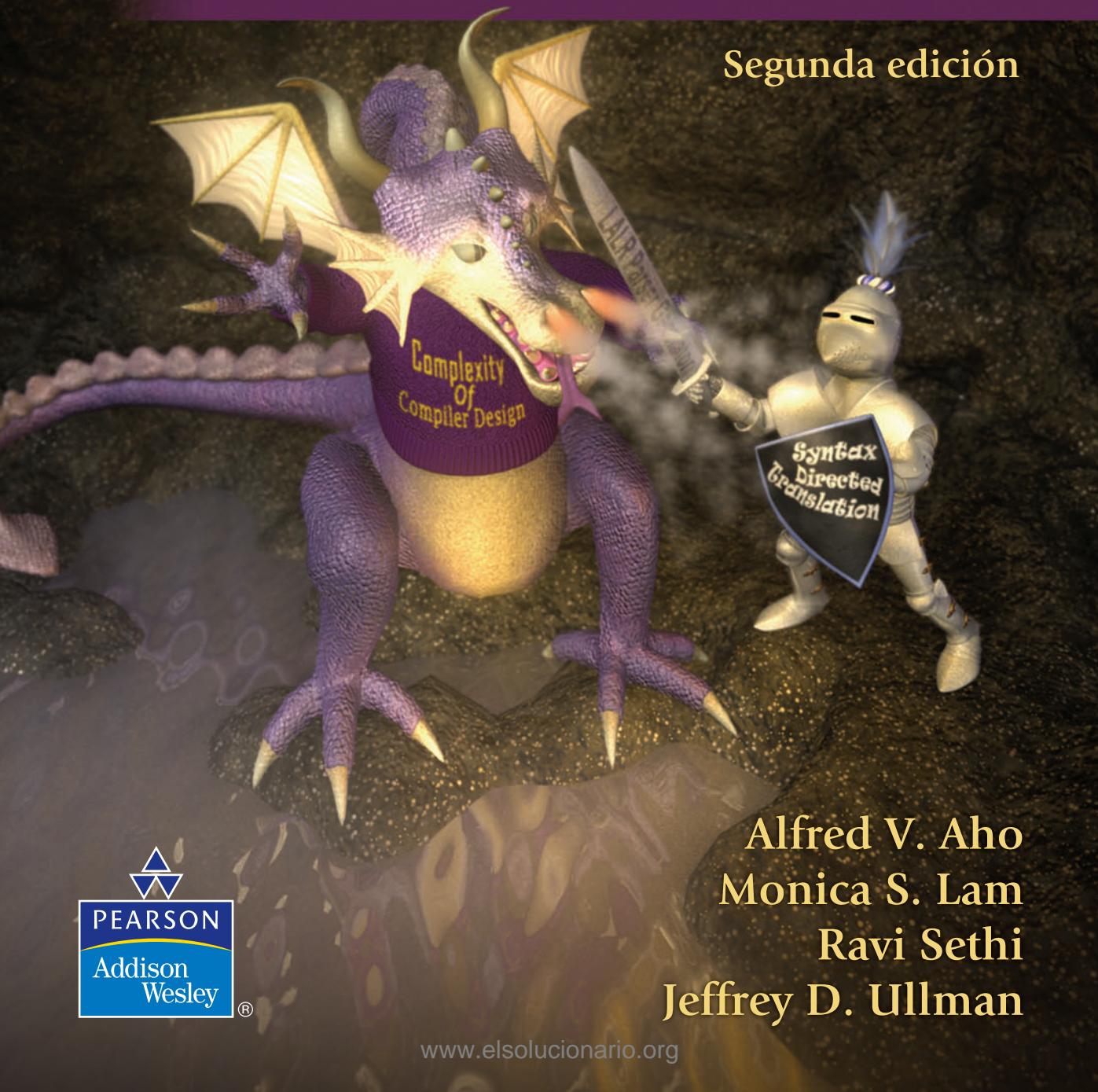


Compiladores

principios, técnicas y herramientas

Segunda edición



Alfred V. Aho
Monica S. Lam

Ravi Sethi
Jeffrey D. Ullman

Compiladores

principios, técnicas y herramientas

Segunda edición

Compiladores

principios, técnicas y herramientas

Segunda edición

Alfred V. Aho

Columbia University

Monica S. Lam

Stanford University

Ravi Sethi

Avaya

Jeffrey D. Ullman

Stanford University

TRADUCCIÓN

Alfonso Vidal Romero Elizondo

Ingeniero en Electrónica y Comunicaciones

REVISIÓN TÉCNICA

Sergio Fuenlabrada Velázquez

Edna M. Miranda Chávez

Adalberto Robles Valadez

Oskar A. Gómez Coronel

Academia de Computación UPIICSA

Instituto Politécnico Nacional

Norma F. Roffe Samaniego

Elda Quiroga González

Departamento de Computación

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

Jesús Alfonso Esparza

Departamento de Computación

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Estado de México



México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

www.elsolucionario.org

AHO, ALFRED V.

COMPILADORES. PRINCIPIOS, TÉCNICAS Y HERRAMIENTAS. Segunda edición

PEARSON EDUCACIÓN, México, 2008

ISBN: 978-970-26-1133-2

Área: Computación

Formato: 18.5 × 23.5 cm

Páginas: 1040

Authorized translation from the English language edition, entitled *Compilers: Principles, techniques and tools, 2nd edition* by Alfred V. Aho, Monica S. Lam Ravi Sethi, Jeffrey D. Ullman, published by Pearson Education, Inc., publishing as Addison-Wesley, Copyright ©2007. All rights reserved.

ISBN: 0321486811

Traducción autorizada de la edición en idioma inglés, *Compilers: Principles, techniques and tools, 2nd edition* de Alfred V. Aho, Monica S. Lam Ravi Sethi, Jeffrey D. Ullman, publicada por Pearson Education, Inc., publicada como Addison-Wesley, Copyright ©2007. Todos los derechos reservados.

Esta edición en español es la única autorizada.

Edición en español

Editor: Luis Miguel Cruz Castillo
e-mail: luis.cruz@pearsoned.com
Editor de desarrollo: Bernardino Gutiérrez Hernández
Supervisor de producción: Enrique Trejo Hernández

Edición en inglés

Publisher	Greg Tobin
Executive Editor	Michael Hirsch
Acquisitions Editor	Matt Goldstein
Project Editor	Katherine Harutunian
Associate Managing Editor	Jeffrey Holcomb
Cover Designer	Joyce Cosentino Wells
Digital Assets Manager	Marianne Groth
Media Producer	Bethany Tidd
Senior Marketing Manager	Michelle Brown
Marketing Assistant	Sarah Milmore
Senior Author Support/ Technology Specialist	Joe Vetere
Senior Manufacturing Buyer	Carol Melville
Cover Image	Scott Ullman of Strange Tonic Productions (www.stragetonnic.com)

SEGUNDA EDICIÓN, 2008

D.R. © 2008 por Pearson Educación de México, S.A. de C.V.
Atlaconulco 500-5o. piso
Col. Industrial Atoto
53519, Naucalpan de Juárez, Edo. de México

Cámara Nacional de la Industria Editorial Mexicana. Reg. Núm. 1031.

Addison-Wesley es una marca registrada de Pearson Educación de México, S.A. de C.V.

Reservados todos los derechos. Ni la totalidad ni parte de esta publicación pueden reproducirse, registrarse o transmitirse, por un sistema de recuperación de información, en ninguna forma ni por ningún medio, sea electrónico, mecánico, fotoquímico, magnético o electroóptico, por fotocopia, grabación o cualquier otro, sin permiso previo por escrito del editor.

El préstamo, alquiler o cualquier otra forma de cesión de uso de este ejemplar requerirá también la autorización del editor o de sus representantes.

ISBN 10: 970-26-1133-4

ISBN 13: 978-970-26-1133-2

Impreso en México. *Printed in Mexico.*

1 2 3 4 5 6 7 8 9 0 - 11 10 09 08

Prefacio

Desde la primera edición de este libro, en 1986, el mundo ha cambiado en forma considerable. Los lenguajes de programación han evolucionado para presentar nuevos problemas de compilación. Las arquitecturas computacionales ofrecen una variedad de recursos, de los cuales el diseñador de compiladores debe sacar ventaja. Tal vez lo más interesante sea que la venerable tecnología de la optimización de código ha encontrado un uso fuera de los compiladores. Ahora se utiliza en herramientas que buscan errores en el software, y lo que es más importante, buscan brechas de seguridad en el código existente. Y gran parte de la tecnología de punta (gramática, expresiones regulares, analizadores y traductores orientados a la sintaxis) tiene todavía un amplio uso.

Por ende, la filosofía que manejamos en la edición anterior de este libro no ha cambiado. Reconocemos que sólo unos cuantos lectores llegarán a crear (o inclusive a mantener) un compilador para un lenguaje de programación importante. Sin embargo, los modelos, la teoría y los algoritmos asociados con un compilador pueden aplicarse a una gran variedad de problemas en el diseño y desarrollo de software. Por lo tanto, destacamos los problemas que se encuentran con más frecuencia durante el diseño de un procesador de lenguaje, sin importar el lenguaje origen ni la máquina de destino.

Cómo utilizar este libro

Se requieren cuando menos dos trimestres, o incluso dos semestres, para cubrir todo el material de este libro (o al menos la mayor parte). Lo ideal es cubrir la primera mitad del texto en un curso universitario y la segunda mitad (que hace énfasis en la optimización de código) en otro curso, a nivel posgrado o maestría. He aquí una descripción general de los capítulos:

El capítulo 1 contiene material de motivación, y también presenta algunos puntos sobre los antecedentes de la arquitectura computacional y los principios de los lenguajes de programación.

El capítulo 2 desarrolla un compilador en miniatura y presenta muchos de los conceptos importantes, que se desarrollarán en los capítulos siguientes. El compilador en sí aparece en el apéndice.

El capítulo 3 cubre el análisis léxico, las expresiones regulares, las máquinas de estado finito y las herramientas para generar exploradores. Este material es fundamental para el procesamiento de texto de todo tipo.

El capítulo 4 cubre los principales métodos de análisis, de arriba-abajo (recursivo descendente, LL) y de abajo-abajo (LR y sus variantes).

El capítulo 5 introduce las ideas principales en las definiciones y las traducciones orientadas a la sintaxis.

El capítulo 6 toma la teoría del capítulo 5 y muestra cómo usarla para generar código intermedio para un lenguaje de programación ordinario.

El capítulo 7 cubre los entornos en tiempo de ejecución, en especial la administración de la pila en tiempo de ejecución y la recolección de basura.

El capítulo 8 trata de la generación de código objeto. Cubre la construcción de los bloques fundamentales, la generación de código a partir de las expresiones y los bloques básicos, así como las técnicas de asignación de registros.

El capítulo 9 introduce la tecnología de la optimización de código, incluyendo los diagramas de flujo, los frameworks de flujo de datos y los algoritmos iterativos para resolver estos frameworks.

El capítulo 10 cubre la optimización a nivel de instrucciones. Se destaca aquí la extracción del paralelismo de las secuencias pequeñas de instrucciones, y cómo programarlas en procesadores individuales que puedan realizar más de función a la vez.

El capítulo 11 habla de la detección y explotación del paralelismo a mayor escala. Aquí, se destacan los códigos numéricos que cuentan con muchos ciclos estrechos que varían a través de los arreglos multidimensionales.

El capítulo 12 explica el análisis entre procedimientos. Cubre el análisis y uso de alias en los apuntadores, y el análisis de los flujos de datos que toma en cuenta la secuencia de llamadas a procedimientos que llegan a un punto dado en el código.

En Columbia, Harvard y Stanford se han impartido cursos basados en el material de este libro. En Columbia se ofrece de manera regular un curso a estudiantes universitarios del último año o estudiantes postgraduados sobre los lenguajes de programación y los traductores, usando el material de los primeros ocho capítulos. Algo que destaca de este curso es un proyecto que dura todo un semestre, en el cual los estudiantes trabajan en pequeños equipos para crear e implementar un pequeño lenguaje que ellos mismos diseñan. Los lenguajes que crean los estudiantes han abarcado diversos dominios de aplicaciones, incluyendo el cálculo de cuantos, la síntesis de música, los gráficos de computadora, juegos, operaciones con matrices y muchas otras áreas más. Los estudiantes utilizan generadores de componentes de compiladores como ANTLR, Lex y Yacc, y las técnicas de traducción orientadas a la sintaxis que se describen en los capítulos 2 y 5 para construir sus compiladores. También se ofrece un curso de seguimiento para graduados, que se enfoca en el material que viene en los capítulos 9 a 12, en donde se enfatiza la generación y optimización de código para las máquinas contemporáneas, incluyendo los procesadores de redes y las arquitecturas con múltiples procesadores.

En Stanford, un curso introductorio de un tetramestre apenas cubre el material que viene en los capítulos 1 a 8, aunque hay una introducción a la optimización de código global del capítulo 9. En el segundo curso de compiladores se cubren los capítulos 9 a 12, además del material avanzado

sobre la recolección de basura del capítulo 7. Los estudiantes utilizan un sistema llamado *Joeq*, basado en Java y desarrollado por la comunidad local, para implementar los algoritmos de análisis de los flujos de datos.

Requisitos previos

El lector debe poseer cierto “conocimiento orientado a las ciencias computacionales”, lo que incluye por lo menos un segundo curso sobre programación, además de cursos sobre estructuras de datos y matemáticas discretas. Es útil tener un conocimiento sobre varios lenguajes de programación.

Ejercicios

Este libro contiene gran cantidad de ejercicios para casi todas las secciones. Para indicar los ejercicios difíciles, o partes de ellos, utilizamos un signo de admiración. Los ejercicios todavía más difíciles tienen doble signo de admiración.

Tareas en línea de Gradiance

Una nueva característica de la segunda edición es que hay un conjunto complementario de tareas en línea, las cuales utilizan una tecnología desarrollada por Gradiance Corp. Los instructores pueden asignar estas tareas a su clase; los estudiantes que no estén inscritos en una clase pueden inscribirse en una “clase ómnibus” que les permita realizar las tareas como tutorial (sin una clase creada por un instructor). Las preguntas de Gradiance tienen la misma apariencia que las preguntas ordinarias, sólo que se muestran sus soluciones. Si realiza una elección incorrecta, recibe asesoría, o retroalimentación, específica para ayudarle a corregir su error. Los profesores usuarios de este libro pueden utilizar Gradiance como apoyo a sus clases, y cada alumno que compre el libro puede obtener un código para acceder a los ejercicios de su profesor. Para mayor información consulte a su representante de Pearson Educación o visite el sitio Web de este libro. Cabe aclarar que esta información está en inglés.

Soporte en World Wide Web

En el sitio Web de este libro:

www.pearsoneducacion.net/aho

Encontrará una fe de erratas, que actualizaremos a medida que vayamos detectando los errores, y materiales de respaldo. Esperamos tener disponibles los apuntes para cada curso relacionando con compiladores a medida que lo impartamos, incluyendo tareas, soluciones y exámenes. Planeamos publicar también descripciones de los compiladores importantes escritos por sus implementadores. Cabe destacar que todo el material y este sitio se encuentran en inglés.

Agradecimientos

El arte de la portada fue realizado por S. D. Ullman de Strange Tonic Productions.

John Bentley nos proporcionó muchos comentarios sobre varios capítulos desde un borrador anterior de este libro. Recibimos valiosos comentarios y fe de erratas de: Domenico Bianculli,

Peter Bosch, Marcio Buss, Marc Eaddy, Stephen Edwards, Vibhav Garg, Kim Hazelwood, Gaurav Kc, Wei Li, Mike Smith, Art Stamness, Krysta Svore, Olivier Tardieu y Jia Zeng. Agradecemos en gran medida la ayuda de todas estas personas. Desde luego, los errores restantes son nuestros.

Además, Monica quisiera agradecer a sus colegas en el equipo del compilador SUIF por una lección de 18 años sobre el proceso de compilación: Gerald Aigner, Dzintars Avots, Saman Amarasinghe, Jennifer Anderson, Michael Carbin, Gerald Cheong, Amer Diwan, Robert French, Anwar Ghuloum, Mary Hall, John Hennessy, David Heine, Shih-Wei Liao, Amy Lim, Benjamin Livshits, Michael Martin, Dror Maydan, Todd Mowry, Brian Murphy, Jeffrey Oplinger, Karen Pieper, Martin Rinard, Olatunji Ruwase, Constantine Sapuntzakis, Patrick Sathyathanathan, Michael Smith, Steven Tjiang, Chau-Wen Tseng, Christopher Unkel, John Whaley, Robert Wilson, Christopher Wilson, y Michael Wolf.

A. V. A., Chatham NJ
M. S. L., Menlo Park CA
R. S., Far Hills NJ
J. D. U., Stanford CA
Junio, 2006

Tabla de contenido

Prefacio	v
1 Introducción	1
1.1 Procesadores de lenguaje	1
1.1.1 Ejercicios para la sección 1.1.	3
1.2 La estructura de un compilador	4
1.2.1 Análisis de léxico.	5
1.2.2 Análisis sintáctico	8
1.2.3 Análisis semántico	8
1.2.4 Generación de código intermedio.	9
1.2.5 Optimización de código	10
1.2.6 Generación de código.	10
1.2.7 Administración de la tabla de símbolos	11
1.2.8 El agrupamiento de fases en pasadas	11
1.2.9 Herramientas de construcción de compiladores	12
1.3 La evolución de los lenguajes de programación.	12
1.3.1 El avance a los lenguajes de alto nivel	13
1.3.2 Impactos en el compilador	14
1.3.3 Ejercicios para la sección 1.3.	14
1.4 La ciencia de construir un compilador	15
1.4.1 Modelado en el diseño e implementación de compiladores	15
1.4.2 La ciencia de la optimización de código	15
1.5 Aplicaciones de la tecnología de compiladores	17
1.5.1 Implementación de lenguajes de programación de alto nivel.	17
1.5.2 Optimizaciones para las arquitecturas de computadoras	19
1.5.3 Diseño de nuevas arquitecturas de computadoras	21
1.5.4 Traducciones de programas.	22
1.5.5 Herramientas de productividad de software	23
1.6 Fundamentos de los lenguajes de programación	25
1.6.1 La distinción entre estático y dinámico	25
1.6.2 Entornos y estados	26
1.6.3 Alcance estático y estructura de bloques	28
1.6.4 Control de acceso explícito	31
1.6.5 Alcance dinámico.	31
1.6.6 Mecanismos para el paso de parámetros.	33

1.6.7	Uso de alias	35
1.6.8	Ejercicios para la sección 1.6.	35
1.7	Resumen del capítulo 1	36
1.8	Referencias para el capítulo 1	38
2	Un traductor simple orientado a la sintaxis	39
2.1	Introducción	40
2.2	Definición de sintaxis.	42
2.2.1	Definición de gramáticas	42
2.2.2	Derivaciones	44
2.2.3	Árboles de análisis sintáctico	45
2.2.4	Ambigüedad	47
2.2.5	Asociatividad de los operadores	48
2.2.6	Precedencia de operadores	48
2.2.7	Ejercicios para la sección 2.2.	51
2.3	Traducción orientada a la sintaxis.	52
2.3.1	Notación postfija	53
2.3.2	Atributos sintetizados	54
2.3.3	Definiciones simples orientadas a la sintaxis	56
2.3.4	Recorridos de los árboles	56
2.3.5	Esquemas de traducción.	57
2.3.6	Ejercicios para la sección 2.3.	60
2.4	Ánálisis sintáctico	60
2.4.1	Análisis sintáctico tipo arriba-abajo	61
2.4.2	Análisis sintáctico predictivo.	64
2.4.3	Cuándo usar las producciones ϵ	65
2.4.4	Diseño de un analizador sintáctico predictivo	66
2.4.5	Recursividad a la izquierda.	67
2.4.6	Ejercicios para la sección 2.4.	68
2.5	Un traductor para las expresiones simples	68
2.5.1	Sintaxis abstracta y concreta	69
2.5.2	Adaptación del esquema de traducción	70
2.5.3	Procedimientos para los no terminales	72
2.5.4	Simplificación del traductor	73
2.5.5	El programa completo	74
2.6	Ánálisis léxico.	76
2.6.1	Eliminación de espacio en blanco y comentarios	77
2.6.2	Lectura adelantada	78
2.6.3	Constantes	78
2.6.4	Reconocimiento de palabras clave e identificadores	79
2.6.5	Un analizador léxico	81
2.6.6	Ejercicios para la sección 2.6.	84
2.7	Tablas de símbolos	85
2.7.1	Tabla de símbolos por alcance.	86
2.7.2	El uso de las tablas de símbolos	89

2.8	Generación de código intermedio	91
2.8.1	Dos tipos de representaciones intermedias	91
2.8.2	Construcción de árboles sintácticos	92
2.8.3	Comprobación estática	97
2.8.4	Código de tres direcciones	99
2.8.5	Ejercicios para la sección 2.8	105
2.9	Resumen del capítulo 2	105
3	Ánálsis léxico	109
3.1	La función del analizador léxico	109
3.1.1	Comparación entre análisis léxico y análisis sintáctico	110
3.1.2	Tokens, patrones y lexemas	111
3.1.3	Atributos para los tokens	112
3.1.4	Errores léxicos	113
3.1.5	Ejercicios para la sección 3.1	114
3.2	Uso de búfer en la entrada	115
3.2.1	Pares de búferes	115
3.2.2	Centinelas	116
3.3	Especificación de los tokens	116
3.3.1	Cadenas y lenguajes	117
3.3.2	Operaciones en los lenguajes	119
3.3.3	Expresiones regulares	120
3.3.4	Definiciones regulares	123
3.3.5	Extensiones de las expresiones regulares	124
3.3.6	Ejercicios para la sección 3.3	125
3.4	Reconocimiento de tokens	128
3.4.1	Diagramas de transición de estados	130
3.4.2	Reconocimiento de las palabras reservadas y los identificadores	132
3.4.3	Finalización del bosquejo	133
3.4.4	Arquitectura de un analizador léxico basado en diagramas de transición de estados	134
3.4.5	Ejercicios para la sección 3.4	136
3.5	El generador de analizadores léxicos Lex	140
3.5.1	Uso de Lex	140
3.5.2	Estructura de los programas en Lex	141
3.5.3	Resolución de conflictos en Lex	144
3.5.4	El operador adelantado	144
3.5.5	Ejercicios para la sección 3.5	146
3.6	Autómatas finitos	147
3.6.1	Autómatas finitos no deterministas	147
3.6.2	Tablas de transición	148
3.6.3	Aceptación de las cadenas de entrada mediante los autómatas	149
3.6.4	Autómatas finitos deterministas	149
3.6.5	Ejercicios para la sección 3.6	151

3.7	De las expresiones regulares a los autómatas	152
3.7.1	Conversión de un AFN a AFD	152
3.7.2	Simulación de un AFN	156
3.7.3	Eficiencia de la simulación de un AFN	157
3.7.4	Construcción de un AFN a partir de una expresión regular	159
3.7.5	Eficiencia de los algoritmos de procesamiento de cadenas	163
3.7.6	Ejercicios para la sección 3.7	166
3.8	Diseño de un generador de analizadores léxicos	166
3.8.1	La estructura del analizador generado	167
3.8.2	Coincidencia de patrones con base en los AFNs	168
3.8.3	AFDs para analizadores léxicos	170
3.8.4	Implementación del operador de preanálisis	171
3.8.5	Ejercicios para la sección 3.8	172
3.9	Optimización de los buscadores por concordancia de patrones basados en AFD	173
3.9.1	Estados significativos de un AFN	173
3.9.2	Funciones calculadas a partir del árbol sintáctico	175
3.9.3	Cálculo de <i>anulable</i> , <i>primerapos</i> y <i>ultimapos</i>	176
3.9.4	Cálculo de <i>siguienteapos</i>	177
3.9.5	Conversión directa de una expresión regular a un AFD	179
3.9.6	Minimización del número de estados de un AFD	180
3.9.7	Minimización de estados en los analizadores léxicos	184
3.9.8	Intercambio de tiempo por espacio en la simulación de un AFD	185
3.9.9	Ejercicios para la sección 3.9	186
3.10	Resumen del capítulo 3	187
3.11	Referencias para el capítulo 3	189
4	Análisis sintáctico	191
4.1	Introducción	192
4.1.1	La función del analizador sintáctico	192
4.1.2	Representación de gramáticas	193
4.1.3	Manejo de los errores sintácticos	194
4.1.4	Estrategias para recuperarse de los errores	195
4.2	Gramáticas libres de contexto	197
4.2.1	La definición formal de una gramática libre de contexto	197
4.2.2	Convenciones de notación	198
4.2.3	Derivaciones	199
4.2.4	Árboles de análisis sintáctico y derivaciones	201
4.2.5	Ambigüedad	203
4.2.6	Verificación del lenguaje generado por una gramática	204
4.2.7	Comparación entre gramáticas libres de contexto y expresiones regulares	205
4.2.8	Ejercicios para la sección 4.2	206
4.3	Escritura de una gramática	209
4.3.1	Comparación entre análisis léxico y análisis sintáctico	209
4.3.2	Eliminación de la ambigüedad	210

4.3.3	Eliminación de la recursividad por la izquierda.	212
4.3.4	Factorización por la izquierda.	214
4.3.5	Construcciones de lenguajes que no son libres de contexto.	215
4.3.6	Ejercicios para la sección 4.3.	216
4.4	Análisis sintáctico descendente	217
4.4.1	Análisis sintáctico de descenso recursivo	219
4.4.2	PRIMERO y SIGUIENTE	220
4.4.3	Gramáticas LL(1)	222
4.4.4	Análisis sintáctico predictivo no recursivo	226
4.4.5	Recuperación de errores en el análisis sintáctico predictivo	228
4.4.6	Ejercicios para la sección 4.4.	231
4.5	Análisis sintáctico ascendente	233
4.5.1	Reducciones	234
4.5.2	Poda de mangos	235
4.5.3	Análisis sintáctico de desplazamiento-reducción	236
4.5.4	Conflictos durante el análisis sintáctico de desplazamiento-reducción	238
4.5.5	Ejercicios para la sección 4.5.	240
4.6	Introducción al análisis sintáctico LR: SLR (LR simple)	241
4.6.1	¿Por qué analizadores sintácticos LR?	241
4.6.2	Los elementos y el autómata LR(0)	242
4.6.3	El algoritmo de análisis sintáctico LR	248
4.6.4	Construcción de tablas de análisis sintáctico SLR.	252
4.6.5	Prefijos viables	256
4.6.6	Ejercicios para la sección 4.6.	257
4.7	Analizadores sintácticos LR más poderosos	259
4.7.1	Elementos LR(1) canónicos.	260
4.7.2	Construcción de conjuntos de elementos LR(1).	261
4.7.3	Tablas de análisis sintáctico LR(1) canónico.	265
4.7.4	Construcción de tablas de análisis sintáctico LALR	266
4.7.5	Construcción eficiente de tablas de análisis sintáctico LALR	270
4.7.6	Compactación de las tablas de análisis sintáctico LR	275
4.7.7	Ejercicios para la sección 4.7.	277
4.8	Uso de gramáticas ambiguas.	278
4.8.1	Precedencia y asociatividad para resolver conflictos	279
4.8.2	La ambigüedad del “else colgante”	281
4.8.3	Recuperación de errores en el análisis sintáctico LR	283
4.8.4	Ejercicios para la sección 4.8.	285
4.9	Generadores de analizadores sintácticos.	287
4.9.1	El generador de analizadores sintácticos Yacc.	287
4.9.2	Uso de Yacc con gramáticas ambiguas.	291
4.9.3	Creación de analizadores léxicos de Yacc con Lex.	294
4.9.4	Recuperación de errores en Yacc.	295
4.9.5	Ejercicios para la sección 4.9.	297
4.10	Resumen del capítulo 4	297
4.11	Referencias para el capítulo 4	300

5 Traducción orientada por la sintaxis	303
5.1 Definiciones dirigidas por la sintaxis	304
5.1.1 Atributos heredados y sintetizados	304
5.1.2 Evaluación de una definición dirigida por la sintaxis en los nodos de un árbol de análisis sintáctico	306
5.1.3 Ejercicios para la sección 5.1	309
5.2 Órdenes de evaluación para las definiciones dirigidas por la sintaxis	310
5.2.1 Gráficos de dependencias	310
5.2.2 Orden de evaluación	312
5.2.3 Definiciones con atributos sintetizados	312
5.2.4 Definiciones con atributos heredados	313
5.2.5 Reglas semánticas con efectos adicionales controlados	314
5.2.6 Ejercicios para la sección 5.2	317
5.3 Aplicaciones de la traducción orientada por la sintaxis	318
5.3.1 Construcción de árboles de análisis sintáctico	318
5.3.2 La estructura de tipos	321
5.3.3 Ejercicios para la sección 5.3	323
5.4 Esquemas de traducción orientados por la sintaxis	323
5.4.1 Esquemas de traducción postfijos	324
5.4.2 Implementación de esquemas de traducción orientados a la sintaxis postfijo con la pila del analizador sintáctico	325
5.4.3 Esquemas de traducción orientados a la sintaxis con acciones dentro de las producciones	327
5.4.4 Eliminación de la recursividad por la izquierda de los esquemas de traducción orientados a la sintaxis	328
5.4.5 Esquemas de traducción orientados a la sintaxis para definiciones con atributos heredados por la izquierda	331
5.4.6 Ejercicios para la sección 5.4	336
5.5 Implementación de definiciones dirigidas por la sintaxis con atributos heredados por la izquierda	337
5.5.1 Traducción durante el análisis sintáctico de descenso recursivo	338
5.5.2 Generación de código al instante	340
5.5.3 Las definiciones dirigidas por la sintaxis con atributos heredados por la izquierda y el análisis sintáctico LL	343
5.5.4 Análisis sintáctico ascendente de las definiciones dirigidas por la sintaxis con atributos heredados por la izquierda	348
5.5.5 Ejercicios para la sección 5.5	352
5.6 Resumen del capítulo 5	353
5.7 Referencias para el capítulo 5	354
6 Generación de código intermedio	357
6.1 Variantes de los árboles sintácticos	358
6.1.1 Grafo dirigido acíclico para las expresiones	359
6.1.2 El método número de valor para construir GDAs	360
6.1.3 Ejercicios para la sección 6.1	362

6.2	Código de tres direcciones	363
6.2.1	Direcciones e instrucciones	364
6.2.2	Cuádruplos	366
6.2.3	Tripletas	367
6.2.4	Forma de asignación individual estática	369
6.2.5	Ejercicios para la sección 6.2	370
6.3	Tipos y declaraciones	370
6.3.1	Expresiones de tipos	371
6.3.2	Equivalencia de tipos	372
6.3.3	Declaraciones	373
6.3.4	Distribución del almacenamiento para los nombres locales	373
6.3.5	Secuencias de las declaraciones	376
6.3.6	Campos en registros y clases	376
6.3.7	Ejercicios para la sección 6.3	378
6.4	Traducción de expresiones	378
6.4.1	Operaciones dentro de expresiones	378
6.4.2	Traducción incremental	380
6.4.3	Direccionamiento de los elementos de un arreglo	381
6.4.4	Traducción de referencias a arreglos	383
6.4.5	Ejercicios para la sección 6.4	384
6.5	Comprobación de tipos	386
6.5.1	Reglas para la comprobación de tipos	387
6.5.2	Conversiones de tipos	388
6.5.3	Sobrecarga de funciones y operadores	390
6.5.4	Inferencia de tipos y funciones polimórficas	391
6.5.5	Un algoritmo para la unificación	395
6.5.6	Ejercicios para la sección 6.5	398
6.6	Flujo de control	399
6.6.1	Expresiones booleanas	399
6.6.2	Código de corto circuito	400
6.6.3	Instrucciones de flujo de control	401
6.6.4	Traducción del flujo de control de las expresiones booleanas	403
6.6.5	Evitar gotos redundantes	405
6.6.6	Valores booleanos y código de salto	408
6.6.7	Ejercicios para la sección 6.6	408
6.7	Parcheo de retroceso (backpatch)	410
6.7.1	Generación de código de una pasada, mediante parcheo de retroceso	410
6.7.2	Técnica de parcheo de retroceso para las expresiones booleanas	411
6.7.3	Instrucciones de flujo de control	413
6.7.4	Instrucciones break, continue y goto	416
6.7.5	Ejercicios para la sección 6.7	417
6.8	Instrucciones switch	418
6.8.1	Traducción de las instrucciones switch	419
6.8.2	Traducción orientada por la sintaxis de las instrucciones switch	420
6.8.3	Ejercicios para la sección 6.8	421

6.9	Código intermedio para procedimientos	422
6.10	Resumen del capítulo 6	424
6.11	Referencias para el capítulo 6	425
7	Entornos en tiempo de ejecución	427
7.1	Organización del almacenamiento	427
7.1.1	Asignación de almacenamiento estática y dinámica	429
7.2	Asignación de espacio en la pila	430
7.2.1	Árboles de activación	430
7.2.2	Registros de activación	433
7.2.3	Secuencias de llamadas	436
7.2.4	Datos de longitud variable en la pila	438
7.2.5	Ejercicios para la sección 7.2	440
7.3	Acceso a los datos no locales en la pila	441
7.3.1	Acceso a los datos sin procedimientos anidados	442
7.3.2	Problemas con los procedimientos anidados	442
7.3.3	Un lenguaje con declaraciones de procedimientos anidados	443
7.3.4	Profundidad de anidamiento	443
7.3.5	Enlace de acceso	445
7.3.6	Manipulación de los enlaces de acceso	447
7.3.7	Enlaces de acceso para los parámetros de procedimientos	448
7.3.8	Estructura de datos Display	449
7.3.9	Ejercicios para la sección 7.3	451
7.4	Administración del montículo	452
7.4.1	El administrador de memoria	453
7.4.2	La jerarquía de memoria de una computadora	454
7.4.3	Localidad en los programas	455
7.4.4	Reducción de la fragmentación	457
7.4.5	Solicitudes de desasignación manual	460
7.4.6	Ejercicios para la sección 7.4	463
7.5	Introducción a la recolección de basura	463
7.5.1	Metas de diseño para los recolectores de basura	464
7.5.2	Capacidad de alcance	466
7.5.3	Recolectores de basura con conteo de referencias	468
7.5.4	Ejercicios para la sección 7.5	470
7.6	Introducción a la recolección basada en el rastreo	470
7.6.1	Un recolector básico “marcar y limpiar”	471
7.6.2	Abstracción básica	473
7.6.3	Optimización de “marcar y limpiar”	475
7.6.4	Recolectores de basura “marcar y compactar”	476
7.6.5	Recolectores de copia	478
7.6.6	Comparación de costos	482
7.6.7	Ejercicios para la sección 7.6	482
7.7	Recolección de basura de pausa corta	483
7.7.1	Recolección de basura incremental	483

7.7.2	Análisis de capacidad alcance incremental	485
7.7.3	Fundamentos de la recolección parcial	487
7.7.4	Recolección de basura generacional	488
7.7.5	El algoritmo del tren	490
7.7.6	Ejercicios para la sección 7.7	493
7.8	Temas avanzados sobre la recolección de basura	494
7.8.1	Recolección de basura paralela y concurrente	495
7.8.2	Reubicación parcial de objetos	497
7.8.3	Recolección conservadora para lenguajes inseguros	498
7.8.4	Referencias débiles	498
7.8.5	Ejercicios para la sección 7.8	499
7.9	Resumen del capítulo 7	500
7.10	Referencias para el capítulo 7	502
8	Generación de código	505
8.1	Cuestiones sobre el diseño de un generador de código	506
8.1.1	Entrada del generador de código	507
8.1.2	El programa destino	507
8.1.3	Selección de instrucciones	508
8.1.4	Asignación de registros	510
8.1.5	Orden de evaluación	511
8.2	El lenguaje destino	512
8.2.1	Un modelo simple de máquina destino	512
8.2.2	Costos del programa y las instrucciones	515
8.2.3	Ejercicios para la sección 8.2	516
8.3	Direcciones en el código destino	518
8.3.1	Asignación estática	518
8.3.2	Asignación de pila	520
8.3.3	Direcciones para los nombres en tiempo de ejecución	522
8.3.4	Ejercicios para la sección 8.3	524
8.4	Bloques básicos y grafos de flujo	525
8.4.1	Bloques básicos	526
8.4.2	Información de siguiente uso	528
8.4.3	Grafos de flujo	529
8.4.4	Representación de los grafos de flujo	530
8.4.5	Ciclos	531
8.4.6	Ejercicios para la sección 8.4	531
8.5	Optimización de los bloques básicos	533
8.5.1	La representación en GDA de los bloques básicos	533
8.5.2	Búsqueda de subexpresiones locales comunes	534
8.5.3	Eliminación del código muerto	535
8.5.4	El uso de identidades algebraicas	536
8.5.5	Representación de referencias a arreglos	537
8.5.6	Asignaciones de apuntadores y llamadas a procedimientos	539
8.5.7	Reensamblado de los bloques básicos a partir de los GDAs	539
8.5.8	Ejercicios para la sección 8.5	541

8.6	Un generador de código simple	542
8.6.1	Descriptores de registros y direcciones	543
8.6.2	El algoritmo de generación de código.	544
8.6.3	Diseño de la función <i>obtenReg</i>	547
8.6.4	Ejercicios para la sección 8.6.	548
8.7	Optimización de mirilla (peephole)	549
8.7.1	Eliminación de instrucciones redundantes de carga y almacenamiento	550
8.7.2	Eliminación de código inalcanzable	550
8.7.3	Optimizaciones del flujo de control	551
8.7.4	Simplificación algebraica y reducción por fuerza	552
8.7.5	Uso de características específicas de máquina	552
8.7.6	Ejercicios para la sección 8.7.	553
8.8	Repartición y asignación de registros.	553
8.8.1	Repartición global de registros	553
8.8.2	Conteos de uso	554
8.8.3	Asignación de registros para ciclos externos	556
8.8.4	Asignación de registros mediante la coloración de grafos	556
8.8.5	Ejercicios para la sección 8.8.	557
8.9	Selección de instrucciones mediante la rescritura de árboles.	558
8.9.1	Esquemas de traducción de árboles	558
8.9.2	Generación de código mediante el revestimiento de un árbol de entrada	560
8.9.3	Coincidencias de los patrones mediante el análisis sintáctico	563
8.9.4	Rutinas para la comprobación semántica	565
8.9.5	Proceso general para igualar árboles	565
8.9.6	Ejercicios para la sección 8.9.	567
8.10	Generación de código óptimo para las expresiones	567
8.10.1	Números de Ershov	567
8.10.2	Generación de código a partir de árboles de expresión etiquetados	568
8.10.3	Evaluación de expresiones con un suministro insuficiente de registros	570
8.10.4	Ejercicios para la sección 8.10.	572
8.11	Generación de código de programación dinámica	573
8.11.1	Evaluación contigua.	574
8.11.2	El algoritmo de programación dinámica.	575
8.11.3	Ejercicios para la sección 8.11.	577
8.12	Resumen del capítulo 8	578
8.13	Referencias para el capítulo 8	579
9	Optimizaciones independientes de la máquina	583
9.1	Las fuentes principales de optimización	584
9.1.1	Causas de redundancia.	584
9.1.2	Un ejemplo abierto: Quicksort	585
9.1.3	Transformaciones que preservan la semántica	586
9.1.4	Subexpresiones comunes globales	588
9.1.5	Propagación de copias	590
9.1.6	Eliminación de código muerto.	591

9.1.7	Movimiento de código	592
9.1.8	Variables de inducción y reducción en fuerza	592
9.1.9	Ejercicios para la sección 9.1	596
9.2	Introducción al análisis del flujo de datos	597
9.2.1	La abstracción del flujo de datos	597
9.2.2	El esquema del análisis del flujo de datos	599
9.2.3	Esquemas del flujo de datos en bloques básicos	600
9.2.4	Definiciones de alcance	601
9.2.5	Análisis de variables vivas	608
9.2.6	Expresiones disponibles	610
9.2.7	Resumen	614
9.2.8	Ejercicios para la sección 9.2	615
9.3	Fundamentos del análisis del flujo de datos	618
9.3.1	Semi-lattices	618
9.3.2	Funciones de transferencia	623
9.3.3	El algoritmo iterativo para los marcos de trabajo generales	626
9.3.4	Significado de una solución de un flujo de datos	628
9.3.5	Ejercicios para la sección 9.3	631
9.4	Propagación de constantes	632
9.4.1	Valores del flujo de datos para el marco de trabajo de propagación de constantes	633
9.4.2	La reunión para el marco de trabajo de propagación de constantes	633
9.4.3	Funciones de transferencia para el marco de trabajo de propagación de constantes	634
9.4.4	Monotonía en el marco de trabajo de propagación de constantes	635
9.4.5	La distributividad del marco de trabajo de propagación de constantes	635
9.4.6	Interpretación de los resultados	637
9.4.7	Ejercicios para la sección 9.4	637
9.5	Eliminación de redundancia parcial	639
9.5.1	Los orígenes de la redundancia	639
9.5.2	¿Puede eliminarse toda la redundancia?	642
9.5.3	El problema del movimiento de código diferido	644
9.5.4	Anticipación de las expresiones	645
9.5.5	El algoritmo de movimiento de código diferido	646
9.5.6	Ejercicios para la sección 9.5	655
9.6	Ciclos en los grafos de flujo	655
9.6.1	Dominadores	656
9.6.2	Ordenamiento “primero en profundidad”	660
9.6.3	Aristas en un árbol de expansión con búsqueda en profundidad	661
9.6.4	Aristas posteriores y capacidad de reducción	662
9.6.5	Profundidad de un grafo de flujo	665
9.6.6	Ciclos naturales	665
9.6.7	Velocidad de convergencia de los algoritmos de flujos de datos iterativos	667
9.6.8	Ejercicios para la sección 9.6	669

9.7	Análisis basado en regiones	672
9.7.1	Regiones	672
9.7.2	Jerarquías de regiones para grafos de flujo reducibles	673
9.7.3	Generalidades de un análisis basado en regiones	676
9.7.4	Suposiciones necesarias sobre las funciones transformación	678
9.7.5	Un algoritmo para el análisis basado en regiones	680
9.7.6	Manejo de grafos de flujo no reducibles	684
9.7.7	Ejercicios para la sección 9.7	686
9.8	Análisis simbólico	686
9.8.1	Expresiones afines de las variables de referencia	687
9.8.2	Formulación del problema de flujo de datos	689
9.8.3	Análisis simbólico basado en regiones	694
9.8.4	Ejercicios para la sección 9.8	699
9.9	Resumen del capítulo 9	700
9.10	Referencias para el capítulo 9	703
10	Paralelismo a nivel de instrucción	707
10.1	Arquitecturas de procesadores	708
10.1.1	Canalizaciones de instrucciones y retrasos de bifurcación	708
10.1.2	Ejecución canalizada	709
10.1.3	Emisión de varias instrucciones	710
10.2	Restricciones de la programación del código	710
10.2.1	Dependencia de datos	711
10.2.2	Búsqueda de dependencias entre accesos a memoria	712
10.2.3	Concesiones entre el uso de registros y el paralelismo	713
10.2.4	Ordenamiento de fases entre la asignación de registros y la programación de código	716
10.2.5	Dependencia del control	716
10.2.6	Soporte de ejecución especulativa	717
10.2.7	Un modelo de máquina básico	719
10.2.8	Ejercicios para la sección 10.2	720
10.3	Programación de bloques básicos	721
10.3.1	Grafos de dependencia de datos	722
10.3.2	Programación por lista de bloques básicos	723
10.3.3	Órdenes topológicos priorizados	725
10.3.4	Ejercicios para la sección 10.3	726
10.4	Programación de código global	727
10.4.1	Movimiento de código primitivo	728
10.4.2	Movimiento de código hacia arriba	730
10.4.3	Movimiento de código hacia abajo	731
10.4.4	Actualización de las dependencias de datos	732
10.4.5	Algoritmos de programación global	732
10.4.6	Técnicas avanzadas de movimiento de código	736
10.4.7	Interacción con los programadores dinámicos	737
10.4.8	Ejercicios para la sección 10.4	737

10.5	Canalización por software.	738
10.5.1	Introducción	738
10.5.2	Canalización de los ciclos mediante software.	740
10.5.3	Asignación de recursos y generación de código	743
10.5.4	Ciclos de ejecución cruzada	743
10.5.5	Objetivos y restricciones de la canalización por software	745
10.5.6	Un algoritmo de canalización por software	749
10.5.7	Programación de grafos de dependencia de datos acíclicos	749
10.5.8	Programación de grafos de dependencia cíclicos	751
10.5.9	Mejoras a los algoritmos de canalización	758
10.5.10	Expansión modular de variables	758
10.5.11	Instrucciones condicionales	761
10.5.12	Soporte de hardware para la canalización por software	762
10.5.13	Ejercicios para la sección 10.5.	763
10.6	Resumen del capítulo 10	765
10.7	Referencias para el capítulo 10	766
11	Optimización para el paralelismo y la localidad	769
11.1	Conceptos básicos	771
11.1.1	Multiprocesadores	772
11.1.2	Paralelismo en las aplicaciones	773
11.1.3	Paralelismo a nivel de ciclo.	775
11.1.4	Localidad de los datos	777
11.1.5	Introducción a la teoría de la transformación afín.	778
11.2	Multiplicación de matrices: un ejemplo detallado	782
11.2.1	El algoritmo de multiplicación de matrices.	782
11.2.2	Optimizaciones	785
11.2.3	Interferencia de la caché.	788
11.2.4	Ejercicios para la sección 11.2.	788
11.3	Espacios de iteraciones.	788
11.3.1	Construcción de espacios de iteraciones a partir de anidamientos de ciclos	788
11.3.2	Orden de ejecución para los anidamientos de ciclos.	791
11.3.3	Formulación de matrices de desigualdades	791
11.3.4	Incorporación de constantes simbólicas	793
11.3.5	Control del orden de ejecución	793
11.3.6	Cambio de ejes	798
11.3.7	Ejercicios para la sección 11.3.	799
11.4	Índices de arreglos afines	801
11.4.1	Accesos afines	802
11.4.2	Accesos afines y no afines en la práctica	803
11.4.3	Ejercicios para la sección 11.4.	804
11.5	Reutilización de datos	804
11.5.1	Tipos de reutilización.	805
11.5.2	Reutilización propia.	806

11.5.3	Reutilización espacial propia	809
11.5.4	Reutilización de grupo	811
11.5.5	Ejercicios para la sección 11.5	814
11.6	Análisis de dependencias de datos de arreglos	815
11.6.1	Definición de la dependencia de datos de los accesos a arreglos	816
11.6.2	Programación lineal entera	817
11.6.3	La prueba del GCD	818
11.6.4	Heurística para resolver programas lineales enteros	820
11.6.5	Solución de programas lineales enteros generales	823
11.6.6	Resumen	825
11.6.7	Ejercicios para la sección 11.6	826
11.7	Búsqueda del paralelismo sin sincronización	828
11.7.1	Un ejemplo introductorio	828
11.7.2	Particionamientos de espacio afín	830
11.7.3	Restricciones de partición de espacio	831
11.7.4	Resolución de restricciones de partición de espacio	835
11.7.5	Un algoritmo simple de generación de código	838
11.7.6	Eliminación de iteraciones vacías	841
11.7.7	Eliminación de las pruebas de los ciclos más internos	844
11.7.8	Transformaciones del código fuente	846
11.7.9	Ejercicios para la sección 11.7	851
11.8	Sincronización entre ciclos paralelos	853
11.8.1	Un número constante de sincronizaciones	853
11.8.2	Grafos de dependencias del programa	854
11.8.3	Tiempo jerárquico	857
11.8.4	El algoritmo de paralelización	859
11.8.5	Ejercicios para la sección 11.8	860
11.9	Canalizaciones	861
11.9.1	¿Qué es la canalización?	861
11.9.2	Soberrelajación sucesiva (Successive Over-Relaxation, SOR): un ejemplo	863
11.9.3	Ciclos completamente permutables	864
11.9.4	Canalización de ciclos completamente permutables	864
11.9.5	Teoría general	867
11.9.6	Restricciones de partición de tiempo	868
11.9.7	Resolución de restricciones de partición de tiempo mediante el Lema de Farkas	872
11.9.8	Transformaciones de código	875
11.9.9	Paralelismo con sincronización mínima	880
11.9.10	Ejercicios para la sección 11.9	882
11.10	Optimizaciones de localidad	884
11.10.1	Localidad temporal de los datos calculados	885
11.10.2	Contracción de arreglos	885
11.10.3	Intercalación de particiones	887
11.10.4	Reunión de todos los conceptos	890
11.10.5	Ejercicios para la sección 11.10	892

11.11	Otros usos de las transformaciones afines.	893
11.11.1	Máquinas con memoria distribuida	894
11.11.2	Procesadores que emiten múltiples instrucciones.	895
11.11.3	Instrucciones con vectores y SIMD	895
11.11.4	Preobtención.	896
11.12	Resumen del capítulo 11	897
11.13	Referencias para el capítulo 11	899
12	Análisis interprocedural	903
12.1	Conceptos básicos	904
12.1.1	Grafos de llamadas	904
12.1.2	Sensibilidad al contexto	906
12.1.3	Cadenas de llamadas	908
12.1.4	Análisis sensible al contexto basado en la clonación	910
12.1.5	Análisis sensible al contexto basado en el resumen	911
12.1.6	Ejercicios para la sección 12.1	914
12.2	¿Por qué análisis interprocedural?	916
12.2.1	Invocación de métodos virtuales	916
12.2.2	Análisis de alias de apuntadores	917
12.2.3	Paralelización	917
12.2.4	Detección de errores de software y vulnerabilidades	917
12.2.5	Inyección de código SQL	918
12.2.6	Desbordamiento de búfer	920
12.3	Una representación lógica del flujo de datos.	921
12.3.1	Introducción a Datalog.	921
12.3.2	Reglas de Datalog	922
12.3.3	Predicados intensionales y extensionales	924
12.3.4	Ejecución de programas en Datalog.	927
12.3.5	Evaluación incremental de programas en Datalog.	928
12.3.6	Reglas problemáticas en Datalog.	930
12.3.7	Ejercicios para la sección 12.3	932
12.4	Un algoritmo simple de análisis de apuntadores	933
12.4.1	Por qué es difícil el análisis de apuntadores	934
12.4.2	Un modelo para apuntadores y referencias	935
12.4.3	Insensibilidad al flujo.	936
12.4.4	La formulación en Datalog	937
12.4.5	Uso de la información de los tipos	938
12.4.6	Ejercicios para la sección 12.4	939
12.5	Análisis interprocedural insensible al contexto	941
12.5.1	Efectos de la invocación a un método	941
12.5.2	Descubrimiento del grafo de llamadas en Datalog.	943
12.5.3	Carga dinámica y reflexión	944
12.5.4	Ejercicios para la sección 12.5	945
12.6	Análisis de apuntadores sensible al contexto	945
12.6.1	Contextos y cadenas de llamadas	946
12.6.2	Agregar contexto a las reglas de Datalog	949

12.6.3	Observaciones adicionales acerca de la sensibilidad	949
12.6.4	Ejercicios para la sección 12.6	950
12.7	Implementación en Datalog mediante BDDs	951
12.7.1	Diagramas de decisiones binarios.	951
12.7.2	Transformaciones en BDDs.	953
12.7.3	Representación de las relaciones mediante BDDs	954
12.7.4	Operaciones relacionales como operaciones BDD	954
12.7.5	Uso de BDDs para el análisis tipo “apunta a”	957
12.7.6	Ejercicios para la sección 12.7	958
12.8	Resumen del capítulo 12	958
12.9	Referencias para el capítulo 12	961
A	Un front-end completo	965
A.1	El lenguaje de código fuente	965
A.2	Main	966
A.3	Analizador léxico	967
A.4	Tablas de símbolos y tipos	970
A.5	Código intermedio para las expresiones	971
A.6	Código de salto para las expresiones booleanas.	974
A.7	Código intermedio para las instrucciones	978
A.8	Analizador sintáctico	981
A.9	Creación del front-end	986
B	Búsqueda de soluciones linealmente independientes	989
	Índice	993

Capítulo 1

Introducción

Los lenguajes de programación son notaciones que describen los cálculos a las personas y las máquinas. Nuestra percepción del mundo en que vivimos depende de los lenguajes de programación, ya que todo el software que se ejecuta en todas las computadoras se escribió en algún lenguaje de programación. Pero antes de poder ejecutar un programa, primero debe traducirse a un formato en el que una computadora pueda ejecutarlo.

Los sistemas de software que se encargan de esta traducción se llaman *compiladores*.

Este libro trata acerca de cómo diseñar e implementar compiladores. Aquí descubriremos que podemos utilizar unas cuantas ideas básicas para construir traductores para una amplia variedad de lenguajes y máquinas. Además de los compiladores, los principios y las técnicas para su diseño se pueden aplicar a tantos dominios distintos aparte, que es probable que un científico computacional las reutilice muchas veces en el transcurso de su carrera profesional. El estudio de la escritura de los compiladores se relaciona con los lenguajes de programación, la arquitectura de las máquinas, la teoría de lenguajes, los algoritmos y la ingeniería de software.

En este capítulo preliminar presentaremos las distintas formas de los traductores de lenguaje, proporcionaremos una descripción general de alto nivel sobre la estructura de un compilador ordinario, y hablaremos sobre las tendencias en los lenguajes de programación y la arquitectura de máquinas que dan forma a los compiladores. Incluirímos algunas observaciones sobre la relación entre el diseño de los compiladores y la teoría de las ciencias computacionales, y un esquema de las aplicaciones de tecnología sobre los compiladores que van más allá de la compilación. Terminaremos con una breve descripción de los conceptos clave de los lenguajes de programación que necesitaremos para nuestro estudio de los compiladores.

1.1 Procesadores de lenguaje

Dicho en forma simple, un compilador es un programa que puede leer un programa en un lenguaje (el lenguaje *fuente*) y traducirlo en un programa equivalente en otro lenguaje (el lenguaje *destino*); vea la figura 1.1. Una función importante del compilador es reportar cualquier error en el programa fuente que detecte durante el proceso de traducción.

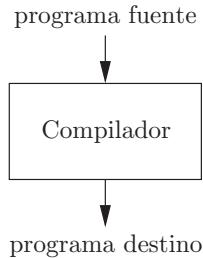


Figura 1.1: Un compilador

Si el programa destino es un programa ejecutable en lenguaje máquina, entonces el usuario puede ejecutarlo para procesar las entradas y producir salidas (resultados); vea la figura 1.2.

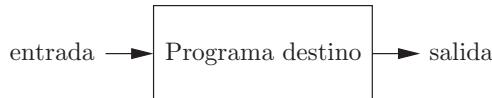


Figura 1.2: Ejecución del programa de destino

Un *intérprete* es otro tipo común de procesador de lenguaje. En vez de producir un programa destino como una traducción, el intérprete nos da la apariencia de ejecutar directamente las operaciones especificadas en el programa de origen (fuente) con las entradas proporcionadas por el usuario, como se muestra en la figura 1.3.



Figura 1.3: Un intérprete

El programa destino en lenguaje máquina que produce un compilador es, por lo general, más rápido que un intérprete al momento de asignar las entradas a las salidas. No obstante, por lo regular, el intérprete puede ofrecer mejores diagnósticos de error que un compilador, ya que ejecuta el programa fuente instrucción por instrucción.

Ejemplo 1.1: Los procesadores del lenguaje Java combinan la compilación y la interpretación, como se muestra en la figura 1.4. Un programa fuente en Java puede compilarse primero en un formato intermedio, llamado *bytecodes*. Después, una máquina virtual los interpreta. Un beneficio de este arreglo es que los bytecodes que se compilan en una máquina pueden interpretarse en otra, tal vez a través de una red.

Para poder lograr un procesamiento más rápido de las entradas a las salidas, algunos compiladores de Java, conocidos como compiladores *just-in-time* (justo a tiempo), traducen los bytecodes en lenguaje máquina justo antes de ejecutar el programa intermedio para procesar la entrada. □

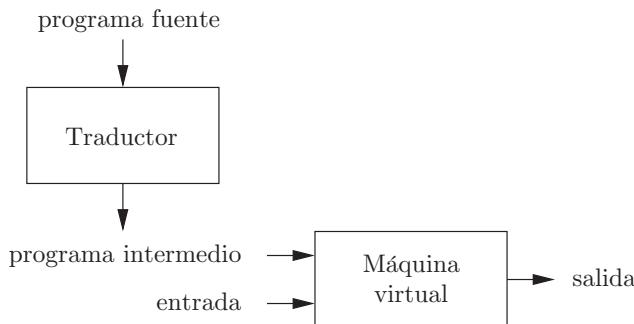


Figura 1.4: Un compilador híbrido

Además de un compilador, pueden requerirse otros programas más para la creación de un programa destino ejecutable, como se muestra en la figura 1.5. Un programa fuente puede dividirse en módulos guardados en archivos separados. La tarea de recolectar el programa de origen se confía algunas veces a un programa separado, llamado *preprocesador*. El preprocesador también puede expandir algunos fragmentos de código abreviados de uso frecuente, llamados macros, en instrucciones del lenguaje fuente.

Después, el programa fuente modificado se alimenta a un compilador. El compilador puede producir un programa destino en ensamblador como su salida, ya que es más fácil producir el lenguaje ensamblador como salida y es más fácil su depuración. A continuación, el lenguaje ensamblador se procesa mediante un programa llamado *ensamblador*, el cual produce código máquina relocalizable como su salida.

A menudo, los programas extensos se compilan en partes, por lo que tal vez haya que enlazar (vincular) el código máquina relocalizable con otros archivos objeto relocalizables y archivos de biblioteca para producir el código que se ejecute en realidad en la máquina. El *enlazador* resuelve las direcciones de memoria externas, en donde el código en un archivo puede hacer referencia a una ubicación en otro archivo. Entonces, el *cargador* reúne todos los archivos objeto ejecutables en la memoria para su ejecución.

1.1.1 Ejercicios para la sección 1.1

Ejercicio 1.1.1: ¿Cuál es la diferencia entre un compilador y un intérprete?

Ejercicio 1.1.2: ¿Cuáles son las ventajas de (a) un compilador sobre un intérprete, y (b) las de un intérprete sobre un compilador?

Ejercicio 1.1.3: ¿Qué ventajas hay para un sistema de procesamiento de lenguajes en el cual el compilador produce lenguaje ensamblador en vez de lenguaje máquina?

Ejercicio 1.1.4: A un compilador que traduce un lenguaje de alto nivel a otro lenguaje de alto nivel se le llama traductor de *source-to-source*. ¿Qué ventajas hay en cuanto al uso de C como lenguaje destino para un compilador?

Ejercicio 1.1.5: Describa algunas de las tareas que necesita realizar un ensamblador.

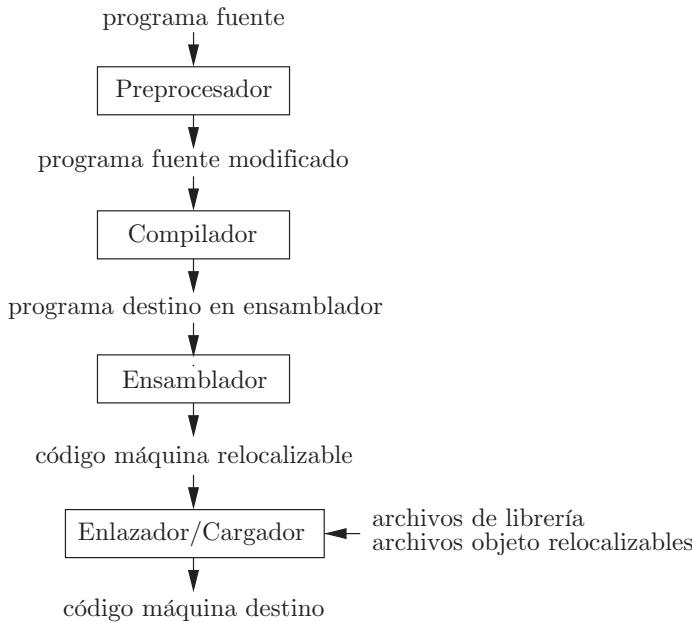


Figura 1.5: Un sistema de procesamiento de lenguaje

1.2 La estructura de un compilador

Hasta este punto, hemos tratado al compilador como una caja simple que mapea un programa fuente a un programa destino con equivalencia semántica. Si abrimos esta caja un poco, podremos ver que hay dos procesos en esta asignación: análisis y síntesis.

La parte del *análisis* divide el programa fuente en componentes e impone una estructura gramatical sobre ellas. Después utiliza esta estructura para crear una representación intermedia del programa fuente. Si la parte del análisis detecta que el programa fuente está mal formado en cuanto a la sintaxis, o que no tiene una semántica consistente, entonces debe proporcionar mensajes informativos para que el usuario pueda corregirlo. La parte del análisis también recolecta información sobre el programa fuente y la almacena en una estructura de datos llamada *tabla de símbolos*, la cual se pasa junto con la representación intermedia a la parte de la síntesis.

La parte de la *síntesis* construye el programa destino deseado a partir de la representación intermedia y de la información en la tabla de símbolos. A la parte del análisis se le llama comúnmente el *front-end* del compilador; la parte de la síntesis (propiamente la traducción) es el *back-end*.

Si examinamos el proceso de compilación con más detalle, podremos ver que opera como una secuencia de *fases*, cada una de las cuales transforma una representación del programa fuente en otro. En la figura 1.6 se muestra una descomposición típica de un compilador en fases. En la práctica varias fases pueden agruparse, y las representaciones intermedias entre las

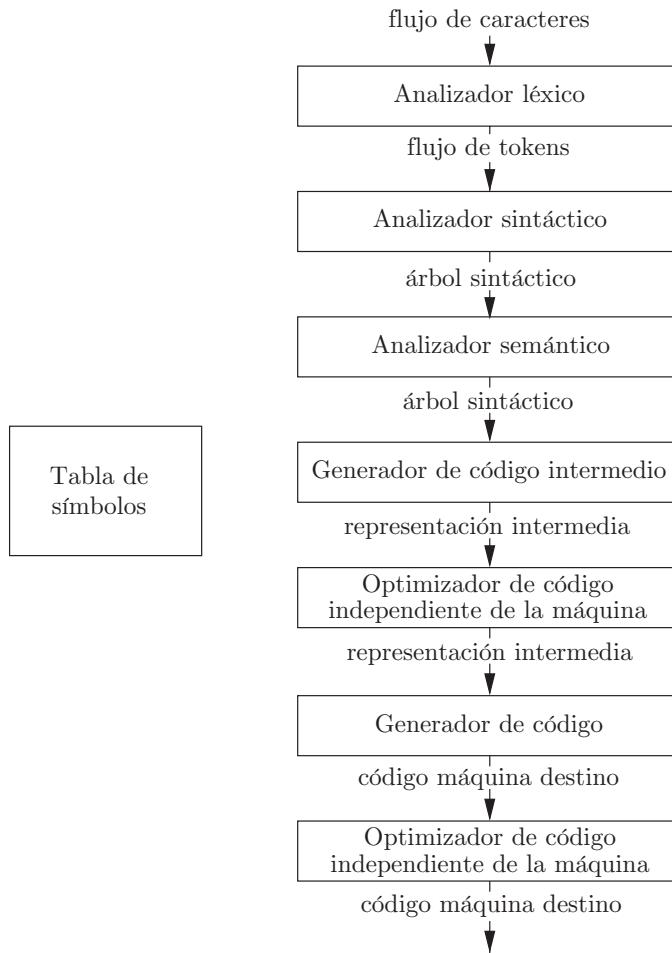


Figura 1.6: Fases de un compilador

fases agrupadas no necesitan construirse de manera explícita. La tabla de símbolos, que almacena información sobre todo el programa fuente, se utiliza en todas las fases del compilador.

Algunos compiladores tienen una fase de optimización de código independiente de la máquina, entre el front-end y el back-end. El propósito de esta optimización es realizar transformaciones sobre la representación intermedia, para que el back-end pueda producir un mejor programa destino de lo que hubiera producido con una representación intermedia sin optimizar. Como la optimización es opcional, puede faltar una de las dos fases de optimización de la figura 1.6.

1.2.1 Análisis de léxico

A la primera fase de un compilador se le llama *análisis de léxico* o *escaneo*. El analizador de léxico lee el flujo de caracteres que componen el programa fuente y los agrupa en secuencias signifi-

cativas, conocidas como *lexemas*. Para cada lexema, el analizador léxico produce como salida un *token* de la forma:

$$\langle \text{nombre-token}, \text{valor-atributo} \rangle$$

que pasa a la fase siguiente, el análisis de la sintaxis. En el token, el primer componente *nombre-token* es un símbolo abstracto que se utiliza durante el análisis sintáctico, y el segundo componente *valor-atributo* apunta a una entrada en la tabla de símbolos para este token. La información de la entrada en la tabla de símbolos se necesita para el análisis semántico y la generación de código.

Por ejemplo, suponga que un programa fuente contiene la instrucción de asignación:

$$\text{posicion} = \text{inicial} + \text{velocidad} * 60 \quad (1.1)$$

Los caracteres en esta asignación podrían agruparse en los siguientes lexemas y mapearse a los siguientes tokens que se pasan al analizador sintáctico:

1. **posicion** es un lexema que se asigna a un token $\langle \mathbf{id}, 1 \rangle$, en donde **id** es un símbolo abstracto que representa la palabra *identificador* y 1 apunta a la entrada en la tabla de símbolos para **posicion**. La entrada en la tabla de símbolos para un identificador contiene información acerca de éste, como su nombre y tipo.
2. El símbolo de asignación **=** es un lexema que se asigna al token $\langle = \rangle$. Como este token no necesita un valor-atributo, hemos omitido el segundo componente. Podríamos haber utilizado cualquier símbolo abstracto como **asignar** para el nombre-token, pero por conveniencia de notación hemos optado por usar el mismo lexema como el nombre para el símbolo abstracto.
3. **inicial** es un lexema que se asigna al token $\langle \mathbf{id}, 2 \rangle$, en donde 2 apunta a la entrada en la tabla de símbolos para **inicial**.
4. **+** es un lexema que se asigna al token $\langle + \rangle$.
5. **velocidad** es un lexema que se asigna al token $\langle \mathbf{id}, 3 \rangle$, en donde 3 apunta a la entrada en la tabla de símbolos para **velocidad**.
6. ***** es un lexema que se asigna al token $\langle * \rangle$.
7. **60** es un lexema que se asigna al token $\langle 60 \rangle$.¹

El analizador léxico ignora los espacios en blanco que separan a los lexemas.

La figura 1.7 muestra la representación de la instrucción de asignación (1.1) después del análisis léxico como la secuencia de tokens.

$$\langle \mathbf{id}, 1 \rangle \langle = \rangle \langle \mathbf{id}, 2 \rangle \langle + \rangle \langle \mathbf{id}, 3 \rangle \langle * \rangle \langle 60 \rangle \quad (1.2)$$

En esta representación, los nombres de los tokens **=**, **+** y ***** son símbolos abstractos para los operadores de asignación, suma y multiplicación, respectivamente.

¹ Hablando en sentido técnico, para el lexema **60** formaríamos un token como $\langle \mathbf{número}, 4 \rangle$, en donde 4 apunta a la tabla de símbolos para la representación interna del entero 60, pero dejaremos la explicación de los tokens para los números hasta el capítulo 2. En el capítulo 3 hablaremos sobre las técnicas para la construcción de analizadores léxicos.

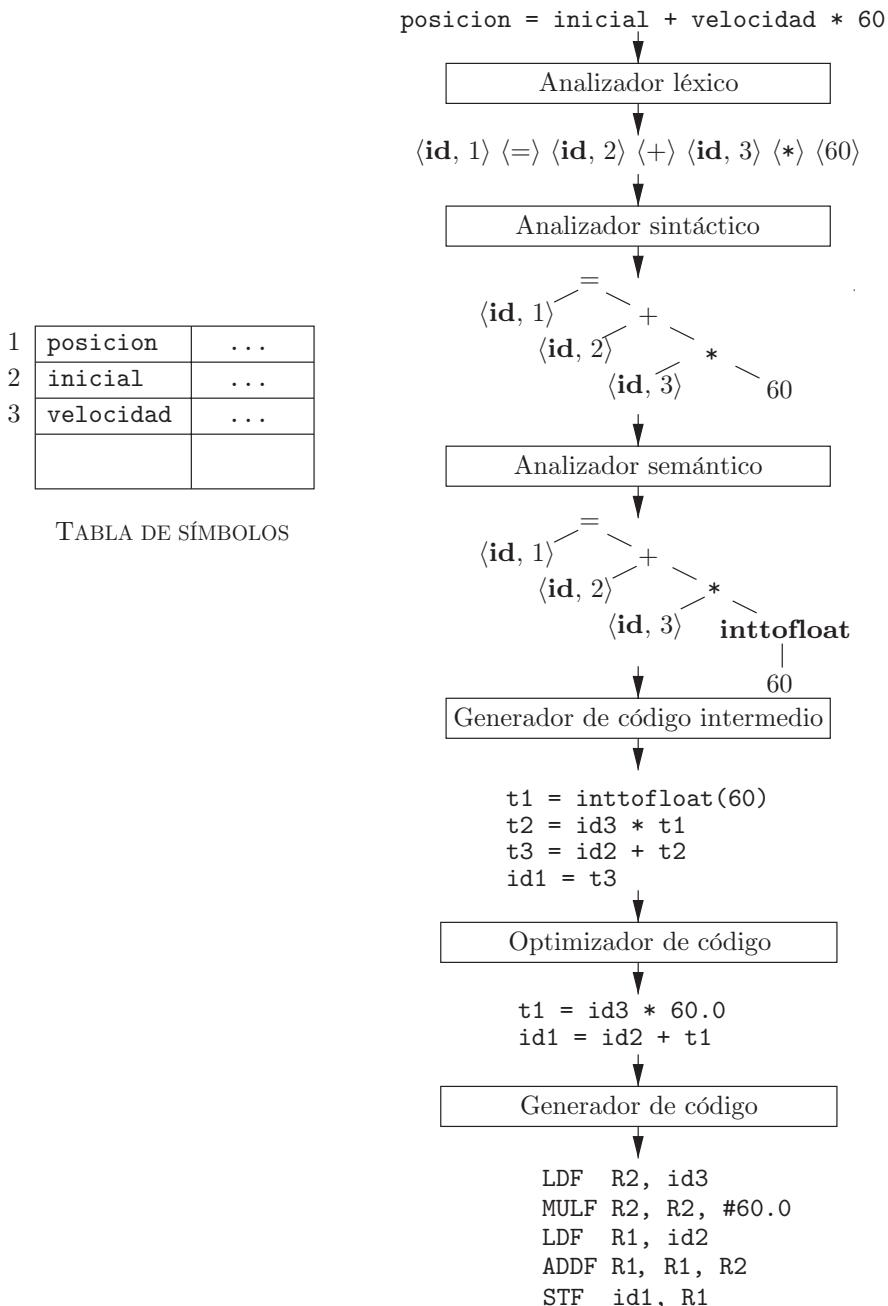


Figura 1.7: Traducción de una instrucción de asignación

1.2.2 Análisis sintáctico

La segunda fase del compilador es el *análisis sintáctico* o *parsing*. El parser (analizador sintáctico) utiliza los primeros componentes de los tokens producidos por el analizador de léxico para crear una representación intermedia en forma de árbol que describa la estructura gramatical del flujo de tokens. Una representación típica es el *árbol sintáctico*, en el cual cada nodo interior representa una operación y los hijos del nodo representan los argumentos de la operación. En la figura 1.7 se muestra un árbol sintáctico para el flujo de tokens (1.2) como salida del analizador sintáctico.

Este árbol muestra el orden en el que deben llevarse a cabo las operaciones en la siguiente asignación:

```
posición = inicial + velocidad * 60
```

El árbol tiene un nodo interior etiquetado como `*`, con `<id, 3>` como su hijo izquierdo, y el entero `60` como su hijo derecho. El nodo `<id, 3>` representa el identificador `velocidad`. El nodo etiquetado como `*` hace explícito que primero debemos multiplicar el valor de `velocidad` por `60`. El nodo etiquetado como `+` indica que debemos sumar el resultado de esta multiplicación al valor de `inicial`. La raíz del árbol, que se etiqueta como `=`, indica que debemos almacenar el resultado de esta suma en la ubicación para el identificador `posición`. Este ordenamiento de operaciones es consistente con las convenciones usuales de la aritmética, las cuales nos indican que la multiplicación tiene mayor precedencia que la suma y, por ende, debe realizarse antes que la suma.

Las fases siguientes del compilador utilizan la estructura gramatical para ayudar a analizar el programa fuente y generar el programa destino. En el capítulo 4 utilizaremos gramáticas libres de contexto para especificar la estructura gramatical de los lenguajes de programación, y hablaremos sobre los algoritmos para construir analizadores sintácticos eficientes de manera automática, a partir de ciertas clases de gramáticas. En los capítulos 2 y 5 veremos que las definiciones orientadas a la sintaxis pueden ayudar a especificar la traducción de las construcciones del lenguaje de programación.

1.2.3 Análisis semántico

El *analizador semántico* utiliza el árbol sintáctico y la información en la tabla de símbolos para comprobar la consistencia semántica del programa fuente con la definición del lenguaje. También recopila información sobre el tipo y la guarda, ya sea en el árbol sintáctico o en la tabla de símbolos, para usarla más tarde durante la generación de código intermedio.

Una parte importante del análisis semántico es la *comprobación (verificación) de tipos*, en donde el compilador verifica que cada operador tenga operandos que coincidan. Por ejemplo, muchas definiciones de lenguajes de programación requieren que el índice de un arreglo sea entero; el compilador debe reportar un error si se utiliza un número de punto flotante para indexar el arreglo.

La especificación del lenguaje puede permitir ciertas conversiones de tipo conocidas como *coerciones*. Por ejemplo, puede aplicarse un operador binario aritmético a un par de enteros o a un par de números de punto flotante. Si el operador se aplica a un número de punto flotante

y a un entero, el compilador puede convertir u obligar a que se convierta en un número de punto flotante.

Dicha conversión aparece en la figura 1.7. Suponga que *posicion*, *inicial* y *velocidad* se han declarado como números de punto flotante, y que el lexema 60 por sí solo forma un entero. El comprobador de tipo en el analizador semántico de la figura 1.7 descubre que se aplica el operador * al número de punto flotante *velocidad* y al entero 60. En este caso, el entero puede convertirse en un número de punto flotante. Observe en la figura 1.7 que la salida del analizador semántico tiene un nodo adicional para el operador **inttofloat**, que convierte de manera explícita su argumento tipo entero en un número de punto flotante. En el capítulo 6 hablaremos sobre la comprobación de tipos y el análisis semántico.

1.2.4 Generación de código intermedio

En el proceso de traducir un programa fuente a código destino, un compilador puede construir una o más representaciones intermedias, las cuales pueden tener una variedad de formas. Los árboles sintácticos son una forma de representación intermedia; por lo general, se utilizan durante el análisis sintáctico y semántico.

Después del análisis sintáctico y semántico del programa fuente, muchos compiladores generan un nivel bajo explícito, o una representación intermedia similar al código máquina, que podemos considerar como un programa para una máquina abstracta. Esta representación intermedia debe tener dos propiedades importantes: debe ser fácil de producir y fácil de traducir en la máquina destino.

En el capítulo 6, consideraremos una forma intermedia llamada *código de tres direcciones*, que consiste en una secuencia de instrucciones similares a ensamblador, con tres operandos por instrucción. Cada operando puede actuar como un registro. La salida del generador de código intermedio en la figura 1.7 consiste en la secuencia de código de tres direcciones.

```

t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

```

(1.3)

Hay varios puntos que vale la pena mencionar sobre las instrucciones de tres direcciones. En primer lugar, cada instrucción de asignación de tres direcciones tiene, por lo menos, un operador del lado derecho. Por ende, estas instrucciones corrigen el orden en el que se van a realizar las operaciones; la multiplicación va antes que la suma en el programa fuente (1.1). En segundo lugar, el compilador debe generar un nombre temporal para guardar el valor calculado por una instrucción de tres direcciones. En tercer lugar, algunas “instrucciones de tres direcciones” como la primera y la última en la secuencia (1.3) anterior, tienen menos de tres operandos.

En el capítulo 6 hablaremos sobre las representaciones intermedias principales que se utilizan en los compiladores. El capítulo 5 introduce las técnicas para la traducción dirigida por la sintaxis, las cuales se aplican en el capítulo 6 para la comprobación de tipos y la generación de código intermedio en las construcciones comunes de los lenguajes de programación, como las expresiones, las construcciones de control de flujo y las llamadas a procedimientos.

1.2.5 Optimización de código

La fase de optimización de código independiente de la máquina trata de mejorar el código intermedio, de manera que se produzca un mejor código destino. Por lo general, mejor significa más rápido, pero pueden lograrse otros objetivos, como un código más corto, o un código de destino que consuma menos poder. Por ejemplo, un algoritmo directo genera el código intermedio (1.3), usando una instrucción para cada operador en la representación tipo árbol que produce el analizador semántico.

Un algoritmo simple de generación de código intermedio, seguido de la optimización de código, es una manera razonable de obtener un buen código de destino. El optimizador puede deducir que la conversión del 60, de entero a punto flotante, puede realizarse de una vez por todas en tiempo de compilación, por lo que se puede eliminar la operación **inttofloat** sustituyendo el entero 60 por el número de punto flotante 60.0. Lo que es más, **t3** se utiliza sólo una vez para transmitir su valor a **id1**, para que el optimizador pueda transformar (1.3) en la siguiente secuencia más corta:

$$\begin{aligned} t1 &= id3 * 60.0 \\ id1 &= id2 + t1 \end{aligned} \tag{1.4}$$

Hay una gran variación en la cantidad de optimización de código que realizan los distintos compiladores. En aquellos que realizan la mayor optimización, a los que se les denomina como “compiladores optimizadores”, se invierte mucho tiempo en esta fase. Hay optimizaciones simples que mejoran en forma considerable el tiempo de ejecución del programa destino, sin reducir demasiado la velocidad de la compilación. Los capítulos 8 en adelante hablan con más detalle sobre las optimizaciones independientes y dependientes de la máquina.

1.2.6 Generación de código

El generador de código recibe como entrada una representación intermedia del programa fuente y la asigna al lenguaje destino. Si el lenguaje destino es código máquina, se seleccionan registros o ubicaciones (localidades) de memoria para cada una de las variables que utiliza el programa. Después, las instrucciones intermedias se traducen en secuencias de instrucciones de máquina que realizan la misma tarea. Un aspecto crucial de la generación de código es la asignación juiciosa de los registros para guardar las variables.

Por ejemplo, usando los registros **R1** y **R2**, el código intermedio en (1.4) podría traducirse en el siguiente código de máquina:

$$\begin{aligned} LDF & R2, id3 \\ MULF & R2, R2, #60.0 \\ LDF & R1, id2 \\ ADDF & R1, R1, R2 \\ STF & id1, R1 \end{aligned} \tag{1.5}$$

El primer operando de cada instrucción especifica un destino. La **F** en cada instrucción nos indica que trata con números de punto flotante. El código en (1.5) carga el contenido de

la dirección `id3` en el registro `R2`, y después lo multiplica con la constante de punto flotante `60.0`. El `#` indica que el número `60.0` se va a tratar como una constante inmediata. La tercera instrucción mueve `id2` al registro `R1` y la cuarta lo suma al valor que se había calculado antes en el registro `R2`. Por último, el valor en el registro `R1` se almacena en la dirección de `id1`, por lo que el código implementa en forma correcta la instrucción de asignación (1.1). El capítulo 8 trata acerca de la generación de código.

En esta explicación sobre la generación de código hemos ignorado una cuestión importante: la asignación de espacio de almacenamiento para los identificadores en el programa fuente. Como veremos en el capítulo 7, la organización del espacio de almacenamiento en tiempo de ejecución depende del lenguaje que se esté compilando. Las decisiones sobre la asignación de espacio de almacenamiento se realizan durante la generación de código intermedio, o durante la generación de código.

1.2.7 Administración de la tabla de símbolos

Una función esencial de un compilador es registrar los nombres de las variables que se utilizan en el programa fuente, y recolectar información sobre varios atributos de cada nombre. Estos atributos pueden proporcionar información acerca del espacio de almacenamiento que se asigna para un nombre, su tipo, su alcance (en qué parte del programa puede usarse su valor), y en el caso de los nombres de procedimientos, cosas como el número y los tipos de sus argumentos, el método para pasar cada argumento (por ejemplo, por valor o por referencia) y el tipo devuelto.

La tabla de símbolos es una estructura de datos que contiene un registro para cada nombre de variable, con campos para los atributos del nombre. La estructura de datos debe diseñarse de tal forma que permita al compilador buscar el registro para cada nombre, y almacenar u obtener datos de ese registro con rapidez. En el capítulo 2 hablaremos sobre las tablas de símbolos.

1.2.8 El agrupamiento de fases en pasadas

El tema sobre las fases tiene que ver con la organización lógica de un compilador. En una implementación, las actividades de varias fases pueden agruparse en una *pasada*, la cual lee un archivo de entrada y escribe en un archivo de salida. Por ejemplo, las fases correspondientes al front-end del análisis léxico, análisis sintáctico, análisis semántico y generación de código intermedio podrían agruparse en una sola pasada. La optimización de código podría ser una pasada opcional. Entonces podría haber una pasada de back-end, consistente en la generación de código para una máquina de destino específica.

Algunas colecciones de compiladores se han creado en base a representaciones intermedias diseñadas con cuidado, las cuales permiten que el front-end para un lenguaje específico se interconecte con el back-end para cierta máquina destino. Con estas colecciones, podemos producir compiladores para distintos lenguajes fuente para una máquina destino, mediante la combinación de distintos front-end con sus back-end para esa máquina de destino. De manera similar, podemos producir compiladores para distintas máquinas destino, mediante la combinación de un front-end con back-end para distintas máquinas destino.

1.2.9 Herramientas de construcción de compiladores

Al igual que cualquier desarrollador de software, el desarrollador de compiladores puede utilizar para su beneficio los entornos de desarrollo de software modernos que contienen herramientas como editores de lenguaje, depuradores, administradores de versiones, profilers, ambientes seguros de prueba, etcétera. Además de estas herramientas generales para el desarrollo de software, se han creado otras herramientas más especializadas para ayudar a implementar las diversas fases de un compilador.

Estas herramientas utilizan lenguajes especializados para especificar e implementar componentes específicos, y muchas utilizan algoritmos bastante sofisticados. Las herramientas más exitosas son las que ocultan los detalles del algoritmo de generación y producen componentes que pueden integrarse con facilidad al resto del compilador. Algunas herramientas de construcción de compiladores de uso común son:

1. *Generadores de analizadores sintácticos (parsers)*, que producen de manera automática analizadores sintácticos a partir de una descripción gramatical de un lenguaje de programación.
2. *Generadores de escáneres*, que producen analizadores de léxicos a partir de una descripción de los tokens de un lenguaje utilizando expresiones regulares.
3. *Motores de traducción orientados a la sintaxis*, que producen colecciones de rutinas para recorrer un árbol de análisis sintáctico y generar código intermedio.
4. *Generadores de generadores de código*, que producen un generador de código a partir de una colección de reglas para traducir cada operación del lenguaje intermedio en el lenguaje máquina para una máquina destino.
5. *Motores de análisis de flujos de datos*, que facilitan la recopilación de información de cómo se transmiten los valores de una parte de un programa a cada una de las otras partes. El análisis de los flujos de datos es una parte clave en la optimización de código.
6. *Kits (conjuntos) de herramientas para la construcción de compiladores*, que proporcionan un conjunto integrado de rutinas para construir varias fases de un compilador.

A lo largo de este libro, describiremos muchas de estas herramientas.

1.3 La evolución de los lenguajes de programación

Las primeras computadoras electrónicas aparecieron en la década de 1940 y se programaban en lenguaje máquina, mediante secuencias de 0's y 1's que indicaban de manera explícita a la computadora las operaciones que debía ejecutar, y en qué orden. Las operaciones en sí eran de muy bajo nivel: mover datos de una ubicación a otra, sumar el contenido de dos registros, comparar dos valores, etcétera. Está demás decir, que este tipo de programación era lenta, tediosa y propensa a errores. Y una vez escritos, los programas eran difíciles de comprender y modificar.

1.3.1 El avance a los lenguajes de alto nivel

El primer paso hacia los lenguajes de programación más amigables para las personas fue el desarrollo de los lenguajes ensambladores a inicios de la década de 1950, los cuales usaban mnemónicos. Al principio, las instrucciones en un lenguaje ensamblador eran sólo representaciones mnemónicas de las instrucciones de máquina. Más adelante, se agregaron macro instrucciones a los lenguajes ensambladores, para que un programador pudiera definir abreviaciones parametrizadas para las secuencias de uso frecuente de las instrucciones de máquina.

Un paso importante hacia los lenguajes de alto nivel se hizo en la segunda mitad de la década de 1950, con el desarrollo de Fortran para la computación científica, Cobol para el procesamiento de datos de negocios, y Lisp para la computación simbólica. La filosofía de estos lenguajes era crear notaciones de alto nivel con las que los programadores pudieran escribir con más facilidad los cálculos numéricos, las aplicaciones de negocios y los programas simbólicos. Estos lenguajes tuvieron tanto éxito que siguen en uso hoy en día.

En las siguientes décadas se crearon muchos lenguajes más con características innovadoras para facilitar que la programación fuera más natural y más robusta. Más adelante, en este capítulo, hablaremos sobre ciertas características clave que son comunes para muchos lenguajes de programación modernos.

En la actualidad existen miles de lenguajes de programación. Pueden clasificarse en una variedad de formas. Una de ellas es por generación. Los *lenguajes de primera generación* son los lenguajes de máquina, los de *segunda generación* son los lenguajes ensambladores, y los de *tercera generación* son los lenguajes de alto nivel, como Fortran, Cobol, Lisp, C, C++, C# y Java. Los *lenguajes de cuarta generación* son diseñados para aplicaciones específicas como NOMAD para la generación de reportes, SQL para las consultas en bases de datos, y Postscript para el formato de texto. El término *lenguaje de quinta generación* se aplica a los lenguajes basados en lógica y restricciones, como Prolog y OPS5.

Otra de las clasificaciones de los lenguajes utiliza el término *imperativo* para los lenguajes en los que un programa especifica *cómo* se va a realizar un cálculo, y *declarativo* para los lenguajes en los que un programa especifica *qué* cálculo se va a realizar. Los lenguajes como C, C++, C# y Java son lenguajes imperativos. En los lenguajes imperativos hay una noción de estado del programa, junto con instrucciones que modifican ese estado. Los lenguajes funcionales como ML y Haskell, y los lenguajes de lógica de restricción como Prolog, se consideran a menudo como lenguajes declarativos.

El término *lenguaje von Neumann* se aplica a los lenguajes de programación cuyo modelo se basa en la arquitectura de computadoras descrita por von Neumann. Muchos de los lenguajes de la actualidad, como Fortran y C, son lenguajes von Neumann.

Un *lenguaje orientado a objetos* es uno que soporta la programación orientada a objetos, un estilo de programación en el que un programa consiste en una colección de objetos que interactúan entre sí. Simula 67 y Smalltalk son de los primeros lenguajes orientados a objetos importantes. Los lenguajes como C++, C#, Java y Ruby son los lenguajes orientados a objetos más recientes.

Los *lenguajes de secuencias de comandos (scripting)* son lenguajes interpretados con operadores de alto nivel diseñados para “unir” cálculos. Estos cálculos se conocían en un principio como “*secuencias de comandos (scripts)*”. Awk, JavaScript, Perl, PHP, Python, Ruby y Tcl son ejemplos populares de lenguajes de secuencias de comandos. Los programas escritos en

lenguajes de secuencias de comandos son a menudo más cortos que los programas equivalentes escritos en lenguajes como C.

1.3.2 Impactos en el compilador

Desde su diseño, los lenguajes de programación y los compiladores están íntimamente relacionados; los avances en los lenguajes de programación impusieron nuevas demandas sobre los escritores de compiladores. Éstos tenían que idear algoritmos y representaciones para traducir y dar soporte a las nuevas características del lenguaje. Desde la década de 1940, la arquitectura de computadoras ha evolucionado también. Los escritores de compiladores no sólo tuvieron que rastrear las nuevas características de un lenguaje, sino que también tuvieron que idear algoritmos de traducción para aprovechar al máximo las nuevas características del hardware.

Los compiladores pueden ayudar a promover el uso de lenguajes de alto nivel, al minimizar la sobrecarga de ejecución de los programas escritos en estos lenguajes. Los compiladores también son imprescindibles a la hora de hacer efectivas las arquitecturas computacionales de alto rendimiento en las aplicaciones de usuario. De hecho, el rendimiento de un sistema computacional es tan dependiente de la tecnología de compiladores, que éstos se utilizan como una herramienta para evaluar los conceptos sobre la arquitectura antes de crear una computadora.

Escribir compiladores es un reto. Un compilador por sí solo es un programa extenso. Además, muchos sistemas modernos de procesamiento de lenguajes manejan varios lenguajes fuente y máquinas destino dentro del mismo framework; es decir, sirven como colecciones de compiladores, y es probable que consistan en millones de líneas de código. En consecuencia, las buenas técnicas de ingeniería de software son esenciales para crear y evolucionar los procesadores modernos de lenguajes.

Un compilador debe traducir en forma correcta el conjunto potencialmente infinito de programas que podrían escribirse en el lenguaje fuente. El problema de generar el código destino óptimo a partir de un programa fuente es indecidible; por ende, los escritores de compiladores deben evaluar las concesiones acerca de los problemas que se deben atacar y la heurística que se debe utilizar para lidiar con el problema de generar código eficiente.

Un estudio de los compiladores es también un estudio sobre cómo la teoría se encuentra con la práctica, como veremos en la sección 1.4.

El propósito de este libro es enseñar la metodología y las ideas fundamentales que se utilizan en el diseño de los compiladores. Este libro no tiene la intención de enseñar todos los algoritmos y técnicas que podrían usarse para construir un sistema de procesamiento de lenguajes de vanguardia. No obstante, los lectores de este texto adquirirán el conocimiento básico y la comprensión para aprender a construir un compilador con relativa facilidad.

1.3.3 Ejercicios para la sección 1.3

Ejercicio 1.3.1: Indique cuál de los siguientes términos:

- | | | |
|-------------------------|---------------------------|--------------------------|
| a) imperativo | b) declarativo | c) von Neumann |
| d) orientado a objetos | e) funcional | f) de tercera generación |
| g) de cuarta generación | h) secuencias de comandos | |

se aplican a los siguientes lenguajes:

- 1) C 2) C++ 3) Cobol 4) Fortran 5) Java
- 6) Lisp 7) ML 8) Perl 9) Python 10) VB.

1.4 La ciencia de construir un compilador

El diseño de compiladores está lleno de bellos ejemplos, en donde se resuelven problemas complicados del mundo real mediante la abstracción de la esencia del problema en forma matemática. Éstos sirven como excelentes ilustraciones de cómo pueden usarse las abstracciones para resolver problemas: se toma un problema, se formula una abstracción matemática que capture las características clave y se resuelve utilizando técnicas matemáticas. La formulación del problema debe tener bases y una sólida comprensión de las características de los programas de computadora, y la solución debe validarse y refinarse en forma empírica.

Un compilador debe aceptar todos los programas fuente conforme a la especificación del lenguaje; el conjunto de programas fuente es infinito y cualquier programa puede ser muy largo, posiblemente formado por millones de líneas de código. Cualquier transformación que realice el compilador mientras traduce un programa fuente debe preservar el significado del programa que se está compilando. Por ende, los escritores de compiladores tienen influencia no sólo sobre los compiladores que crean, sino en todos los programas que compilan sus compiladores. Esta capacidad hace que la escritura de compiladores sea en especial gratificante; no obstante, también hace que el desarrollo de los compiladores sea todo un reto.

1.4.1 Modelado en el diseño e implementación de compiladores

El estudio de los compiladores es principalmente un estudio de la forma en que diseñamos los modelos matemáticos apropiados y elegimos los algoritmos correctos, al tiempo que logramos equilibrar la necesidad de una generalidad y poder con la simpleza y la eficiencia.

Algunos de los modelos más básicos son las máquinas de estados finitos y las expresiones regulares, que veremos en el capítulo 3. Estos modelos son útiles para describir las unidades de léxico de los programas (palabras clave, identificadores y demás) y para describir los algoritmos que utiliza el compilador para reconocer esas unidades. Además, entre los modelos esenciales se encuentran las gramáticas libres de contexto, que se utilizan para describir la estructura sintáctica de los lenguajes de programación, como el anidamiento de los paréntesis o las instrucciones de control. En el capítulo 4 estudiaremos las gramáticas. De manera similar, los árboles son un modelo importante para representar la estructura de los programas y su traducción a código objeto, como veremos en el capítulo 5.

1.4.2 La ciencia de la optimización de código

El término “optimización” en el diseño de compiladores se refiere a los intentos que realiza un compilador por producir código que sea más eficiente que el código obvio. Por lo tanto, “optimización” es un término equivocado, ya que no hay forma en que se pueda garantizar que el código producido por un compilador sea tan rápido o más rápido que cualquier otro código que realice la misma tarea.

En los tiempos modernos, la optimización de código que realiza un compilador se ha vuelto tanto más importante como más compleja. Es más compleja debido a que las arquitecturas de los procesadores se han vuelto más complejas, con lo que ofrecen más oportunidades de mejorar la forma en que se ejecuta el código. Es más importante, ya que las computadoras paralelas masivas requieren de una optimización considerable, pues de lo contrario su rendimiento sufre por grados de magnitud. Con la posible prevalencia de las máquinas multinúcleo (computadoras con chips que contienen grandes números de procesadores), todos los compiladores tendrán que enfrentarse al problema de aprovechar las máquinas con múltiples procesadores.

Es difícil, si no es que imposible, construir un compilador robusto a partir de “arreglos”. Por ende, se ha generado una teoría extensa y útil sobre el problema de optimizar código. El uso de una base matemática rigurosa nos permite mostrar que una optimización es correcta y que produce el efecto deseable para todas las posibles entradas. Empezando en el capítulo 9, veremos de qué manera son necesarios los modelos como grafos, matrices y programas lineales si el compilador debe producir un código bien optimizado.

Por otro lado, no es suficiente sólo con la pura teoría. Al igual que muchos problemas del mundo real, no hay respuestas perfectas. De hecho, la mayoría de las preguntas que hacemos en la optimización de un compilador son indecidibles. Una de las habilidades más importantes en el diseño de los compiladores es la de formular el problema adecuado a resolver. Para empezar, necesitamos una buena comprensión del comportamiento de los programas, junto con un proceso extenso de experimentación y evaluación para validar nuestras intuiciones.

Las optimizaciones de compiladores deben cumplir con los siguientes objetivos de diseño:

- La optimización debe ser correcta; es decir, debe preservar el significado del programa compilado.
- La optimización debe mejorar el rendimiento de muchos programas.
- El tiempo de compilación debe mantenerse en un valor razonable.
- El esfuerzo de ingeniería requerido debe ser administrable.

Es imposible hacer mucho énfasis en la importancia de estar en lo correcto. Es trivial escribir un compilador que genere código rápido, ¡si el código generado no necesita estar correcto! Es tan difícil optimizar los compiladores en forma apropiada, que nos atrevemos a decir que ¡ningún compilador optimizador está libre de errores! Por ende, el objetivo más importante en la escritura de un compilador es que sea correcto.

El segundo objetivo es que el compilador debe tener efectividad a la hora de mejorar el rendimiento de muchos programas de entrada. Por lo general, el rendimiento indica la velocidad de ejecución del programa. En especial en las aplicaciones incrustadas, también puede ser conveniente minimizar el tamaño del código generado. Y en el caso de los dispositivos móviles, también es conveniente que el código reduzca el consumo de energía. Por lo general, las mismas optimizaciones que agilizan el tiempo de ejecución también ahorran energía. Además del rendimiento, los aspectos de capacidad de uso como el reporte de errores y la depuración también son importantes.

En tercer lugar, debemos lograr que el tiempo de compilación sea corto para poder soportar un ciclo rápido de desarrollo y depuración. Este requerimiento se ha vuelto más fácil de

cumplir a medida que las máquinas se hacen más rápidas. A menudo, un programa primero se desarrolla y se depura sin optimizaciones. No sólo se reduce el tiempo de compilación, sino que, lo más importante, los programas no optimizados son más fáciles de depurar, ya que las optimizaciones que introduce un compilador, por lo común, oscurecen la relación entre el código fuente y el código objeto. Al activar las optimizaciones en el compilador, algunas veces se exponen nuevos problemas en el programa fuente; en consecuencia, hay que probar otra vez el código optimizado. Algunas veces, la necesidad de prueba adicional impide el uso de optimizaciones en aplicaciones, en especial si su rendimiento no es crítico.

Por último, un compilador es un sistema complejo; debemos mantener este sistema simple, para asegurar que los costos de ingeniería y de mantenimiento del compilador sean manejables. Existe un número infinito de optimizaciones de programas que podríamos implementar, y se requiere de muy poco esfuerzo para crear una optimización correcta y efectiva. Debemos dar prioridad a las optimizaciones, implementando sólo las que conlleven a los mayores beneficios en los programas fuente que encontremos en la práctica.

Por lo tanto, al estudiar los compiladores no sólo aprendemos a construir uno, sino también la metodología general para resolver problemas complejos y abiertos. El método que se utilice en el desarrollo de los compiladores implica tanto teoría como experimentación. Por lo general, empezamos formulando el problema según nuestras intuiciones acerca de cuáles son las cuestiones importantes.

1.5 Aplicaciones de la tecnología de compiladores

El diseño de compiladores no es sólo acerca de los compiladores; muchas personas utilizan la tecnología que aprenden al estudiar compiladores en la escuela y nunca, hablando en sentido estricto, han escrito (ni siquiera parte de) un compilador para un lenguaje de programación importante. La tecnología de compiladores tiene también otros usos importantes. Además, el diseño de compiladores impacta en otras áreas de las ciencias computacionales. En esta sección veremos un repaso acerca de las interacciones y aplicaciones más importantes de esta tecnología.

1.5.1 Implementación de lenguajes de programación de alto nivel

Un lenguaje de programación de alto nivel define una abstracción de programación: el programador expresa un algoritmo usando el lenguaje, y el compilador debe traducir el programa en el lenguaje de destino. Por lo general, es más fácil programar en los lenguajes de programación de alto nivel, aunque son menos eficientes; es decir, los programas destino se ejecutan con más lentitud. Los programadores que utilizan un lenguaje de bajo nivel tienen más control sobre un cálculo y pueden, en principio, producir código más eficiente. Por desgracia, los programas de menor nivel son más difíciles de escribir y (peor aún) menos portables, más propensos a errores y más difíciles de mantener. La optimización de los compiladores incluye técnicas para mejorar el rendimiento del código generado, con las cuales se desplaza la ineficiencia que dan las abstracciones de alto nivel.

Ejemplo 1.2: La palabra clave **registro** en el lenguaje de programación C es uno de los primeros ejemplos de la interacción entre la tecnología de los compiladores y la evolución del lenguaje. Cuando el lenguaje C se creó a mediados de la década de 1970, era necesario dejar que un programador controlara qué variables del programa debían residir en los registros. Este control ya no fue necesario cuando se desarrollaron las técnicas efectivas de asignación de recursos, por lo cual la mayoría de los programas modernos ya no utilizan esta característica del lenguaje.

De hecho, los programas que utilizan la palabra clave **registro** (**register**) pueden perder eficiencia, debido a que, por lo general, los programadores no son los mejores jueces en las cuestiones de bajo nivel, como la asignación de registros. La elección óptima en la asignación de registros depende en gran parte de las cuestiones específicas sobre la arquitectura de una máquina. Las decisiones de la administración de recursos como cablear (distribuir) a bajo nivel podrían de hecho afectar el rendimiento, en especial si el programa se ejecuta en máquinas distintas a la máquina para la cual se escribió. □

Los diversos cambios en cuanto a la elección popular de lenguajes de programación han sido orientados al aumento en los niveles de abstracción. C fue el lenguaje de programación de sistemas predominante en la década de 1980; muchos de los nuevos proyectos que empezaron en la década de 1990 eligieron a C++; Java, que se introdujo en 1995, ganó popularidad con rapidez a finales de la década de 1990. Las características nuevas de los lenguajes de programación que se introdujeron en cada generación incitaron a realizar nuevas investigaciones en la optimización de los compiladores. A continuación veremos las generalidades acerca de las principales características de los lenguajes que han estimulado avances considerables en la tecnología de los compiladores.

Prácticamente todos los lenguajes de programación comunes, incluyendo a C, Fortran y Cobol, soportan los tipos de datos de conjuntos definidos por el usuario, como los arreglos y las estructuras, y el flujo de control de alto nivel, como los ciclos y las invocaciones a procedimientos. Si sólo tomamos cada instrucción de alto nivel u operación de acceso a datos y la traducimos directamente a código máquina, el resultado sería muy ineficiente. Se ha desarrollado un cuerpo de optimizaciones de compilador, conocido como *optimizaciones de flujo de datos*, para analizar el flujo de datos a través del programa y eliminar las redundancias en estas instrucciones. Son efectivas para generar código que se asemeje al código que escribe un programador experimentado a un nivel más bajo.

La orientación a objetos se introdujo por primera vez en Simula en 1967, y se ha incorporado en lenguajes como Smalltalk, C++, C# y Java. Las ideas clave de la orientación a objetos son:

1. La abstracción de datos, y
2. La herencia de propiedades,

de las cuales se ha descubierto que facilitan el mantenimiento de los programas, y los hacen más modulares. Los programas orientados a objetos son diferentes de los programas escritos en muchos otros lenguajes, ya que consisten de muchos más procedimientos, pero más pequeños (a los cuales se les llama *métodos*, en términos de orientación a objetos). Por ende, las optimizaciones de los compiladores deben ser capaces de realizar bien su trabajo a través de los límites de los procedimientos del programa fuente. El uso de procedimientos en línea, que viene siendo cuando se sustituye la llamada a un procedimiento por su cuerpo, es muy útil en este caso. También se han desarrollado optimizaciones para agilizar los envíos de los métodos virtuales.

Java tiene muchas características que facilitan la programación, muchas de las cuales se han introducido anteriormente en otros lenguajes. El lenguaje Java ofrece seguridad en los tipos; es decir, un objeto no puede usarse como objeto de un tipo que no esté relacionado. Todos los accesos a los arreglos se verifican para asegurar que se encuentren dentro de los límites del arreglo. Java no tiene apuntadores y no permite la aritmética de apuntadores. Tiene una herramienta integrada para la recolección de basura, la cual libera de manera automática la memoria de variables que ya no se encuentran en uso. Mientras que estas características facilitan el proceso de la programación, provocan una sobrecarga en tiempo de ejecución. Se han desarrollado optimizaciones en el compilador para reducir la sobrecarga, por ejemplo, mediante la eliminación de las comprobaciones de rango innecesarias y la asignación de objetos a los que no se puede acceder más allá de un procedimiento en la pila, en vez del heap. También se han desarrollado algoritmos efectivos para minimizar la sobrecarga de la recolección de basura.

Además, Java está diseñado para dar soporte al código portable y móvil. Los programas se distribuyen como bytecodes de Java, el cual debe interpretarse o compilarse en código nativo en forma dinámica, es decir, en tiempo de ejecución. La compilación dinámica también se ha estudiado en otros contextos, en donde la información se extrae de manera dinámica en tiempo de ejecución, y se utiliza para producir un código más optimizado. En la optimización dinámica, es importante minimizar el tiempo de compilación, ya que forma parte de la sobrecarga en la ejecución. Una técnica de uso común es sólo compilar y optimizar las partes del programa que se van a ejecutar con frecuencia.

1.5.2 Optimizaciones para las arquitecturas de computadoras

La rápida evolución de las arquitecturas de computadoras también nos ha llevado a una insaciable demanda de nueva tecnología de compiladores. Casi todos los sistemas de alto rendimiento aprovechan las dos mismas técnicas básicas: *paralelismo* y *jerarquías de memoria*. Podemos encontrar el paralelismo en varios niveles: a *nivel de instrucción*, en donde varias operaciones se ejecutan al mismo tiempo y a *nivel de procesador*, en donde distintos subprocesos de la misma aplicación se ejecutan en distintos hilos. Las jerarquías de memoria son una respuesta a la limitación básica de que podemos construir un almacenamiento muy rápido o muy extenso, pero no un almacenamiento que sea tanto rápido como extenso.

Paralelismo

Todos los microprocesadores modernos explotan el paralelismo a nivel de instrucción. Sin embargo, este paralelismo puede ocultarse al programador. Los programas se escriben como si todas las instrucciones se ejecutaran en secuencia; el hardware verifica en forma dinámica las dependencias en el flujo secuencial de instrucciones y las ejecuta en paralelo siempre que sea posible. En algunos casos, la máquina incluye un programador (scheduler) de hardware que puede modificar el orden de las instrucciones para aumentar el paralelismo en el programa. Ya sea que el hardware reordene o no las instrucciones, los compiladores pueden reordenar las instrucciones para que el paralelismo a nivel de instrucción sea más efectivo.

El paralelismo a nivel de instrucción también puede aparecer de manera explícita en el conjunto de instrucciones. Las máquinas VLIW (Very Long Instruction Word, Palabra de instrucción

muy larga) tienen instrucciones que pueden ejecutar varias operaciones en paralelo. El Intel IA64 es un ejemplo de tal arquitectura. Todos los microprocesadores de alto rendimiento y propósito general incluyen también instrucciones que pueden operar sobre un vector de datos al mismo tiempo. Las técnicas de los compiladores se han desarrollado para generar código de manera automática para dichas máquinas, a partir de programas secuenciales.

Los multiprocesadores también se han vuelto frecuentes; incluso es común que hasta las computadoras personales tengan múltiples procesadores. Los programadores pueden escribir código multihilo para los multiprocesadores, o código en paralelo que un compilador puede generar de manera automática, a partir de programas secuenciales. Dicho compilador oculta de los programadores los detalles sobre cómo encontrar el paralelismo en un programa, distribuir los cálculos en la máquina y minimizar la sincronización y los cálculos entre cada procesador. Muchas aplicaciones de computación científica y de ingeniería hacen un uso intensivo de los cálculos y pueden beneficiarse mucho con el procesamiento en paralelo. Se han desarrollado técnicas de paralelización para traducir de manera automática los programas científicos en código para multiprocesadores.

Jerarquías de memoria

Una jerarquía de memoria consiste en varios niveles de almacenamiento con distintas velocidades y tamaños, en donde el nivel más cercano al procesador es el más rápido, pero también el más pequeño. El tiempo promedio de acceso a memoria de un programa se reduce si la mayoría de sus accesos se satisfacen a través de los niveles más rápidos de la jerarquía. Tanto el paralelismo como la existencia de una jerarquía de memoria mejoran el rendimiento potencial de una máquina, pero el compilador debe aprovecharlos de manera efectiva para poder producir un rendimiento real en una aplicación.

Las jerarquías de memoria se encuentran en todas las máquinas. Por lo general, un procesador tiene un pequeño número de registros que consisten en cientos de bytes, varios niveles de caché que contienen desde kilobytes hasta megabytes, memoria física que contiene desde megabytes hasta gigabytes y, por último, almacenamiento secundario que contiene gigabytes y mucho más. De manera correspondiente, la velocidad de los accesos entre los niveles adyacentes de la jerarquía puede diferir por dos o tres órdenes de magnitud. A menudo, el rendimiento de un sistema se limita no sólo a la velocidad del procesador, sino también por el rendimiento del subsistema de memoria. Aunque desde un principio los compiladores se han enfocado en optimizar la ejecución del procesador, ahora se pone más énfasis en lograr que la jerarquía de memoria sea más efectiva.

El uso efectivo de los registros es quizás el problema individual más importante en la optimización de un programa. A diferencia de los registros que tienen que administrarse de manera explícita en el software, las cachés y las memorias físicas están ocultas del conjunto de instrucciones y el hardware se encarga de administrarlas. La experiencia nos ha demostrado que las directivas de administración de caché que implementa el hardware no son efectivas en algunos casos, en especial con el código científico que tiene estructuras de datos extensas (por lo general, arreglos). Es posible mejorar la efectividad de la jerarquía de memoria, cambiando la distribución de los datos, o cambiando el orden de las instrucciones que acceden a los datos. También podemos cambiar la distribución del código para mejorar la efectividad de las cachés de instrucciones.

1.5.3 Diseño de nuevas arquitecturas de computadoras

En los primeros días del diseño de arquitecturas de computadoras, los compiladores se desarrollaron después de haber creado las máquinas. Eso ha cambiado. Desde que la programación en lenguajes de alto nivel es la norma, el rendimiento de un sistema computacional se determina no sólo por su velocidad en general, sino también por la forma en que los compiladores pueden explotar sus características. Por ende, en el desarrollo de arquitecturas de computadoras modernas, los compiladores se desarrollan en la etapa de diseño del procesador, y se utiliza el código compilado, que se ejecuta en simuladores, para evaluar las características propuestas sobre la arquitectura.

RISC

Uno de los mejores ejemplos conocidos sobre cómo los compiladores influenciaron el diseño de la arquitectura de computadoras fue la invención de la arquitectura RISC (Reduced Instruction-Set Computer, Computadora con conjunto reducido de instrucciones). Antes de esta invención, la tendencia era desarrollar conjuntos de instrucciones cada vez más complejos, destinados a facilitar la programación en ensamblador; estas arquitecturas se denominaron CISC (Complex Instruction-Set Computer, Computadora con conjunto complejo de instrucciones). Por ejemplo, los conjuntos de instrucciones CISC incluyen modos de direccionamiento de memoria complejos para soportar los accesos a las estructuras de datos, e instrucciones para invocar procedimientos que guardan registros y pasan parámetros en la pila.

A menudo, las optimizaciones de compiladores pueden reducir estas instrucciones a un pequeño número de operaciones más simples, eliminando las redundancias presentes en las instrucciones complejas. Por lo tanto, es conveniente crear conjuntos simples de instrucciones; los compiladores pueden utilizarlas con efectividad y el hardware es mucho más sencillo de optimizar.

La mayoría de las arquitecturas de procesadores de propósito general, como PowerPC, SPARC, MIPS, Alpha y PA-RISC, se basan en el concepto RISC. Aunque la arquitectura x86 (el microprocesador más popular) tiene un conjunto de instrucciones CISC, muchas de las ideas desarrolladas para las máquinas RISC se utilizan en la implementación del procesador. Además, la manera más efectiva de usar una máquina x86 de alto rendimiento es utilizar sólo sus instrucciones simples.

Arquitecturas especializadas

En las últimas tres décadas se han propuesto muchos conceptos sobre la arquitectura, entre los cuales se incluyen las máquinas de flujo de datos, las máquinas VLIW (Very Long Instruction Word, Palabra de instrucción muy larga), los arreglos SIMD (Single Instruction, Multiple Data, Una sola instrucción, varios datos) de procesadores, los arreglos sistólicos, los multiprocesadores con memoria compartida y los multiprocesadores con memoria distribuida. El desarrollo de cada uno de estos conceptos arquitectónicos se acompañó por la investigación y el desarrollo de la tecnología de compiladores correspondiente.

Algunas de estas ideas han incursionado en los diseños de las máquinas enbebidas. Debido a que pueden caber sistemas completos en un solo chip, los procesadores ya no necesitan ser unidades primarias preempaquetadas, sino que pueden personalizarse para lograr una mejor efectividad en costo para una aplicación específica. Por ende, a diferencia de los procesadores de propósito general, en donde las economías de escala han llevado a las arquitecturas

computacionales a convergir, los procesadores de aplicaciones específicas exhiben una diversidad de arquitecturas computacionales. La tecnología de compiladores se necesita no sólo para dar soporte a la programación para estas arquitecturas, sino también para evaluar los diseños arquitectónicos propuestos.

1.5.4 Traducciones de programas

Mientras que, por lo general, pensamos en la compilación como una traducción de un lenguaje de alto nivel al nivel de máquina, la misma tecnología puede aplicarse para realizar traducciones entre distintos tipos de lenguajes. A continuación se muestran algunas de las aplicaciones más importantes de las técnicas de traducción de programas.

Traducción binaria

La tecnología de compiladores puede utilizarse para traducir el código binario para una máquina al código binario para otra máquina distinta, con lo cual se permite a una máquina ejecutar los programas que originalmente eran compilados para otro conjunto de instrucciones. Varias compañías de computación han utilizado la tecnología de la traducción binaria para incrementar la disponibilidad de software en sus máquinas. En especial, debido al dominio en el mercado de la computadora personal x86, la mayoría de los títulos de software están disponibles como código x86. Se han desarrollado traductores binarios para convertir código x86 en código Alpha y Sparc. Además, Transmeta Inc. utilizó la traducción binaria en su implementación del conjunto de instrucciones x86. En vez de ejecutar el complejo conjunto de instrucciones x86 directamente en el hardware, el procesador Transmeta Crusoe es un procesador VLIW que se basa en la traducción binaria para convertir el código x86 en código VLIW nativo.

La traducción binaria también puede usarse para ofrecer compatibilidad inversa. Cuando el procesador en la Apple Macintosh se cambió del Motorola MC 68040 al PowerPC en 1994, se utilizó la traducción binaria para permitir que los procesadores PowerPC ejecutaran el código heredado del MC 68040.

Síntesis de hardware

No sólo la mayoría del software está escrito en lenguajes de alto nivel; incluso hasta la mayoría de los diseños de hardware se describen en lenguajes de descripción de hardware de alto nivel, como Verilog y VHDL (Very High-Speed Integrated Circuit Hardware Description Language, Lenguaje de descripción de hardware de circuitos integrados de muy alta velocidad). Por lo general, los diseños de hardware se describen en el nivel de transferencia de registros (RTL), en donde las variables representan registros y las expresiones representan la lógica combinacional. Las herramientas de síntesis de hardware traducen las descripciones RTL de manera automática en compuertas, las cuales a su vez se asignan a transistores y, en un momento dado, a un esquema físico. A diferencia de los compiladores para los lenguajes de programación, estas herramientas a menudo requieren horas para optimizar el circuito. También existen técnicas para traducir diseños a niveles más altos, como al nivel de comportamiento o funcional.

Intérpretes de consultas de bases de datos

Además de especificar el software y el hardware, los lenguajes son útiles en muchas otras aplicaciones. Por ejemplo, los lenguajes de consulta, en especial SQL (Structured Query Language,

Lenguaje de consulta estructurado), se utilizan para realizar búsquedas en bases de datos. Las consultas en las bases de datos consisten en predicados que contienen operadores relacionales y booleanos. Pueden interpretarse o compilarse en comandos para buscar registros en una base de datos que cumplan con ese predicado.

Simulación compilada

La simulación es una técnica general que se utiliza en muchas disciplinas científicas y de ingeniería para comprender un fenómeno, o para validar un diseño. Por lo general, las entradas de los simuladores incluyen la descripción del diseño y los parámetros específicos de entrada para esa ejecución específica de la simulación. Las simulaciones pueden ser muy costosas. Por lo general, necesitamos simular muchas alternativas de diseño posibles en muchos conjuntos distintos de entrada, y cada experimento puede tardar días en completarse, en una máquina de alto rendimiento. En vez de escribir un simulador para interpretar el diseño, es más rápido compilar el diseño para producir código máquina que simule ese diseño específico en forma nativa. La simulación compilada puede ejecutarse muchos grados de magnitud más rápido que un método basado en un intérprete. La simulación compilada se utiliza en muchas herramientas de alta tecnología que simulan diseños escritos en Verilog o VHDL.

1.5.5 Herramientas de productividad de software

Sin duda, los programas son los artefactos de ingeniería más complicados que se hayan producido jamás; consisten en muchos, muchos detalles, cada uno de los cuales debe corregirse para que el programa funcione por completo. Como resultado, los errores proliferan en los programas; éstos pueden hacer que un sistema falle, producir resultados incorrectos, dejar un sistema vulnerable a los ataques de seguridad, o incluso pueden llevar a fallas catastróficas en sistemas críticos. La prueba es la técnica principal para localizar errores en los programas.

Un enfoque complementario interesante y prometedor es utilizar el análisis de flujos de datos para localizar errores de manera estática (es decir, antes de que se ejecute el programa). El análisis de flujos de datos puede buscar errores a lo largo de todas las rutas posibles de ejecución, y no sólo aquellas ejercidas por los conjuntos de datos de entrada, como en el caso del proceso de prueba de un programa. Muchas de las técnicas de análisis de flujos de datos, que se desarrollaron en un principio para las optimizaciones de los compiladores, pueden usarse para crear herramientas que ayuden a los programadores en sus tareas de ingeniería de software.

El problema de encontrar todos los errores en los programas es indiscutible. Puede diseñarse un análisis de flujos de datos para advertir a los programadores acerca de todas las posibles instrucciones que violan una categoría específica de errores. Pero si la mayoría de estas advertencias son falsas alarmas, los usuarios no utilizarán la herramienta. Por ende, los detectores prácticos de errores en general no son sólidos ni están completos. Es decir, tal vez no encuentren todos los errores en el programa, y no se garantiza que todos los errores reportados sean errores verdaderos. Sin embargo, se han desarrollado diversos análisis estáticos que han demostrado ser efectivos para buscar errores, como el realizar referencias a apuntadores nulos o previamente liberados, en programas reales. El hecho de que los detectores de errores puedan ser poco sólidos los hace considerablemente distintos a las optimizaciones de código. Los optimizadores deben ser conservadores y no pueden alterar la semántica del programa, bajo ninguna circunstancia.

Para dar equilibrio a esta sección, vamos a mencionar varias formas en las que el análisis de los programas, basado en técnicas que se desarrollaron originalmente para optimizar el código en los compiladores, ha mejorado la productividad del software. De especial importancia son las técnicas que detectan en forma estática cuando un programa podría tener una vulnerabilidad de seguridad.

Comprobación (verificación) de tipos

La comprobación de tipos es una técnica efectiva y bien establecida para captar las inconsistencias en los programas. Por ejemplo, puede usarse para detectar errores en donde se aplique una operación al tipo incorrecto de objeto, o si los parámetros que se pasan a un procedimiento no coinciden con su firma. El análisis de los programas puede ir más allá de sólo encontrar los errores de tipo, analizando el flujo de datos a través de un programa. Por ejemplo, si a un apuntador se le asigna `null` y se desreferencia justo después, es evidente que el programa tiene un error.

La misma tecnología puede utilizarse para detectar una variedad de huecos de seguridad, en donde un atacante proporciona una cadena u otro tipo de datos que el programa utiliza sin cuidado. Una cadena proporcionada por el usuario podría etiquetarse con el tipo de “peligrosa”. Si no se verifica el formato apropiado de la cadena, entonces se deja como “peligrosa”, y si una cadena de este tipo puede influenciar el flujo de control del código en cierto punto del programa, entonces hay una falla potencial en la seguridad.

Comprobación de límites

Es más fácil cometer errores cuando se programa en un lenguaje de bajo nivel que en uno de alto nivel. Por ejemplo, muchas brechas de seguridad en los sistemas se producen debido a los desbordamientos en las entradas y salidas de los programas escritos en C. Como C no comproueba los límites de los arreglos, es responsabilidad del usuario asegurar que no se acceda a los arreglos fuera de los límites. Si no se comprueba que los datos suministrados por el usuario pueden llegar a desbordar un elemento, el programa podría caer en el truco de almacenar los datos del usuario fuera del espacio asociado a este elemento. Un atacante podría manipular los datos de entrada que hagan que el programa se comporte en forma errónea y comprometa la seguridad del sistema. Se han desarrollado técnicas para encontrar los desbordamientos de búfer en los programas, pero con un éxito limitado.

Si el programa se hubiera escrito en un lenguaje seguro que incluya la comprobación automática de los rangos, este problema no habría ocurrido. El mismo análisis del flujo de datos que se utiliza para eliminar las comprobaciones de rango redundantes podría usarse también para localizar los desbordamientos probables de un elemento. Sin embargo, la principal diferencia es que al no poder eliminar una comprobación de rango sólo se produciría un pequeño incremento en el tiempo de ejecución, mientras que el no identificar un desbordamiento potencial del búfer podría comprometer la seguridad del sistema. Por ende, aunque es adecuado utilizar técnicas simples para optimizar las comprobaciones de rangos, los análisis sofisticados, como el rastreo de los valores de los apuntadores entre un procedimiento y otro, son necesarios para obtener resultados de alta calidad en las herramientas para la detección de errores.

Herramientas de administración de memoria

La recolección de basura es otro excelente ejemplo de la concesión entre la eficiencia y una combinación de la facilidad de uso y la confiabilidad del software. La administración automática de la memoria elimina todos los errores de administración de memoria (por ejemplo, las “fugas de memoria”), que son una fuente importante de problemas en los programas en C y C++. Se han desarrollado varias herramientas para ayudar a los programadores a detectar los errores de administración de memoria. Por ejemplo, Purify es una herramienta que se emplea mucho, la cual capta en forma dinámica los errores de administración de memoria, a medida que ocurren. También se han desarrollado herramientas que ayudan a identificar algunos de estos problemas en forma estática.

1.6 Fundamentos de los lenguajes de programación

En esta sección hablaremos sobre la terminología más importante y las distinciones que aparecen en el estudio de los lenguajes de programación. No es nuestro objetivo abarcar todos los conceptos o todos los lenguajes de programación populares. Asumimos que el lector está familiarizado por lo menos con uno de los lenguajes C, C++, C# o Java, y que tal vez conozca otros.

1.6.1 La distinción entre estático y dinámico

Una de las cuestiones más importantes a las que nos enfrentamos al diseñar un compilador para un lenguaje es la de qué decisiones puede realizar el compilador acerca de un programa. Si un lenguaje utiliza una directiva que permite al compilador decidir sobre una cuestión, entonces decimos que el lenguaje utiliza una directiva *estática*, o que la cuestión puede decidirse en *tiempo de compilación*. Por otro lado, se dice que una directiva que sólo permite realizar una decisión a la hora de ejecutar el programa es una *directiva dinámica*, o que requiere una decisión en *tiempo de ejecución*.

Una de las cuestiones en las que nos debemos de concentrar es en el alcance de las declaraciones. El *alcance* de una declaración de *x* es la región del programa en la que los usos de *x* se refieren a esta declaración. Un lenguaje utiliza el *alcance estático* o *alcance léxico* si es posible determinar el alcance de una declaración con sólo ver el programa. En cualquier otro caso, el lenguaje utiliza un *alcance dinámico*. Con el alcance dinámico, a medida que se ejecuta el programa, el mismo uso de *x* podría referirse a una de varias declaraciones distintas de *x*.

La mayoría de los lenguajes, como C y Java, utilizan el alcance estático. En la sección 1.6.3 hablaremos sobre éste.

Ejemplo 1.3: Como otro ejemplo de la distinción entre estático y dinámico, considere el uso del término “static” según se aplica a los datos en la declaración de una clase en Java. En Java, una variable es un nombre para una ubicación en memoria que se utiliza para almacenar un valor de datos. Aquí, “static” no se refiere al alcance de la variable, sino a la habilidad del compilador para determinar la ubicación en memoria en la que puede encontrarse la variable declarada. Una declaración como:

```
public static int x;
```

hace de x una *variable de clase* e indica que sólo hay una copia de x , sin importar cuántos objetos se creen de esta clase. Lo que es más, el compilador puede determinar una ubicación en memoria en la que se almacene este entero x . En contraste, si se hubiera omitido la palabra “static” de esta declaración, entonces cada objeto de la clase tendría su propia ubicación en la que se guardara x , y el compilador no podría determinar todos estos lugares antes de ejecutar el programa. \square

1.6.2 Entornos y estados

Otra distinción importante que debemos hacer al hablar sobre los lenguajes de programación es si los cambios que ocurren a medida que el programa se ejecuta afectan a los valores de los elementos de datos, o si afectan a la interpretación de los nombres para esos datos. Por ejemplo, la ejecución de una asignación como $x = y + 1$ cambia el valor denotado por el nombre x . Dicho en forma más específica, la asignación cambia el valor en cualquier ubicación denotada por x .

Tal vez sea menos claro que la ubicación denotada por x puede cambiar en tiempo de ejecución. Por ejemplo, como vimos en el ejemplo 1.3, si x no es una variable estática (o de “clase”), entonces cada objeto de la clase tiene su propia ubicación para una instancia de la variable x . En este caso, la asignación para x puede cambiar cualquiera de esas variables de “instancia”, dependiendo del objeto en el que se aplique un método que contenga esa asignación.

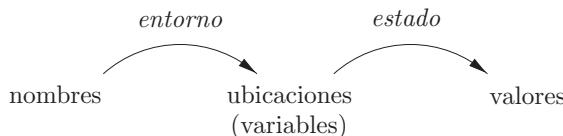


Figura 1.8: Asignación de dos etapas, de nombres a valores

La asociación de nombres con ubicaciones en memoria (el *almacén*) y después con valores puede describirse mediante dos asignaciones que cambian a medida que se ejecuta el programa (vea la figura 1.8):

1. El *entorno* es una asignación de nombres a ubicaciones de memoria. Como las variables se refieren a ubicaciones (“l-values” en la terminología de C), podríamos definir de manera alternativa un entorno como una asignación de nombres a variables.
2. El *estado* es una asignación de las ubicaciones en memoria a sus valores. Es decir, el estado asigna l-values a sus correspondientes r-values, en la terminología de C.

Los entornos cambian de acuerdo a las reglas de alcance de un lenguaje.

Ejemplo 1.4: Considere el fragmento de un programa en C en la figura 1.9. El entero i se declara como una variable global, y también se declara como una variable local para la función f . Cuando f se ejecuta, el entorno se ajusta de manera que el nombre i se refiera a la ubicación reservada para la i que es local para f , y cualquier uso de i , como la asignación $i = 3$ que

```

...
int i;          /* i global */
...
void f(...) {
    int i;          /* i local */
    ...
    i = 3;          /* uso de la i local */
    ...
}
...
x = i + 1;      /* uso de la i global */

```

Figura 1.9: Dos declaraciones del nombre *i*

se muestra en forma explícita, se refiere a esa ubicación. Por lo general, a la *i* local se le otorga un lugar en la pila en tiempo de ejecución.

Cada vez que se ejecuta una función *g* distinta de *f*, los usos de *i* no pueden referirse a la *i* que es local para *f*. Los usos del nombre *i* en *g* deben estar dentro del alcance de alguna otra declaración de *i*. Un ejemplo es la instrucción *x = i+1* mostrada en forma explícita, la cual se encuentra dentro de un procedimiento cuya definición no se muestra. La *i* en *i+1* se refiere supuestamente a la *i* global. Al igual que en la mayoría de los lenguajes, las declaraciones en C deben anteponer su uso, por lo que una función que esté antes de la *i* global no puede hacer referencia a ella. □

Las asignaciones de entorno y de estado en la figura 1.8 son dinámicas, pero hay unas cuantas excepciones:

1. *Comparación entre enlace estático y dinámico* de los nombres con las ubicaciones. La mayoría de la vinculación de los nombres con las ubicaciones es dinámica, y veremos varios métodos para esta vinculación a lo largo de esta sección. Algunas declaraciones, como la *i* global en la figura 1.9, pueden recibir una ubicación en memoria una sola vez, a medida que el compilador genera el código objeto.²
2. *Comparación entre enlace estático y dinámico* de las ubicaciones con los valores. Por lo general, el enlace de las ubicaciones con los valores (la segunda etapa en la figura 1.8) es dinámica también, ya que no podemos conocer el valor en una ubicación sino hasta ejecutar el programa. Las constantes declaradas son una excepción. Por ejemplo, la siguiente definición en C:

```
#define ARRAYSIZE 1000
```

²Técnicamente, el compilador de C asignará una ubicación en memoria virtual para la *i* global, dejando la responsabilidad al cargador y al sistema operativo de determinar en qué parte de la memoria física de la máquina se ubicará *i*. Sin embargo, no debemos preocuparnos por las cuestiones de “reubicación” tales como éstas, que no tienen impacto sobre la compilación. En vez de ello, tratamos el espacio de direcciones que utiliza el compilador para su código de salida como si otorgara las ubicaciones de memoria física.

Nombres, identificadores y variables

Aunque los términos “nombre” y “variable” a menudo se refieren a lo mismo, los utilizamos con cuidado para diferenciar entre los nombres en tiempo de compilación y las ubicaciones en tiempo de ejecución denotadas por los nombres.

Un *identificador* es una cadena de caracteres, por lo general letras o dígitos, que se refiere a (identifica) una entidad, como un objeto de datos, un procedimiento, una clase o un tipo. Todos los identificadores son nombres, pero no todos los nombres son identificadores. Los nombres también pueden ser expresiones. Por ejemplo, el nombre *x.y* podría denotar el campo *y* de una estructura denotada por *x*. Aquí, *x* y *y* son identificadores, mientras que *x.y* es un nombre, pero no un identificador. A los nombres compuestos como *x.y* se les llama nombres *calificados*.

Una *variable* se refiere a una ubicación específica en memoria. Es común que el mismo identificador se declare más de una vez; cada una de esas declaraciones introduce una nueva variable. Aun cuando cada identificador se declara sólo una vez, un identificador que sea local para un procedimiento recursivo hará referencia a las distintas ubicaciones de memoria, en distintas ocasiones.

enlaza el nombre **ARRAYSIZE** con el valor 1000 en forma estática. Podemos determinar este enlace analizando la instrucción, y sabemos que es imposible que esta vinculación cambie cuando se ejecute el programa.

1.6.3 Alcance estático y estructura de bloques

La mayoría de los lenguajes, incluyendo a C y su familia, utilizan el alcance estático. Las reglas de alcance para C se basan en la estructura del programa; el alcance de una declaración se determina en forma implícita, mediante el lugar en el que aparece la declaración en el programa. Los lenguajes posteriores, como C++, Java y C#, también proporcionan un control explícito sobre los alcances, a través del uso de palabras clave como **public**, **private** y **protected**.

En esta sección consideramos las reglas de alcance estático para un lenguaje con bloques, en donde un *bloque* es una agrupación de declaraciones e instrucciones. C utiliza las llaves { y } para delimitar un bloque; el uso alternativo de **begin** y **end** para el mismo fin se remonta hasta Algol.

Ejemplo 1.5: Para una primera aproximación, la directiva de alcance estático de C es la siguiente:

1. Un programa en C consiste en una secuencia de declaraciones de nivel superior de variables y funciones.
2. Las funciones pueden contener declaraciones de variables, en donde las variables incluyen variables locales y parámetros. El alcance de una declaración de este tipo se restringe a la función en la que aparece.

Procedimientos, funciones y métodos

Para evitar decir “procedimientos, funciones o métodos” cada vez que queremos hablar sobre un subprograma que pueda llamarse, nos referiremos, por lo general, a todos ellos como “procedimientos”. La excepción es que al hablar en forma explícita de los programas en lenguajes como C, que sólo tienen funciones, nos referiremos a ellos como “funciones”. O, si hablamos sobre un lenguaje como Java, que sólo tiene métodos, utilizaremos ese término.

Por lo general, una función devuelve un valor de algún tipo (el “tipo de retorno”), mientras que un procedimiento no devuelve ningún valor. C y los lenguajes similares, que sólo tienen funciones, tratan a los procedimientos como funciones con un tipo de retorno especial “void”, para indicar que no hay valor de retorno. Los lenguajes orientados a objetos como Java y C++ utilizan el término “métodos”. Éstos pueden comportarse como funciones o procedimientos, pero se asocian con una clase específica.

3. El alcance de una declaración de nivel superior de un nombre x consiste en todo el programa que le sigue, con la excepción de las instrucciones que se encuentran dentro de una función que también tiene una declaración de x .

Los detalles adicionales en relación con la directiva de alcance estático de C tratan con las declaraciones de variables dentro de instrucciones. Examinaremos dichas declaraciones a continuación y en el ejemplo 1.6. \square

En C, la sintaxis de los bloques se da en base a lo siguiente:

1. Un tipo de instrucción es un bloque. Los bloques pueden aparecer en cualquier parte en la que puedan aparecer otros tipos de instrucciones, como las instrucciones de asignación.
2. Un bloque es una secuencia de declaraciones que va seguida de una secuencia de instrucciones, todas rodeadas por llaves.

Observe que esta sintaxis permite anidar bloques, uno dentro de otro. Esta propiedad de anidamiento se conoce como *estructura de bloques*. La familia C de lenguajes tiene estructura de bloques, con la excepción de que una función tal vez no se defina dentro de otra.

Decimos que una declaración D “pertenece” a un bloque B , si B es el bloque anidado más cercano que contiene a D ; es decir, D se encuentra dentro de B , pero no dentro de cualquier bloque que esté anidado dentro de B .

La regla de alcance estático para las declaraciones de variables en un lenguaje estructurado por bloques es la siguiente: Si la declaración D del nombre x pertenece al bloque B , entonces el alcance de D es todo B , excepto para cualquier bloque B' anidado a cualquier profundidad dentro de B , en donde x se vuelve a declarar. Aquí, x se vuelve a declarar en B' si alguna otra declaración D' del mismo nombre x pertenece a B' .

Una forma equivalente de expresar esta regla es enfocándose en un uso de un nombre x . Digamos que B_1, B_2, \dots, B_k sean todos los bloques que rodean este uso de x , en donde B_k es el más pequeño, anidado dentro de B_{k-1} , que a su vez se anida dentro de B_{k-2} , y así en lo sucesivo. Hay que buscar la i más grande de tal forma que haya una declaración de x que pertenezca a B_i . Este uso de x se refiere a la declaración en B_i . De manera alternativa, este uso de x se encuentra dentro del alcance de la declaración en B_i .

```
main() {
    int a = 1;
    int b = 1;
    {
        int b = 2;                                B1
        {
            int a = 3;                            B2
            {
                cout << a << b;                B3
            }
            {
                int b = 4;                            B4
                cout << a << b;                B4
            }
            cout << a << b;
        }
        cout << a << b;
    }
}
```

Figura 1.10: Bloques en un programa en C++

Ejemplo 1.6: El programa en C++ de la figura 1.10 tiene cuatro bloques, con varias definiciones de las variables a y b . Como ayuda de memoria, cada declaración inicializa su variable con el número del bloque al que pertenece.

Por ejemplo, considere la declaración `int a = 1` en el bloque B_1 . Su alcance es todo B_1 , excepto los bloques anidados (tal vez con más profundidad) dentro de B_1 que tengan su propia declaración de a . B_2 , que está anidado justo dentro de B_1 , no tiene una declaración de a , pero B_3 sí. B_4 no tiene una declaración de a , por lo que el bloque B_3 es el único lugar en todo el programa que se encuentra fuera del alcance de la declaración del nombre a que pertenece a B_1 . Esto es, el alcance incluye a B_4 y a todo B_2 , excepto la parte de B_2 que se encuentra dentro de B_3 . Los alcances de las cinco declaraciones se sintetizan en la figura 1.11.

Desde otro punto de vista, vamos a considerar la instrucción de salida en el bloque B_4 y a enlazar las variables a y b que se utilizan ahí con las declaraciones apropiadas. La lista de bloques circundantes, en orden de menor a mayor tamaño, es B_4, B_2, B_1 . Observe que B_3 no rodea al punto en cuestión. B_4 tiene una declaración de b , por lo que es a esta declaración a la que este uso de b hace referencia, y el valor de b que se imprime es 4. Sin embargo, B_4 no tiene una declaración de a , por lo que ahora analizamos B_2 . Ese bloque no tiene una declaración de

DECLARACIÓN	ALCANCE
int a = 1;	$B_1 - B_3$
int b = 1;	$B_1 - B_2$
int b = 2;	$B_2 - B_4$
int a = 3;	B_3
int b = 4;	B_4

Figura 1.11: Alcances de las declaraciones en el ejemplo 1.6

a tampoco, por lo que continuamos con B_1 . Por fortuna, hay una declaración `int a = 1` que pertenece a ese bloque, por lo que el valor de *a* que se imprime es 1. Si no hubiera dicha declaración, el programa tendría error. \square

1.6.4 Control de acceso explícito

Las clases y las estructuras introducen un nuevo alcance para sus miembros. Si *p* es un objeto de una clase con un campo (miembro) *x*, entonces el uso de *x* en *p.x* se refiere al campo *x* en la definición de la clase. En analogía con la estructura de bloques, el alcance de la declaración de un miembro *x* en una clase *C* se extiende a cualquier subclase *C'*, excepto si *C'* tiene una declaración local del mismo nombre *x*.

Mediante el uso de palabras clave como **public**, **private** y **protected**, los lenguajes orientados a objetos como C++ o Java proporcionan un control explícito sobre el acceso a los nombres de los miembros en una superclase. Estas palabras clave soportan el *encapsulamiento* mediante la restricción del acceso. Por ende, los nombres privados reciben de manera intencional un alcance que incluye sólo las declaraciones de los métodos y las definiciones asociadas con esa clase, y con cualquier clase “amiga” (friend: el término de C++). Los nombres protegidos son accesibles para las subclases. Los nombres públicos son accesibles desde el exterior de la clase.

En C++, la definición de una clase puede estar separada de las definiciones de algunos o de todos sus métodos. Por lo tanto, un nombre *x* asociado con la clase *C* puede tener una región del código que se encuentra fuera de su alcance, seguida de otra región (la definición de un método) que se encuentra dentro de su alcance. De hecho, las regiones dentro y fuera del alcance pueden alternar, hasta que se hayan definido todos los métodos.

1.6.5 Alcance dinámico

Técnicamente, cualquier directiva de alcance es dinámica si se basa en un factor o factores que puedan conocerse sólo cuando se ejecute el programa. Sin embargo, el término *alcance dinámico* se refiere, por lo general, a la siguiente directiva: el uso de un nombre *x* se refiere a la declaración de *x* en el procedimiento que se haya llamado más recientemente con dicha declaración. El alcance dinámico de este tipo aparece sólo en situaciones especiales. Vamos a considerar dos ejemplos de directivas dinámicas: la expansión de macros en el preprocesador de C y la resolución de métodos en la programación orientada a objetos.

Declaraciones y definiciones

Los términos aparentemente similares “declaración” y “definición” para los conceptos de los lenguajes de programación, son en realidad bastante distintos. Las declaraciones nos hablan acerca de los tipos de cosas, mientras que las definiciones nos hablan sobre sus valores. Por ende, `int i` es una declaración de *i*, mientras que `i = 1` es una definición de *i*.

La diferencia es más considerable cuando tratamos con métodos u otros procedimientos. En C++, un método se declara en la definición de una clase, proporcionando los tipos de los argumentos y el resultado del método (que a menudo se le conoce como la *firma* del método). Despues el método se define (es decir, se proporciona el código para ejecutar el método) en otro lugar. De manera similar, es común definir una función de C en un archivo y declararla en otros archivos, en donde se utiliza esta función.

Ejemplo 1.7: En el programa en C de la figura 1.12, el identificador *a* es una macro que representa la expresión $(x + 1)$. Pero, ¿qué es *x*? No podemos resolver *x* de manera estática, es decir, en términos del texto del programa.

```
#define a (x+1)
int x = 2;
void b() { int x = 1; printf("%d\n", a); }
void c() { printf("%d\n", a); }
void main() { b(); c(); }
```

Figura 1.12: Una macro para la cual se debe calcular el alcance de sus nombres en forma dinámica

De hecho, para poder interpretar a *x*, debemos usar la regla de alcance dinámico ordinaria. Examinamos todas las llamadas a funciones que se encuentran activas, y tomamos la función que se haya llamado más recientemente y que tenga una declaración de *x*. A esta declaración es a la que se refiere *x*.

En el ejemplo de la figura 1.12, la función *main* primero llama a la función *b*. A medida que *b* se ejecuta, imprime el valor de la macro *a*. Como debemos sustituir $(x + 1)$ por *a*, resolvemos este uso de *x* con la declaración `int x = 1` en la función *b*. La razón es que *b* tiene una declaración de *x*, por lo que el término $(x + 1)$ en `printf` en *b* se refiere a esta *x*. Por ende, el valor que se imprime es 1.

Una vez que *b* termina y que se hace la llamada a *c*, debemos imprimir de nuevo el valor de la macro *a*. No obstante, la única *x* accesible para *c* es la *x* global. Por lo tanto, la instrucción `printf` en *c* se refiere a esta declaración de *x*, y se imprime el valor de 2. \square

La resolución dinámica del alcance también es esencial para los procedimientos polimórficos, aquellos que tienen dos o más definiciones para el mismo nombre, dependiendo sólo de los

Analogía entre el alcance estático y dinámico

Aunque podría haber cualquier número de directivas estáticas o dinámicas para el alcance, hay una interesante relación entre la regla de alcance estático normal (estructurado por bloques) y la directiva dinámica normal. En cierto sentido, la regla dinámica es para el tiempo lo que la regla estática es para el espacio. Mientras que la regla estática nos pide buscar la declaración cuya unidad (bloque) sea la más cercana en rodear la ubicación física del uso, la regla dinámica nos pide que busquemos la declaración cuya unidad (invocación a un procedimiento) sea la más cercana en rodear el momento del uso.

tipos de los argumentos. En algunos lenguajes como ML (vea la sección 7.3.3), es posible determinar los tipos en forma estática para todos los usos de los nombres, en cuyo caso el compilador puede sustituir cada uso del nombre de un procedimiento p por una referencia al código para el procedimiento apropiado. No obstante, en otros lenguajes como Java y C++, hay momentos en los que el compilador no puede realizar esa determinación.

Ejemplo 1.8: Una característica distintiva de la programación orientada a objetos es la habilidad de cada objeto de invocar el método apropiado, en respuesta a un mensaje. En otras palabras, el procedimiento al que se llama cuando se ejecuta $x.m()$ depende de la clase del objeto denotado por x en ese momento. A continuación se muestra un ejemplo típico:

1. Hay una clase C con un método llamado $m()$.
2. D es una subclase de C , y D tiene su propio método llamado $m()$.
3. Hay un uso de m de la forma $x.m()$, en donde x es un objeto de la clase C .

Por lo general, es imposible saber en tiempo de compilación si x será de la clase C o de la subclase D . Si la aplicación del método ocurre varias veces, es muy probable que algunas se realicen con objetos denotados por x , que estén en la clase C pero no en D , mientras que otras estarán en la clase D . No es sino hasta el tiempo de ejecución que se puede decidir cuál definición de m es la correcta. Por ende, el código generado por el compilador debe determinar la clase del objeto x , y llamar a uno de los dos métodos llamados m . \square

1.6.6 Mecanismos para el paso de parámetros

Todos los lenguajes de programación tienen una noción de un procedimiento, pero pueden diferir en cuanto a la forma en que estos procedimientos reciben sus argumentos. En esta sección vamos a considerar cómo se asocian los *parámetros actuales* (los parámetros que se utilizan en la llamada a un procedimiento) con los *parámetros formales* (los que se utilizan en la definición del procedimiento). El mecanismo que se utilice será el que determine la forma en que el código de secuencia de llamadas tratará a los parámetros. La gran mayoría de los lenguajes utilizan la “llamada por valor”, la “llamada por referencia”, o ambas. Vamos a explicar estos términos, junto con otro método conocido como “llamada por nombre”, que es principalmente de interés histórico.

Llamada por valor

En la *llamada por valor*, el parámetro actual se evalúa (si es una expresión) o se copia (si es una variable). El valor se coloca en la ubicación que pertenece al correspondiente parámetro formal del procedimiento al que se llamó. Este método se utiliza en C y en Java, además de ser una opción común en C++, así como en la mayoría de los demás lenguajes. La llamada por valor tiene el efecto de que todo el cálculo que involucra a los parámetros formales, y que realiza el procedimiento al que se llamó, es local para ese procedimiento, y los parámetros actuales en sí no pueden modificarse.

Sin embargo, observe que en C podemos pasar un apuntador a una variable para permitir que el procedimiento al que se llamó modifique la variable. De igual forma, los nombres de arreglos que se pasan como parámetros en C, C++ o Java, ofrecen al procedimiento que se llamó lo que es en efecto un apuntador, o una referencia al mismo arreglo. Por ende, si a es el nombre de un arreglo del procedimiento que hace la llamada, y se pasa por valor al correspondiente parámetro formal x , entonces una asignación tal como $x[i] = 2$ en realidad modifica el elemento del arreglo $a[2]$. La razón es que, aunque x obtiene una copia del valor de a , ese valor es en realidad un apuntador al inicio del área de memoria en donde se encuentra el arreglo llamado a .

De manera similar, en Java muchas variables son en realidad referencias, o apuntadores, a las cosas que representan. Esta observación se aplica a los arreglos, las cadenas y los objetos de todas las clases. Aún y cuando Java utiliza la llamada por valor en forma exclusiva, cada vez que pasamos el nombre de un objeto a un procedimiento al que se llamó, el valor que recibe ese procedimiento es, en efecto, un apuntador al objeto. En consecuencia, el procedimiento al que se llamó puede afectar al valor del objeto en sí.

Llamada por referencia

En la *llamada por referencia*, la dirección del parámetro actual se pasa al procedimiento al que se llamó como el valor del correspondiente parámetro formal. Los usos del parámetro formal en el código del procedimiento al que se llamó se implementan siguiendo este apuntador hasta la ubicación indicada por el procedimiento que hizo la llamada. Por lo tanto, las modificaciones al parámetro formal aparecen como cambios para el parámetro actual.

No obstante, si el parámetro actual es una expresión, entonces ésta se evalúa antes de la llamada y su valor se almacena en su propia ubicación. Los cambios al parámetro formal cambian esta ubicación, pero no pueden tener efecto sobre los datos del procedimiento que hizo la llamada.

La llamada por referencia se utiliza para los parámetros “ref” en C++, y es una opción en muchos otros lenguajes. Es casi esencial cuando el parámetro formal es un objeto, arreglo o estructura grande. La razón es que la llamada estricta por valor requiere que el procedimiento que hace la llamada copie todo el parámetro actual en el espacio que pertenece al correspondiente parámetro formal. Este proceso de copiado se vuelve extenso cuando el parámetro es grande. Como dijimos al hablar sobre la llamada por valor, los lenguajes como Java resuelven el problema de pasar arreglos, cadenas u otros objetos copiando sólo una referencia a esos objetos. El efecto es que Java se comporta como si usara la llamada por referencia para cualquier cosa que no sea un tipo básico como entero o real.

Llamada por nombre

Hay un tercer mecanismo (la llamada por nombre) que se utilizó en uno de los primeros lenguajes de programación: Algol 60. Este mecanismo requiere que el procedimiento al que se llamó se ejecute como si el parámetro actual se sustituyera literalmente por el parámetro formal en su código, como si el procedimiento formal fuera una macro que representa al parámetro actual (cambiando los nombres locales en el procedimiento al que se llamó, para que sean distintos). Cuando el parámetro actual es una expresión en vez de una variable, se producen ciertos comportamientos no intuitivos, razón por la cual este mecanismo no es popular hoy en día.

1.6.7 Uso de alias

Hay una consecuencia interesante del paso por parámetros tipo llamada por referencia o de su simulación, como en Java, en donde las referencias a los objetos se pasan por valor. Es posible que dos parámetros formales puedan referirse a la misma ubicación; se dice que dichas variables son *alias* una de la otra. Como resultado, dos variables cualesquiera, que dan la impresión de recibir sus valores de dos parámetros formales distintos, pueden convertirse en alias una de la otra, también.

Ejemplo 1.9: Suponga que a es un arreglo que pertenece a un procedimiento p , y que p llama a otro procedimiento $q(x,y)$ con una llamada $q(a,a)$. Suponga también que los parámetros se pasan por valor, pero que los nombres de los arreglos son en realidad referencias a la ubicación en la que se almacena el arreglo, como en C o los lenguajes similares. Ahora, las variables x y y se han convertido en alias una de la otra. El punto importante es que si dentro de q hay una asignación $x[10] = 2$, entonces el valor de $y[10]$ también se convierte en 2. \square

Resulta que es esencial comprender el uso de los alias y los mecanismos que los crean si un compilador va a optimizar un programa. Como veremos al inicio del capítulo 9, hay muchas situaciones en las que sólo podemos optimizar código si podemos estar seguros de que ciertas variables no son alias. Por ejemplo, podríamos determinar que $x = 2$ es el único lugar en el que se asigna la variable x . De ser así, entonces podemos sustituir el uso de x por el uso de 2; por ejemplo, sustituimos $a = x+3$ por el término más simple $a = 5$. Pero suponga que hay otra variable y que sirve como alias para x . Entonces una asignación $y = 4$ podría tener el efecto inesperado de cambiar a x . Esto también podría significar que sería un error sustituir $a = x+3$ por $a = 5$, ya que el valor correcto de a podría ser 7.

1.6.8 Ejercicios para la sección 1.6

Ejercicio 1.6.1: Para el código en C estructurado por bloques de la figura 1.13(a), indique los valores asignados a w , x , y y z .

Ejercicio 1.6.2: Repita el ejercicio 1.6.1 para el código de la figura 1.13(b).

Ejercicio 1.6.3: Para el código estructurado por bloques de la figura 1.14, suponiendo el alcance estático usual de las declaraciones, dé el alcance para cada una de las doce declaraciones.

```

int w, x, y, z;
int i = 4; int j = 5;
{
    int j = 7;
    i = 6;
    w = i + j;
}
x = i + j;
{
    int i = 8;
    y = i + j;
}
z = i + j;

```

(a) Código para el ejercicio 1.6.1

```

int w, x, y, z;
int i = 3; int j = 4;
{
    int i = 5;
    w = i + j;
}
x = i + j;
{
    int j = 6;
    i = 7;
    y = i + j;
}
z = i + j;

```

(b) Código para el ejercicio 1.6.2

Figura 1.13: Código estructurado por bloques

```

{
    int w, x, y, z;      /* Block B1 */
    {
        int x, z;        /* Block B2 */
        {
            int w, x;    /* Block B3 */
        }
    {
        int w, x;        /* Block B4 */
        {
            int y, z;    /* Block B5 */
        }
    }
}

```

Figura 1.14: Código estructurado por bloques para el ejercicio 1.6.3

Ejercicio 1.6.4: ¿Qué imprime el siguiente código en C?

```

#define a (x+1)
int x = 2;
void b() { x = a; printf("%d\n", x); }
void c() { int x = 1; printf("%d\n"), a; }
void main() { b(); c(); }

```

1.7 Resumen del capítulo 1

- ◆ *Procesadores de lenguaje.* Un entorno de desarrollo integrado de software incluye muchos tipos distintos de procesadores de lenguaje, como compiladores, intérpretes, ensambladores, enlazadores, cargadores, depuradores, profilers.
- ◆ *Fases del compilador.* Un compilador opera como una secuencia de fases, cada una de las cuales transforma el programa fuente de una representación intermedia a otra.

- ◆ *Lenguajes máquina y ensamblador.* Los lenguajes máquina fueron los lenguajes de programación de la primera generación, seguidos de los lenguajes ensambladores. La programación en estos lenguajes requería de mucho tiempo y estaba propensa a errores.
- ◆ *Modelado en el diseño de compiladores.* El diseño de compiladores es una de las fases en las que la teoría ha tenido el mayor impacto sobre la práctica. Entre los modelos que se han encontrado de utilidad se encuentran: autómatas, gramáticas, expresiones regulares, árboles y muchos otros.
- ◆ *Optimización de código.* Aunque el código no puede verdaderamente “optimizarse”, esta ciencia de mejorar la eficiencia del código es tanto compleja como muy importante. Constituye una gran parte del estudio de la compilación.
- ◆ *Lenguajes de alto nivel.* A medida que transcurre el tiempo, los lenguajes de programación se encargan cada vez más de las tareas que se dejaban antes al programador, como la administración de memoria, la comprobación de consistencia en los tipos, o la ejecución del código en paralelo.
- ◆ *Compiladores y arquitectura de computadoras.* La tecnología de compiladores ejerce una influencia sobre la arquitectura de computadoras, así como también se ve influenciada por los avances en la arquitectura. Muchas innovaciones modernas en la arquitectura dependen de la capacidad de los compiladores para extraer de los programas fuente las oportunidades de usar con efectividad las capacidades del hardware.
- ◆ *Productividad y seguridad del software.* La misma tecnología que permite a los compiladores optimizar el código puede usarse para una variedad de tareas de análisis de programas, que van desde la detección de errores comunes en los programas, hasta el descubrimiento de que un programa es vulnerable a uno de los muchos tipos de intrusiones que han descubierto los “hackers”.
- ◆ *Reglas de alcance.* El *alcance* de una declaración de x es el contexto en el cual los usos de x se refieren a esta declaración. Un lenguaje utiliza el *alcance estático* o *alcance léxico* si es posible determinar el alcance de una declaración con sólo analizar el programa. En cualquier otro caso, el lenguaje utiliza un *alcance dinámico*.
- ◆ *Entornos.* La asociación de nombres con ubicaciones en memoria y después con los valores puede describirse en términos de *entornos*, los cuales asignan los nombres a las ubicaciones en memoria, y los *estados*, que asignan las ubicaciones a sus valores.
- ◆ *Estructura de bloques.* Se dice que los lenguajes que permiten anidar bloques tienen *estructura de bloques*. Un nombre x en un bloque anidado B se encuentra en el alcance de una declaración D de x en un bloque circundante, si no existe otra declaración de x en un bloque intermedio.
- ◆ *Paso de parámetros.* Los parámetros se pasan de un procedimiento que hace la llamada al procedimiento que es llamado, ya sea por valor o por referencia. Cuando se pasan objetos grandes por valor, los valores que se pasan son en realidad referencias a los mismos objetos, lo cual resulta en una llamada por referencia efectiva.

- ◆ *Uso de alias.* Cuando los parámetros se pasan (de manera efectiva) por referencia, dos parámetros formales pueden referirse al mismo objeto. Esta posibilidad permite que un cambio en una variable cambie a la otra.

1.8 Referencias para el capítulo 1

Para el desarrollo de los lenguajes de programación que se crearon y han estado en uso desde 1967, incluyendo Fortran, Algol, Lisp y Simula, vea [7]. Para los lenguajes que se crearon para 1982, incluyendo C, C++, Pascal y Smalltalk, vea [1].

La Colección de compiladores de GNU, gcc, es una fuente popular de compiladores de código fuente abierto para C, C++, Fortran, Java y otros lenguajes [2]. Phoenix es un kit de herramientas para construir compiladores que proporciona un framework integrado para construir las fases de análisis del programa, generación de código y optimización de código de los compiladores que veremos en este libro [3].

Para obtener más información acerca de los conceptos de los lenguajes de programación, recomendamos [5,6]. Para obtener más información sobre la arquitectura de computadoras y el impacto que tiene en la compilación, sugerimos [4].

1. Bergin, T. J. y R. G. Gibson, *History of Programming Languages*, ACM Press, Nueva York, 1996.
2. <http://gcc.gnu.org/>.
3. <http://research.microsoft.com/phoenix/default.aspx>.
4. Hennessy, J. L. y D. A. Patterson, *Computer Organization and Design: The Hardware/Software Interface*, Morgan-Kaufmann, San Francisco, CA, 2004.
5. Scott M. L., *Programming Language Pragmatics*, segunda edición, Morgan-Kaufmann, San Francisco, CA, 2006.
6. Sethi, R., *Programming Languages: Concepts and Constructs*, Addison-Wesley, 1996.
7. Wexelblat, R. L., *History of Programming Languages*, Academic Press, Nueva York, 1981.

Capítulo 2

Un traductor simple orientado a la sintaxis

Este capítulo es una introducción a las técnicas de compilación que veremos en los capítulos 3 a 6 de este libro. Presenta las técnicas mediante el desarrollo de un programa funcional en Java que traduce instrucciones representativas de un lenguaje de programación en un código de tres direcciones, una representación intermedia. En este capítulo haremos énfasis en el frontend de un compilador, en especial en el análisis léxico, el análisis sintáctico (parsing), y la generación de código intermedio. En los capítulos 7 y 8 veremos cómo generar instrucciones de máquina a partir del código de tres direcciones.

Empezaremos con algo pequeño, creando un traductor orientado a la sintaxis que asigne expresiones aritméticas infijo a expresiones postfijo. Despues extenderemos este traductor para que asigne fragmentos de código (como se muestra en la figura 2.1) a un código de tres direcciones de la forma que se presenta en la figura 2.2.

El traductor funcional en Java aparece en el apéndice A. El uso de Java es conveniente, pero no esencial. De hecho, las ideas en este capítulo son anteriores a la creación de Java y de C.

```
{  
    int i; int j; float[100] a; float v; float x;  
    while ( true ) {  
        do i = i+1; while ( a[i] < v );  
        do j = j-1; while ( a[j] > v );  
        if ( i >= j ) break;  
        x = a[i]; a[i] = a[j]; a[j] = x;  
    }  
}
```

Figura 2.1: Un fragmento de código que se va a traducir

```

1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
4: j = j - 1
5: t2 = a [ j ]
6: if t2 > v goto 4
7: ifFalse i >= j goto 9
8: goto 14
9: x = a [ i ]
10: t3 = a [ j ]
11: a [ i ] = t3
12: a [ j ] = x
13: goto 1
14:

```

Figura 2.2: Código intermedio simplificado para el fragmento del programa de la figura 2.1

2.1 Introducción

La fase de análisis de un compilador descompone un programa fuente en piezas componentes y produce una representación interna, a la cual se le conoce como código intermedio. La fase de síntesis traduce el código intermedio en el programa destino.

El análisis se organiza de acuerdo con la “sintaxis” del lenguaje que se va a compilar. La *sintaxis* de un lenguaje de programación describe el formato apropiado de sus programas, mientras que la *semántica* del lenguaje define lo que sus programas significan; es decir, lo que hace cada programa cuando se ejecuta. Para especificar la sintaxis, en la sección 2.2 presentamos una notación de uso popular, llamada gramáticas sin contexto o BNF (Forma de Backus-Naur). Con las notaciones que se tienen disponibles, es mucho más difícil describir la semántica de un lenguaje que la sintaxis. Para especificar la semántica, deberemos, por lo tanto, usar descripciones informales y ejemplos sugerentes.

Además de especificar la sintaxis de un lenguaje, se puede utilizar una gramática libre de contexto para ayudar a guiar la traducción de los programas. En la sección 2.3 presentamos una técnica de compilación orientada a la gramática, conocida como *traducción orientada a la sintaxis*. En la sección 2.4 presentamos el análisis sintáctico (parsing).

El resto de este capítulo es una guía rápida a través del modelo de un front-end de un compilador en la figura 2.3. Empezamos con el analizador sintáctico. Por cuestión de simplicidad, consideramos la traducción orientada a la sintaxis de las expresiones infijas al formato postfijo, una notación en la cual los operadores aparecen después de sus operandos. Por ejemplo, el formato postfijo de la expresión $9 - 5 + 2$ es $95 - 2 +$. La traducción al formato postfijo es lo bastante completa como para ilustrar el análisis sintáctico, y a la vez lo bastante simple como para que se pueda mostrar el traductor por completo en la sección 2.5. El traductor simple maneja expresiones como $9 - 5 + 2$, que consisten en dígitos separados por signos positivos y negativos. Una razón para empezar con dichas expresiones simples es que el analizador sintáctico puede trabajar directamente con los caracteres individuales para los operadores y los operandos.

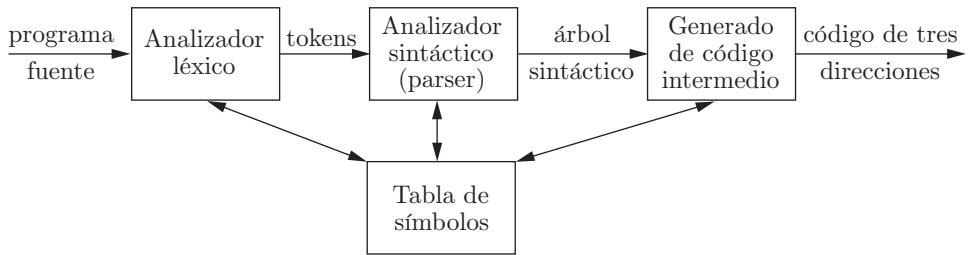
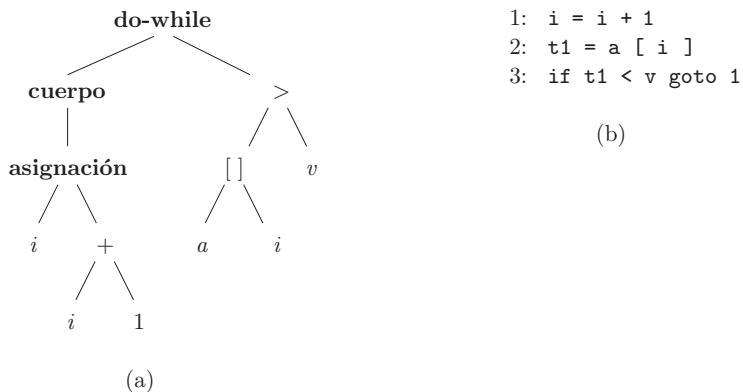


Figura 2.3: Un modelo de una front-end de un compilador

Un analizador léxico permite que un traductor maneje instrucciones de varios caracteres como identificadores, que se escriben como secuencias de caracteres, pero se tratan como unidades conocidas como *tokens* durante el análisis sintáctico; por ejemplo, en la expresión `cuenta+1`, el identificador `cuenta` se trata como una unidad. El analizador léxico en la sección 2.6 permite que aparezcan números, identificadores y “espacio en blanco” (espacios, tabuladores y caracteres de nueva línea) dentro de las expresiones.

A continuación, consideraremos la generación de código intermedio. En la figura 2.4 se ilustran dos formas de código intermedio. Una forma, conocida como *árboles sintácticos abstractos* o simplemente *árboles sintácticos*, representa la estructura sintáctica jerárquica del programa fuente. En el modelo de la figura 2.3, el analizador sintáctico produce un árbol sintáctico, que se traduce posteriormente en código de tres direcciones. Algunos compiladores combinan el análisis sintáctico y la generación de código intermedio en un solo componente.

Figura 2.4: Código intermedio para “`do i=i+1; while (a[i] < v);`”

La raíz del árbol sintáctico abstracto en la figura 2.4(a) representa un ciclo do-while completo. El hijo izquierdo de la raíz representa el cuerpo del ciclo, que consiste sólo de la asignación `i=i+1`. El hijo derecho de la raíz representa la condición `a[i]<v`. En la sección 2.8(a) aparece una implementación de los árboles sintácticos.

La otra representación intermedia común, que se muestra en la figura 2.4(b), es una secuencia de instrucciones de “tres direcciones”; en la figura 2.2 aparece un ejemplo más completo. Esta forma de código intermedio recibe su nombre de las instrucciones de la forma $x = y \text{ op } z$, en donde **op** es un operador binario, y y z son las direcciones para los operandos, y x es la dirección del resultado de la operación. Una instrucción de tres direcciones lleva a cabo cuando menos una operación, por lo general, un cálculo, una comparación o una bifurcación.

En el apéndice A reunimos las técnicas descritas en este capítulo para crear un front-end de un compilador en Java. La interfaz traduce las instrucciones en instrucciones en lenguaje ensamblador.

2.2 Definición de sintaxis

En esta sección presentamos una notación (la “gramática libre de contexto”, o simplemente “gramática”) que se utiliza para especificar la sintaxis de un lenguaje. Utilizaremos las gramáticas a lo largo de este libro para organizar los front-ends de los compiladores.

Una gramática describe en forma natural la estructura jerárquica de la mayoría de las instrucciones de un lenguaje de programación. Por ejemplo, una instrucción if-else en Java puede tener la siguiente forma:

if (*expr*) *instr* **else** *instr*

Esto es, una instrucción if-else es la concatenación de la palabra clave **if**, un paréntesis abierto, una expresión, un paréntesis cerrado, una instrucción, la palabra clave **else** y otra instrucción. Mediante el uso de la variable *expr* para denotar una expresión y la variable *instr* para denotar una instrucción, esta regla de estructuración puede expresarse de la siguiente manera:

instr → **if** (*expr*) *instr* **else** *instr*

en donde la flecha se lee como “puede tener la forma”. A dicha regla se le llama *producción*. En una producción, los elementos léxicos como la palabra clave **if** y los paréntesis se llaman *terminales*. Las variables como *expr* e *instr* representan secuencias de terminales, y se llaman *no terminales*.

2.2.1 Definición de gramáticas

Una *gramática libre de contexto* tiene cuatro componentes:

1. Un conjunto de símbolos *terminales*, a los que algunas veces se les conoce como “tokens”. Los terminales son los símbolos elementales del lenguaje definido por la gramática.
2. Un conjunto de *no terminales*, a las que algunas veces se les conoce como “variables sintácticas”. Cada no terminal representa un conjunto de cadenas o terminales, de una forma que describiremos más adelante.
3. Un conjunto de *producciones*, en donde cada producción consiste en un no terminal, llamada *encabezado* o *lado izquierdo* de la producción, una flecha y una secuencia de

Comparación entre tokens y terminales

En un compilador, el analizador léxico lee los caracteres del programa fuente, los agrupa en unidades con significado léxico llamadas lexemas, y produce como salida tokens que representan estos lexemas. Un token consiste en dos componentes, el nombre del token y un valor de atributo. Los nombres de los tokens son símbolos abstractos que utiliza el analizador sintáctico para su análisis. A menudo, a estos tokens les llamamos *terminales*, ya que aparecen como símbolos terminales en la gramática para un lenguaje de programación. El valor de atributo, si está presente, es un apuntador a la tabla de símbolos que contiene información adicional acerca del token. Esta información adicional no forma parte de la gramática, por lo que en nuestra explicación sobre el análisis sintáctico, a menudo nos referimos a los tokens y los terminales como sinónimos.

terminales y no terminales, llamada *cuerpo* o *lado derecho* de la producción. La intención intuitiva de una producción es especificar una de las formas escritas de una instrucción; si el no terminal del encabezado representa a una instrucción, entonces el cuerpo representa una forma escrita de la instrucción.

4. Una designación de una de los no terminales como el símbolo *inicial*.

Para especificar las gramáticas presentamos sus producciones, en donde primero se listan las producciones para el símbolo inicial. Suponemos que los dígitos, los signos como $<$ y \leq , y las cadenas en negritas como **while** son terminales. Un nombre en cursiva es un no terminal, y se puede asumir que cualquier nombre o símbolo que no esté en cursiva es un terminal.¹ Por conveniencia de notación, las producciones con el mismo no terminal que el encabezado pueden agrupar sus cuerpos, con los cuerpos alternativos separados por el símbolo $|$, que leemos como “o”.

Ejemplo 2.1: Varios ejemplos en este capítulo utilizan expresiones que consisten en dígitos y signos positivos y negativos; por ejemplo, las cadenas como $9-5+2$, $3-1$ o 7 . Debido a que debe aparecer un signo positivo o negativo entre dos dígitos, nos referimos a tales expresiones como “listas de dígitos separados por signos positivos o negativos”. La siguiente gramática describe la sintaxis de estas expresiones. Las producciones son:

$$\text{lista} \rightarrow \text{lista} + \text{dígito} \quad (2.1)$$

$$\text{lista} \rightarrow \text{lista} - \text{dígito} \quad (2.2)$$

$$\text{lista} \rightarrow \text{dígito} \quad (2.3)$$

$$\text{dígito} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \quad (2.4)$$

¹Utilizaremos letras individuales en cursiva para fines adicionales, en especial cuando estudiemos las gramáticas con detalle en el capítulo 4. Por ejemplo, vamos a usar X , Y y Z para hablar sobre un símbolo que puede ser o no terminal. No obstante, cualquier nombre en cursiva que contenga dos o más caracteres seguirá representando un no terminal.

Los cuerpos de las tres producciones con la *lista* no terminal como encabezado pueden agruparse de la siguiente manera equivalente:

$$\textit{lista} \rightarrow \textit{lista} + \textit{dígito} \mid \textit{lista} - \textit{dígito} \mid \textit{dígito}$$

De acuerdo con nuestras convenciones, los terminales de la gramática son los siguientes símbolos:

$$+ \ - \ 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9$$

Los no terminales son los nombres en cursiva *lista* y *dígito*, en donde *lista* es el símbolo inicial, ya que sus producciones se dan primero. \square

Decimos que una producción es *para* un no terminal, si el no terminal es el encabezado de la producción. Una cadena de terminales es una secuencia de cero o más terminales. La cadena de cero terminales, escrita como ϵ , se llama cadena *vacía*.²

2.2.2 Derivaciones

Una gramática deriva cadenas empezando con el símbolo inicial y sustituyendo en forma repetida un no terminal, mediante el cuerpo de una producción para ese no terminal. Las cadenas de terminales que pueden derivarse del símbolo inicial del *lenguaje* definido por la gramática.

Ejemplo 2.2: El lenguaje definido por la gramática del ejemplo 2.1 consiste en listas de dígitos separadas por signos positivos y negativos. Las diez producciones para el *dígito* no terminal le permiten representar a cualquiera de los terminales $0, 1, \dots, 9$. De la producción (2.3), un dígito por sí solo es una lista. Las producciones (2.1) y (2.2) expresan la regla que establece que cualquier lista seguida de un signo positivo o negativo, y después de otro dígito, forma una nueva lista.

Las producciones (2.1) a (2.4) son todo lo que necesitamos para definir el lenguaje deseado. Por ejemplo, podemos deducir que $9-5+2$ es una *lista* de la siguiente manera:

- 9 es una *lista* por la producción (2.3), ya que 9 es un *dígito*.
- $9-5$ es una *lista* por la producción (2.2), ya que 9 es una *lista* y 5 es un *dígito*.
- $9-5+2$ es una *lista* por la producción (2.1), ya que $9-5$ es una *lista* y 2 es un *dígito*. \square

Ejemplo 2.3: Un tipo de lista algo distinto es la lista de parámetros en la llamada a una función. En Java, los parámetros se encierran entre paréntesis, como en la llamada `max(x,y)` de la función `max` con los parámetros `x` y `y`. Un matiz de dichas listas es que una lista vacía de parámetros puede encontrarse entre los terminales (y `)`). Podemos empezar a desarrollar una gramática para dichas secuencias con las siguientes producciones:

²Técnicamente, ϵ puede ser una cadena de cero símbolos de cualquier alfabeto (colección de símbolos).

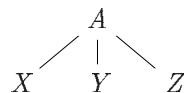
$$\begin{aligned}
 \text{llamada} &\rightarrow \mathbf{id} (\text{paramsopc}) \\
 \text{paramsopc} &\rightarrow \text{params} \mid \epsilon \\
 \text{params} &\rightarrow \text{params} , \text{ param} \mid \text{param}
 \end{aligned}$$

Observe que el segundo cuerpo posible para *paramsopc* (“lista de parámetros opcionales”) es ϵ , que representa la cadena vacía de símbolos. Es decir, *paramsopc* puede sustituirse por la cadena vacía, por lo que una *llamada* puede consistir en un nombre de función, seguido de la cadena de dos terminales $()$. Observe que las producciones para *params* son análogas para las de *lista* en el ejemplo 2.1, con una coma en lugar del operador aritmético $+$ o $-$, y *param* en vez de *dígito*. No hemos mostrado las producciones para *param*, ya que los parámetros son en realidad expresiones arbitrarias. En breve hablaremos sobre las producciones apropiadas para las diversas construcciones de los lenguajes, como expresiones, instrucciones, etcétera. \square

El *análisis sintáctico (parsing)* es el problema de tomar una cadena de terminales y averiguar cómo derivarla a partir del símbolo inicial de la gramática, y si no puede derivarse a partir de este símbolo, entonces hay que reportar los errores dentro de la cadena. El análisis sintáctico es uno de los problemas más fundamentales en todo lo relacionado con la compilación; los principales métodos para el análisis sintáctico se describen en el capítulo 4. En este capítulo, por cuestión de simplicidad, empezamos con programas fuente como 9-5+2 en los que cada carácter es un terminal; en general, un programa fuente tiene lexemas de varios caracteres que el analizador léxico agrupa en tokens, cuyos primeros componentes son los terminales procesados por el analizador sintáctico.

2.2.3 Árboles de análisis sintáctico

Un árbol de análisis sintáctico muestra, en forma gráfica, la manera en que el símbolo inicial de una gramática deriva a una cadena en el lenguaje. Si el no terminal *A* tiene una producción $A \rightarrow XYZ$, entonces un árbol de análisis sintáctico podría tener un nodo interior etiquetado como *A*, con tres hijos llamados *X*, *Y* y *Z*, de izquierda a derecha:



De manera formal, dada una gramática libre de contexto, un *árbol de análisis sintáctico* de acuerdo con la gramática es un árbol con las siguientes propiedades:

1. La raíz se etiqueta con el símbolo inicial.
2. Cada hoja se etiqueta con un terminal, o con ϵ .
3. Cada nodo interior se etiqueta con un no terminal.
4. Si *A* es el no terminal que etiqueta a cierto nodo interior, y X_1, X_2, \dots, X_n son las etiquetas de los hijos de ese nodo de izquierda a derecha, entonces debe haber una producción $A \rightarrow X_1 X_2 \dots X_n$. Aquí, cada una de las etiquetas X_1, X_2, \dots, X_n representa a un símbolo

Terminología de árboles

Las estructuras de datos tipo árbol figuran de manera prominente en la compilación.

- Un árbol consiste en uno o más *nodos*. Los nodos pueden tener *etiquetas*, que en este libro, por lo general, serán símbolos de la gramática. Al dibujar un árbol, con frecuencia representamos los nodos mediante estas etiquetas solamente.
- Sólo uno de los nodos es la *raíz*. Todos los nodos, excepto la raíz, tienen un *padre* único; la raíz no tiene padre. Al dibujar árboles, colocamos el padre de un nodo encima de ese nodo y dibujamos una línea entre ellos. Entonces, la raíz es el nodo más alto (superior).
- Si el nodo *N* es el padre del nodo *M*, entonces *M* es *hijo* de *N*. Los hijos de nuestro nodo se llaman *hermanos*. Tienen un orden, *partiendo desde la izquierda*, por lo que al dibujar árboles, ordenamos los hijos de un nodo dado en esta forma.
- Un nodo sin hijos se llama *hoja*. Los otros nodos (los que tienen uno o más hijos) son *nodos interiores*.
- Un *descendiente* de un nodo *N* es ya sea el mismo *N*, un hijo de *N*, un hijo de un hijo de *N*, y así en lo sucesivo, para cualquier número de niveles. Decimos que el nodo *N* es un *ancestro* del nodo *M*, si *M* es descendiente de *N*.

que puede ser o no un terminal. Como un caso especial, si $A \rightarrow \epsilon$ es una producción, entonces un nodo etiquetado como *A* puede tener un solo hijo, etiquetado como ϵ .

Ejemplo 2.4: La derivación de 9-5+2 en el ejemplo 2.2 se ilustra mediante el árbol en la figura 2.5. Cada nodo en el árbol se etiqueta mediante un símbolo de la gramática. Un nodo interior y su hijo corresponden a una producción; el nodo interior corresponde al encabezado de la producción, el hijo corresponde al cuerpo.

En la figura 2.5, la raíz se etiqueta como *lista*, el símbolo inicial de la gramática en el ejemplo 2.1. Los hijos de la raíz se etiquetan, de izquierda a derecha, como *lista*, $+$ y *dígito*. Observe que

$$\textit{lista} \rightarrow \textit{lista} + \textit{dígito}$$

es una producción en la gramática del ejemplo 2.1. El hijo izquierdo de la raíz es similar a la raíz, con un hijo etiquetado como $-$ en vez de $+$. Los tres nodos etiquetados como *dígito* tienen cada uno un hijo que se etiqueta mediante un dígito. \square

De izquierda a derecha, las hojas de un árbol de análisis sintáctico forman la *derivación* del árbol: la cadena que se *genera* o *deriva* del no terminal en la raíz del árbol de análisis sintáctico. En la figura 2.5, la derivación es 9-5+2; por conveniencia, todas las hojas se muestran en el nivel inferior. Por lo tanto, no es necesario alinear las hojas de esta forma. Cualquier árbol

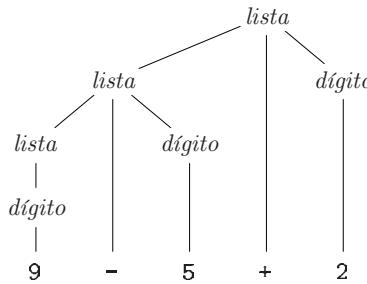


Figura 2.5: Árbol de análisis sintáctico para $9-5+2$, de acuerdo con la gramática en el ejemplo 2.1

imparte un orden natural de izquierda a derecha a sus hojas, con base en la idea de que si X y Y son dos hijos con el mismo parente, y X está a la izquierda de Y , entonces todos los descendientes de X están a la izquierda de los descendientes de Y .

Otra definición del lenguaje generado por una gramática es como el conjunto de cadenas que puede generar cierto árbol de análisis sintáctico. Al proceso de encontrar un árbol de análisis sintáctico para una cadena dada de terminales se le llama *analizar sintácticamente* esa cadena.

2.2.4 Ambigüedad

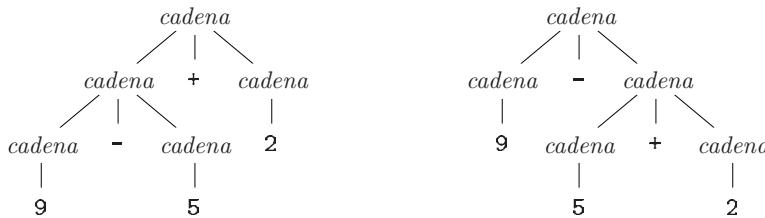
Tenemos que ser cuidadosos al hablar sobre *la estructura* de una cadena, de acuerdo a una gramática. Una gramática puede tener más de un árbol de análisis sintáctico que genere una cadena dada de terminales. Se dice que dicha gramática es *ambigua*. Para mostrar que una gramática es ambigua, todo lo que debemos hacer es buscar una cadena de terminales que sea la derivación de más de un árbol de análisis sintáctico. Como una cadena con más de un árbol de análisis sintáctico tiene, por lo general, más de un significado, debemos diseñar gramáticas no ambiguas para las aplicaciones de compilación, o utilizar gramáticas ambiguas con reglas adicionales para resolver las ambigüedades.

Ejemplo 2.5: Suponga que utilizamos una sola *cadena* no terminal y que no diferenciamos entre los dígitos y las listas, como en el ejemplo 2.1. Podríamos haber escrito la siguiente gramática:

cadena \rightarrow *cadena* + *cadena* | *cadena* - *cadena* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Mezclar la noción de *dígito* y *lista* en la *cadena* no terminal tiene sentido superficial, ya que un *dígito* individual es un caso especial de una *lista*.

No obstante, la figura 2.6 muestra que una expresión como $9-5+2$ tiene más de un árbol de análisis sintáctico con esta gramática. Los dos árboles para $9-5+2$ corresponden a las dos formas de aplicar paréntesis a la expresión: $(9-5)+2$ y $9-(5+2)$. Este segundo uso de los paréntesis proporciona a la expresión el valor inesperado de 2, en vez del valor ordinario de 6. La gramática del ejemplo 2.1 no permite esta interpretación. \square

Figura 2.6: Dos árboles de análisis sintáctico para $9-5+2$

2.2.5 Asociatividad de los operadores

Por convención, $9+5+2$ es equivalente a $(9+5)+2$ y $9-5-2$ es equivalente a $(9-5)-2$. Cuando un operando como 5 tiene operadores a su izquierda y a su derecha, se requieren convenciones para decidir qué operador se aplica a ese operando. Decimos que el operador $+$ se *asocia* por la izquierda, porque un operando con signos positivos en ambos lados de él pertenece al operador que está a su izquierda. En la mayoría de los lenguajes de programación, los cuatro operadores aritméticos (suma, resta, multiplicación y división) son asociativos por la izquierda.

Algunos operadores comunes, como la exponenciación, son asociativos por la derecha. Como otro ejemplo, el operador de asignación $=$ en C y sus descendientes es asociativo por la derecha; es decir, la expresión $a=b=c$ se trata de la misma forma que la expresión $a=(b=c)$.

Las cadenas como $a=b=c$ con un operador asociativo por la derecha se generan mediante la siguiente gramática:

$$\begin{aligned} \text{derecha} &\rightarrow \text{letra} = \text{derecha} \mid \text{letra} \\ \text{letra} &\rightarrow a \mid b \mid \dots \mid z \end{aligned}$$

El contraste entre un árbol de análisis sintáctico para un operador asociativo por la izquierda como $-$, y un árbol de análisis sintáctico para un operador asociativo por la derecha como $=$, se muestra en la figura 2.7. Observe que el árbol de análisis sintáctico para $9-5-2$ crece hacia abajo y a la izquierda, mientras que el árbol de análisis sintáctico para $a=b=c$ crece hacia abajo y a la derecha.

2.2.6 Precedencia de operadores

Considere la expresión $9+5*2$. Hay dos posibles interpretaciones de esta expresión: $(9+5)*2$ o $9+(5*2)$. Las reglas de asociatividad para $+$ y $*$ se aplican a las ocurrencias del mismo operador, por lo que no resuelven esta ambigüedad. Las reglas que definen la precedencia relativa de los operadores son necesarias cuando hay más de un tipo de operador presente.

Decimos que $*$ tiene *mayor precedencia* que $+$, si $*$ recibe sus operandos antes que $+$. En la aritmética ordinaria, la multiplicación y la división tienen mayor precedencia que la suma y la resta. Por lo tanto, $*$ recibe el 5 tanto en $9+5*2$ como en $9*5+2$; es decir, las expresiones son equivalentes a $9+(5*2)$ y $(9*5)+2$, respectivamente.

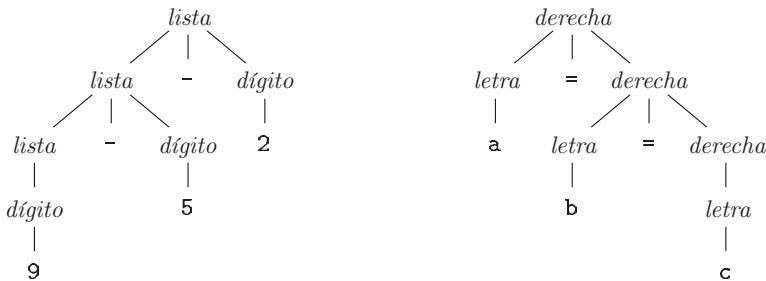


Figura 2.7: Árboles de análisis sintáctico para las gramáticas asociativas por la izquierda y por la derecha

Ejemplo 2.6: Podemos construir una gramática para expresiones aritméticas a partir de una tabla que muestre la asociatividad y la precedencia de los operadores. Empezamos con los cuatro operadores aritméticos comunes y una tabla de precedencia, mostrando los operadores en orden de menor a mayor precedencia. Los operadores en la misma línea tienen la misma asociatividad y precedencia:

$$\begin{array}{ll} \text{asociativo por la izquierda:} & + - \\ \text{asociativo por la derecha:} & * / \end{array}$$

Creamos dos no terminales llamadas *expr* y *term* para los dos niveles de precedencia, y un no terminal adicional llamado *factor* para generar unidades básicas en las expresiones. Las unidades básicas en las expresiones son dígitos y expresiones entre paréntesis.

$$\text{factor} \rightarrow \text{dígito} \mid (\text{expr})$$

Ahora consideraremos los operadores binarios, $*$ y $/$, que tienen la mayor precedencia. Como estos operadores asocian por la izquierda, las producciones son similares a las de las listas que asocian por la izquierda.

$$\begin{array}{ll} \text{term} \rightarrow & \text{term} * \text{factor} \\ & \mid \text{term} / \text{factor} \\ & \mid \text{factor} \end{array}$$

De manera similar, *expr* genera listas de términos separados por los operadores aditivos:

$$\begin{array}{ll} \text{expr} \rightarrow & \text{expr} + \text{term} \\ & \mid \text{expr} - \text{term} \\ & \mid \text{term} \end{array}$$

Por lo tanto, la gramática resultante es:

$$\begin{array}{ll} \text{expr} \rightarrow & \text{expr} + \text{term} \mid \text{expr} - \text{term} \mid \text{term} \\ \text{term} \rightarrow & \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor} \\ \text{factor} \rightarrow & \text{dígito} \mid (\text{expr}) \end{array}$$

Generalización de la gramática de expresiones del ejemplo 2.6

Podemos considerar un factor como una expresión que no puede “separarse” mediante ningún operador. Al usar el término “separar”, queremos indicar que al colocar un operador enseguida de cualquier factor, en cualquier lado, ninguna pieza del factor, excepto en su totalidad, se convertirá en un operando de ese operador. Si el factor es una expresión con paréntesis, éstos la protegen contra dicha “separación”, mientras que si el factor es un solo operando, no puede separarse.

Un término (que no sea también un factor) es una expresión que puede separarse mediante los operadores de la mayor precedencia: * y /, pero no mediante los operadores de menor precedencia. Una expresión (que no sea un término o factor) puede separarse mediante cualquier operador.

Podemos generalizar esta idea con cualquier número n de niveles de precedencia. Necesitamos $n + 1$ no terminales. El primero, como *factor* en el ejemplo 2.6, nunca podrá separarse. Por lo general, los cuerpos de producciones para esta no terminal son sólo operandos individuales y expresiones con paréntesis. Entonces, para cada nivel de precedencia hay un no terminal que representa a las expresiones que pueden separarse sólo mediante operadores en ese nivel, o en uno superior. Por lo general, las producciones para esta no terminal tienen cuerpos que representan los usos de los operadores en ese nivel, más un cuerpo que sólo es la no terminal para el siguiente nivel superior.

Con esta gramática, una expresión es una lista de términos separados por los signos + o -, y un término es una lista de factores separados por los signos * o /. Observe que cualquier expresión con paréntesis es un factor, así que con los paréntesis podemos desarrollar expresiones que tengan un anidamiento con profundidad arbitraria (y árboles con una profundidad arbitraria). \square

Ejemplo 2.7: Las palabras clave nos permiten reconocer instrucciones, ya que la mayoría de éstas empiezan con una palabra clave o un carácter especial. Las excepciones a esta regla incluyen las asignaciones y las llamadas a procedimientos. Las instrucciones definidas por la gramática (ambigua) en la figura 2.8 son legales en Java.

En la primera producción para *instr*, el terminal **id** representa a cualquier identificador. Las producciones para *expresión* no se muestran. Las instrucciones de asignación especificadas por la primera producción son legales en Java, aunque Java trata a = como un operador de asignación que puede aparecer dentro de una expresión. Por ejemplo, Java permite **a=b=c**, mientras que esta gramática no.

El no terminal *instrs* genera una lista posiblemente vacía de instrucciones. La segunda producción para *instrs* genera la lista vacía ϵ . La primera producción genera una lista posiblemente vacía de instrucciones seguidas por una instrucción.

La colocación de los signos de punto y coma es sutil; aparecen al final de cada cuerpo que no termina en *instr*. Este enfoque evita la acumulación de puntos y comas después de instrucciones como if y while, que terminan con subinstrucciones anidadas. Cuando la subinstrucción anidada sea una instrucción o un do-while, se generará un punto y coma como parte de la subinstrucción. \square

$$\begin{array}{lcl}
 \text{instr} & \rightarrow & \text{id} = \text{expresión} ; \\
 & | & \text{if} (\text{expresión}) \text{ instr} \\
 & | & \text{if} = \text{expresión} ; \text{ instr} \text{ else} \text{ instr} \\
 & | & \text{while} (\text{expresión}) \text{ instr} \\
 & | & \text{do} \text{ instr} \text{ while} (\text{expresión}) ; \\
 & | & \{ \text{instrs} \} \\
 \\
 \text{instrs} & \rightarrow & \text{instrs} \text{ instr} \\
 & | & \epsilon
 \end{array}$$

Figura 2.8: Una gramática para un subconjunto de instrucciones de Java

2.2.7 Ejercicios para la sección 2.2

Ejercicio 2.2.1: Considere la siguiente gramática libre de contexto:

$$S \rightarrow S S + \mid S S * \mid a$$

- Muestre cómo puede generarse la cadena **aa+a*** mediante esta gramática.
- Construya un árbol de análisis sintáctico para esta cadena.
- ¿Qué lenguaje genera esta gramática? Justifique su respuesta.

Ejercicio 2.2.2: ¿Qué lenguaje se genera mediante las siguientes gramáticas? En cada caso, justifique su respuesta.

- $S \rightarrow 0 S 1 \mid 0 1$
- $S \rightarrow + S S \mid - S S \mid a$
- $S \rightarrow S (S) S \mid \epsilon$
- $S \rightarrow a S b S \mid b S a S \mid \epsilon$
- $S \rightarrow a \mid S + S \mid S S \mid S^* \mid (S)$

Ejercicio 2.2.3: ¿Cuáles de las gramáticas en el ejercicio 2.2.2 son ambiguas?

Ejercicio 2.2.4: Construya gramáticas libres de contexto no ambiguas para cada uno de los siguientes lenguajes. En cada caso muestre que su gramática es correcta.

- Expresiones aritméticas en notación prefija.
- Listas asociativas por la izquierda de identificadores separados por comas.
- Listas asociativas por la derecha de identificadores separados por comas.
- Expresiones aritméticas de enteros e identificadores con los cuatro operadores binarios **+, -, *, /.**

! e) Agregue el operador unario de suma y de resta a los operadores aritméticos de (d).

Ejercicio 2.2.5:

- a) Muestre que todas las cadenas binarias generadas por la siguiente gramática tienen valores que pueden dividirse entre 3. *Sugerencia:* Use la inducción en el número de nodos en un árbol de análisis sintáctico.

$$num \rightarrow 11 \mid 1001 \mid num\ 0 \mid num\ num$$

- b) ¿La gramática genera todas las cadenas binarias con valores divisibles entre 3?

Ejercicio 2.2.6: Construya una gramática libre de contexto para los números romanos.

2.3 Traducción orientada a la sintaxis

La traducción orientada a la sintaxis se realiza uniendo reglas o fragmentos de un programa a las producciones en una gramática. Por ejemplo, considere una expresión *expr* generada por la siguiente producción:

$$expr \rightarrow expr_1 + term$$

Aquí, *expr* es la suma de las dos subexpresiones *expr₁* y *term*. El subíndice en *expr₁* se utiliza sólo para diferenciar la instancia de *expr* en el cuerpo de la producción, del encabezado de la producción. Podemos traducir *expr* explotando su estructura, como en el siguiente seudocódigo:

```
traducir expr1;
traducir term;
manejar +;
```

Mediante una variante de este seudocódigo, vamos a construir un árbol sintáctico para *expr* en la sección 2.8, creando árboles sintácticos para *expr₁* y *term*, y después manearemos el + construyéndole un nodo. Por conveniencia, el ejemplo en esta sección es la traducción de las expresiones infijas a la notación postfijo.

Esta sección introduce dos conceptos relacionados con la traducción orientada a la sintaxis:

- *Atributos.* Un *atributo* es cualquier cantidad asociada con una construcción de programación. Algunos ejemplos de atributos son los tipos de datos de las expresiones, el número de instrucciones en el código generado, o la ubicación de la primera instrucción en el código generado para una construcción, entre muchas otras posibilidades. Como utilizamos símbolos de la gramática (no terminales y terminales) para representar las construcciones de programación, extendemos la noción de los atributos, de las construcciones a los símbolos que las representan.

- *Esquemas de traducción (orientada a la sintaxis).* Un *esquema de traducción* es una notación para unir los fragmentos de un programa a las producciones de una gramática. Los fragmentos del programa se ejecutan cuando se utiliza la producción durante el análisis sintáctico. El resultado combinado de todas estas ejecuciones de los fragmentos, en el orden inducido por el análisis sintáctico, produce la traducción del programa al cual se aplica este proceso de análisis/síntesis.

Utilizaremos las traducciones orientadas a la sintaxis a lo largo de este capítulo para traducir las expresiones infijas en notación postfija, para evaluar expresiones y para construir árboles sintácticos para las construcciones de programación. En el capítulo 5 aparece una discusión más detallada sobre los formalismos orientados a la sintaxis.

2.3.1 Notación postfija

Los ejemplos en esta sección manejan la traducción a la notación postfija. La *notación postfija* para una expresión E puede definirse de manera inductiva, como se muestra a continuación:

1. Si E es una variable o constante, entonces la notación postfija para E es la misma E .
2. Si E es una expresión de la forma $E_1 \mathbf{op} E_2$, en donde **op** es cualquier operador binario, entonces la notación postfija para E es $E'_1 E'_2 \mathbf{op}$, en donde E'_1 y E'_2 son las notaciones postfija para E_1 y E_2 , respectivamente.
3. Si E es una expresión con paréntesis de la forma (E_1) , entonces la notación postfija para E es la misma que la notación postfija para E_1 .

Ejemplo 2.8: La notación postfija para $(9-5)+2$ es $95-2+$. Es decir, las traducciones de 9, 5 y 2 son las mismas constantes, en base a la regla (1). Entonces, la traducción de $9-5$ es $95-$ en base a la regla (2). La traducción de $(9-5)$ es la misma en base a la regla (3). Habiendo traducido la subexpresión con paréntesis, podemos aplicar la regla (2) a toda la expresión, con $(9-5)$ en el papel de E_1 y 2 en el papel de E_2 , para obtener el resultado $95-2+$.

Como otro ejemplo, la notación postfija para $9-(5+2)$ es $952+-$. Es decir, primero se traduce $5+2$ a $52+$, y esta expresión se convierte en el segundo argumento del signo negativo. \square

No se necesitan paréntesis en la notación postfija, debido a que la posición y la *aridad* (número de argumentos) de los operadores sólo permiten una decodificación de una expresión postfija. El “truco” es explorar varias veces la cadena postfija partiendo de la izquierda, hasta encontrar un operador. Después, se busca a la izquierda el número apropiado de operandos, y se agrupa este operador con sus operandos. Se evalúa el operador con los operandos y se sustituyen por el resultado. Después se repite el proceso, continuando a la derecha y buscando otro operador.

Ejemplo 2.9: Considere la expresión postfija $952+-3*$. Si exploramos partiendo de la izquierda, primero encontramos el signo positivo. Si buscamos a su izquierda encontramos los operandos 5 y 2. Su suma, 7, sustituye a $52+$ y tenemos la cadena $97-3*$. Ahora, el operador

de más a la izquierda es el signo negativo, y sus operandos son 9 y 7. Si los sustituimos por el resultado de la resta nos queda 23*. Por último, el signo de multiplicación se asigna a 2 y 3, lo cual produce el resultado de 6. \square

2.3.2 Atributos sintetizados

La idea de asociar cantidades con construcciones de programación (por ejemplo, valores y tipos con expresiones) puede expresarse en términos de gramáticas. Asociamos los atributos con los no terminales y terminales. Después, unimos reglas a las producciones de la gramática; estas reglas describen la forma en que se calculan los atributos en esos nodos del árbol de análisis sintáctico en donde la producción en cuestión se utiliza para relacionar un nodo con sus hijos.

Una *definición orientada a la sintaxis* se asocia:

1. Con cada símbolo de gramática, un conjunto de atributos.
2. Con cada producción, un conjunto de *reglas semánticas* para calcular los valores de los atributos asociados con los símbolos que aparecen en la producción.

Los atributos pueden evaluarse de la siguiente forma. Para una cadena de entrada x dada, se construye un árbol de análisis sintáctico para x . Después, se aplican las reglas semánticas para evaluar los atributos en cada nodo del árbol de análisis sintáctico, como se indica a continuación.

Suponga que un nodo N en un árbol de análisis sintáctico se etiqueta mediante el símbolo de gramática X . Escribimos $X.a$ para denotar el valor del atributo a de X en ese nodo. A un árbol de análisis sintáctico que muestra los valores de los atributos en cada nodo se le conoce como árbol de análisis sintáctico *anotado*. Por ejemplo, la figura 2.9 muestra un árbol de análisis sintáctico anotado para $9-5+2$ con un atributo t asociado con los no terminales *expr* y *term*. El valor $9-5+2$ del atributo en la raíz es la notación postfija para $9-5+2$. En breve veremos cómo se calculan estas expresiones.

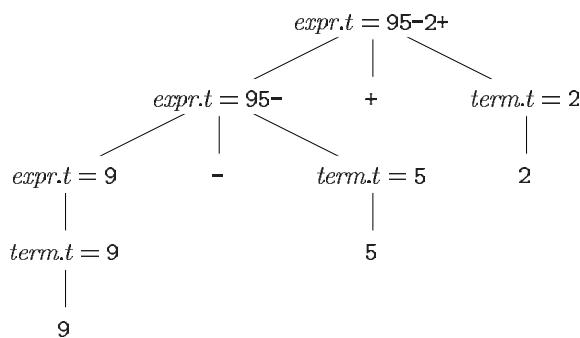


Figura 2.9: Valores de los atributos en los nodos de un árbol de análisis sintáctico

Se dice que un atributo está *sintetizado* si su valor en el nodo N de un árbol de análisis sintáctico se determina mediante los valores de los atributos del hijo de N y del mismo N . Los atributos sintetizados tienen la atractiva propiedad de que pueden evaluarse durante un

recorrido de abajo hacia arriba transversal de un árbol de análisis sintáctico. En la sección 5.1.1 hablaremos sobre otro tipo importante de atributo: el atributo “heredado”. De manera informal, los atributos heredados tienen su valor en un nodo de un árbol de análisis sintáctico que se determina mediante los valores de los atributos en el mismo nodo, en su padre y en sus hermanos del árbol.

Ejemplo 2.10: El árbol de análisis sintáctico anotado en la figura 2.9 se basa en la definición orientada a la sintaxis de la figura 2.10 para traducir expresiones que consisten en dígitos separados por los signos positivo o negativo, a la notación postfija. Cada no terminal tiene un atributo t con valor de cadena, el cual representa a la notación postfija para la expresión generada por esa no terminal en un árbol de análisis sintáctico. El símbolo \parallel en la regla semántica es el operador para la concatenación de cadenas.

PRODUCCIÓN	REGLAS SEMÁNTICAS
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \parallel term.t \parallel '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \parallel term.t \parallel '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

Figura 2.10: Definición orientada a la sintaxis para la traducción de infija a postfija

La forma postfija de un dígito es el mismo dígito; por ejemplo, la regla semántica asociada con la producción $term \rightarrow 9$ define a $term.t$ como el mismo 9, cada vez que se utiliza esta producción en un nodo de un árbol de análisis sintáctico. Los otros dígitos se traducen de manera similar. Como otro ejemplo, cuando se aplica la producción $expr \rightarrow term$, el valor de $term.t$ se convierte en el valor de $expr.t$.

La producción $expr \rightarrow expr_1 + term$ deriva a una expresión que contiene un operador de suma.³ El operando izquierdo del operador de suma se proporciona mediante $expr_1$, y el operando derecho mediante $term$. La regla semántica

$$expr.t = expr_1.t \parallel term.t \parallel '+'$$

asociada con esta producción construye el valor del atributo $expr.t$ mediante una concatenación de las formas postfijas $expr_1.t$ y $term.t$ de los operandos izquierdo y derecho, respectivamente, y después adjunta el signo de suma. Esta regla es una formalización de la definición de “expresión postfija”. \square

³En ésta y en muchas otras reglas, el mismo no terminal (en este caso, $expr$) aparece varias veces. El propósito del subíndice 1 en $expr_1$ es diferenciar las dos ocurrencias de $expr$ en la producción; el “1” no forma parte del no terminal. Para obtener más detalles consulte el cuadro titulado “Usos de un no terminal para diferenciar convenciones”.

Usos de un no terminal para diferenciar convenciones

A menudo, en las reglas tenemos la necesidad de diferenciar entre varios usos del mismo no terminal en el encabezado y cuerpo de una producción; para ilustrar esto, vea el ejemplo 2.10. La razón es que en el árbol de análisis sintáctico, distintos nodos etiquetados por el mismo no terminal, por lo general, tienen distintos valores para sus traducciones. Nosotros vamos a adoptar la siguiente convención: el no terminal aparece sin subíndice en el encabezado y con distintos subíndices en el cuerpo. Todas son ocurrencias del mismo no terminal, y el subíndice no forma parte de su nombre. No obstante, hay que alertar al lector sobre la diferencia entre los ejemplos de traducciones específicas, en donde se utiliza esta convención, y las producciones genéricas como $A \rightarrow X_1 X_2, \dots, X_n$, en donde las X con subíndices representan una lista arbitraria de símbolos de gramática, y *no* son instancias de un no terminal específico llamado X .

2.3.3 Definiciones simples orientadas a la sintaxis

La definición orientada a la sintaxis en el ejemplo 2.10 tiene la siguiente propiedad importante: la cadena que representa la traducción del no terminal en el encabezado de cada producción es la concatenación de las traducciones de los no terminales en el cuerpo de la producción, en el mismo orden que en la producción, con algunas cadenas adicionales opcionales entrelazadas. Una definición orientada a la sintaxis con esta propiedad se denomina como *simple*.

Ejemplo 2.11: Considere la primera producción y regla semántica de la figura 2.10:

PRODUCCIÓN	REGLA SEMÁNTICA	(2.5)
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \parallel term.t \parallel '+'$	

Aquí, la traducción $expr.t$ es la concatenación de las traducciones de $expr_1$ y $term$, seguida por el símbolo $+$. Observe que $expr_1$ y $term$ aparecen en el mismo orden, tanto en el cuerpo de la producción como en la regla semántica. No hay símbolos adicionales antes o entre sus traducciones. En este ejemplo, el único símbolo adicional ocurre al final. \square

Cuando hablamos sobre los esquemas de traducción, veremos que puede implementarse una definición simple orientada a la sintaxis con sólo imprimir las cadenas adicionales, en el orden en el que aparezcan en la definición.

2.3.4 Recorridos de los árboles

Utilizaremos los recorridos de los árboles para describir la evaluación de los atributos y especificar la ejecución de los fragmentos de código en un esquema de traducción. Un *recorrido* de un árbol empieza en la raíz y visita cada nodo del árbol en cierto orden.

Un recorrido del tipo *primero en profundidad* empieza en la raíz y visita en forma recursiva los hijos de cada nodo en cualquier orden, no necesariamente de izquierda a derecha. Se llama “primero en profundidad” debido a que visita cuando pueda a un hijo de un nodo que no haya sido visitado, de manera que visita a los nodos que estén a cierta distancia (“profundidad”) de la raíz lo más rápido que pueda.

El procedimiento *visitar(N)* en la figura 2.11 es un recorrido primero en profundidad, el cual visita a los hijos de un nodo en el orden de izquierda a derecha, como se muestra en la figura 2.12. En este recorrido hemos incluido la acción de evaluar las traducciones en cada nodo, justo antes de terminar con el nodo (es decir, después de que se hayan calculado las traducciones en el hijo). En general, las acciones asociadas con un recorrido pueden ser cualquier cosa que elijamos, o ninguna acción en sí.

```
procedure visitar(nodo N) {
    for ( cada hijo (C de N, de izquierda a derecha ) {
        visitar(C);
    }
    evaluar las reglas semánticas en el nodo N;
}
```

Figura 2.11: Un recorrido tipo “primero en profundidad” de un árbol

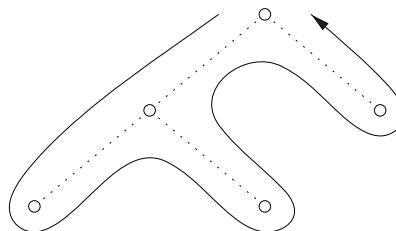


Figura 2.12: Ejemplo de un recorrido tipo “primero en profundidad” de un árbol

Una definición orientada a la sintaxis no impone un orden específico para la evaluación de los atributos en un árbol de análisis sintáctico; cualquier orden de evaluación que calcule un atributo *a* después de todos los demás atributos de los que *a* dependa es aceptable. Los atributos sintetizados pueden evaluarse durante cualquier recorrido de *abajo-arriba*; es decir, un recorrido que evalúe los atributos en un nodo después de haber evaluado los atributos de sus hijos. En general, tanto con los atributos sintetizados como con los heredados, la cuestión acerca del orden de evaluación es bastante compleja; vea la sección 5.2.

2.3.5 Esquemas de traducción

La definición orientada a la sintaxis en la figura 2.10 genera una traducción uniendo cadenas como atributos para los nodos en el árbol de análisis sintáctico. Ahora consideraremos un método alternativo que no necesita manipular cadenas; produce la misma traducción en forma incremental, mediante la ejecución de fragmentos del programa.

Recorridos preorden y postorden

Los recorridos en preorden y postorden son dos casos especiales de recorridos tipo “primero en profundidad”, en los cuales se visitan los hijos de cada nodo de izquierda a derecha.

A menudo, recorremos un árbol para realizar cierta acción específica en cada nodo. Si la acción se realiza cuando visitamos a un nodo por primera vez, entonces podemos referirnos al recorrido como un *recorrido en preorden*. De manera similar, si la acción se realiza justo antes de dejar un nodo por última vez, entonces decimos que es un *recorrido en postorden* del árbol. El procedimiento *visitar(N)* en la figura 2.11 es un ejemplo de un recorrido postorden.

Los recorridos en preorden y postorden definen los ordenamientos correspondientes en los nodos, con base en el momento en el que se va a realizar una acción en un nodo. El *recorrido en preorden* de un (sub)árbol con raíz en el nodo N consiste en N , seguido por los recorridos preorden de los subárboles de cada uno de sus hijos, si existen, empezando desde la izquierda. El *recorrido en postorden* de un (sub)árbol con raíz en N consiste en los recorridos en postorden de cada uno de los subárboles de los hijos de N , si existen, empezando desde la izquierda, seguidos del mismo N .

Un esquema de traducción orientado a la sintaxis es una notación para especificar una traducción, uniendo los fragmentos de un programa a las producciones en una gramática. Un esquema de traducción es como una definición orientada a la sintaxis, sólo que el orden de evaluación de las reglas semánticas se especifica en forma explícita.

Los fragmentos de un programa incrustados dentro de los cuerpos de las producciones se llaman *acciones semánticas*. La posición en la que debe ejecutarse una acción se muestra encerrada entre llaves y se escribe dentro del cuerpo de producción, como en el siguiente ejemplo:

resto → + *term* {print('+'')} *resto*₁

Veremos esas reglas a la hora de considerar una forma alternativa de gramática para las expresiones, en donde el no terminal llamada *resto* representa “todo excepto el primer término de una expresión”. En la sección 2.4.5 hablaremos sobre esta forma de gramática. De nuevo, el subíndice en *resto*₁ diferencia esta instancia del no terminal *resto* en el cuerpo de la producción, de la instancia de *resto* en el encabezado de la producción.

Al dibujar un árbol de análisis sintáctico para un esquema de traducción, para indicar una acción le construimos un hijo adicional, conectado mediante una línea punteada al nodo que corresponde a la cabeza de la producción. Por ejemplo, la porción del árbol de análisis sintáctico para la producción y acción anteriores se muestra en la figura 2.13. El nodo para una acción semántica no tiene hijos, por lo que la acción se realiza la primera vez que se ve el nodo.

Ejemplo 2.12: El árbol de análisis sintáctico en la figura 2.14 tiene instrucciones “print” en hojas adicionales, las cuales se adjuntan mediante líneas punteadas a los nodos interiores del árbol de análisis sintáctico. El esquema de traducción aparece en la figura 2.15. La gramática subyacente genera expresiones que consisten en dígitos separados por los signos positivo y ne-

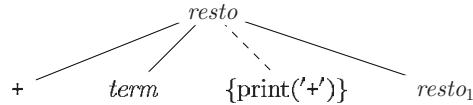
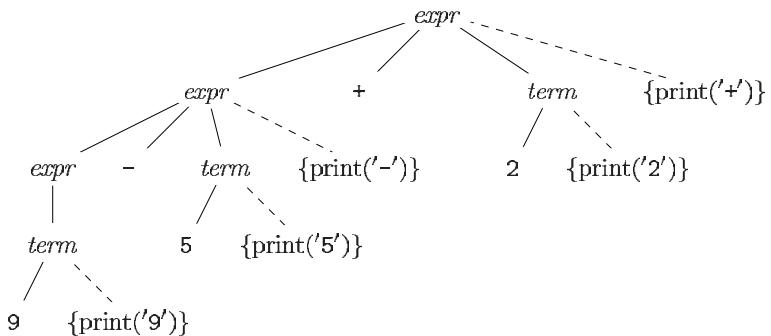


Figura 2.13: Se construye una hoja adicional para una acción semántica

gativo. Las acciones incrustadas en los cuerpos de producción traducen dichas expresiones a la notación postfija, siempre y cuando realicemos un recorrido del tipo “primero en profundidad” de izquierda a derecha del árbol, y ejecutemos cada instrucción “print” a medida que visitemos sus hojas.

Figura 2.14: Acciones para traducir $9-5+2$ a $95-2+$

$expr$	\rightarrow	$expr_1 + term$	$\{print('+'\')\}$
$expr$	\rightarrow	$expr_1 - term$	$\{print('-\')\}$
$expr$	\rightarrow	$term$	
$term$	\rightarrow	0	$\{print('0')\}$
$term$	\rightarrow	1	$\{print('1')\}$
	\dots		
$term$	\rightarrow	9	$\{print('9')\}$

Figura 2.15: Acciones para traducir a la notación postfija

La raíz de la figura 2.14 representa la primera producción en la figura 2.15. En un recorrido en postorden, primero realizamos todas las acciones en el subárbol de más a la izquierda de la raíz, para el operando izquierdo, también etiquetado $expr$ como la raíz. Despues visitamos la hoja $+$ en la cual no hay acción. Despues realizamos las acciones en el subárbol para el operando derecho $term$ y, por ultimo, la acción semántica $\{print('+'\')\}$ en el nodo adicional.

Como las producciones para $term$ sólo tienen un dígito en el lado derecho, las acciones para las producciones imprimen ese dígito. No es necesario ningún tipo de salida para la producción $expr \rightarrow term$, y sólo se necesita imprimir el operador en la acción para cada una de las pri-

meras dos producciones. Al ejecutarse durante un recorrido en postorden del árbol de análisis sintáctico, las acciones en la figura 2.14 imprimen 95-2+. \square

Observe que, aunque los esquemas en las figuras 2.10 y 2.15 producen la misma traducción, la construyen en forma distinta; la figura 2.10 adjunta cadenas como atributos a los nodos en el árbol de análisis sintáctico, mientras que el esquema en la figura 2.15 imprime la traducción en forma incremental, a través de acciones semánticas.

Las acciones semánticas en el árbol de análisis sintáctico en la figura 2.14 traducen la expresión infija 9-5+2 a 95-2+, imprimiendo cada carácter en 9-5+2 sólo una vez, sin utilizar espacio de almacenamiento para la traducción de las subexpresiones. Cuando se crea la salida en forma incremental de este modo, el orden en el que se imprimen los caracteres es importante.

La implementación de un esquema de traducción debe asegurar que las acciones semánticas se realicen en el orden en el que aparecerían durante un recorrido postorden de un árbol de análisis sintáctico. La implementación en realidad no necesita construir un árbol de análisis sintáctico (a menudo no lo hace), siempre y cuando asegure que las acciones semánticas se realizan como si construyéramos un árbol de análisis sintáctico y después ejecutáramos las acciones durante un recorrido postorden.

2.3.6 Ejercicios para la sección 2.3

Ejercicio 2.3.1: Construya un esquema de traducción orientado a la sintaxis, que traduzca expresiones aritméticas de la notación infija a la notación prefija, en la cual un operador aparece antes de sus operandos; por ejemplo, $-xy$ es la notación prefija para $x - y$. Proporcione los árboles de análisis sintáctico anotados para las entradas 9-5+2 y 9-5*2.

Ejercicio 2.3.2: Construya un esquema de traducción orientado a la sintaxis, que traduzca expresiones aritméticas de la notación postfija a la notación infija. Proporcione los árboles de análisis sintáctico anotados para las entradas 95-2* y 952*-.

Ejercicio 2.3.3: Construya un esquema de traducción orientado a la sintaxis, que traduzca enteros a números romanos.

Ejercicio 2.3.4: Construya un esquema de traducción orientado a la sintaxis, que traduzca números romanos a enteros.

Ejercicio 2.3.5: Construya un esquema de traducción orientado a la sintaxis, que traduzca expresiones aritméticas postfijo a sus expresiones aritméticas infijas equivalentes.

2.4 Análisis sintáctico

El análisis sintáctico (parsing) es el proceso de determinar cómo puede generarse una cadena de terminales mediante una gramática. Al hablar sobre este problema, es más útil pensar en que se va a construir un árbol de análisis sintáctico, aun cuando un compilador tal vez no lo construya en la práctica. No obstante, un analizador sintáctico debe ser capaz de construir el árbol en principio, o de lo contrario no se puede garantizar que la traducción sea correcta.

Esta sección introduce un método de análisis sintáctico conocido como “descenso recursivo”, el cual puede usarse tanto para el análisis sintáctico, como para la implementación de traductores orientados a la sintaxis. En la siguiente sección aparece un programa completo en Java, que implementa el esquema de traducción de la figura 2.15. Una alternativa viable es usar una herramienta de software para generar un traductor directamente de un esquema de traducción. La sección 4.9 describe dicha herramienta: Yacc; puede implementar el esquema de traducción de la figura 2.15 sin necesidad de modificarlo.

Para cualquier gramática libre de contexto, hay un analizador sintáctico que se tarda, como máximo, un tiempo $O(n^3)$ en analizar una cadena de n terminales. Pero, por lo general, el tiempo cúbico es demasiado costoso. Por fortuna, para los lenguajes de programación reales podemos diseñar una gramática que pueda analizarse con rapidez. Los algoritmos de tiempo lineal bastan para analizar en esencia todos los lenguajes que surgen en la práctica. Los analizadores sintácticos de los lenguajes de programación casi siempre realizan un escaneo de izquierda a derecha sobre la entrada, buscando por adelantado un terminal a la vez, y construyendo las piezas del árbol de análisis sintáctico a medida que avanzan.

La mayoría de los métodos de análisis sintáctico se adaptan a una de dos clases, llamadas métodos *descendente* y *ascendente*. Estos términos se refieren al orden en el que se construyen los nodos en el árbol de análisis sintáctico. En los analizadores tipo descendente, la construcción empieza en la raíz y procede hacia las hojas, mientras que en los analizadores tipo ascendente, la construcción empieza en las hojas y procede hacia la raíz. La popularidad de los analizadores tipo arriba-abajo se debe a que pueden construirse analizadores eficientes con más facilidad a mano, mediante métodos descendentes. No obstante, el análisis sintáctico tipo ascendente puede manejar una clase más extensa de gramáticas y esquemas de traducción, por lo que las herramientas de software para generar analizadores sintácticos directamente a partir de las gramáticas utilizan con frecuencia éste método.

2.4.1 Análisis sintáctico tipo arriba-abajo

Para introducir el análisis sintáctico tipo arriba-abajo, consideremos una gramática que se adapta bien a esta clase de métodos. Más adelante en esta sección, consideraremos la construcción de los analizadores sintácticos descendentes en general. La gramática en la figura 2.16 genera un subconjunto de las instrucciones de C o Java. Utilizamos los terminales en negrita **if** y **for** para las palabras clave “if” y “for”, respectivamente, para enfatizar que estas secuencias de caracteres se tratan como unidades, es decir, como símbolos terminales individuales. Además, el terminal **expr** representa expresiones; una gramática más completa utilizaría un no terminal *expr* y tendría producciones para el no terminal *expr*. De manera similar, **otras** es un terminal que representa a otras construcciones de instrucciones.

La construcción descendente de un árbol de análisis sintáctico como el de la figura 2.17 se realiza empezando con la raíz, etiquetada con el no terminal inicial *instr*, y realizando en forma repetida los siguientes dos pasos:

1. En el nodo N , etiquetado con el no terminal A , se selecciona una de las producciones para A y se construyen hijos en N para los símbolos en el cuerpo de la producción.
2. Se encuentra el siguiente nodo en el que se va a construir un subárbol, por lo general, el no terminal no expandido de más a la izquierda del árbol.

$$\begin{array}{lcl}
 instr & \rightarrow & \mathbf{expr} ; \\
 & | & \mathbf{if} \ (\mathbf{expr}) \ instr \\
 & | & \mathbf{for} \ (\mathit{expopc} ; \mathit{expopc} ; \mathit{expopc}) \ instr \\
 & | & \mathbf{otras} \\
 \\
 \mathit{expopc} & \rightarrow & \epsilon \\
 & | & \mathbf{expr}
 \end{array}$$

Figura 2.16: Una gramática para algunas instrucciones en C y Java

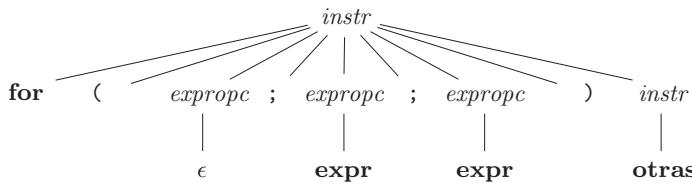


Figura 2.17: Un árbol de análisis sintáctico, de acuerdo con la gramática en la figura 2.16

Para algunas gramáticas, los pasos anteriores pueden implementarse durante un solo escaneo de izquierda a derecha de la cadena de entrada. El terminal actual escaneo en la entrada se conoce con frecuencia como el símbolo de *preanálisis*. En un principio, el símbolo preanálisis es el primer (es decir, el de más a la izquierda) terminal de la cadena de entrada. La figura 2.18 ilustra la construcción del árbol de análisis sintáctico en la figura 2.17 para la siguiente cadena de entrada:

for (; expr ; expr) otras

En un principio, el terminal **for** es el símbolo de preanálisis, y la parte conocida del árbol de análisis sintáctico consiste en la raíz, etiquetada con el no terminal inicial *instr* en la figura 2.18(a). El objetivo es construir el resto del árbol de análisis sintáctico de tal forma que la cadena generada por el árbol de análisis sintáctico coincida con la cadena de entrada.

Para que ocurra una coincidencia, el no terminal *instr* en la figura 2.18(a) debe derivar una cadena que empiece con el símbolo de preanálisis **for**. En la gramática de la figura 2.16, sólo hay una producción para *instr* que puede derivar dicha cadena, por lo que la seleccionamos y construimos los hijos de la raíz, etiquetados con los símbolos en el cuerpo de la producción. Esta expansión del árbol de análisis sintáctico se muestra en la figura 2.18(b).

Cada una de las tres instantáneas en la figura 2.18 tiene flechas que marcan el símbolo de preanálisis en la entrada y el nodo en el árbol de análisis sintáctico que se está considerando. Una vez que se construyen los hijos en un nodo, a continuación consideraremos el hijo de más a la izquierda. En la figura 2.18(b), los hijos acaban de construirse en la raíz, y se está considerando el hijo de más a la izquierda, etiquetado con **for**.

Cuando el nodo que se está considerando en el árbol de análisis sintáctico es para un terminal, y éste coincide con el símbolo de preanálisis, entonces avanzamos tanto en el árbol de análisis sintáctico como en la entrada. El siguiente terminal en la entrada se convierte en el

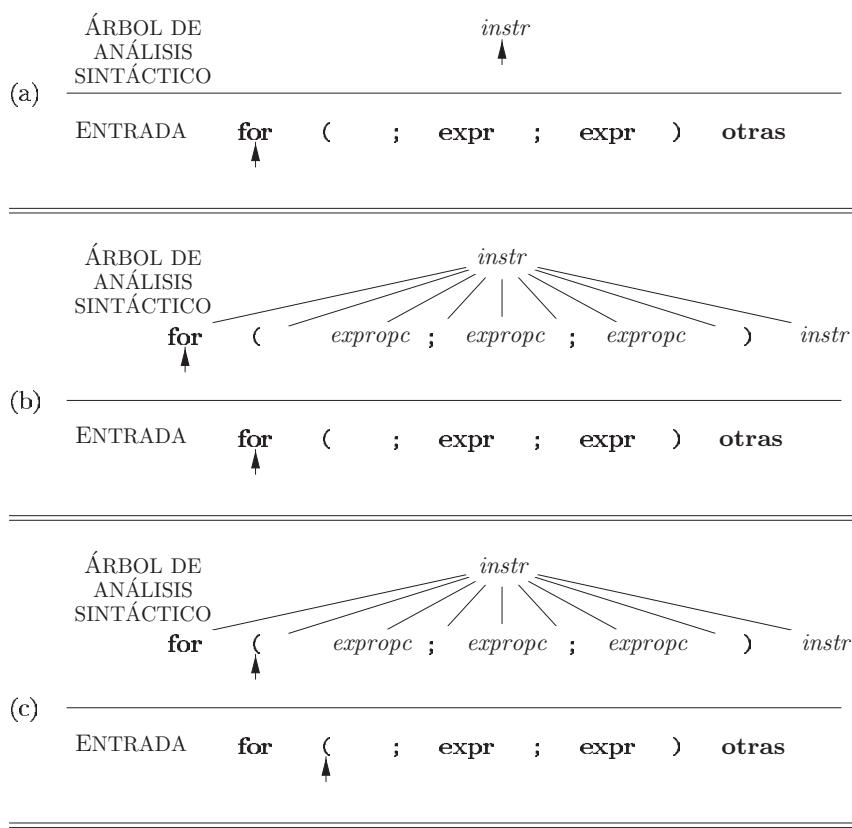


Figura 2.18: Análisis sintáctico tipo descendente mientras se explora la entrada de izquierda a derecha

nuevo símbolo de preanálisis, y se considera el siguiente hijo en el árbol de análisis sintáctico. En la figura 2.18(c), la flecha en el árbol de análisis sintáctico ha avanzado al siguiente hijo de la raíz y la flecha en la entrada ha avanzado al siguiente terminal, que es `(`. Un avance más llevará la flecha en el árbol de análisis sintáctico al hijo etiquetado con el no terminal `exp`, y llevará la flecha en la entrada al terminal `;`.

En el nodo no terminal etiquetado como `exp`, repetimos el proceso de seleccionar una producción para un no terminal. Las producciones con ϵ como el cuerpo (“producciones ϵ ”) requieren un tratamiento especial. Por el momento, las utilizaremos como una opción predeterminada cuando no puedan usarse otras producciones; regresaremos a ellas en la sección 2.4.3. Con el no terminal `exp` y el símbolo de preanálisis `;`, se utiliza la producción ϵ , ya que `;` no coincide con la otra única producción para `exp`, que tiene la terminal `expr` como cuerpo.

En general, la selección de una producción para un no terminal puede requerir del proceso de prueba y error; es decir, tal vez tengamos que probar una producción y retroceder para probar otra, si la primera no es adecuada. Una producción es inadecuada si, después de usarla,

no podemos completar el árbol para que coincida con la cadena de entrada. Sin embargo, no es necesario retroceder en un caso especial importante conocido como análisis sintáctico predictivo, el cual veremos a continuación.

2.4.2 Análisis sintáctico predictivo

El *análisis sintáctico de descenso recursivo* es un método de análisis sintáctico descendente, en el cual se utiliza un conjunto de procedimientos recursivos para procesar la entrada. Un procedimiento se asocia con cada no terminal de una gramática. Aquí consideraremos una forma simple de análisis sintáctico de descenso recursivo, conocido como *análisis sintáctico predictivo*, en el cual el símbolo de preanálisis determina sin ambigüedad el flujo de control a través del cuerpo del procedimiento para cada no terminal. La secuencia de llamadas a procedimientos durante el análisis de una entrada define en forma implícita un árbol de análisis sintáctico para la entrada, y puede usarse para crear un árbol de análisis sintáctico explícito, si se desea.

El analizador sintáctico predictivo en la figura 2.19 consiste en procedimientos para las no terminales *instr* y *expropc* de la gramática en la figura 2.16, junto con un procedimiento adicional llamado *coincidir*, el cual se utiliza para simplificar el código para *instr* y *expropc*. El procedimiento *coincidir(t)* compara su argumento *t* con el símbolo de preanálisis y avanza al siguiente terminal de entrada si coinciden. Por ende, *coincidir* cambia el valor de la variable *preanálisis*, una variable global que contiene el terminal de entrada que se acaba de explorar.

El análisis sintáctico empieza con una llamada del procedimiento para el no terminal inicial, *instr*. Con la misma entrada que en la figura 2.18 preanálisis es en un principio el primer **for** terminal. El procedimiento *instr* ejecuta el código correspondiente a la siguiente producción:

$$instr \rightarrow \mathbf{for} (\ expropc ; \ expropc ; \ expropc) \ instr$$

En el código para el cuerpo de la producción (es decir, el caso **for** del procedimiento *instr*), cada terminal se relaciona con el símbolo de preanálisis, y cada no terminal conduce a una llamada de su procedimiento, en la siguiente secuencia de llamadas:

```
coincidir(for); coincidir('(');
expropc(); coincidir(''); expropc(); coincidir(''); expropc();
coincidir(')'); instr();
```

El análisis sintáctico predictivo se basa en información acerca de los primeros símbolos que pueden generarse mediante el cuerpo de una producción. Dicho en forma más precisa, vamos a suponer que α es una cadena de símbolos de la gramática (terminales y no terminales). Definimos $\text{PRIMERO}(\alpha)$ como el conjunto de terminales que aparecen como los primeros símbolos de una o más cadenas de terminales generadas a partir de α . Si α es ϵ o puede generar a ϵ , entonces α también está en $\text{PRIMERO}(\alpha)$.

Los detalles de cómo se calcula $\text{PRIMERO}(\alpha)$ se encuentran en la sección 4.4.2. Aquí vamos a usar el razonamiento “ad hoc” para deducir los símbolos en $\text{PRIMERO}(\alpha)$; por lo general, α empieza con un terminal, que por ende es el único símbolo en $\text{PRIMERO}(\alpha)$, o empieza con un no terminal cuyos cuerpos de producciones empiezan con terminales, en cuyo caso estos terminales son los únicos miembros de $\text{PRIMERO}(\alpha)$.

Por ejemplo, con respecto a la gramática de la figura 2.16, los siguientes son cálculos correctos de PRIMERO .

```

void instr() {
    switch ( preanálisis ) {
        case expr:
            coincidir(expr); coincidir(''); break;
        case if:
            coincidir(if); coincidir('('); coincidir(expr); coincidir(')'); instr();
            break;
        case for:
            coincidir(for); coincidir('(');
            expopc(); coincidir(''); expopc(); coincidir(''); expopc();
            coincidir(')'); instr(); break;
        case otras:
            coincidir(otras); break;
        default:
            reportar("error de sintaxis");
    }
}

void expopc() {
    if ( preanálisis == expr ) coincidir(expr);
}

void coincidir(terminal t) {
    if ( preanálisis == t ) preanálisis = siguienteTerminal;
    else reportar("error de sintaxis");
}

```

Figura 2.19: Seudocódigo para un analizador sintáctico predictivo

$$\begin{aligned}
 \text{PRIMERO}(\textit{instr}) &= \{\textbf{expr}, \textbf{if}, \textbf{for}, \textbf{otras}\} \\
 \text{PRIMERO}(\textbf{expr} ;) &= \{\textbf{expr}\}
 \end{aligned}$$

Los PRIMEROS conjuntos deben considerarse si hay dos producciones $A \rightarrow \alpha$ y $A \rightarrow \beta$. Si ignoramos las producciones ϵ por el momento, el análisis sintáctico predictivo requiere que $\text{PRIMERO}(\alpha)$ y $\text{PRIMERO}(\beta)$ estén separados. Así, puede usarse el símbolo de preanálisis para decidir qué producción utilizar; si el símbolo de preanálisis está en $\text{PRIMERO}(\alpha)$, se utiliza α . En caso contrario, si el símbolo de preanálisis está en $\text{PRIMERO}(\beta)$, se utiliza β .

2.4.3 Cuándo usar las producciones ϵ

Nuestro analizador sintáctico predictivo utiliza una producción ϵ como valor predeterminado si no se puede utilizar otra producción. Con la entrada de la figura 2.18, después de relacionar los terminales **for** y **(**, el símbolo de preanálisis es **;**. En este punto se hace una llamada a *expopc*, y se ejecuta el código

```
if ( preanálisis == expr ) coincidir(expr);
```

en su cuerpo. El no terminal *expopc* tiene dos producciones, con los cuerpos **expr** y ϵ . El símbolo de preanálisis “;” no coincide con el terminal **expr**, por lo que no se puede aplicar la producción con el cuerpo **expr**. De hecho, el procedimiento regresa sin cambiar el símbolo de preanálisis o hacer cualquier otra cosa. Hacer nada corresponde a aplicar una producción ϵ .

En forma más general, considere una variante de las producciones en la figura 2.16, en donde *expopc* genera un no terminal de una expresión en vez del terminal **expr**:

$$\begin{array}{rcl} \textit{expopc} & \rightarrow & \textit{expr} \\ & | & \epsilon \end{array}$$

Por ende, *expopc* genera una expresión usando el no terminal *expr* o genera ϵ . Mientras se analiza *expopc*, si el símbolo de preanálisis no se encuentra en $\text{PRIMERO}(\textit{expr})$, entonces se utiliza la producción ϵ .

Para obtener más información sobre cuándo usar las producciones ϵ , vea la explicación de las gramáticas LL(1) en la sección 4.4.3.

2.4.4 Diseño de un analizador sintáctico predictivo

Podemos generalizar la técnica introducida de manera informal en la sección 2.4.2 para aplicarla a cualquier gramática que tenga un conjunto PRIMERO separado para los cuerpos de las producciones que pertenezcan a cualquier no terminal. También veremos que al tener un esquema de traducción (es decir, una gramática con acciones embebidas) es posible ejecutar esas acciones como parte de los procedimientos diseñados para el analizador sintáctico.

Recuerde que un *analizador sintáctico predictivo* es un programa que consiste en un procedimiento para cada no terminal. El procedimiento para el no terminal *A* realiza dos cosas:

1. Decide qué producción *A* debe utilizar, examinando el símbolo de preanálisis. La producción con el cuerpo α (en donde α no es ϵ , la cadena vacía) se utiliza si el símbolo de preanálisis está en $\text{PRIMERO}(\alpha)$. Si hay un conflicto entre dos cuerpos no vacíos por cualquier símbolo de preanálisis, entonces no podemos usar este método de análisis sintáctico en esta gramática. Además, la producción ϵ para *A*, si existe, se utiliza si el símbolo de preanálisis no se encuentra en el conjunto PRIMERO para cualquier otro cuerpo de producción para *A*.
2. Después, el procedimiento imita el cuerpo de la producción elegida. Es decir, los símbolos del cuerpo se “ejecutan” en turno, empezando desde la izquierda. Un no terminal se “ejecuta” mediante una llamada al procedimiento para ese no terminal, y un terminal que coincide con el símbolo de preanálisis se “ejecuta” leyendo el siguiente símbolo de entrada. Si en cualquier punto el terminal en el cuerpo no coincide con el símbolo de preanálisis, se reporta un error de sintaxis.

La figura 2.19 es el resultado de aplicar estas reglas a la gramática de figura 2.16.

Así como un esquema de traducción se forma extendiendo una gramática, un traductor orientado a la sintaxis puede formarse mediante la extensión de un analizador sintáctico predictivo. En la sección 5.4 se proporciona un algoritmo para este fin. La siguiente construcción limitada es suficiente por ahora:

1. Se construye un analizador sintáctico predictivo, ignorando las acciones en las producciones.
2. Se copian las acciones del esquema de traducción al analizador sintáctico. Si una acción aparece después del símbolo de la gramática X en la producción p , entonces se copia después de la implementación de X en el código para p . En caso contrario, si aparece al principio de la producción, entonces se copia justo antes del código para el cuerpo de la producción.

En la sección 2.5 construiremos un traductor de este tipo.

2.4.5 Recursividad a la izquierda

Es posible que un analizador sintáctico de descenso recursivo entre en un ciclo infinito. Se produce un problema con las producciones “recursivas por la izquierda” como

$$\text{expr} \rightarrow \text{expr} + \text{term}$$

en donde el símbolo más a la izquierda del cuerpo es el mismo que el no terminal en el encabezado de la producción. Suponga que el procedimiento para expr decide aplicar esta producción. El cuerpo empieza con expr , de manera que el procedimiento para expr se llama en forma recursiva. Como el símbolo de preanálisis cambia sólo cuando coincide un terminal en el cuerpo, no se realizan cambios en la entrada entre las llamadas recursivas de expr . Como resultado, la segunda llamada a expr hace exactamente lo que hizo la primera llamada, lo cual significa que se realiza una tercera llamada a expr y así sucesivamente, por siempre.

Se puede eliminar una producción recursiva por la izquierda, reescribiendo la producción problemática. Considere un terminal A con dos producciones:

$$A \rightarrow A\alpha \mid \beta$$

en donde α y β son secuencias de terminales y no terminales que no empiezan con A . Por ejemplo, en

$$\text{expr} \rightarrow \text{expr} + \text{term} \mid \text{term}$$

del no terminal $A = \text{expr}$, la cadena $\alpha = + \text{term}$, y la cadena $\beta = \text{term}$.

Se dice que el no terminal A y su producción son *recursivas por la izquierda*, ya que la producción $A \rightarrow A\alpha$ tiene a la misma A como el símbolo de más a la izquierda en el lado derecho.⁴ La aplicación repetida de esta producción genera una secuencia de α 's a la derecha de A , como en la figura 2.20(a). Cuando por fin A se sustituye por β , tenemos una β seguida por una secuencia de cero o más α 's.

⁴En una gramática recursiva a la izquierda general, en vez de la producción $A \rightarrow A\alpha$, el no terminal A puede derivar en $A\alpha$ a través de producciones intermedias.

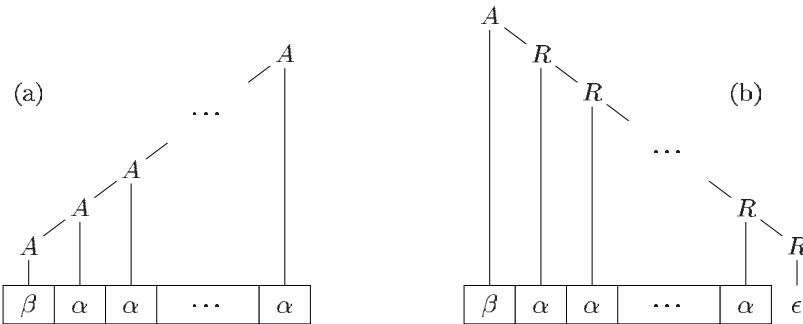


Figura 2.20: Formas recursivas por la izquierda y por la derecha para generar una cadena

Podemos lograr el mismo efecto que en la figura 2.20(b) si reescribimos las producciones para A de la siguiente manera, usando un nuevo no terminal R :

$$\begin{array}{lcl} A & \rightarrow & \beta R \\ R & \rightarrow & \alpha R \mid \epsilon \end{array}$$

El no terminal R y su producción $R \rightarrow \alpha R$ son *recursivos por la derecha*, ya que esta producción para R tiene al mismo R como el último símbolo del lado derecho. Las producciones recursivas por la derecha conducen a árboles que crecen hacia abajo y a la derecha, como en la figura 2.20(b). Los árboles que crecen hacia abajo y a la derecha dificultan más la traducción de expresiones que contienen operadores asociativos por la izquierda, como el signo menos resta. No obstante, en la sección 2.5.2 veremos que de todas formas puede obtenerse la traducción apropiada de expresiones a la notación postfijo mediante un diseño cuidadoso del esquema de traducción.

En la sección 4.3.3 consideraremos formas más generales de recursividad por la izquierda, y mostraremos cómo puede eliminarse toda la recursividad por la izquierda de una gramática.

2.4.6 Ejercicios para la sección 2.4

Ejercicio 2.4.1: Construya analizadores sintácticos de descenso recursivo, empezando con las siguientes gramáticas:

a) $S \rightarrow + S S \mid - S S \mid a$

b) $S \rightarrow S (S) S \mid \epsilon$

c) $S \rightarrow 0 S 1 \mid 0 1$

2.5 Un traductor para las expresiones simples

Ahora vamos a construir un traductor orientado a la sintaxis usando las técnicas de las últimas tres secciones, en forma de un programa funcional en Java, que traduce expresiones aritméticas a la forma postfija. Para mantener el programa inicial lo bastante pequeño como para poder

administrarlo, empezaremos con expresiones que consisten en dígitos separados por los signos binarios de suma y resta. En la sección 2.6 extenderemos el programa para traducir expresiones que incluyan números y otros operadores. Es conveniente estudiar la traducción de expresiones con detalle, ya que aparecen como un constructor en muchos lenguajes.

A menudo, un esquema de traducción orientado a la sintaxis sirve como la especificación para un traductor. El esquema en la figura 2.21 (repetido de la figura 2.15) define la traducción que se va a realizar aquí.

$expr$	\rightarrow	$expr + term$	$\{ \text{print}('+' \})$
		$expr - term$	$\{ \text{print}(' - ' \})$
		$term$	
$term$	\rightarrow	0	$\{ \text{print}('0' \})$
		1	$\{ \text{print}('1' \})$
		...	
		9	$\{ \text{print}('9' \})$

Figura 2.21: Acciones para traducir a la notación postfija

A menudo, la gramática subyacente de un esquema dado tiene que modificarse antes de poder analizarlo con un analizador sintáctico predictivo. En especial, la gramática subyacente del esquema de la figura 2.21 es recursiva por la izquierda, y como vimos en la última sección, un analizador sintáctico predictivo no puede manejar una gramática recursiva por la izquierda.

Parece que tenemos un conflicto: por una parte, necesitamos una gramática que facilite la traducción, por otra parte necesitamos una gramática considerablemente distinta que facilite el análisis sintáctico. La solución es empezar con la gramática para facilitar la traducción y transformarla con cuidado para facilitar el análisis sintáctico. Al eliminar la recursividad por la izquierda en la figura 2.21, podemos obtener una gramática adecuada para usarla en un traductor predictivo de descenso recursivo.

2.5.1 Sintaxis abstracta y concreta

Un punto inicial útil para diseñar un traductor es una estructura de datos llamada árbol sintáctico abstracto. En un árbol sintáctico abstracto para una expresión, cada nodo interior representa a un operador; los hijos del nodo representan a los operandos del operador. De manera más general, cualquier construcción de programación puede manejarse al formar un operador para la construcción y tratar como operandos a los componentes con significado semántico de esa construcción.

En el árbol sintáctico abstracto para $9-5+2$ en la figura 2.22, la raíz representa al operador $+$. Los subárboles de la raíz representan a las subexpresiones $9-5$ y 2 . El agrupamiento de $9-5$ como un operando refleja la evaluación de izquierda a derecha de los operadores en el mismo nivel de precedencia. Como $-$ y $+$ tienen la misma precedencia, $9-5+2$ es equivalente a $(9-5)+2$.

Los árboles sintácticos abstractos, conocidos simplemente como *árboles sintácticos*, se parecen en cierto grado a los árboles de análisis sintáctico. No obstante, en el árbol sintáctico los nodos interiores representan construcciones de programación mientras que, en el árbol de

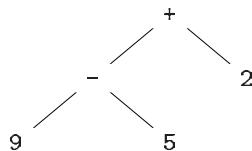


Figura 2.22: Árbol sintáctico para 9-5+2

análisis sintáctico, los nodos interiores representan no terminales. Muchos no terminales de una gramática representan construcciones de programación, pero otros son “ayudantes” de un tipo distinto, como los que representan términos, factores o demás variaciones de expresiones. En el árbol sintáctico, estos ayudantes, por lo general, no son necesarios y por ende se descartan. Para enfatizar el contraste, a un árbol de análisis sintáctico se le conoce algunas veces como *árbol sintáctico concreto*, y a la gramática subyacente se le llama *sintaxis completa* para el lenguaje.

En el árbol sintáctico de la figura 2.22, cada nodo interior se asocia con un operador, sin nodos “ayudantes” para *producciones individuales* (una producción cuyo cuerpo consiste en un no terminal individual, y nada más) como $expr \rightarrow term$ o para producciones ϵ como $resto \rightarrow \epsilon$.

Es conveniente que un esquema de traducción se base en una gramática cuyos árboles de análisis sintáctico estén lo más cerca de los árboles sintácticos que sea posible. El agrupamiento de subexpresiones mediante la gramática en la figura 2.21 es similar a su agrupamiento en los árboles sintácticos. Por ejemplo, las subexpresiones para el operador de suma se proporcionan mediante $expr$ y $term$ en el cuerpo de la producción $expr + term$.

2.5.2 Adaptación del esquema de traducción

La técnica de eliminación de recursividad por la izquierda trazada en la figura 2.20 también se puede aplicar a las producciones que contengan acciones semánticas. En primer lugar, la técnica se extiende a varias producciones para A . En nuestro ejemplo, A es $expr$ y hay dos producciones recursivas a la izquierda para $expr$, y una que no es recursiva. La técnica transforma las producciones $A \rightarrow A\alpha \mid A\beta \mid \gamma$ en lo siguiente:

$$\begin{aligned} A &\rightarrow \gamma R \\ R &\rightarrow \alpha R \mid \beta R \mid \epsilon \end{aligned}$$

En segundo lugar, debemos transformar producciones que hayan incrustado acciones, no sólo terminales y no terminales. Las acciones semánticas incrustadas en las producciones simplemente se transportan en la transformación, como si fueran terminales.

Ejemplo 2.13: Considere el esquema de traducción de la figura 2.21. Suponga que:

$$\begin{aligned} A &= expr \\ \alpha &= + term \{ \text{print('+')}\} \\ \beta &= - term \{ \text{print('')} \} \\ \gamma &= term \end{aligned}$$

Entonces, la transformación con eliminación de recursividad por la izquierda produce el esquema de traducción de la figura 2.23. Las producciones *expr* en la figura 2.21 se han transformado en las producciones para *expr*, y un nuevo no terminal *resto* desempeña el papel de *R*. Las producciones para *term* se repiten de la figura 2.21. La figura 2.24 muestra cómo se traduce 9-5+2, mediante la gramática de la figura 2.23. \square

$$\begin{array}{lcl}
 \textit{expr} & \rightarrow & \textit{term resto} \\
 \textit{resto} & \rightarrow & + \textit{term} \{ \text{print('+'}) \} \textit{resto} \\
 & | & - \textit{term} \{ \text{print('-'}) \} \textit{resto} \\
 & | & \epsilon \\
 \textit{term} & \rightarrow & 0 \{ \text{print('0')} \} \\
 & | & 1 \{ \text{print('1')} \} \\
 & | & \dots \\
 & | & 9 \{ \text{print('9')} \}
 \end{array}$$

Figura 2.23: Esquema de traducción, después eliminar la recursividad por la izquierda

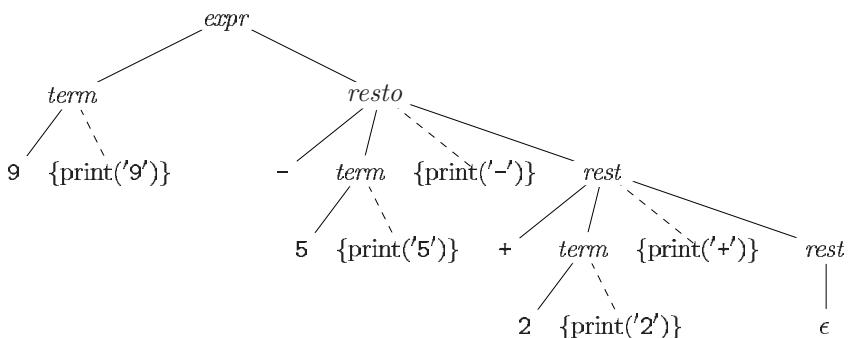


Figura 2.24: Traducción de 9-5+2 a 95-2+

La eliminación de la recursividad por la izquierda debe realizarse con cuidado, para asegurar que se preserve el orden de las acciones semánticas. Por ejemplo, el esquema transformado en la figura 2.23 tiene las acciones $\{ \text{print('+'}) \}$ y $\{ \text{print('-'}) \}$ en medio del cuerpo de una producción, en cada caso entre los no terminales *term* y *resto*. Si las acciones se desplazaran al final, después de *resto*, entonces las traducciones serían incorrectas. Dejamos como ejercicio al lector demostrar que 9-5+2 se traduciría entonces en forma incorrecta a 952+-, la notación postfija para 9-(5+2), en vez del término 95-2+ deseado, la notación postfija para (9-5)+2.

2.5.3 Procedimientos para los no terminales

Las funciones *expr*, *resto* y *term* en la figura 2.25 implementan el esquema de traducción orientada a la sintaxis de la figura 2.23. Estas funciones imitan los cuerpos de las producciones de los no terminales correspondientes. La función *expr* implementa la producción $expr \rightarrow term resto$, mediante la llamada a *term()* seguida de *resto()*.

```

void expr() {
    term(); resto();
}

void resto() {
    if ( preanálisis == '+' ) {
        coincidir('+'); term(); print('+'); resto();
    }
    else if ( preanálisis == '-' ) {
        coincidir('-'); term(); print('-'); resto();
    }
    else { } /* no hace nada con la entrada */ ;
}

void term() {
    if ( preanálisis es un dígito ) {
        t = preanálisis; coincidir(preanálisis); print(t);
    }
    else reportar("error de sintaxis");
}

```

Figura 2.25: Seudocódigo para los no terminales *expr*, *resto* y *term*

La función *resto* implementa las tres producciones para el no terminal *resto* en la figura 2.23. Aplica la primera producción si el símbolo de preanálisis es un signo positivo, la segunda producción si el símbolo de preanálisis es negativo, y la producción $resto \rightarrow \epsilon$ en todos los demás casos. Las primeras dos producciones para *resto* se implementan mediante las primeras dos bifurcaciones de la instrucción **if** en el procedimiento *resto*. Si el símbolo de preanálisis es +, el signo positivo se relaciona mediante la llamada *coincidir('+')*. Después de la llamada *term()*, la acción semántica se implementa escribiendo un carácter de signo positivo. La segunda producción es similar, con - en vez de +. Como la tercera producción para *resto* tiene a ϵ del lado derecho, la última cláusula **else** en la función *resto* no hace nada.

Las diez producciones para *term* generan los diez dígitos. Como cada una de estas producciones genera un dígito y lo imprime, el mismo código en la figura 2.25 las implementa todas. Si la prueba tiene éxito, la variable *t* guarda el dígito representado por *preanálisis*, para poder

escribirlo después de la llamada a *coincidir*. Observe que *coincidir* cambia el símbolo de preanálisis, por lo que el dígito necesita guardarse para imprimirllo después.⁵

2.5.4 Simplificación del traductor

Antes de mostrar un programa completo, haremos dos transformaciones de simplificación al código de la figura 2.25. Las simplificaciones incorporarán el procedimiento *resto* al procedimiento *expr*. Cuando se traducen expresiones con varios niveles de precedencia, dichas simplificaciones reducen el número de procedimientos necesarios.

En primer lugar, ciertas llamadas recursivas pueden sustituirse por iteraciones. Cuando la última instrucción que se ejecuta en el cuerpo de un procedimiento es una llamada recursiva al mismo procedimiento, se dice que la llamada es *recursiva por la cola*. Por ejemplo, en la función *resto*, las llamadas de *resto()* con los símbolos de preanálisis + y - son recursivas por la cola, ya que en cada una de estas bifurcaciones, la llamada recursiva a *resto* es la última instrucción ejecutada por esa llamada de *resto*.

Para un procedimiento sin parámetros, una llamada recursiva por la cola puede sustituirse simplemente por un salto al inicio del procedimiento. El código para *resto* puede reescribirse como el seudocódigo de la figura 2.26. Siempre y cuando el símbolo de preanálisis sea un signo positivo o negativo, el procedimiento *resto* relaciona el signo, llama a *term* para relacionar un dígito y continúa el proceso. En caso contrario, sale del ciclo while y regresa de *resto*.

```
void resto() {
    while( true ) {
        if ( preanálisis == '+' ) {
            coincidir('+'); term(); print('+'); continue;
        }
        else if preanálisis == '-' ) {
            coincidir('-'); term(); print('-'); continue;
        }
        break ;
    }
}
```

Figura 2.26: Eliminación de la recursividad por la cola en el procedimiento *resto* de la figura 2.25

En segundo lugar, el programa Java completo incluirá un cambio más. Una vez que las llamadas recursivas por la cola a *resto* en la figura 2.25 se sustituyan por iteraciones, la única llamada restante a *resto* es desde el interior del procedimiento *expr*. Por lo tanto, los dos procedimientos pueden integrarse en uno, sustituyendo la llamada *resto()* por el cuerpo del procedimiento *resto*.

⁵Como optimización menor, podríamos imprimir antes de llamar a *coincidir* para evitar la necesidad de guardar el dígito. En general, es riesgoso cambiar el orden de las acciones y los símbolos de la gramática, ya que podríamos modificar lo que hace la traducción.

2.5.5 El programa completo

En la figura 2.27 aparece el programa Java completo para nuestro traductor. La primera línea de la figura 2.27, empezando con `import`, proporciona acceso al paquete `java.io` para la entrada y salida del sistema. El resto del código consiste en las dos clases `Analizador` y `Postfijo`. La clase `Analizador` contiene la variable `preanálisis` y las funciones `Analizador`, `expr`, `term` y `coincidir`.

La ejecución empieza con la función `main`, que se define en la clase `Postfijo`. La función `main` crea una instancia `analizar` de la clase `Analizador` y llama a su función `expr` para analizar una expresión.

La función `Analizador`, que tiene el mismo nombre que su clase, es un *constructor*; se llama de manera automática cuando se crea un objeto de la clase. De la definición al principio de la clase `Analizador`, podemos ver que el constructor `Analizador` inicializa la variable `preanálisis` leyendo un token. Los tokens, que consisten en caracteres individuales, los suministra la rutina de entrada del sistema `read`, la cual lee el siguiente carácter del archivo de entrada. Observe que `preanálisis` se declara como entero, en vez de carácter, para anticipar el hecho de que en secciones posteriores introduciremos tokens adicionales además de los caracteres individuales.

La función `expr` es el resultado de la simplificación que vimos en la sección 2.5.4; implementa a los no terminales `expr` y `rest` de la figura 2.23. El código para `expr` en la figura 2.27 llama a `term` y después tiene un ciclo `while` que evalúa infinitamente si `preanálisis` coincide con `'+'` o con `'-'`. El control sale de este ciclo `while` al llegar a la instrucción `return`. Dentro del ciclo, se utilizan las herramientas de entrada/salida de la clase `System` para escribir un carácter.

La función `term` utiliza la rutina `isDigit` de la clase `Character` de Java para probar si el símbolo de preanálisis es un dígito. La rutina `isDigit` espera ser aplicada sobre un carácter; no obstante, `preanálisis` se declara como entero, anticipando las extensiones a futuro. La construcción `(char) preanálisis` convierte `preanálisis` a un carácter. En una pequeña modificación de la figura 2.25, la acción semántica de escribir el carácter `preanálisis` ocurre antes de la llamada a `coincidir`.

La función `coincidir` comprueba los terminales; lee el siguiente terminal de la entrada si hay una coincidencia con el símbolo `preanálisis`, e indica un error en cualquier otro caso, mediante la ejecución de la siguiente instrucción:

```
throw new Error("error de sintaxis");
```

Este código crea una nueva excepción de la clase `Error` y proporciona la cadena `error de sintaxis` como mensaje de error. Java no requiere que las excepciones `Error` se declaren en una cláusula `throws`, ya que su uso está destinado sólo para los eventos anormales que nunca deberían ocurrir.⁶

⁶El manejo de errores puede simplificarse con las herramientas para manejo de excepciones de Java. Un método sería definir una nueva excepción, por decir `ErrorSintaxis`, que extiende a la clase `Exception` del sistema. Después, se lanza `ErrorSintaxis` en vez de `Error` al detectar un error, ya sea en `term` o en `coincidir`. Más adelante, se maneja la excepción en `main`, encerrando la llamada `analizar.expr()` dentro de una instrucción `try` que atrape a la excepción `ErrorSintaxis`, se escribe un mensaje y termina el programa. Tendríamos que agregar una clase `ErrorSintaxis` al programa en la figura 2.27. Para completar la extensión, además de `IOException`, las funciones `coincidir` y `term` ahora deben declarar que pueden lanzar a `ErrorSintaxis`. La función `expr`, que llama a estas dos funciones, también debe declarar que puede lanzar a `ErrorSintaxis`.

```

import java.io.*;
class Analizador {
    static int preanálisis;

    public Analizador() throws IOException {
        preanálisis = System.in.read();
    }

    void expr() throws IOException {
        term();
        while(true) {
            if( preanálisis == '+' ) {
                coincidir('+'); term(); System.out.write('+');
            }
            else if( preanálisis == '-' ) {
                coincidir('-'); term(); System.out.write('-');
            }
            else return;
        }
    }

    void term() throws IOException {
        if( Character.isDigit((char) preanálisis) ) {
            System.out.write((char) preanálisis); coincidir(preanálisis);
        }
        else throw new Error("error de sintaxis");
    }

    void coincidir(int t) throws IOException {
        if(preanálisis == t) preanálisis = System.in.read();
        else throw new Error("error de sintaxis");
    }
}

public class Postfijo {
    public static void main(String[] args) throws IOException {
        Analizador analizar = new Analizador();
        analizar.expr(); System.out.write('\n');
    }
}

```

Figura 2.27: Programa en Java para traducir expresiones infijas al formato postfijo

Unas cuantas características importantes sobre Java

Las siguientes notas pueden ser de utilidad para aquellos que no estén familiarizados con Java, a la hora de leer el código de la figura 2.27:

- Una clase en Java consiste en una secuencia de definiciones de variables y funciones.
- Es necesario utilizar paréntesis para encerrar las listas de parámetros de las funciones, aun cuando no haya parámetros; por ende, escribimos `expr()` y `term()`. Estas funciones son en realidad procedimientos, ya que no devuelven valores; esto se indica mediante la palabra clave `void` antes del nombre de la función.
- Las funciones se comunican enviando parámetros “por valor” o accediendo a los datos compartidos. Por ejemplo, las funciones `expr()` y `term()` examinan el símbolo de preanálisis usando la variable `preanálisis` de la clase, a la que todos pueden acceder, ya que todos pertenecen a la misma clase `Analizador`.
- Al igual que C, Java usa `=` para la asignación, `==` para la igualdad y `!=` para la desigualdad.
- La cláusula “`throws IOException`” en la definición de `term()` declara que se puede producir una excepción llamada `IOException`. Dicha excepción ocurre si no hay entrada que leer cuando la función `coincidir` usa la rutina `read`. Cualquier función que llame a `coincidir` debe también declarar que puede ocurrir una excepción `IOException` durante su propia ejecución.

2.6 Análisis léxico

Un analizador léxico lee los caracteres de la entrada y los agrupa en “objetos token”. Junto con un símbolo de terminal que se utiliza para las decisiones de análisis sintáctico, un objeto token lleva información adicional en forma de valores de atributos. Hasta ahora, no hemos tenido la necesidad de diferenciar entre los términos “token” y “terminal”, ya que el analizador sintáctico ignora los valores de los atributos que lleva un token. En esta sección, un token es un terminal junto con información adicional.

A una secuencia de caracteres de entrada que conforman un solo token se le conoce como *lexema*. Por ende, podemos decir que el analizador léxico aísla a un analizador sintáctico de la representación tipo lexema de los tokens.

El analizador léxico en esta sección permite que aparezcan números, identificadores y “espacio en blanco” (espacios, tabuladores y nuevas líneas) dentro de las expresiones. Puede usarse para extender el traductor de expresiones de la sección anterior. Como la gramática de expresiones de la figura 2.21 debe extenderse para permitir números e identificadores, apro-

vecharemos esta oportunidad para permitir también la multiplicación y la división. El esquema de traducción extendido aparece en la figura 2.28.

$expr$	\rightarrow	$expr + term$	$\{ \text{print}('+' \})$
		$expr - term$	$\{ \text{print}(' - ') \}$
		$term$	
$term$	\rightarrow	$term * factor$	$\{ \text{print}('*') \}$
		$term / factor$	$\{ \text{print}('/') \}$
		$factor$	
$factor$	\rightarrow	$(expr)$	
		num	$\{ \text{print}(num.valor) \}$
		id	$\{ \text{print}(id.lexema) \}$

Figura 2.28: Acciones para traducir a la notación postfijo

En la figura 2.28, se asume que el terminal **num** tiene un atributo **num.valor**, el cual nos da el valor entero correspondiente a esta ocurrencia de **num**. El terminal **id** tiene un atributo con valor de cadena escrito como **id.lexema**; asumimos que esta cadena es el lexema actual que comprende esta instancia del **id** del token.

Al final de esta sección, ensamblaremos en código Java los fragmentos de seudocódigo que se utilizan para ilustrar el funcionamiento de un analizador léxico. El método en esta sección es adecuado para los analizadores léxicos escritos a mano. La sección 3.5 describe una herramienta llamada Lex que genera un analizador léxico a partir de una especificación. En la sección 2.7 consideraremos las tablas de símbolos o las estructuras de datos para guardar información acerca de los identificadores.

2.6.1 Eliminación de espacio en blanco y comentarios

El traductor de expresiones en la sección 2.5 puede ver cada carácter en la entrada, por lo que los caracteres extraños, como los espacios en blanco, harán que falle. La mayoría de los lenguajes permiten que aparezcan cantidades arbitrarias de espacio en blanco entre los tokens. Los comentarios se ignoran de igual forma durante el análisis sintáctico, por lo que también pueden tratarse como espacio en blanco.

Si el analizador léxico elimina el espacio en blanco, el analizador sintáctico nunca tendrá que considerarlo. La alternativa de modificar la gramática para incorporar el espacio en blanco en la sintaxis no es tan fácil de implementar.

El seudocódigo en la figura 2.29 omite el espacio en blanco, leyendo los caracteres de entrada hasta llegar a un espacio en blanco, un tabulador o una nueva línea. La variable *vistazo* contiene el siguiente carácter de entrada. Los números de línea y el contexto son útiles dentro de los mensajes de error para ayudar a señalar los errores; el código usa la variable *línea* para contar los caracteres de nueva línea en la entrada.

```

for ( ; ; vistazo = siguiente carácter de entrada ) {
    if ( vistazo es un espacio en blanco o un tabulador ) no hacer nada;
    else if ( vistazo es una nueva línea ) línea = línea+1;
    else break;
}

```

Figura 2.29: Omisión del espacio en blanco

2.6.2 Lectura adelantada

Tal vez un analizador léxico tenga que leer algunos caracteres de preanálisis, antes de que pueda decidir sobre el token que va a devolver al analizador sintáctico. Por ejemplo, un analizador léxico para C o Java debe leer por adelantado después de ver el carácter `>`. Si el siguiente carácter es `=`, entonces `>` forma parte de la secuencia de caracteres `>=`, el lexema para el token del operador “mayor o igual que”. En caso contrario, `>` por sí solo forma el operador “mayor que”, y el analizador léxico ha leído un carácter de más.

Un método general para leer por adelantado en la entrada es mantener un búfer de entrada, a partir del cual el analizador léxico puede leer y devolver caracteres. Los búferes de entrada pueden justificarse tan sólo por cuestión de eficiencia, ya que por lo general es más eficiente obtener un bloque de caracteres que obtener un carácter a la vez. Un apuntador lleva el registro de la porción de la entrada que se ha analizado; para regresar un carácter se mueve el apuntador hacia atrás. En la sección 3.2 veremos las técnicas para el uso de búfer en la entrada.

Por lo general, basta con leer un carácter de preanálisis, así que una solución simple es utilizar una variable, por decir *vistazo*, para guardar el siguiente carácter de entrada. El analizador léxico en esta sección lee un carácter de preanálisis mientras recolecta dígitos para números, o caracteres para identificadores; por ejemplo, lee más allá del `1` para diferenciar entre `1` y `10`, y lee más allá de `t` para diferenciar entre `t` y `true`.

El analizador léxico lee de preanálisis sólo cuando debe hacerlo. Un operador como `*` puede identificarse sin necesidad de leer por adelantado. En tales casos, *vistazo* se establece a un espacio en blanco, que se omitirá cuando se llame al analizador léxico para buscar el siguiente token. La aserción invariante en esta sección es que, cuando el analizador léxico devuelve un token, la variable *vistazo* contiene el carácter que está más allá del lexema para el token actual, o contiene un espacio en blanco.

2.6.3 Constantes

Cada vez que aparece un solo dígito en una gramática para expresiones, parece razonable permitir una constante entera arbitraria en su lugar. Las constantes enteras pueden permitirse al crear un símbolo terminal, por decir **num**, para éstas, o al incorporar la sintaxis de constantes enteras en la gramática. El trabajo de recolectar caracteres en enteros y calcular su valor numérico colectivo se asigna por lo general a un analizador léxico, para que los números puedan tratarse como unidades individuales durante el análisis sintáctico y la traducción.

Cuando aparece una secuencia de dígitos en el flujo de entrada, el analizador léxico pasa al analizador sintáctico un token que consiste en la terminal **num**, junto con un atributo con

valor de entero, el cual se calcula a partir de los dígitos. Si escribimos tokens como n-uplas encerradas entre los signos $\langle \rangle$, la entrada $31 + 28 + 59$ se transforma en la siguiente secuencia:

$$\langle \mathbf{num}, 31 \rangle \langle + \rangle \langle \mathbf{num}, 28 \rangle \langle + \rangle \langle \mathbf{num}, 59 \rangle$$

Aquí, el símbolo terminal $+$ no tiene atributos, por lo que su n-upla es simplemente $\langle + \rangle$. El pseudocódigo en la figura 2.30 lee los dígitos en un entero y acumula el valor del entero, usando la variable v .

```

if ( vistazo contiene un dígito ) {
    v = 0;
    do {
        v = v * 10 + valor entero del dígito vistazo;
        vistazo = siguiente carácter de entrada;
    } while ( vistazo contiene un dígito );
    return token (num, v);
}

```

Figura 2.30: Agrupamiento de dígitos en enteros

2.6.4 Reconocimiento de palabras clave e identificadores

La mayoría de los lenguajes utilizan cadenas de caracteres fijas tales como **for**, **do** e **if**, como signos de puntuación o para identificar las construcciones. A dichas cadenas de caracteres se les conoce como *palabras clave*.

Las cadenas de caracteres también se utilizan como identificadores para nombrar variables, arreglos, funciones y demás. Las gramáticas tratan de manera rutinaria a los identificadores como terminales para simplificar el analizador sintáctico, el cual por consiguiente puede esperar el mismo terminal, por decir **id**, cada vez que aparece algún identificador en la entrada. Por ejemplo, en la siguiente entrada:

$$\text{cuenta} = \text{cuenta} + \text{incremento}; \quad (2.6)$$

el analizador trabaja con el flujo de terminales **id** = **id** + **id**. El token para **id** tiene un atributo que contiene el lexema. Si escribimos los tokens como n-uplas, podemos ver que las n-uplas para el flujo de entrada (2.6) son:

$$\langle \mathbf{id}, \text{"cuenta"} \rangle \langle = \rangle \langle \mathbf{id}, \text{"cuenta"} \rangle \langle + \rangle \langle \mathbf{id}, \text{"incremento"} \rangle \langle ; \rangle$$

Por lo general, las palabras clave cumplen con las reglas para formar identificadores, por lo que se requiere un mecanismo para decidir cuándo un lexema forma una palabra clave y cuándo un identificador. El problema es más fácil de resolver si las palabras clave son *reservadas*; es decir, si no pueden utilizarse como identificadores. Entonces, una cadena de caracteres forma a un identificador sólo si no es una palabra clave.

El analizador léxico en esta sección resuelve dos problemas, utilizando una tabla para guardar cadenas de caracteres:

- *Representación simple.* Una tabla de cadenas puede aislar al resto del compilador de la representación de las cadenas, ya que las fases del compilador pueden trabajar con referencias o apuntadores a la cadena en la tabla. Las referencias también pueden manipularse con más eficiencia que las mismas cadenas.
- *Palabras reservadas.* Las palabras reservadas pueden implementarse mediante la inicialización de la tabla de cadenas con las cadenas reservadas y sus tokens. Cuando el analizador léxico lee una cadena o lexema que podría formar un identificador, primero verifica si el lexema se encuentra en la tabla de cadenas. De ser así, devuelve el token de la tabla; en caso contrario, devuelve un token con el terminal **id**.

En Java, una tabla de cadenas puede implementarse como una hash table, usando una clase llamada *Hashtable*. La siguiente declaración:

```
Hashtable palabras = new Hashtable();
```

establece a *palabras* como una hash table predeterminada, que asigna claves a valores. La utilizaremos para asignar los lexemas a los tokens. El seudocódigo en la figura 2.31 utiliza la operación *get* para buscar palabras reservadas.

```
if ( vistazo contiene una letra ) {
    recolectar letras o dígitos en un búfer b;
    s = cadena formada a partir de los caracteres en b;
    w = token devuelto por palabras.get(s);
    if ( w no es null ) return w;
    else {
        Introducir el par clave-valor (s, {id, s}) en palabras
        return token {id, s};
    }
}
```

Figura 2.31: Distinción entre palabras clave e identificadores

Este seudocódigo recolecta de la entrada una cadena *s*, la cual consiste en letras y dígitos, empezando con una letra. Suponemos que *s* se hace lo más larga posible; es decir, el analizador léxico continuará leyendo de la entrada mientras siga encontrando letras y dígitos. Cuando encuentra algo distinto a una letra o dígito, por ejemplo, espacio en blanco, el lexema se copia a un búfer *b*. Si la tabla tiene una entrada para *s*, entonces se devuelve el token obtenido por *palabras.get*. Aquí, *s* podría ser una palabra clave, con la que se sembró inicializado la tabla *palabras*, o podría ser un identificador que se introdujo antes en la tabla. En caso contrario, el token **id** y el atributo *s* se instalan en la tabla y se devuelven.

2.6.5 Un analizador léxico

Hasta ahora en esta sección, los fragmentos de seudocódigo se juntan para formar una función llamada *escanear* que devuelve objetos token, de la siguiente manera:

```
Token escanear() {
    omitir espacio en blanco, como en la sección 2.6.1;
    manejar los números, como en la sección 2.6.3;
    manejar las palabras reservadas e identificadores, como en la sección 2.6.4;
    /* Si llegamos aquí, tratar el carácter de lectura de preanálisis vistazo como token */
    Token t = new Token(vistazo);
    vistazo = espacio en blanco /* inicialización, como vimos en la sección 2.6.2 */;
    return t;
}
```

El resto de esta sección implementa la función *explorar* como parte de un paquete de Java para el análisis léxico. El paquete, llamado **analizador lexico** tiene clases para tokens y una clase **AnalizadorLexico** que contiene la función *escanear*.

Las clases para los tokens y sus campos se ilustran en la figura 2.32; sus métodos no se muestran. La clase **Token** tiene un campo llamado **etiqueta**, el cual se usa para las decisiones sobre el análisis sintáctico. La subclase **Num** agrega un campo **valor** para un valor de entero. La subclase **Palabra** agrega un campo **lexema**, el cual se utiliza para las palabras reservadas y los identificadores.

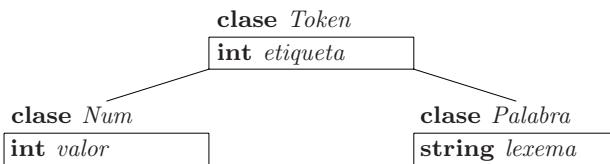


Figura 2.32: La clase *Token* y las subclases *Num* y *Palabra*

Cada clase se encuentra en un archivo por sí sola. A continuación se muestra el archivo para la clase **Token**:

```
1) package analizadorlexico;           // Archivo Token.java
2) public class Token {
3)     public final int etiqueta;
4)     public Token(int t) {etiqueta = t; }
5) }
```

La línea 1 identifica el paquete **analizadorlexico**. El campo **etiqueta** se declara en la línea 3 como **final**, de forma que no puede modificarse una vez establecido. El constructor **Token** en la línea 4 se utiliza para crear objetos token, como en

```
new Token('+')
```

lo cual crea un nuevo objeto de la clase **Token** y establece su campo **etiqueta** a una representación entera de '+'. Por cuestión de brevedad, omitimos el habitual método **toString**, que devolvería una cadena adecuada para su impresión.

En donde el seudocódigo tenga terminales como **num** e **id**, el código de Java utiliza constantes enteras. La clase **Etiqueta** implementa a dichas constantes:

```

1) package analizadorlexico;           // Archivo Etiqueta.java
2) public class Etiqueta {
3)     public final static int
4)         NUM = 256, ID = 257, TRUE = 258, FALSE = 259;
5) }
```

Además de los campos con valor de entero NUM e ID, esta clase define dos campos adicionales, TRUE y FALSE, para un uso futuro; se utilizarán para ilustrar el tratamiento de las palabras clave reservadas.⁷

Los campos en la clase **Etiqueta** son **public**, por lo que pueden usarse fuera del paquete. Son **static**, por lo que sólo hay una instancia o copia de estos campos. Los campos son **final**, por lo que pueden establecerse sólo una vez. En efecto, estos campos representan constantes. En C se logra un efecto similar mediante el uso de instrucciones de definición, para permitir que nombres como NUM se utilicen como constantes simbólicas, por ejemplo:

```
#define NUM 256
```

El código en Java se refiere a **Etiqueta.NUM** y **Etiqueta.ID** en lugares en donde el seudocódigo se refiere a los terminales **num** e **id**. El único requerimiento es que **Etiqueta.NUM** y **Etiqueta.ID** deben inicializarse con valores distintos, que difieran uno del otro y de las constantes que representan tokens de un solo carácter, como '+' o '*'.

```

1) package analizadorlexico;           // Archivo Num.java
2) public class Num extends Token {
3)     public final int valor;
4)     public Num(int v) { super(Etiqueta.NUM); valor = v; }
5) }

1) package analizadorlexico;           // Archivo Palabra.java
2) public class Palabra extends Token {
3)     public final String lexema;
4)     public Palabra(int t, String s) {
5)         super(t); lexema = new String(s);
6)     }
7) }
```

Figura 2.33: Las subclases Num y Palabra de Token

Las clases **Num** y **Palabra** aparecen en la figura 2.33. La clase **Num** extiende a **Token** mediante la declaración de un campo entero **valor** en la línea 3. El constructor **Num** en la línea 4 llama a **super(Etiqueta.NUM)**, que establece el campo **etiqueta** en la superclase **Token** a **Etiqueta.NUM**.

⁷Por lo general, los caracteres ASCII se convierten en enteros entre 0 y 255. Por lo tanto, utilizaremos enteros mayores de 255 para los terminales.

```

1) package analizadorlexico;           // Archivo AnalizadorLexico.java
2) import java.io.*; import java.util.*;
3) public class AnalizadorLexico {
4)     public int linea = 1;
5)     private char vistazo = ' ';
6)     private Hashtable palabras = new Hashtable();
7)     void reservar(Palabra t) { palabras.put(t.lexema, t); }
8)     public AnalizadorLexico() {
9)         reservar( new Palabra(Etiqueta.TRUE, "true") );
10)        reservar( new Palabra(Etiqueta.FALSE, "false") );
11)    }
12)    public Token explorar() throws IOException {
13)        for( ; ; vistazo = (char)System.in.read() ) {
14)            if( vistazo == ' ' || vistazo == '\t' ) continue;
15)            else if( vistazo == '\n' ) linea = linea + 1;
16)            else break;
17)        }
/* continúa en la figura 2.35 */

```

Figura 2.34: Código para un analizador léxico, parte 1 de 2

La clase **Palabra** se utiliza para palabras reservadas e identificadores, por lo que el constructor **Palabra** en la línea 4 espera dos parámetros: un lexema y un valor entero correspondiente para **etiqueta**. Podemos crear un objeto para la palabra reservada **true** ejecutando lo siguiente:

```
new Palabra(Etiqueta.TRUE, "true")
```

lo cual crea un Nuevo objeto con el campo **etiqueta** establecido en **etiqueta.TRUE** y el campo **lexema** establecido a la cadena **"true"**.

La clase **AnalizadorLexico** para el análisis léxico aparece en las figuras 2.34 y 2.35. La variable entera **linea** en la línea 4 cuenta las líneas de entrada, y la variable carácter **vistazo** en la línea 5 contiene el siguiente carácter de entrada.

Las palabras reservadas se manejan en las líneas 6 a 11. La tabla **palabras** se declara en la línea 6. La función auxiliar **reservar** en la línea 7 coloca un par cadena-palabra en la tabla. Las líneas 9 y 10 en el constructor **AnalizadorLexico** inicializan la tabla. Utilizan el constructor **Palabra** para crear objetos tipo palabra, los cuales se pasan a la función auxiliar **reservar**. Por lo tanto, la tabla se inicializa con las palabras reservadas **"true"** y **"false"** antes de la primera llamada de **explorar**.

El código para **escanear** en las figuras 2.34-2.35 implementa los fragmentos de seudocódigo en esta sección. La instrucción **for** en las líneas 13 a 17 omite los caracteres de espacio en blanco, tabuladores y de nueva línea. Cuando el control sale de la instrucción **for**, **vistazo** contiene un carácter que no es espacio en blanco.

El código para leer una secuencia de dígitos está en las líneas 18 a 25. La función **isDigit** es de la clase **Character** integrada en Java. Se utiliza en la línea 18 para comprobar si **vistazo**

```

18)     if( Character.isDigit(vistazo) ) {
19)         int v = 0;
20)         do {
21)             v = 10*v + Character.digit(vistazo, 10);
22)             vistazo = (char)System.in.read();
23)         } while( Character.isDigit(vistazo) );
24)         return new Num(v);
25)     }
26)     if( Character.isLetter(vistazo) ) {
27)         StringBuffer b = new StringBuffer();
28)         do {
29)             b.append(vistazo);
30)             peek = (char)System.in.read();
31)         } while( Character.isLetterOrDigit(vistazo) );
32)         String s = b.toString();
33)         Palabra w = (Palabra)palabras.get(s);
34)         if (w != null) return w;
35)         w = new Palabra(Etiqueta.ID, s);
36)         palabras.put(s, w);
37)         return w;
38)     }
39)     Token t = new Token(vistazo);
40)     vistazo = ' ';
41)     return t;
42) }
43) }
```

Figura 2.35: Código para un analizador léxico, parte 2 de 2

es un dígito. De ser así, el código en las líneas 19 a 24 acumula el valor entero de la secuencia de dígitos en la entrada y devuelve un nuevo objeto `Num`.

Las líneas 26 a 38 analizan palabras reservadas e identificadores. Las palabras clave `true` y `false` ya se han reservado en las líneas 9 y 10. Así, si llegamos a la línea 35 entonces la cadena `s` no está reservada, por lo cual debe ser el lexema para un identificador. La línea 35, por lo tanto, devuelve un nuevo objeto `palabra` con `lexema` establecido a `s` y `etiqueta` establecida a `Etiqueta.ID`. Por último, las líneas 39 a 41 devuelven el carácter actual como un token y establecen `vistazo` a un espacio en blanco, que se eliminará la próxima vez que se llame a `escanear`.

2.6.6 Ejercicios para la sección 2.6

Ejercicio 2.6.1: Extienda el analizador léxico de la sección 2.6.5 para eliminar los comentarios, que se definen de la siguiente manera:

- Un comentario empieza con `//` e incluye a todos los caracteres hasta el fin de esa línea.

- b) Un comentario empieza con `/*` e incluye a todos los caracteres hasta la siguiente ocurrencia de la secuencia de caracteres `*/`.

Ejercicio 2.6.2: Extienda el analizador léxico en la sección 2.6.5 para que reconozca los operadores relacionales `<`, `<=`, `==`, `!=`, `>=`, `>`.

Ejercicio 2.6.3: Extienda el analizador léxico en la sección 2.6.5 para que reconozca los valores de punto flotante como `2.`, `3.14` y `.5`.

2.7 Tablas de símbolos

Las *tablas de símbolos* son estructuras de datos que utilizan los compiladores para guardar información acerca de las construcciones de un programa fuente. La información se recolecta en forma incremental mediante las fases de análisis de un compilador, y las fases de síntesis la utilizan para generar el código destino. Las entradas en la tabla de símbolos contienen información acerca de un identificador, como su cadena de caracteres (o lexema), su tipo, su posición en el espacio de almacenamiento, y cualquier otra información relevante. Por lo general, las tablas de símbolos necesitan soportar varias declaraciones del mismo identificador dentro de un programa.

En la sección 1.6.1 vimos que el alcance de una declaración es la parte de un programa a la cual se aplica esa declaración. Vamos a implementar los alcances mediante el establecimiento de una tabla de símbolos separada para cada alcance. Un bloque de programa con declaraciones⁸ tendrá su propia tabla de símbolos, con una entrada para cada declaración en el bloque. Este método también funciona para otras construcciones que establecen alcances; por ejemplo, una clase tendrá su propia tabla, con una entrada para cada campo y cada método.

Esta sección contiene un módulo de tabla de símbolos, adecuado para usarlo con los fragmentos del traductor en Java de este capítulo. El módulo se utilizará como está, cuando ensamblemos el traductor completo en el apéndice A. Mientras tanto, por cuestión de simplicidad, el ejemplo principal de esta sección es un lenguaje simplificado, que sólo contiene las construcciones clave que tocan las tablas de símbolos; en especial, los bloques, las declaraciones y los factores. Omitiremos todas las demás construcciones de instrucciones y expresiones, para poder enfocarnos en las operaciones con la tabla de símbolos. Un programa consiste en bloques con declaraciones opcionales e “instrucciones” que consisten en identificadores individuales. Cada instrucción de este tipo representa un uso del identificador. He aquí un programa de ejemplo en este lenguaje:

(2.7)

Los ejemplos de la estructura de bloques en la sección 1.6.3 manejaron las definiciones y usos de nombres; la entrada (2.7) consiste únicamente de definiciones y usos de nombres.

La tarea que vamos a realizar es imprimir un programa revisado, en el cual se han eliminado las declaraciones y cada “instrucción” tiene su identificador, seguido de un signo de punto y coma y de su tipo.

⁸En C, por ejemplo, los bloques de programas son funciones o secciones de funciones que se separan mediante llaves, y que tienen una o más declaraciones en su interior.

¿Quién crea las entradas en la tabla de símbolos?

El analizador léxico, el analizador sintáctico y el analizador semántico son los que crean y utilizan las entradas en la tabla de símbolos durante la fase de análisis. En este capítulo, haremos que el analizador sintáctico cree las entradas. Con su conocimiento de la estructura sintáctica de un programa, por lo general, un analizador sintáctico está en una mejor posición que el analizador léxico para diferenciar entre las distintas declaraciones de un identificador.

En algunos casos, un analizador léxico puede crear una entrada en la tabla de símbolos, tan pronto como ve los caracteres que conforman un lexema. Más a menudo, el analizador léxico sólo puede devolver un token al analizador sintáctico, por decir **id**, junto con un apuntador al lexema. Sin embargo, sólo el analizador sintáctico puede decidir si debe utilizar una entrada en la tabla de símbolos que se haya creado antes, o si debe crear una entrada nueva para el identificador.

Ejemplo 2.14: En la entrada anterior (2.7), el objetivo es producir lo siguiente:

```
{ { x:int; y:bool; } x:int; y:char; }
```

Las primeras **x** y **y** son del bloque interno de la entrada (2.7). Como este uso de **x** se refiere a la declaración de **x** en el bloque externo, va seguido de **int**, el tipo de esa declaración. El uso de **y** en el bloque interno se refiere a la declaración de **y** en ese mismo bloque y, por lo tanto, tiene el tipo booleano. También vemos los usos de **x** y **y** en el bloque externo, con sus tipos, según los proporcionan las declaraciones del bloque externo: entero y carácter, respectivamente. \square

2.7.1 Tabla de símbolos por alcance

El término “alcance del identificador *x*” en realidad se refiere al alcance de una declaración específica de *x*. El término *alcance* por sí solo se refiere a una parte del programa que es el alcance de una o más declaraciones.

Los alcances son importantes, ya que el mismo identificador puede declararse para distintos fines en distintas partes de un programa. A menudo, los nombres comunes como **i** y **x** tienen varios usos. Como otro ejemplo, las subclases pueden volver a declarar el nombre de un método para redefinirlo de una superclase.

Si los bloques pueden anidarse, varias declaraciones del mismo identificador pueden aparecer dentro de un solo bloque. La siguiente sintaxis produce bloques anidados cuando *instrs* puede generar un bloque:

$$\text{bloque} \rightarrow \text{'{' } \text{decls } \text{instrs } \text{''}$$

Colocamos las llaves entre comillas simples en la sintaxis para diferenciarlas de las llaves para las acciones semánticas. Con la gramática en la figura 2.38, *decls* genera una secuencia opcional de declaraciones y *instrs* genera una secuencia opcional de instrucciones.

Optimización de las tablas de símbolos para los bloques

Las implementaciones de las tablas de símbolos para los bloques pueden aprovechar la regla del bloque anidado más cercano. El anidamiento asegura que la cadena de tablas de símbolos aplicables forme una pila. En la parte superior de la pila se encuentra la tabla para el bloque actual. Debajo de ella en la pila están las tablas para los bloques circundantes. Por ende, las tablas de símbolos pueden asignarse y desasignarse en forma parecida a una pila.

Algunos compiladores mantienen una sola hash table de entradas accesibles; es decir, de entradas que no se ocultan mediante una declaración en un bloque anidado. Dicha hash table soporta búsquedas esenciales en tiempos constantes, a expensas de insertar y eliminar entradas al entrar y salir de los bloques. Al salir de un bloque B , el compilador debe deshacer cualquier modificación a la hash table debido a las declaraciones en el bloque B . Para ello puede utilizar una pila auxiliar, para llevar el rastro de las modificaciones a la tabla hash mientras se procesa el bloque B .

Inclusive, una instrucción puede ser un bloque, por lo que nuestro lenguaje permite bloques anidados, en donde puede volver a declararse un identificador.

La regla del *bloque anidado más cercano* nos indica que un identificador x se encuentra en el alcance de la declaración anidada más cercana de x ; es decir, la declaración de x que se encuentra al examinar los bloques desde adentro hacia fuera, empezando con el bloque en el que aparece x .

Ejemplo 2.15: El siguiente seudocódigo utiliza subíndices para diferenciar entre las distintas declaraciones del mismo identificador:

```

1)  {   int x1; int y1;
2)    {   int w2; bool y2; int z2;
3)      ... w2 ...; ... x1 ...; ... y2 ...; ... z2 ...;
4)    }
5)      ... w0 ...; ... x1 ...; ... y1 ...;
6)  }

```

El subíndice no forma parte de un identificador; es, de hecho, el número de línea de la declaración que se aplica al identificador. Por ende, todas las ocurrencias de x están dentro del alcance de la declaración en la línea 1. La ocurrencia de y en la línea 3 está en el alcance de la declaración de y en la línea 2, ya que y se volvió a declarar dentro del bloque interno. Sin embargo, la ocurrencia de y en la línea 5 está dentro del alcance de la declaración de y en la línea 1.

La ocurrencia de w en la línea 5 se encuentra supuestamente dentro del alcance de una declaración de w fuera de este fragmento del programa; su subíndice 0 denota una declaración que es global o externa para este bloque.

Por último, z se declara y se utiliza dentro del bloque anidado, pero no puede usarse en la línea 5, ya que la declaración anidada se aplica sólo al bloque anidado. \square

La regla del bloque anidado más cercano puede implementarse mediante el encadenamiento de las tablas de símbolos. Es decir, la tabla para un bloque anidado apunta a la tabla para el bloque circundante.

Ejemplo 2.16: La figura 2.36 muestra tablas de símbolos para el seudocódigo del ejemplo 2.15. B_1 es para el bloque que empieza en la línea 1 y B_2 es para el bloque que empieza en la línea 2. En la parte superior de la figura hay una tabla de símbolos adicional B_0 para cualquier declaración global o predeterminada que proporcione el lenguaje. Durante el tiempo que analizamos las líneas 2 a 4, el entorno se representa mediante una referencia a la tabla de símbolos inferior (la de B_2). Cuando avanzamos a la línea 5, la tabla de símbolos para B_2 se vuelve inaccesible, y el entorno se refiere en su lugar a la tabla de símbolos para B_1 , a partir de la cual podemos llegar a la tabla de símbolos global, pero no a la tabla para B_2 . \square

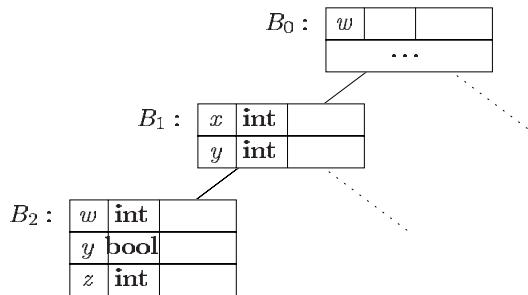


Figura 2.36: Tablas de símbolos encadenadas para el ejemplo 2.15

La implementación en Java de las tablas de símbolos encadenadas en la figura 2.37 define a una clase `Ent`, abreviación de *entorno*.⁹ La clase `Ent` soporta tres operaciones:

- *Crear una nueva tabla de símbolos.* El constructor `Ent(p)` en las líneas 6 a 8 de la figura 2.37 crea un objeto `Ent` con una hash table llamada `tabla`. El objeto se encadena al parámetro con valor de entorno `p`, estableciendo el campo `siguiente` a `p`. Aunque los objetos `Ent` son los que forman una cadena, es conveniente hablar de las tablas que se van a encadenar.
- *put* una nueva entrada en la tabla actual. La hash table contiene pares clave-valor, en donde:
 - La *clave* es una cadena, o más bien una referencia a una cadena. Podríamos usar de manera alternativa referencias a objetos token para identificadores como las claves.
 - El *valor* es una entrada de la clase `Símbolo`. El código en las líneas 9 a 11 no necesita conocer la estructura de una entrada; es decir, el código es independiente de los campos y los métodos en la clase `Símbolo`.

⁹“Entorno” es otro término para la colección de tablas de símbolos que son relevantes en un punto dado en el programa.

```

1) package símbolos;           // Archivo Ent.java
2) import java.util.*;
3) public class Ent {
4)     private Hashtable tabla;
5)     protected Ent ant;
6)
7)     public Ent(Ent p) {
8)         tabla = new Hashtable(); ant = p;
9)
10)    public void put(String s, Simbolo sim) {
11)        tabla.put(s, sim);
12)
13)    public Simbolo get(String s) {
14)        for( Ent e = this; e != null; e = e.ant ) {
15)            Simbolo encontro = (Simbolo)(e.tabla.get(s));
16)            if( encontro != null ) return encontro;
17)        }
18)        return null;
19)    }

```

Figura 2.37: La clase *Ent* implementa a las tablas de símbolos encadenadas

- *get* una entrada para un identificador buscando en la cadena de tablas, empezando con la tabla para el bloque actual. El código para esta operación en las líneas 12 a 18 devuelve una entrada en la tabla de símbolos o *null*.

El encadenamiento de las tablas de símbolos produce una estructura tipo árbol, ya que puede anidarse más de un bloque dentro de un bloque circundante. Las líneas punteadas en la figura 2.36 son un recordatorio de que las tablas de símbolos encadenadas pueden formar un árbol.

2.7.2 El uso de las tablas de símbolos

En efecto, la función de una tabla de símbolos es pasar información de las declaraciones a los usos. Una acción semántica “coloca” (*put*) información acerca de un identificador *x* en la tabla de símbolos, cuando se analiza la declaración de *x*. Posteriormente, una acción semántica asociada con una producción como *factor* → **id** “obtiene” (*get*) información acerca del identificador, de la tabla de símbolos. Como la traducción de una expresión $E_1 \text{ op } E_2$, para un operador **op** ordinario, depende sólo de las traducciones de E_1 y E_2 , y no directamente de la tabla de símbolos, podemos agregar cualquier número de operadores sin necesidad de cambiar el flujo básico de información de las declaraciones a los usos, a través de la tabla de símbolos.

Ejemplo 2.17: El esquema de traducción en la figura 2.38 ilustra cómo puede usarse la clase *Ent*. El esquema de traducción se concentra en los alcances, las declaraciones y los usos. Implementa la traducción descrita en el ejemplo 2.14. Como dijimos antes, en la entrada

<i>programa</i>	\rightarrow	$\{ \ sup = \mathbf{null}; \ }$
	<i>bloque</i>	
<i>bloque</i>	\rightarrow	$'\{'$
		$\{ \ guardado = sup;$
		$sup = \mathbf{new} \ Ent(sup);$
		$\text{print}(" \{ \ "); \}$
<i>decls instrs '}'</i>		$\{ \ sup = guardado;$
		$\text{print}(" \} \ "); \}$
<i>decls</i>	\rightarrow	<i>decls decl</i>
	$ $	ϵ
<i>decl</i>	\rightarrow	tipo id ;
		$\{ \ s = \mathbf{new} \ Símbolo;$
		$s.type = \mathbf{tipo}.lexema$
		$sup.put(\mathbf{id}.lexema, s); \}$
<i>instrs</i>	\rightarrow	<i>instrs instr</i>
	$ $	ϵ
<i>instr</i>	\rightarrow	<i>bloque</i>
	$ $	<i>factor</i> ;
		$\{ \ \text{print}(" \; \ "); \}$
<i>factor</i>	\rightarrow	id
		$\{ \ s = sup.get(\mathbf{id}.lexema);$
		$\text{print}(\mathbf{id}.lexema);$
		$\text{print}(" \: "); \}$
		$\text{print}(s.tipo);$

Figura 2.38: El uso de las tablas de símbolos para traducir un lenguaje con bloques

```
{ int x;  char y;  { bool y; x; y; } x; y; }
```

el esquema de traducción elimina las declaraciones y produce

```
{ { x:int;  y:bool; } x:int;  y:char; }
```

Observe que los cuerpos de las producciones se alinearon en la figura 2.38, para que todos los símbolos de gramática aparezcan en una columna y todas las acciones en una segunda columna. Como resultado, por lo general, los componentes del cuerpo se esparcen a través de varias líneas.

Ahora, consideremos las acciones semánticas. El esquema de traducción crea y descarta las tablas de símbolos al momento de entrar y salir de los bloques, respectivamente. La variable *sup* denota la tabla superior, en el encabezado de una cadena de tablas. La primera producción de la gramática subyacente es *programa* \rightarrow *bloque*. La acción semántica antes de *bloque* inicializa *sup* a **null**, sin entradas.

La segunda producción, *bloque* \rightarrow '{' *decls instrs* '}', tiene acciones al momento en que se entra y se sale del bloque. Al entrar al bloque, antes de *decls*, una acción semántica guarda una referencia a la tabla actual, usando una variable local llamada *guardado*. Cada uso de esta producción tiene su propia variable local *guardado*, distinta de la variable local para cualquier otro uso de esta producción. En un analizador sintáctico de descenso recursivo, *guardado* sería local para el *bloque* for del procedimiento. En la sección 7.2 veremos el tratamiento de las variables locales de una función recursiva. El siguiente código:

```
sup = new Ent(sup);
```

establece la variable *sup* a una tabla recién creada que está encadenada al valor anterior de *sup*, justo antes de entrar al bloque. La variable *sup* es un objeto de la clase *Ent*; el código para el constructor *Ent* aparece en la figura 2.37.

Al salir del bloque, después de '}', una acción semántica restaura *sup* al valor que tenía guardado al momento de entrar al bloque. En realidad, las tablas forman una pila; al restaurar *sup* a su valor guardado, se saca el efecto de las declaraciones en el bloque.¹⁰ Por ende, las declaraciones en el bloque no son visibles fuera del mismo.

Una declaración, *decls* \rightarrow **tipo** **id** produce una nueva entrada para el identificador declarado. Asumimos que los tokens **tipo** e **id** tienen cada uno un atributo asociado, que es el tipo y el lexema, respectivamente, del identificador declarado. En vez de pasar por todos los campos de un objeto de símbolo *s*, asumiremos que hay un campo *tipo* que proporciona el tipo del símbolo. Creamos un nuevo objeto de símbolo *s* y asignamos su tipo de manera apropiada, mediante *s.tipo = tipo.lexema*. La entrada completa se coloca en la tabla de símbolos superior mediante *sup.put(id.lexema, s)*.

La acción semántica en la producción *factor* \rightarrow **id** utiliza la tabla de símbolos para obtener la entrada para el identificador. La operación *get* busca la primera entrada en la cadena de tablas, empezando con *sup*. La entrada que se obtiene contiene toda la información necesaria acerca del identificador, como su tipo. \square

2.8 Generación de código intermedio

El front-end de un compilador construye una representación intermedia del programa fuente, a partir de la cual el back-end genera el programa destino. En esta sección consideraremos representaciones intermedias para expresiones e instrucciones, y veremos ejemplos de cómo producir dichas representaciones.

2.8.1 Dos tipos de representaciones intermedias

Como sugerimos en la sección 2.1 y especialmente en la figura 2.4, las dos representaciones intermedias más importantes son:

¹⁰En vez de guardar y restaurar tablas en forma explícita, una alternativa podría ser agregar las operaciones estáticas *push* y *pop* a la clase *Ent*.

- Los *árboles*, incluyendo los de análisis sintáctico y los sintácticos (abstractos).
- Las representaciones lineales, en especial el “código de tres direcciones”.

Los árboles sintácticos abstractos, o simplemente sintácticos, se presentaron en la sección 2.5.1, y en la sección 5.3.1 volveremos a examinarlos de una manera más formal. Durante el análisis sintáctico, se crean los nodos del árbol sintáctico para representar construcciones de programación importantes. A medida que avanza el análisis, se agrega información a los nodos en forma de atributos asociados con éstos. La elección de atributos depende de la traducción a realizar.

Por otro lado, el código de tres direcciones es una secuencia de pasos de programa elementales, como la suma de dos valores. A diferencia del árbol, no hay una estructura jerárquica. Como veremos en el capítulo 9, necesitamos esta representación si vamos a realizar algún tipo de optimización importante de código. En ese caso, dividimos la extensa secuencia de instrucciones de tres direcciones que forman un programa en “bloques básicos”, que son secuencias de instrucciones que siempre se ejecutan una después de la otra, sin bifurcaciones.

Además de crear una representación intermedia, el front-end de un compilador comprueba que el programa fuente siga las reglas sintácticas y semánticas del lenguaje fuente. A esta comprobación se le conoce como *comprobación estática*; en general, “estático” significa “realizado por el compilador”.¹¹ La comprobación estática asegura que se detecten ciertos tipos de errores de programación, incluyendo los conflictos de tipos, y que se reporten durante la compilación.

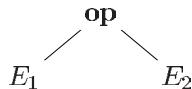
Es posible que un compilador construya un árbol sintáctico al mismo tiempo que emite los pasos del código de tres direcciones. No obstante, es común que los compiladores emitan el código de tres direcciones mientras el analizador sintáctico “avanza en el proceso” de construir un árbol sintáctico, sin construir en realidad la estructura de datos tipo árbol completa. En vez de ello, el compilador almacena los nodos y sus atributos necesarios para la comprobación semántica o para otros fines, junto con la estructura de datos utilizada para el análisis sintáctico. Al hacer esto, las partes del árbol sintáctico que son necesarias para construir el código de tres direcciones están disponibles cuando se necesitan, pero desaparecen cuando ya no son necesarias. En el capítulo 5 veremos los detalles relacionados con este proceso.

2.8.2 Construcción de árboles sintácticos

Primero vamos a proporcionar un esquema de traducción para construir árboles sintácticos, y después, en la sección 2.8.4, mostraremos cómo puede modificarse este esquema para emitir código de tres direcciones, junto con, o en vez de, el árbol sintáctico.

En la sección 2.5.1 vimos que el árbol sintáctico

¹¹ Su contraparte, “dinámico”, significa “mientras el programa se ejecuta”. Muchos lenguajes también realizan ciertas comprobaciones dinámicas. Por ejemplo, un lenguaje orientado a objetos como Java algunas veces debe comprobar los tipos durante la ejecución de un programa, ya que el método que se aplique a un objeto puede depender de la subclase específica de ese objeto.



representa una expresión que se forma al aplicar el operador **op** a las subexpresiones representadas por E_1 y E_2 . Pueden crearse árboles sintácticos para cualquier construcción, no solo expresiones. Cada construcción se representa mediante un nodo, con hijos para los componentes con significado semántico de la construcción. Por ejemplo, los componentes con significado semántico de una instrucción `while` en C:

while (*expr*) *instr*

son la expresión *expr* y la instrucción *instr*.¹² El nodo del árbol sintáctico para una instrucción `while` de este tipo tiene un operador, al cual llamamos **while**, y dos hijos: los árboles sintácticos para *expr* y para *instr*.

El esquema de traducción en la figura 2.39 construye árboles sintácticos para un lenguaje representativo, pero bastante limitado, de expresiones e instrucciones. Todos los no terminales en el esquema de traducción tienen un atributo *n*, que es un nodo del árbol sintáctico. Los nodos se implementan como objetos de la clase *Nodo*.

La clase *Nodo* tiene dos subclases intermedias: *Expr* para todo tipo de expresiones, e *Instr* para todo tipo de instrucciones. Cada tipo de instrucción tiene una subclase correspondiente de *Instr*; por ejemplo, el operador **while** corresponde a la subclase *While*. Un nodo del árbol sintáctico para el operador **while**, con los hijos *x* y *y* se crea mediante el siguiente seudocódigo:

new *While*(*x,y*)

el cual crea un objeto de la clase *While* mediante una llamada al constructor constructora *While*, con el mismo nombre que la clase. Al igual que los constructores corresponden a los operadores, los parámetros de los constructores corresponden a los operandos en la sintaxis abstracta.

Cuando estudiemos el código detallado en el apéndice A, veremos cómo se colocan los métodos en donde pertenecen en esta jerarquía de clases. En esta sección hablaremos sólo de algunos métodos, de manera informal.

Vamos a considerar cada una de las producciones y reglas de la figura 2.39, una a la vez. Primero explicaremos las producciones que definen distintos tipos de instrucciones, y después las producciones que definen nuestros tipos limitados de expresiones.

Árboles sintácticos para las instrucciones

Para cada construcción de una instrucción, definimos un operador en la sintaxis abstracta. Para las construcciones que empiezan con una palabra clave, vamos a usar la palabra clave para el operador. Por ende, hay un operador **while** para las instrucciones `while` y un operador **do** para

¹²El paréntesis derecho sólo sirve para separar la expresión de la instrucción. El paréntesis izquierdo en realidad no tiene significado; está ahí sólo para facilitar la legibilidad, ya que sin él, C permitiría paréntesis desbalanceados.

<i>programa</i>	\rightarrow	<i>bloque</i>	$\{ \text{return } \text{bloque}.n; \}$
<i>bloque</i>	\rightarrow	'{' <i>instrs</i> '}'	$\{ \text{bloque}.n = \text{instrs}.n; \}$
<i>instrs</i>	\rightarrow	<i>instrs</i> ₁ <i>instr</i>	$\{ \text{instrs}.n = \mathbf{new} \text{ Sec}(\text{instrs}_1.n, \text{instr}.n); \}$
		ϵ	$\{ \text{instrs}.n = \mathbf{null}; \}$
<i>instr</i>	\rightarrow	<i>expr</i> ;	$\{ \text{instr}.n = \mathbf{new} \text{ Eval}(\text{expr}.n); \}$
		if (<i>expr</i>) <i>instr</i> ₁	$\{ \text{instr}.n = \mathbf{new} \text{ If}(\text{expr}.n, \text{instr}_1.n); \}$
		while (<i>expr</i>) <i>instr</i> ₁	$\{ \text{instr}.n = \mathbf{new} \text{ While}(\text{expr}.n, \text{instr}_1.n); \}$
		do <i>instr</i> ₁ while (<i>expr</i>);	$\{ \text{instr}.n = \mathbf{new} \text{ Do}(\text{instr}_1.n, \text{expr}.n); \}$
		<i>bloque</i>	$\{ \text{instr}.n = \text{bloque}.n; \}$
<i>expr</i>	\rightarrow	<i>rel</i> = <i>expr</i> ₁	$\{ \text{expr}.n = \mathbf{new} \text{ Asigna}('=', \text{rel}.n, \text{expr}_1.n); \}$
		<i>rel</i>	$\{ \text{expr}.n = \text{rel}.n; \}$
<i>rel</i>	\rightarrow	<i>rel</i> ₁ < <i>adic</i>	$\{ \text{rel}.n = \mathbf{new} \text{ Rel}('<', \text{rel}_1.n, \text{adic}.n); \}$
		<i>rel</i> ₁ \leq <i>adic</i>	$\{ \text{rel}.n = \mathbf{new} \text{ Rel}(' \leq ', \text{rel}_1.n, \text{adic}.n); \}$
		<i>adic</i>	$\{ \text{rel}.n = \text{adic}.n; \}$
<i>adic</i>	\rightarrow	<i>adic</i> ₁ + <i>term</i>	$\{ \text{adic}.n = \mathbf{new} \text{ Op}('+', \text{adic}_1.n, \text{term}.n); \}$
		<i>term</i>	$\{ \text{adic}.n = \text{term}.n; \}$
<i>term</i>	\rightarrow	<i>term</i> ₁ * <i>factor</i>	$\{ \text{term}.n = \mathbf{new} \text{ Op}('*', \text{term}_1.n, \text{factor}.n); \}$
		<i>factor</i>	$\{ \text{term}.n = \text{factor}.n; \}$
<i>factor</i>	\rightarrow	(<i>expr</i>)	$\{ \text{factor}.n = \text{expr}.n; \}$
		num	$\{ \text{factor}.n = \mathbf{new} \text{ Num}(\text{num}.valor); \}$

Figura 2.39: Construcción de árboles sintácticos para expresiones e instrucciones

las instrucciones do-while. Las instrucciones condicionales pueden manejarse mediante la definición de dos operadores **ifelse** e **if** para las instrucciones **if** con y sin una parte **else**, respectivamente. En nuestro lenguaje de ejemplo simple, no utilizamos **else**, por lo cual sólo tenemos una instrucción **if**. Agregar **else** implicaría ciertos problemas de análisis sintáctico, lo cual veremos en la sección 4.8.2.

Cada operador de instrucción tiene una clase correspondiente con el mismo nombre, con la primera letra en mayúscula; por ejemplo, la clase *If* corresponde a **if**. Además, definimos la subclase *Sec*, que representa a una secuencia de instrucciones. Esta subclase corresponde a la no terminal *instrs* de la gramática. Cada una de estas clases es subclase de *Instr*, que a su vez es una subclase de *Nodo*.

El esquema de traducción en la figura 2.39 ilustra la construcción de los nodos de árboles sintácticos. Una regla típica es la de las instrucciones **if**:

$$instr \rightarrow if(expr) instr_1 \quad \{ instr.n = \mathbf{new} \ If(expr.n, instr_1.n); \}$$

Los componentes importantes de la instrucción **if** son *expr* e *instr₁*. La acción semántica define el nodo *instr.n* como un nuevo objeto de la subclase *If*. El código para el constructor de *If* no se muestra. Crea un nuevo nodo etiquetado como **if**, con los nodos *expr.n* e *instr₁.n* como hijos.

Las instrucciones de expresiones no empiezan con una palabra clave, por lo que definimos un nuevo operador **eval** y la clase *Eval*, que es una subclase de *Instr*, para representar las expresiones que son instrucciones. La regla relevante es:

$$instr \rightarrow expr; \quad \{ instr.n = \mathbf{new} \ Eval(expr.n); \}$$

Representación de los bloques en los árboles sintácticos

La construcción de la instrucción restante en la figura 2.39 es el bloque, el cual consiste en una secuencia de instrucciones. Considera las siguientes reglas:

$$instr \rightarrow bloque \quad \{ instr.n = bloque.n; \}$$

$$bloque \rightarrow '{' instrs '}' \quad \{ bloque.n = instrs.n; \}$$

La primera dice que cuando una instrucción es un bloque, tiene el mismo árbol sintáctico que el bloque. La segunda regla dice que el árbol sintáctico para el no terminal *bloque* es simplemente el árbol sintáctico para la secuencia de instrucciones en el bloque.

Por cuestión de simplicidad, el lenguaje en la figura 2.39 no incluye declaraciones. Aun cuando las declaraciones se incluyen en el apéndice A, veremos que el árbol sintáctico para un bloque sigue siendo el árbol sintáctico para las instrucciones en el bloque. Como la información de las declaraciones está incorporada en la tabla de símbolos, no se necesita en el árbol sintáctico. Por lo tanto, los bloques (con o sin declaraciones) parecen ser sólo otra construcción de instrucción en el código intermedio.

Una secuencia de instrucciones se representa mediante el uso de una hoja **null** para una instrucción vacía, y un operador **sec** para una secuencia de instrucciones, como se muestra a continuación:

$$instrs \rightarrow instrs_1 \ instr \quad \{ instrs.n = \mathbf{new} \ Sec(instrs_1.n, instr.n); \}$$

Ejemplo 2.18: En la figura 2.40, vemos parte de un árbol sintáctico que representa a un bloque, o lista de instrucciones. Hay dos instrucciones en la lista, siendo la primera una instrucción **if** y la segunda una instrucción **while**. No mostramos la parte del árbol que está por encima de esta lista de instrucciones, y sólo mostramos como triángulo a cada uno de los subárboles necesarios: dos árboles de expresiones para las condiciones de las instrucciones **if** y **while**, y dos árboles de instrucciones para sus subinstrucciones. \square

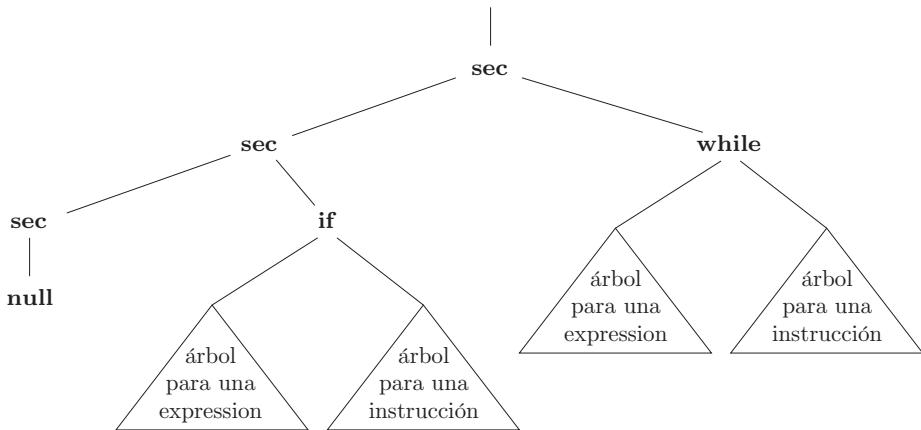


Figura 2.40: Parte de un árbol sintáctico para una lista de instrucciones que consiste en una instrucción **if** y una instrucción **while**

Árboles sintácticos para las expresiones

Anteriormente manejamos la precedencia más alta de ***** sobre **+** mediante el uso de los tres no terminales *expr*, *term* y *factor*. El número de no terminales es precisamente uno más el número de niveles de precedencia en las expresiones, como sugerimos en la sección 2.2.6. En la figura 2.39, tenemos dos operadores de comparación, **<** y **\leq** en un nivel de precedencia, así como los operadores **+** y ***** comunes, por lo que hemos agregado un no terminal adicional, llamado *adic*.

La sintaxis abstracta nos permite agrupar operadores “similares” para reducir el número de casos y subclases de nodos en una implementación de expresiones. En este capítulo, “similar” significa que las reglas de comprobación de tipos y generación de código para los operadores son similares. Por ejemplo, comúnmente los operadores **+** y ***** pueden agruparse, ya que pueden manejarse de la misma forma; sus requerimientos en relación con los tipos de operandos son los mismos, y cada uno produce una instrucción individual de tres direcciones que aplica un operador a dos valores. En general, el agrupamiento de operadores en la sintaxis abstracta se basa en las necesidades de las fases posteriores del compilador. La tabla en la figura 2.41 especifica la correspondencia entre la sintaxis concreta y abstracta para varios de los operadores de Java.

En la sintaxis concreta, todos los operadores son asociativos por la izquierda, excepto el operador de asignación **=**, el cual es asociativo por la derecha. Los operadores en una línea

SINTAXIS CONCRETA	SINTAXIS ABSTRACTA
=	asigna
	cond
&&	cond
== !=	rel
< <= >= >	rel
+ -	op
* / %	op
!	not
- <i>unario</i>	menos
[]	acceso

Figura 2.41: Sintaxis concreta y abstracta para varios operadores de Java

tienen la misma precedencia; es decir, == y != tienen la misma precedencia. Las líneas están en orden de menor a mayor precedencia; por ejemplo, == tiene mayor precedencia que los operadores && y =. El subíndice *unario* en - *unario* es sólo para distinguir un signo de menos unario a la izquierda, como en -2, de un signo de menos binario, como en 2-a. El operador [] representa el acceso a los arreglos, como en a[i].

La columna de sintaxis abstracta especifica el agrupamiento de los operadores. El operador de asignación = está en un grupo por sí solo. El grupo **cond** contiene los operadores booleanos condicionales && y ||. El grupo **rel** contiene los operadores de comparación relacionales en las líneas para == y <. El grupo **op** contiene los operadores aritméticos como + y *. El signo menos unario, la negación booleana y el acceso a los arreglos se encuentran en grupos por sí solos.

La asignación entre la sintaxis concreta y abstracta en la figura 2.41 puede implementarse escribiendo un esquema de traducción. Las producciones para los no terminales *expr*, *rel*, *adic*, *term* y *factor* en la figura 2.39 especifican la sintaxis concreta para un subconjunto representativo de los operadores en la figura 2.41. Las acciones semánticas en estas producciones crean nodos de árboles sintácticos. Por ejemplo, la regla

$$term \rightarrow term_1 * factor \{ term.n = \mathbf{new} \; Op('*', term_1.n, factor.n); \}$$

crea un nodo de la clase *Op*, que implementa a los operadores agrupados bajo **op** en la figura 2.41. El constructor *Op* tiene un parámetro '*' para identificar al operador actual, además de los nodos *term₁.n* y *factor.n* para las subexpresiones.

2.8.3 Comprobación estática

Las comprobaciones estáticas son comprobaciones de consistencia que se realizan durante la compilación. No solo aseguran que un programa pueda compilarse con éxito, sino que también tienen el potencial para atrapar los errores de programación en forma anticipada, antes de ejecutar un programa. La comprobación estática incluye lo siguiente:

- *Comprobación sintáctica.* Hay más en la sintaxis que las gramáticas. Por ejemplo, las restricciones como la de que un identificador se declare cuando menos una vez en un alcance

o que una instrucción `break` debe ir dentro de un ciclo o de una instrucción `switch`, son sintácticas, aunque no están codificadas en, o implementadas por, una gramática que se utiliza para el análisis sintáctico.

- *Comprobación de tipos.* Las reglas sobre los tipos de un lenguaje aseguran que un operador o función se aplique al número y tipo de operandos correctos. Si es necesaria la conversión entre tipos, por ejemplo, cuando se suma un entero a un número de punto flotante, entonces el comprobador de tipos puede insertar un operador en el árbol sintáctico para representar esa conversión. A continuación hablaremos sobre la conversión de tipos, usando el término común “coerción”.

L-value y R-value

Ahora consideraremos algunas comprobaciones estáticas simples que pueden realizarse durante la construcción de un árbol sintáctico para un programa fuente. En general, tal vez haya que realizar comprobaciones estáticas complejas, para lo cual primero hay que construir una representación intermedia y después analizarla.

Hay una diferencia entre el significado de los identificadores a la izquierda y el lado derecho de una asignación. En cada una de las siguientes asignaciones:

```
i = 5;
i = i + 1;
```

el lado derecho especifica un valor entero, mientras que el lado izquierdo especifica en dónde se va a almacenar el valor. Los términos *l-value* y *r-value* se refieren a los valores que son apropiados en los lados izquierdo y derecho de una asignación, respectivamente. Es decir, los *r-value* son lo que generalmente consideramos como “valores”, mientras que los *l-value* son las ubicaciones.

La comprobación estática debe asegurar que el lado izquierdo de una asignación denote a un *l-value*. Un identificador como `i` tiene un *l-value*, al igual que un acceso a un arreglo como `a[2]`. Pero una constante como `2` no es apropiada en el lado izquierdo de la asignación, ya que tiene un *r-value*, pero no un *l-value*.

Comprobación de tipos

La comprobación de tipos asegura que el tipo de una construcción coincida con lo que espera su contexto. Por ejemplo, en la siguiente instrucción `if`:

```
if ( expr ) instr
```

se espera que la expresión `expr` tenga el tipo **boolean**.

Las reglas de comprobación de tipos siguen la estructura operador/operando de la sintaxis abstracta. Suponga que el operador `rel` representa a los operadores relacionales como `<=`. La regla de tipos para el grupo de operadores `rel` es que sus dos operandos deben tener el mismo tipo, y el resultado tiene el tipo booleano. Utilizando el atributo *tipo* para el tipo de una expresión, dejemos que `E` consista de `rel` aplicado a `E1` y `E2`. El tipo de `E` puede comprobarse al momento de construir su nodo, mediante la ejecución de código como el siguiente:

```
if (  $E_1.tipo == E_2.tipo$  )  $E.tipo = boolean;$ 
else error;
```

La idea de relacionar los tipos actuales con los esperados se sigue aplicando, aún en las siguientes situaciones:

- *Coerciones.* Una *coerción* ocurre cuando el tipo de un operando se convierte en forma automática al tipo esperado por el operador. En una expresión como $2 * 3.14$, la transformación usual es convertir el entero 2 en un número de punto flotante equivalente, 2.0, y después realizar una operación de punto flotante con el par resultante de operandos de punto flotante. La definición del lenguaje especifica las coerciones disponibles. Por ejemplo, la regla actual para **rel** que vimos antes podría ser que $E_1.tipo$ y $E_2.tipo$ puedan convertirse al mismo tipo. En tal caso, sería legal comparar, por decir, un entero con un valor de punto flotante.
- *Sobrecarga.* El operador **+** en Java representa la suma cuando se aplica a enteros; significa concatenación cuando se aplica a cadenas. Se dice que un símbolo está *sobrecargado* si tiene distintos significados, dependiendo de su contexto. Por ende, **+** está sobrecargado en Java. Para determinar el significado de un operador sobrecargado, hay que considerar los tipos conocidos de sus operandos y resultados. Por ejemplo, sabemos que el **+** en $z = x + y$ es concatenación si sabemos que cualquiera de las variables **x**, **y** o **z** es de tipo cadena. No obstante, si también sabemos que alguna de éstas es de tipo entero, entonces tenemos un error en los tipos y no hay significado para este uso de **+**.

2.8.4 Código de tres direcciones

Una vez que se construyen los árboles de sintaxis, se puede realizar un proceso más detallado de análisis y síntesis mediante la evaluación de los atributos, y la ejecución de fragmentos de código en los nodos del árbol. Para ilustrar las posibilidades, vamos a recorrer árboles sintáticos para generar código de tres direcciones. En específico, le mostraremos cómo escribir funciones para procesar el árbol sintáctico y, como efecto colateral, emitir el código de tres direcciones necesario.

Instrucciones de tres direcciones

El código de tres direcciones es una secuencia de instrucciones de la forma

$$x = y \mathbf{op} z$$

en donde x , y y z son nombres, constantes o valores temporales generados por el compilador; y **op** representa a un operador.

Manejaremos los arreglos usando las siguientes dos variantes de instrucciones:

$$\begin{aligned} x [y] &= z \\ x &= y [z] \end{aligned}$$

La primera coloca el valor de z en la ubicación $x[y]$, y la segunda coloca el valor de $y[z]$ en la ubicación x .

Las instrucciones de tres direcciones se ejecutan en secuencia numérica, a menos que se les obligue a hacer lo contrario mediante un salto condicional o incondicional. Elegimos las siguientes instrucciones para el flujo de control:

<code>ifFalse x goto L</code>	si x es falsa, ejecutar a continuación la instrucción etiquetada como L
<code>ifTrue x goto L</code>	si x es verdadera, ejecutar a continuación la instrucción etiquetada como L
<code>goto L</code>	ejecutar a continuación la instrucción etiquetada como L

Para unir una etiqueta L a cualquier instrucción, se le antepone el prefijo L:. Una instrucción puede tener más de una etiqueta.

Por último, necesitamos instrucciones para copiar un valor. La siguiente instrucción de tres direcciones copia el valor de y a x :

$x = y$

Traducción de instrucciones

Las instrucciones se traducen en código de tres direcciones mediante el uso de instrucciones de salto, para implementar el flujo de control a través de la instrucción. La distribución de la figura 2.42 ilustra la traducción de `if expr then instr1`. La instrucción de salto en la siguiente distribución:

`ifFalse x goto después`

salta sobre la traducción de $instr_1$ si $expr$ se evalúa como **false**. Las demás construcciones de instrucciones se traducen de manera similar, usando saltos apropiados alrededor del código para sus componentes.

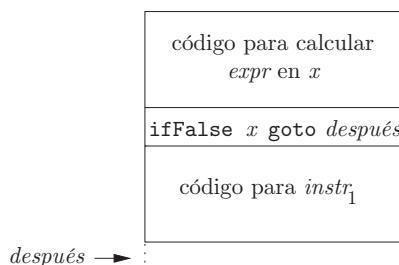


Figura 2.42: Distribución de código para las instrucciones if

Para fines concretos, mostramos el seudocódigo para la clase *If* en la figura 2.43. La clase *If* es una subclase de *Instr*, al igual que las clases para las demás construcciones de instrucciones. Cada subclase de *Instr* tiene un constructor (en este caso, *If*) y una función *gen*, a la cual llamamos para generar el código de tres direcciones para este tipo de instrucción.

```

class If extends Instr {
    Expr E; Instr S;
    public If(Expr x, Instr y) { E = x; S = y; despues = nuevaetiqueta(); }
    public void gen() {
        Expr n = E.r_value();
        emitir("iffalse" + n.toString() + " goto " + despues);
        S.gen();
        emitir(despues + ":");

    }
}

```

Figura 2.43: La función *gen* en la clase *If* genera código de tres direcciones

El constructor de *If* en la figura 2.43 crea nodos de árbol sintáctico para las instrucciones *if*. Se llama con dos parámetros, un nodo de expresión *x* y un nodo de instrucción *y*, los cuales guarda como atributos *E* y *S*. El constructor también asigna al atributo *despues* una nueva etiqueta única, llamando a la función *nuevaetiqueta()*. La etiqueta se utilizará de acuerdo con la distribución en la figura 2.42.

Una vez que se construye el árbol sintáctico completo para un programa fuente, se hace una llamada a la función *gen* en la raíz del árbol sintáctico. Como un programa es un bloque en nuestro lenguaje simple, la raíz del árbol sintáctico representa la secuencia de instrucciones en el bloque. Todas las clases de instrucciones contienen una función *gen*.

El seudocódigo para la función *gen* de la clase *If* en la figura 2.43 es representativo. Llama a *E.r_value()* para traducir la expresión *E* (la expresión con valor booleano que forma parte de las instrucciones *if*) y guarda el nodo de resultado devuelto por *E*. En breve hablaremos sobre la traducción de las expresiones. Después, la función *gen* emite un salto condicional y llama a *S.gen()* para traducir la subinstrucción *S*.

Traducción de expresiones

Ahora ilustraremos la traducción de las expresiones, para lo cual consideraremos expresiones que contengan operadores binarios **op**, accesos a arreglos y asignaciones, además de constantes e identificadores. Por cuestión de simplicidad, en un acceso a un arreglo *y*[*z*], requerimos que *y* sea un identificador.¹³ Para una explicación detallada sobre la generación de código intermedio para las expresiones, vea la sección 6.4.

Vamos a usar el enfoque sencillo de generar una instrucción de tres direcciones para cada nodo de operador en el árbol sintáctico para una expresión. No se genera código para los identificadores y las constantes, ya que éstos pueden aparecer como direcciones en las instrucciones. Si un nodo *x* de la clase *Expr* tiene un operador **op**, entonces se emite una instrucción para calcular el valor en el nodo *x* y convertirlo en un nombre “temporal” generado por el compilador, por decir, *t*. Por ende, *i*-*j*+*k* se traduce en dos instrucciones:

¹³Este lenguaje simple soporta *a*[*a*[*n*]], pero no *a*[*m*] [*n*]. Observe que *a*[*a*[*n*]] tiene la forma *a*[*E*], en donde *E* es *a*[*n*].

```
t1 = i - j
t2 = t1 + k
```

Con los accesos a arreglos y las asignaciones surge la necesidad de diferenciar entre los *l-values* y los *r-values*. Por ejemplo, $2*a[i]$ puede traducirse convirtiendo el *r-value* de $a[i]$ en un nombre temporal, como en:

```
t1 = a [ i ]
t2 = 2 * t1
```

Pero no podemos simplemente usar un nombre temporal en lugar de $a[i]$, si $a[i]$ aparece en el lado izquierdo de una asignación.

El método simple utiliza las dos funciones *l-value* y *r-value*, que aparecen en las figuras 2.44 y 2.45, respectivamente. Cuando la función *r-value* se aplica a un nodo x que no es hoja, genera instrucciones para convertir a x en un nombre temporal, y devuelve un nuevo nodo que representa a este temporal. Cuando se aplica la función *l-value* a un nodo que no es hoja, también genera instrucciones para calcular los subárboles debajo de x , y devuelve un nodo que representa la “dirección” para x .

Describiremos primero la función *l-value*, ya que tiene menos casos. Al aplicarse a un nodo x , la función *l-value* simplemente devuelve x si es el nodo para un identificador (es decir, si x es de la clase *Id*). En nuestro lenguaje simple, el único otro caso en donde una expresión tiene un *l-value* ocurre cuando x representa un acceso a un arreglo, como $a[i]$. En este caso, x tendrá la forma $Acceso(y, z)$, en donde la clase *Acceso* es una subclase de *Expr*, y representa el nombre del arreglo al que se va a acceder y z representa el desplazamiento (índice) del elemento elegido en ese arreglo. Del seudocódigo en la figura 2.44, la función *l-value* llama a *r-value(z)* para generar instrucciones, si es necesario, para calcular el *r-value* de z . Despues construye y y devuelve un nuevo nodo *Acceso*, con hijos para el nombre del arreglo y y el *r-value* de z .

```
Expr l-value(x : Expr) {
    if ( x es un nodo Id ) return x;
    else if ( x es un nodo Acceso(y, z) y y es un nodo Id ) {
        return new Acceso(y, r-value(z));
    }
    else error;
}
```

Figura 2.44: Seudocódigo para la función *l-value*

Ejemplo 2.19: Cuando el nodo x representa el acceso al arreglo $a[2*k]$, la llamada a *l-value(x)* genera una instrucción

```
t = 2 * k
```

y devuelve un nuevo nodo x' que representa el *l-value* $a[t]$, en donde t es un nuevo nombre temporal.

En detalle, se llega al fragmento de código

```
    return new Acceso(y, r-value(z));
```

en donde y es el nodo para \mathbf{a} y z es el nodo para la expresión $2*k$. La llamada a $r\text{-value}(z)$ genera código para la expresión $2*k$ (es decir, la instrucción de tres direcciones $t = 2 * k$) y devuelve el nuevo nodo z' , que representa al nombre temporal t . Ese nodo z' se convierte en el valor del segundo campo en el nuevo nodo *Acceso* llamado x' que se crea. \square

```
Expr r-value(x : Expr) {
    if (  $x$  es un nodo Id o Constante ) return  $x$ ;
    else if (  $x$  es un nodo Op (op,  $y$ ,  $z$ ) o Rel (op,  $y$ ,  $z$ ) ) {
         $t$  = nuevo temporal;
        emitir cadena para  $t = r\text{-value}(y)$  op  $r\text{-value}(z)$ ;
        return un nuevo nodo para  $t$ ;
    }
    else if (  $x$  es un nodo Acceso ( $y$ ,  $z$ ) ) {
         $t$  = nuevo temporal;
        llamar a l-value( $x$ ), que devuelve Acceso ( $y$ ,  $z'$ );
        emitir cadena para  $t = Acceso$  ( $y$ ,  $z'$ );
        return un nuevo nodo para  $t$ ;
    }
    else if (  $x$  es un nodo Asigna ( $y$ ,  $z$ ) ) {
         $z' = r\text{-value}(z)$ ;
        emitir cadena para l-value( $y$ ) =  $z'$ ;
        return  $z'$ ;
    }
}
```

Figura 2.45: Seudocódigo para la función *r-value*

La función *r-value* en la figura 2.45 genera instrucciones y devuelve un nodo que posiblemente es nuevo. Cuando x representa a un identificador o a una constante, *r-value* devuelve la misma x . En todos los demás casos, devuelve un nodo *Id* para un nuevo nombre temporal t . Los casos son los siguientes:

- Cuando x representa a $y \mathbf{op} z$, el código primero calcula $y' = r\text{-value}(y)$ y $z' = r\text{-value}(z)$. Crea un nuevo nombre temporal t y genera una instrucción $t = y' \mathbf{op} z'$ (dicho en forma más precisa, una instrucción que se forma a partir de las representaciones de cadena de t , y' , *op* y z'). Devuelve un nodo para el identificador t .
- Cuando x representa un acceso a un arreglo $y[z]$, podemos reutilizar la función *l-value*. La llamada a *l-value*(x) devuelve un acceso $y[z']$, en donde z' representa a un identificador que contiene el desplazamiento para el acceso al arreglo. El código crea un nuevo nombre temporal t , genera una instrucción basada en $t = y[z']$ y devuelve un nodo para t .

- Cuando x representa a $y = z$, entonces el código primero calcula $z' = r\text{-value}(z)$. Genera una instrucción con base en $l\text{-value}(y) = z'$ y devuelve el nodo z' .

Ejemplo 2.20: Cuando se aplica al árbol sintáctico para

$a[i] = 2*a[j-k]$

la función $r\text{-value}$ genera

```
t3 = j - k
t2 = a [ t3 ]
t1 = 2 * t2
a [ i ] = t1
```

Es decir, la raíz es un nodo *Asigna* con el primer argumento $a[i]$ y el segundo argumento $2*a[j-k]$. Por ende, se aplica el tercer caso y la función $r\text{-value}$ evalúa en forma recursiva a $2*a[j-k]$. La raíz de este subárbol es el nodo *Op* para $*$, que produce la creación de un nuevo nombre temporal $t1$, antes de que se evalúe el operando izquierdo 2, y después el operando derecho. La constante 2 no genera un código de tres direcciones, y su $r\text{-value}$ se devuelve como nodo *Constante* con el valor 2.

El operando derecho $a[j-k]$ es un nodo *Acceso*, el cual provoca la creación de un nuevo nombre temporal $t2$, antes de que se haga una llamada a la función $l\text{-value}$ en este nodo. Despues se hace una llamada recursiva a $r\text{-value}$ sobre la expresión $j-k$. Como efecto colateral de esta llamada se genera la instrucción de tres direcciones $t3 = j - k$, una vez que se crea el nuevo nombre temporal $t3$. Despues, al regresar a la llamada de $l\text{-value}$ sobre $a[j-k]$, al nombre temporal $t2$ se le asigna el $r\text{-value}$ de la expresión de acceso completa, es decir, $t2 = a [t3]$.

Ahora regresamos a la llamada de $r\text{-value}$ sobre el nodo *Op* $2*a[j-k]$, que anteriormente creó el nombre temporal $t1$. Una instrucción de tres direcciones $t1 = 2 * t2$ se genera como un efecto colateral, para evaluar esta expresión de multiplicación. Por último, la llamada a $r\text{-value}$ en toda la expresión se completa llamando a $l\text{-value}$ sobre el lado izquierdo $a[i]$ y después generando una instrucción de tres direcciones $a [i] = t1$, en donde el lado derecho de la asignación se asigna al lado izquierdo. \square

Mejor código para las expresiones

Podemos mejorar nuestra función $r\text{-value}$ en la figura 2.45 y generar menos instrucciones de tres direcciones, de varias formas:

- Reducir el número de instrucciones de copia en una fase de optimización subsiguiente. Por ejemplo, el par de instrucciones $t = i+1$ y $i = t$ pueden combinarse en $i = i+1$, si no hay usos subsiguientes de t .
- Generar menos instrucciones en primer lugar, tomando en cuenta el contexto. Por ejemplo, si el lado izquierdo de una asignación de tres direcciones es un acceso a un arreglo $a[t]$, entonces el lado derecho debe ser un nombre, una constante o un nombre temporal, cada uno de los cuales sólo utiliza una dirección. Pero si el lado izquierdo es un nombre x , entonces el lado derecho puede ser una operación y $op\ z$ que utilice dos direcciones.

Podemos evitar ciertas instrucciones de copia modificando las funciones de traducción para generar una instrucción parcial que calcule, por ejemplo $j+k$, pero que no se comprometa a indicar en dónde se va a colocar el resultado, lo cual se indica mediante una dirección **null** para el resultado:

$$\mathbf{null} = j + k \quad (2.8)$$

La dirección de resultado nula se sustituye después por un identificador o un nombre temporal, según sea apropiado. Se sustituye por un identificador si $j+k$ está en el lado derecho de una asignación, como en $i=j+k;$, en cuyo caso la expresión (2.8) se convierte en

$$i = j + k$$

Pero, si $j+k$ es una subexpresión, como en $j+k+1$, entonces la dirección de resultado nula en (2.8) se sustituye por un nuevo nombre temporal **t**, y se genera una nueva instrucción parcial:

$$\begin{aligned} t &= j + k \\ \mathbf{null} &= t + 1 \end{aligned}$$

Muchos compiladores realizan todo el esfuerzo posible por generar código que sea tan bueno o mejor que el código ensamblador escrito a mano que producen los expertos. Si se utilizan las técnicas de optimización de código como las del capítulo 9, entonces una estrategia efectiva podría ser utilizar un método simple para la generación de código intermedio, y depender del optimizador de código para eliminar las instrucciones innecesarias.

2.8.5 Ejercicios para la sección 2.8

Ejercicio 2.8.1: Las instrucciones **for** en C y Java tienen la forma:

$$\mathbf{for} (\mathit{expr}_1 ; \mathit{expr}_2 ; \mathit{expr}_3) \mathit{instr}$$

La primera expresión se ejecuta antes del ciclo; por lo general se utiliza para inicializar el índice del ciclo. La segunda expresión es una prueba que se realiza antes de cada iteración del ciclo; se sale del ciclo si la expresión se convierte en 0. El ciclo en sí puede considerarse como la instrucción `{ instr expr_3 ; }`. La tercera expresión se ejecuta al final de cada iteración; por lo general se utiliza para incrementar el índice del ciclo. El significado de la instrucción **for** es similar a:

$$\mathit{expr}_1 ; \mathbf{while} (\mathit{expr}_2) \{ \mathit{instr} \mathit{expr}_3; \}$$

Defina una clase *For* para las instrucciones **for**, de manera similar a la clase *If* en la figura 2.43.

Ejercicio 2.8.2: El lenguaje de programación C no tiene un tipo booleano. Muestre cómo un compilador de C podría traducir una instrucción **if** en código de tres direcciones.

2.9 Resumen del capítulo 2

Las técnicas orientadas a la sintaxis que vimos en este capítulo pueden usarse para construir interfaces de usuario (front-end) de compiladores, como las que se muestran en la figura 2.46.

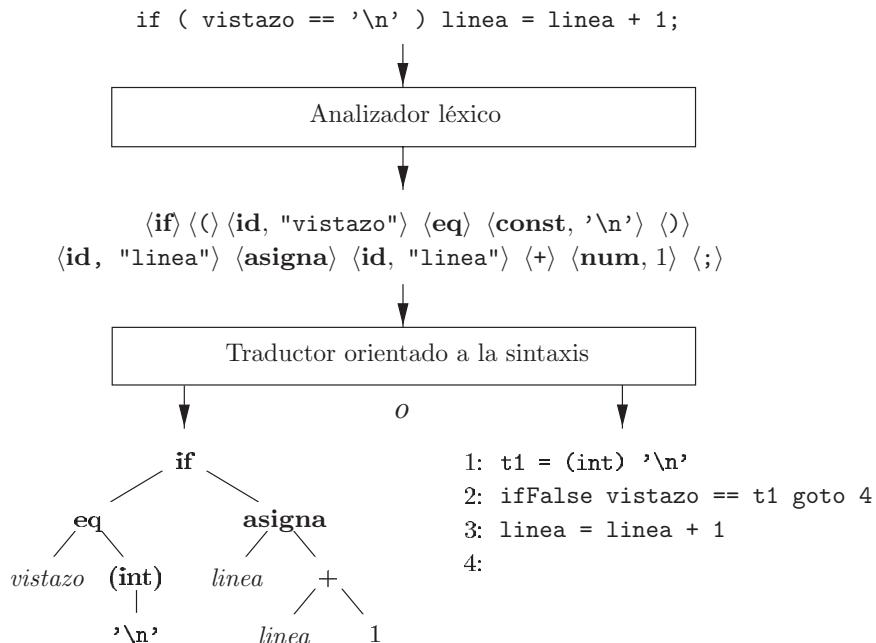


Figura 2.46: Dos posibles traducciones de una instrucción

- ♦ El punto inicial para un traductor orientado a la sintaxis es una gramática para el lenguaje fuente. Una *gramática* describe la estructura jerárquica de los programas. Se define en términos de símbolos elementales, conocidos como *terminales*, y de símbolos variables llamados *no terminales*. Estos símbolos representan construcciones del lenguaje. Las reglas o *producciones* de una gramática consisten en un no terminal conocido como el *encabezado* o *lado izquierdo* de una producción, y de una secuencia de terminales y no terminales, conocida como el *cuerpo* o *lado derecho* de la producción. Un terminal se designa como el símbolo *inicial*.
- ♦ Al especificar un traductor, es conveniente adjuntar atributos a la construcción de programación, en donde un *atributo* es cualquier cantidad asociada con una construcción. Como las construcciones se representan mediante símbolos de la gramática, el concepto de los atributos se extiende a los símbolos de la gramática. Algunos ejemplos de atributos incluyen un valor entero asociado con un terminal **num** que representa números, y una cadena asociada con un terminal **id** que representa identificadores.
- ♦ Un *análizador léxico* lee la entrada un carácter a la vez, y produce como salida un flujo de *tokens*, en donde un token consiste en un símbolo terminal, junto con información adicional en la forma de valores de atributos. En la figura 2.46, los tokens se escriben como n-uplas encerradas entre < >. El token <id, "vistazo"> consiste en la terminal **id** y en un apuntador a la entrada en la tabla de símbolos que contiene la cadena "vistazo". El

traductor utiliza la tabla para llevar la cuenta de las palabras reservadas e identificadores que ya se han analizado.

- ♦ El *análisis sintáctico* es el problema de averiguar cómo puede derivarse una cadena de terminales del símbolo de inicio de la gramática, sustituyendo en forma repetida un no terminal por el cuerpo de una de sus producciones. En general, un analizador sintáctico construye un árbol de análisis sintáctico, en el cual la raíz se etiqueta con el símbolo inicial, cada nodo que no es hoja corresponde a una producción y cada hoja se etiqueta con un terminal o con la cadena vacía, ϵ . El árbol de análisis sintáctico deriva la cadena de terminales en las hojas, y se lee de izquierda a derecha.
- ♦ Los analizadores sintácticos eficientes pueden crearse a mano, mediante un método tipo descendente (de la raíz hasta las hojas de un árbol de análisis sintáctico) llamado análisis sintáctico predictivo. Un *analizador sintáctico predictivo* tiene un procedimiento para cada no terminal; los cuerpos de los procedimientos imitan las producciones para los no terminales; y, el flujo de control a través de los cuerpos de los procedimientos puede determinarse sin ambigüedades, analizando un símbolo de preanálisis en el flujo de entrada. En el capítulo 4 podrá ver otros métodos para el análisis sintáctico.
- ♦ La traducción orientada a la sintaxis se realiza adjuntando reglas o fragmentos de programa a las producciones en una gramática. En este capítulo hemos considerado sólo los atributos *sintetizados*: el valor de un atributo sintetizado en cualquier nodo x puede depender sólo de los atributos en los hijos de x , si los hay. Una *definición orientada a la sintaxis* adjunta reglas a las producciones; las reglas calculan los valores de los atributos. Un *esquema de traducción* incrusta fragmentos de programa llamados *acciones semánticas* en los cuerpos de las producciones. Las acciones se ejecutan en el orden en el que se utilizan las producciones durante el análisis sintáctico.
- ♦ El resultado del análisis sintáctico es una representación del programa fuente, conocido como *código intermedio*. En la figura 2.46 se ilustran dos formas principales de código intermedio. Un *árbol sintáctico abstracto* tiene nodos para las construcciones de programación; los hijos de un nodo proporcionan las subconstrucciones significativas. De manera alternativa, el *código de tres direcciones* es una secuencia de instrucciones, en la cual cada instrucción lleva a cabo una sola operación.
- ♦ Las *tablas de símbolos* son estructuras de datos que contienen información acerca de los identificadores. La información se coloca en la tabla de símbolos cuando se analiza la declaración de un identificador. Una acción semántica obtiene información de la tabla de símbolos cuando el identificador se vuelve a utilizar, por ejemplo, como factor en una expresión.

Capítulo 3

Análisis léxico

En este capítulo le mostraremos cómo construir un analizador léxico. Para implementar un analizador léxico a mano, es útil empezar con un diagrama o cualquier otra descripción de los lexemas de cada token. De esta manera, podemos escribir código para identificar cada ocurrencia de cada lexema en la entrada, y devolver información acerca del token identificado.

Podemos producir también un analizador léxico en forma automática, especificando los patrones de los lexemas a un *generador de analizadores léxicos*, y compilando esos patrones en código que funcione como un analizador léxico. Este método facilita la modificación de un analizador léxico, ya que sólo tenemos que reescribir los patrones afectados, y no todo el programa. Agiliza también el proceso de implementar el analizador, ya que el programador especifica el software en el nivel más alto de los patrones y se basa en el generador para producir el código detallado. En la sección 3.5 presentaremos un generador de analizadores léxicos llamado *Lex* (o *Flex*, en una presentación más reciente).

Empezaremos el estudio de los generadores de analizadores léxicos mediante la presentación de las expresiones regulares, una notación conveniente para especificar los patrones de lexemas. Mostraremos cómo esta notación puede transformarse, primero en autómatas no deterministas y después en autómatas determinista. Estas últimas dos notaciones pueden usarse como entrada para un “controlador”, es decir, código que simula a estos autómatas y los utiliza como guía para determinar el siguiente token. Este control y la especificación del autómata forman el núcleo del analizador léxico.

3.1 La función del analizador léxico

Como la primera fase de un compilador, la principal tarea del analizador léxico es leer los caracteres de la entrada del programa fuente, agruparlos en lexemas y producir como salida una secuencia de tokens para cada lexema en el programa fuente. El flujo de tokens se envía al analizador sintáctico para su análisis. Con frecuencia el analizador léxico interactúa también con la tabla de símbolos. Cuando el analizador léxico descubre un lexema que constituye a un iden-

tificador, debe introducir ese lexema en la tabla de símbolos. En algunos casos, el analizador léxico puede leer la información relacionada con el tipo de información de la tabla de símbolos, como ayuda para determinar el token apropiado que debe pasar al analizador sintáctico.

En la figura 3.1 se sugieren estas interacciones. Por lo regular, la interacción se implementa haciendo que el analizador sintáctico llame al analizador léxico. La llamada, sugerida por el comando *obtenerSiguienteToken*, hace que el analizador léxico lea los caracteres de su entrada hasta que pueda identificar el siguiente lexema y producirlo para el siguiente token, el cual devuelve al analizador sintáctico.

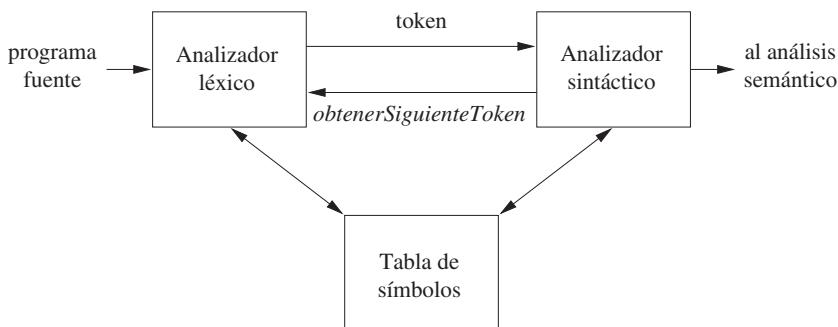


Figura 3.1: Interacciones entre el analizador léxico y el analizador sintáctico

Como el analizador léxico es la parte del compilador que lee el texto de origen, debe realizar otras tareas aparte de identificar lexemas. Una de esas tareas es eliminar los comentarios y el *espacio en blanco* (caracteres de espacio, nueva línea, tabulador y tal vez otros caracteres que se utilicen para separar tokens en la entrada). Otra de las tareas es correlacionar los mensajes de error generados por el compilador con el programa fuente. Por ejemplo, el analizador léxico puede llevar el registro del número de caracteres de nueva línea vistos, para poder asociar un número de línea con cada mensaje de error. En algunos compiladores, el analizador léxico crea una copia del programa fuente con los mensajes de error insertados en las posiciones apropiadas. Si el programa fuente utiliza un preprocesador de macros, la expansión de las macros también pueden formar parte de las tareas del analizador léxico.

Algunas veces, los analizadores léxicos se dividen en una cascada de dos procesos:

- a) El *escaneo* consiste en los procesos simples que no requieren la determinación de tokens de la entrada, como la eliminación de comentarios y la compactación de los caracteres de espacio en blanco consecutivos en uno solo.
- b) El propio *análisis léxico* es la porción más compleja, en donde el escanear produce la secuencia de tokens como salida.

3.1.1 Comparación entre análisis léxico y análisis sintáctico

Existen varias razones por las cuales la parte correspondiente al análisis de un compilador se separa en fases de análisis léxico y análisis sintáctico (parsing).

1. La sencillez en el diseño es la consideración más importante. La separación del análisis léxico y el análisis sintáctico a menudo nos permite simplificar por lo menos una de estas tareas. Por ejemplo, un analizador sintáctico que tuviera que manejar los comentarios y el espacio en blanco como unidades sintácticas sería mucho más complejo que uno que asumiera que el analizador léxico ya ha eliminado los comentarios y el espacio en blanco. Si vamos a diseñar un nuevo lenguaje, la separación de las cuestiones léxicas y sintácticas puede llevar a un diseño más limpio del lenguaje en general.
2. Se mejora la eficiencia del compilador. Un analizador léxico separado nos permite aplicar técnicas especializadas que sirven sólo para la tarea léxica, no para el trabajo del análisis sintáctico. Además, las técnicas de búfer especializadas para leer caracteres de entrada pueden agilizar la velocidad del compilador en forma considerable.
3. Se mejora la portabilidad del compilador. Las peculiaridades específicas de los dispositivos de entrada pueden restringirse al analizador léxico.

3.1.2 Tokens, patrones y lexemas

Al hablar sobre el análisis léxico, utilizamos tres términos distintos, pero relacionados:

- Un *token* es un par que consiste en un nombre de token y un valor de atributo opcional. El nombre del token es un símbolo abstracto que representa un tipo de unidad léxica; por ejemplo, una palabra clave específica o una secuencia de caracteres de entrada que denotan un identificador. Los nombres de los tokens son los símbolos de entrada que procesa el analizador sintáctico. A partir de este momento, en general escribiremos el nombre de un token en negrita. Con frecuencia nos referiremos a un token por su nombre.
- Un *patrón* es una descripción de la forma que pueden tomar los lexemas de un token. En el caso de una palabra clave como token, el patrón es sólo la secuencia de caracteres que forman la palabra clave. Para los identificadores y algunos otros tokens, el patrón es una estructura más compleja que se *relaciona* mediante muchas cadenas.
- Un *lexema* es una secuencia de caracteres en el programa fuente, que coinciden con el patrón para un token y que el analizador léxico identifica como una instancia de ese token.

Ejemplo 3.1: La figura 3.2 proporciona algunos tokens comunes, sus patrones descritos de manera informal y algunos lexemas de ejemplo. La siguiente instrucción en C nos servirá para ver cómo se utilizan estos conceptos en la práctica:

```
printf("Total = %d\n", puntuacion);
```

tanto `printf` como `puntuacion` son lexemas que coinciden con el patrón para el token **id**, y `"Total = %d\n"` es un lexema que coincide con **literal**. □

En muchos lenguajes de programación, las siguientes clases cubren la mayoría, si no es que todos, los tokens:

TOKEN	DESCRIPCIÓN INFORMAL	LEXEMAS DE EJEMPLO
if	caracteres i, f	if
else	caracteres e, l, s, e	Else
comparacion	< o > o <= o >= o == o !=	<=, !=
id	letra seguida por letras y dígitos	pi, puntuacion, D2
numero	cualquier constante numérica	3.14159, 0, 6.02e23
literal	cualquier cosa excepto ", rodeada por "'s	"core dumped"

Figura 3.2: Ejemplos de tokens

1. Un token para cada palabra clave. El patrón para una palabra clave es el mismo que para la palabra clave en sí.
2. Los tokens para los operadores, ya sea en forma individual o en clases como el token **comparacion**, mencionado en la figura 3.2.
3. Un token que representa a todos los identificadores.
4. Uno o más tokens que representan a las constantes, como los números y las cadenas de literales.
5. Tokens para cada signo de puntuación, como los paréntesis izquierdo y derecho, la coma y el signo de punto y coma.

3.1.3 Atributos para los tokens

Cuando más de un lexema puede coincidir con un patrón, el analizador léxico debe proporcionar a las subsiguientes fases del compilador información adicional sobre el lexema específico que coincidió. Por ejemplo, el patrón para el token **numero** coincide con 0 y con 1, pero es en extremo importante para el generador de código saber qué lexema se encontró en el programa fuente. Por ende, en muchos casos el analizador léxico devuelve al analizador sintáctico no sólo el nombre de un token, sino un valor de atributo que describe al lexema que representa ese token; el nombre del token influye en las decisiones del análisis sintáctico, mientras que el valor del atributo influye en la traducción de los tokens después del análisis sintáctico.

Vamos a suponer que los tokens tienen cuando menos un atributo asociado, aunque este atributo tal vez tenga una estructura que combine varias piezas de información. El ejemplo más importante es el token **id**, en donde debemos asociar con el token una gran cantidad de información. Por lo general, la información sobre un identificador (por ejemplo, su lexema, su tipo y la ubicación en la que se encontró por primera vez, en caso de que haya que emitir un mensaje de error sobre ese identificador) se mantiene en la tabla de símbolos. Por lo tanto, el valor de atributo apropiado para un identificador es un apuntador a la entrada en la tabla de símbolos para ese identificador.

Problemas difíciles durante el reconocimiento de tokens

Por lo general, dado el patrón que describe a los lexemas de un token, es muy sencillo reconocer los lexemas que coinciden cuando ocurren en la entrada. No obstante, en algunos lenguajes no es tan evidente cuando hemos visto una instancia de un lexema que corresponde a un token. El siguiente ejemplo se tomó de Fortran, en el formato fijo todavía permitido en Fortran 90. En la siguiente instrucción:

```
DO 5 I = 1.25
```

no es evidente que el primer lexema es D05I, una instancia del token identificador, hasta que vemos el punto que va después del 1. Observe que los espacios en blanco en el lenguaje Fortran de formato fijo se ignoran (una convención arcaica). Si hubiéramos visto una coma en vez del punto, tendríamos una instrucción do:

```
DO 5 I = 1,25
```

en donde el primer lexema es la palabra clave DO.

Ejemplo 3.2: Los nombres de los tokens y los valores de atributo asociados para la siguiente instrucción en Fortran:

```
E = M * C ** 2
```

se escriben a continuación como una secuencia de pares.

```
<id, apuntador a la entrada en la tabla de símbolos para E>
<asigna_op>
<id, apuntador a la entrada en la tabla de símbolos para M>
<mult_op>
<id, apuntador a la entrada en la tabla de símbolos para C>
<exp_op>
<numero, valor entero 2>
```

Observe que en ciertos pares en especial en los operadores, signos de puntuación y palabras clave, no hay necesidad de un valor de atributo. En este ejemplo, el token **numero** ha recibido un atributo con valor de entero. En la práctica, un compilador ordinario almacenaría en su lugar una cadena de caracteres que represente a la constante, y que utilice como valor de atributo para **numero** un apuntador a esa cadena. □

3.1.4 Errores léxicos

Sin la ayuda de los demás componentes es difícil para un analizador léxico saber que hay un error en el código fuente. Por ejemplo, si la cadena **fi** se encuentra por primera vez en un programa en C en el siguiente contexto:

```
fi ( a == f(x)) ...
```

un analizador léxico no puede saber si `fi` es una palabra clave `if` mal escrita, o un identificador de una función no declarada. Como `fi` es un lexema válido para el token `id`, el analizador léxico debe regresar el token `id` al analizador sintáctico y dejar que alguna otra fase del compilador (quizá el analizador sintáctico en este caso) mande un error debido a la transposición de las letras.

Sin embargo, suponga que surge una situación en la cual el analizador léxico no puede proceder, ya que ninguno de los patrones para los tokens coincide con algún prefijo del resto de la entrada. La estrategia de recuperación más simple es la recuperación en “modo de pánico”. Eliminamos caracteres sucesivos del resto de la entrada, hasta que el analizador léxico pueda encontrar un token bien formado al principio de lo que haya quedado de entrada. Esta técnica de recuperación puede confundir al analizador sintáctico, pero en un entorno de computación interactivo, puede ser bastante adecuado.

Otras de las posibles acciones de recuperación de errores son:

1. Eliminar un carácter del resto de la entrada.
2. Insertar un carácter faltante en el resto de la entrada.
3. Sustituir un carácter por otro.
4. Transponer dos caracteres adyacentes.

Las transformaciones como éstas pueden probarse en un intento por reparar la entrada. La estrategia más sencilla es ver si un prefijo del resto de la entrada puede transformarse en un lexema válido mediante una transformación simple. Esta estrategia tiene sentido, ya que en la práctica la mayoría de los errores léxicos involucran a un solo carácter. Una estrategia de corrección más general es encontrar el menor número de transformaciones necesarias para convertir el programa fuente en uno que consista sólo de lexemas válidos, pero este método se considera demasiado costoso en la práctica como para que valga la pena realizarlo.

3.1.5 Ejercicios para la sección 3.1

Ejercicio 3.1.1: Divida el siguiente programa en C++:

```
float cuadradoLimitado(x) float x {
    /* devuelve x al cuadrado, pero nunca más de 100 */
    return (x<=-10.0||x>=10.0)?100:x*x;
}
```

en lexemas apropiados, usando la explicación de la sección 3.1.2 como guía. ¿Qué lexemas deberían obtener valores léxicos asociados? ¿Cuáles deberían ser esos valores?

! Ejercicio 3.1.2: Los lenguajes Indecidibles como HTML o XML son distintos de los de programación convencionales, en que la puntuación (etiquetas) es muy numerosa (como en HTML) o es un conjunto definible por el usuario (como en XML). Además, a menudo las etiquetas pueden tener parámetros. Sugiera cómo dividir el siguiente documento de HTML:

```

He aquí una foto de <B>mi casa</B>:
<P><IMG SRC = "casa.gif"><BR>
Vea <A HREF = "masImg.html">Más Imágenes</A> si
le gustó ésa.<P>

```

en los lexemas apropiados. ¿Qué lexemas deberían obtener valores léxicos asociados, y cuáles deberían ser esos valores?

3.2 Uso de búfer en la entrada

Antes de hablar sobre el problema de reconocer lexemas en la entrada, vamos a examinar algunas formas en las que puede agilizarse la simple pero importante tarea de leer el programa fuente. Esta tarea se dificulta debido a que a menudo tenemos que buscar uno o más caracteres más allá del siguiente lexema para poder estar seguros de que tenemos el lexema correcto. El recuadro titulado “Problemas difíciles durante el reconocimiento de tokens” en la sección 3.1 nos dio un ejemplo extremo, pero hay muchas situaciones en las que debemos analizar por lo menos un carácter más por adelantado. Por ejemplo, no podemos estar seguros de haber visto el final de un identificador sino hasta ver un carácter que no es letra ni dígito, y que, por lo tanto, no forma parte del lexema para **id**. En C, los operadores de un solo carácter como `-`, `=` o `<` podrían ser también el principio de un operador de dos caracteres, como `->`, `==` o `<=`. Por ende, vamos a presentar un esquema de dos búferes que se encarga de las lecturas por adelantado extensas sin problemas. Después consideraremos una mejora en la que se utilizan “centinelas” para ahorrar tiempo al verificar el final de los búferes.

3.2.1 Pares de búferes

Debido al tiempo requerido para procesar caracteres y al extenso número de caracteres que se deben procesar durante la compilación de un programa fuente extenso, se han desarrollado técnicas especializadas de uso de búferes para reducir la cantidad de sobrecarga requerida en el procesamiento de un solo carácter de entrada. Un esquema importante implica el uso de dos búferes que se recargan en forma alterna, como se sugiere en la figura 3.3.

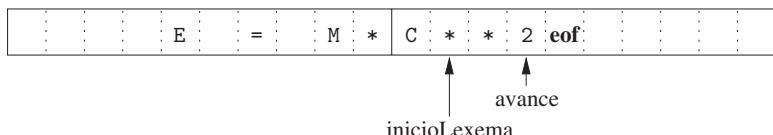


Figura 3.3: Uso de un par de búferes de entrada

Cada búfer es del mismo tamaño N , y por lo general N es del tamaño de un bloque de disco (es decir, 4 096 bytes). Mediante el uso de un comando de lectura del sistema podemos leer N caracteres y colocarlos en un búfer, en vez de utilizar una llamada al sistema por cada carácter. Si quedan menos de N caracteres en el archivo de entrada, entonces un carácter especial,

representado por **eof**, marca el final del archivo fuente y es distinto a cualquiera de los posibles caracteres del programa fuente.

Se mantienen dos apuntadores a la entrada:

1. El apuntador **inicioLexema** marca el inicio del lexema actual, cuya extensión estamos tratando de determinar.
2. El apuntador **avance** explora por adelantado hasta encontrar una coincidencia en el patrón; durante el resto del capítulo cubriremos la estrategia exacta mediante la cual se realiza esta determinación.

Una vez que se determina el siguiente lexema, **avance** se coloca en el carácter que se encuentra en su extremo derecho. Después, una vez que el lexema se registra como un valor de atributo de un token devuelto al analizador sintáctico, **inicioLexema** se coloca en el carácter que va justo después del lexema que acabamos de encontrar. En la figura 3.3 vemos que **avance** ha pasado del final del siguiente lexema, ****** (el operador de exponenciación en Fortran), y debe retractarse una posición a su izquierda.

Para desplazar a **avance** hacia delante primero tenemos que probar si hemos llegado al final de uno de los búferes, y de ser así, debemos recargar el otro búfer de la entrada, y mover **avance** al principio del búfer recién cargado. Siempre y cuando no tengamos que alejarnos tanto del lexema como para que la suma de su longitud más la distancia que nos alejamos sea mayor que N , nunca habrá peligro de sobrescribir el lexema en su búfer antes de poder determinarlo.

3.2.2 Centinelas

Si utilizamos el esquema de la sección 3.2.1 en la forma descrita, debemos verificar, cada vez que movemos el apuntador **avance**, que no nos hayamos salido de uno de los búferes; si esto pasa, entonces también debemos recargar el otro búfer. Así, por cada lectura de caracteres hacemos dos pruebas: una para el final del búfer y la otra para determinar qué carácter se lee (esta última puede ser una bifurcación de varias vías). Podemos combinar la prueba del final del búfer con la prueba del carecer actual si extendemos cada búfer para que contenga un valor **centinela** al final. El centinela es un carácter especial que no puede formar parte del programa fuente, para lo cual una opción natural es el carácter **eof**.

La figura 3.4 muestra el mismo arreglo que la figura 3.3, pero con los centinelas agregados. Observe que **eof** retiene su uso como marcador del final de toda la entrada. Cualquier **eof** que aparezca en otra ubicación distinta al final de un búfer significa que llegamos al final de la entrada. La figura 3.5 sintetiza el algoritmo para mover **avance** hacia delante. Observe cómo la primera prueba, que puede formar parte de una bifurcación de varias vías con base en el carácter al que apunta **avance**, es la única prueba que hacemos, excepto en el caso en el que en realidad nos encontramos al final de un búfer o de la entrada.

3.3 Especificación de los tokens

Las expresiones regulares son una notación importante para especificar patrones de lexemas. Aunque no pueden expresar todos los patrones posibles, son muy efectivas para especificar los tipos de patrones que en realidad necesitamos para los tokens. En esta sección estudiaremos

¿Se nos puede acabar el espacio de los búferes?

En la mayoría de los lenguajes modernos, los lexemas son cortos y basta con uno o dos caracteres de lectura adelantada. Por ende, un tamaño de búfer N alrededor de los miles es más que suficiente, y el esquema de doble búfer de la sección 3.2.1 funciona sin problemas. No obstante, existen ciertos riesgos. Por ejemplo, si las cadenas de caracteres pueden ser muy extensas, que pueden extenderse en varias líneas, entonces podríamos enfrentarnos a la posibilidad de que un lexema sea más grande que N . Para evitar problemas con las cadenas de caracteres extensas, podemos tratarlas como una concatenación de componentes, uno de cada línea sobre la cual se escribe la cadena. Por ejemplo, en Java es convencional representar las cadenas extensas escribiendo una parte en cada línea y concatenando las partes con un operador `+` al final de cada parte.

Un problema más difícil ocurre cuando pueda ser necesario un proceso de lectura adelantada arbitrariamente extenso. Por ejemplo, algunos lenguajes como PL/I no tratan a las palabras clave como *reservadas*; es decir, podemos usar identificadores con el mismo nombre que una palabra clave como `DECLARE`. Si presentamos al analizador léxico el texto de un programa en PL/I que empiece como `DECLARE (ARG1, ARG2, ...` no puede estar seguro si `DECLARE` es una palabra clave y `ARG1`, etcétera son variables que se están declarando, o si `DECLARE` es el nombre de un procedimiento con sus argumentos. Por esta razón, los lenguajes modernos tienden a reservar sus palabras clave. Pero si no lo hacen, podemos tratar a una palabra clave como `DECLARE` como un identificador ambiguo, y dejar que el analizador sintáctico resuelva esta cuestión, tal vez en conjunto con la búsqueda en la tabla de símbolos.

la notación formal para las expresiones regulares, y en la sección 3.5 veremos cómo se utilizan estas expresiones en un generador de analizadores léxicos. Después, en la sección 3.7 veremos cómo construir el analizador léxico, convirtiendo las expresiones regulares en un autómata que realice el reconocimiento de los tokens especificados.

3.3.1 Cadenas y lenguajes

Un *alfabeto* es un conjunto finito de símbolos. Algunos ejemplos típicos de símbolos son las letras, los dígitos y los signos de puntuación. El conjunto $\{0, 1\}$ es el *alfabeto binario*. ASCII es un ejemplo importante de un alfabeto; se utiliza en muchos sistemas de software. Unicode,

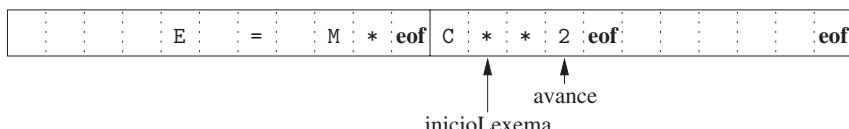


Figura 3.4: Centinelas al final de cada búfer

```

switch ( *avance++ ) {
    case eof:
        if (avance está al final del primer búfer ) {
            recargar el segundo búfer;
            avance = inicio del segundo búfer;
        }
        else if (avance está al final del segundo búfer ) {
            recargar el primer búfer;
            avance = inicio del primer búfer;
        }
        else /* eof dentro de un búfer marca el final de la entrada */
            terminar el análisis léxico;
        break;
    Casos para los demás caracteres
}

```

Figura 3.5: Código de lectura por adelantado con centinelas

Implementación de bifurcaciones de varias vías

Podríamos imaginar que la instrucción `switch` en la figura 3.5 requiere muchos pasos para ejecutarse, y que colocar el caso `eof` primero no es una elección inteligente. En realidad, no importa en qué orden presentemos los casos para cada carácter. En la práctica, una bifurcación de varias vías, dependiendo del carácter de entrada, se realiza en un paso, saltando a una dirección encontrada en un arreglo de direcciones, indexado mediante caracteres.

que incluye aproximadamente 10 000 caracteres de los alfabetos alrededor del mundo, es otro ejemplo importante de un alfabeto.

Una *cadena* sobre un alfabeto es una secuencia finita de símbolos que se extraen de ese alfabeto. En la teoría del lenguaje, los términos “oración” y “palabra” a menudo se utilizan como sinónimos de “cadena”. La longitud de una cadena s , que por lo general se escribe como $|s|$, es el número de ocurrencias de símbolos en s . Por ejemplo, `banana` es una cadena con una longitud de seis. La *cadena vacía*, representada por ϵ , es la cadena de longitud cero.

Un *lenguaje* es cualquier conjunto contable de cadenas sobre algún alfabeto fijo. Esta definición es demasiado amplia. Los lenguajes abstractos como \emptyset , el *conjunto vacío*, o $\{\epsilon\}$, el conjunto que contiene sólo la cadena vacía, son lenguajes bajo esta definición. También lo son el conjunto de todos los programas en C que están bien formados en sentido sintáctico, y el conjunto de todas las oraciones en inglés gramáticamente correctas, aunque estos últimos dos lenguajes son difíciles de especificar con exactitud. Observe que la definición de “lenguaje” no requiere que se atribuya algún significado a las cadenas en el lenguaje. En el capítulo 5 veremos los métodos para definir el “significado” de las cadenas.

Términos para partes de cadenas

Los siguientes términos relacionados con cadenas son de uso común:

1. Un *prefijo* de la cadena s es cualquier cadena que se obtiene al eliminar cero o más símbolos del final de s . Por ejemplo, **ban**, **banana** y ϵ son prefijos de **banana**.
2. Un *sufijo* de la cadena s es cualquier cadena que se obtiene al eliminar cero o más símbolos del principio de s . Por ejemplo, **nana**, **banana** y ϵ son sufijos de **banana**.
3. Una *subcadena* de s se obtiene al eliminar cualquier prefijo y cualquier sufijo de s . Por ejemplo, **banana**, **nan** y ϵ son subcadenas de **banana**.
4. Los prefijos, sufijos y subcadenas *propios* de una cadena s son esos prefijos, sufijos y subcadenas, respectivamente, de s que no son ϵ ni son iguales a la misma s .
5. Una *subsecuencia* de s es cualquier cadena que se forma mediante la eliminación de cero o más posiciones no necesariamente consecutivas de s . Por ejemplo, **baan** es una subsecuencia de **banana**.

Si x y y son cadenas, entonces la *concatenación* de x y y , denotada por xy , es la cadena que se forma al unir y con x . Por ejemplo, si $x = \text{super}$ y $= \text{mercado}$, entonces $xy = \text{supermercado}$. La cadena vacía es la identidad en la concatenación; es decir, para cualquier cadena s , $\epsilon s = s\epsilon = s$.

Si pensamos en la concatenación como un producto, podemos definir la “exponenciación” de cadenas de la siguiente forma. Defina a s^0 para que sea ϵ , y para todas las $i > 0$, defina a s^i para que sea $s^{i-1}s$. Como $\epsilon s = s$, resulta que $s^1 = s$. Entonces $s^2 = ss$, $s^3 = sss$, y así sucesivamente.

3.3.2 Operaciones en los lenguajes

En el análisis léxico, las operaciones más importantes en los lenguajes son la unión, la concatenación y la cerradura, las cuales se definen de manera formal en la figura 3.6. La unión es la operación familiar que se hace con los conjuntos. La concatenación de lenguajes es cuando se concatenan todas las cadenas que se forman al tomar una cadena del primer lenguaje y una cadena del segundo lenguaje, en todas las formas posibles. La *cerradura (Kleene)* de un lenguaje L , que se denota como L^* , es el conjunto de cadenas que se obtienen al concatenar L cero o más veces. Observe que L^0 , la “concatenación de L cero veces”, se define como $\{\epsilon\}$, y por inducción, L^i es $L^{i-1}L$. Por último, la cerradura positiva, denotada como L^+ , es igual que la cerradura de Kleene, pero sin el término L^0 . Es decir, ϵ no estará en L^+ a menos que esté en el mismo L .

OPERACIÓN	DEFINICIÓN Y NOTACIÓN
Unión de L y M	$L \cup M = \{ s \mid s \text{ está en } L \text{ o } s \text{ está en } M \}$
Concatenación de L y M	$LM = \{ st \mid s \text{ está en } L \text{ y } t \text{ está en } M \}$
Cerradura de Kleene de L	$L^* = \bigcup_{i=0}^{\infty} L^i$
Cerradura positivo de L	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Figura 3.6: Definiciones de las operaciones en los lenguajes

Ejemplo 3.3: Sea L el conjunto de letras $\{A, B, \dots, Z, a, b, \dots, z\}$ y sea D el conjunto de dígitos $\{0, 1, \dots, 9\}$. Podemos pensar en L y D de dos formas, en esencia equivalentes. Una forma es que L y D son, respectivamente, los alfabetos de las letras mayúsculas y minúsculas, y de los dígitos. La segunda forma es que L y D son lenguajes cuyas cadenas tienen longitud de uno. He aquí algunos otros lenguajes que pueden construirse a partir de los lenguajes L y D , mediante los operadores de la figura 3.6:

1. $L \cup D$ es el conjunto de letras y dígitos; hablando en sentido estricto, el lenguaje con 62 cadenas de longitud uno, y cada una de las cadenas es una letra o un dígito.
2. LD es el conjunto de 520 cadenas de longitud dos, cada una de las cuales consiste en una letra, seguida de un dígito.
3. L^4 es el conjunto de todas las cadenas de 4 letras.
4. L^* es el conjunto de todas las cadenas de letras, incluyendo ϵ , la cadena vacía.
5. $L(L \cup D)^*$ es el conjunto de todas las cadenas de letras y dígitos que empiezan con una letra.
6. D^+ es el conjunto de todas las cadenas de uno o más dígitos.

□

3.3.3 Expresiones regulares

Suponga que deseamos describir el conjunto de identificadores válidos de C. Es casi el mismo lenguaje descrito en el punto (5) anterior; la única diferencia es que se incluye el guión bajo entre las letras.

En el ejemplo 3.3, pudimos describir los identificadores proporcionando nombres a los conjuntos de letras y dígitos, y usando los operadores de unión, concatenación y cerradura del lenguaje. Este proceso es tan útil que se ha empezado a utilizar comúnmente una notación conocida como *expresiones regulares*, para describir a todos los lenguajes que puedan construirse a partir de estos operadores, aplicados a los símbolos de cierto alfabeto. En esta notación, si *letra*_ se establece de manera que represente a cualquier letra o al guión bajo, y *dígito*_ se

establece de manera que represente a cualquier dígito, entonces podríamos describir el lenguaje de los identificadores de C mediante lo siguiente:

$$\text{letra_} (\text{ letra_} \mid \text{dígito})^*$$

La barra vertical de la expresión anterior significa la unión, los paréntesis se utilizan para agrupar las subexpresiones, el asterisco indica “cero o más ocurrencias de”, y la yuxtaposición de *letra_* con el resto de la expresión indica la concatenación.

Las expresiones regulares se construyen en forma recursiva a partir de las expresiones regulares más pequeñas, usando las reglas que describiremos a continuación. Cada expresión regular *r* denota un lenguaje $L(r)$, el cual también se define en forma recursiva, a partir de los lenguajes denotados por las subexpresiones de *r*. He aquí las reglas que definen las expresiones regulares sobre cierto alfabeto Σ , y los lenguajes que denotan dichas expresiones.

BASE: Hay dos reglas que forman la base:

1. ϵ es una expresión regular, y $L(\epsilon)$ es $\{\epsilon\}$; es decir, el lenguaje cuyo único miembro es la cadena vacía.
2. Si a es un símbolo en Σ , entonces **a** es una expresión regular, y $L(\mathbf{a}) = \{a\}$, es decir, el lenguaje con una cadena, de longitud uno, con a en su única posición. Tenga en cuenta que por convención usamos cursiva para los símbolos, y negrita para su correspondiente expresión regular.¹

INDUCCIÓN: Hay cuatro partes que constituyen la inducción, mediante la cual las expresiones regulares más grandes se construyen a partir de las más pequeñas. Suponga que *r* y *s* son expresiones regulares que denotan a los lenguajes $L(r)$ y $L(s)$, respectivamente.

1. $(r)|(s)$ es una expresión regular que denota el lenguaje $L(r) \cup L(s)$.
2. $(r)(s)$ es una expresión regular que denota el lenguaje $L(r)L(s)$.
3. $(r)^*$ es una expresión regular que denota a $(L(r))^*$.
4. (r) es una expresión regular que denota a $L(r)$. Esta última regla dice que podemos agregar pares adicionales de paréntesis alrededor de las expresiones, sin cambiar el lenguaje que denotan.

Según su definición, las expresiones regulares a menudo contienen pares innecesarios de paréntesis. Tal vez sea necesario eliminar ciertos pares de paréntesis, si adoptamos las siguientes convenciones:

- a) El operador unario $*$ tiene la precedencia más alta y es asociativo a la izquierda.
- b) La concatenación tiene la segunda precedencia más alta y es asociativa a la izquierda.

¹Sin embargo, al hablar sobre los caracteres específicos del conjunto de caracteres ASCII, por lo general, usaremos la fuente de teletipo para el carácter y su expresión regular.

- c) | tiene la precedencia más baja y es asociativo a la izquierda.

Por ejemplo, bajo estas convenciones podemos sustituir la expresión regular $(\mathbf{a})|((\mathbf{b})^*(\mathbf{c}))$ por $\mathbf{a}|\mathbf{b}^*\mathbf{c}$. Ambas expresiones denotan el conjunto de cadenas que son una sola a o cero o más b s seguidas por una c .

Ejemplo 3.4: Sea $\Sigma = \{a, b\}$.

1. La expresión regular $\mathbf{a}|\mathbf{b}$ denota el lenguaje $\{a, b\}$.
2. $(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$ denota a $\{aa, ab, ba, bb\}$, el lenguaje de todas las cadenas de longitud dos sobre el alfabeto Σ . Otra expresión regular para el mismo lenguaje sería $\mathbf{a}\mathbf{a}|\mathbf{a}\mathbf{b}|\mathbf{b}\mathbf{a}|\mathbf{b}\mathbf{b}$.
3. \mathbf{a}^* denota el lenguaje que consiste en todas las cadenas de cero o más a s, es decir, $\{\epsilon, a, aa, aaa, \dots\}$.
4. $(\mathbf{a}|\mathbf{b})^*$ denota el conjunto de todas las cadenas que consisten en cero o más instancias de a o b , es decir, todas las cadenas de a s y b s: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$. Otra expresión regular para el mismo lenguaje es $(\mathbf{a}^*\mathbf{b}^*)^*$.
5. $\mathbf{a}|\mathbf{a}^*\mathbf{b}$ denota el lenguaje $\{a, b, ab, aab, aaab, \dots\}$, es decir, la cadena a y todas las cadenas que consisten en cero o más a s y que terminan en b .

□

A un lenguaje que puede definirse mediante una expresión regular se le llama *conjunto regular*. Si dos expresiones regulares r y s denotan el mismo conjunto regular, decimos que son *equivalentes* y escribimos $r = s$. Por ejemplo, $(\mathbf{a}|\mathbf{b}) = (\mathbf{b}|\mathbf{a})$. Hay una variedad de leyes algebraicas para las expresiones regulares; cada ley afirma que las expresiones de dos formas distintas son equivalentes. La figura 3.7 muestra parte de las leyes algebraicas para las expresiones regulares arbitrarias r , s y t .

LEY	DESCRIPCIÓN
$r s = s r$	es comutativo
$r (s t) = (r s) t$	es asociativo
$r(st) = (rs)t$	La concatenación es asociativa
$r(s t) = rs rt; (s t)r = sr tr$	La concatenación se distribuye sobre
$\epsilon r = r\epsilon = r$	ϵ es la identidad para la concatenación
$r^* = (r \epsilon)^*$	ϵ se garantiza en un cerradura
$r^{**} = r^*$	* es idempotente

Figura 3.7: Leyes algebraicas para las expresiones regulares

3.3.4 Definiciones regulares

Por conveniencia de notación, tal vez sea conveniente poner nombres a ciertas expresiones regulares, y utilizarlos en las expresiones subsiguientes, como si los nombres fueran símbolos por sí mismos. Si Σ es un alfabeto de símbolos básicos, entonces una *definición regular* es una secuencia de definiciones de la forma:

$$\begin{array}{ll} d_1 & \rightarrow r_1 \\ d_2 & \rightarrow r_2 \\ \dots & \\ d_n & \rightarrow r_n \end{array}$$

en donde:

1. Cada d_i es un nuevo símbolo, que no está en Σ y no es el mismo que cualquier otro d .
2. Cada r_i es una expresión regular sobre el alfabeto $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$.

Al restringir r_i a Σ y a las ds definidas con anterioridad, evitamos las definiciones recursivas y podemos construir una expresión regular sobre Σ solamente, para cada r_i . Para ello, primero sustituimos los usos de d_1 en r_2 (que no puede usar ninguna de las ds , excepto d_1), después sustituimos los usos de d_1 y d_2 en r_3 por r_1 y (la sustitución de) r_2 , y así en lo sucesivo. Por último, en r_n sustituimos cada d_i , para $i = 1, 2, \dots, n-1$, por la versión sustituida de r_i , cada una de las cuales sólo tiene símbolos de Σ .

Ejemplo 3.5: Los identificadores de C son cadenas de letras, dígitos y guiones bajos. He aquí una definición regular para el lenguaje de los identificadores de C. Por convención, usaremos cursiva para los símbolos definidos en las definiciones regulares.

$$\begin{array}{ll} letra_ & \rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z \mid - \\ dígito & \rightarrow 0 \mid 1 \mid \dots \mid 9 \\ id & \rightarrow letra_ (letra_ \mid dígito)^* \end{array}$$

□

Ejemplo 3.6: Los números sin signo (enteros o de punto flotante) son cadenas como 5280, 0.01234, 6.336E4 o 1.89E-4. La definición regular

$$\begin{array}{ll} digito & \rightarrow 0 \mid 1 \mid \dots \mid 9 \\ digitos & \rightarrow digito \ digito^* \\ fraccionOpcional & \rightarrow . \ digitos \mid \epsilon \\ exponenteOpcional & \rightarrow (E (+ \mid - \mid \epsilon) \ digitos) \mid \epsilon \\ numero & \rightarrow digitos \ fraccionOpcional \ exponenteOpcional \end{array}$$

es una especificación precisa para este conjunto de cadenas. Es decir, una *fraccionOpcional* es un punto decimal (punto) seguido de uno o más dígitos, o se omite (la cadena vacía). Un *exponenteOpcional*, si no se omite, es la letra E seguida de un signo opcional + o -, seguido de uno o más dígitos. Observe que por lo menos un dígito debe ir después del punto, de manera que *numero* no coincide con 1., pero sí con 1.0. □

3.3.5 Extensiones de las expresiones regulares

Desde que Kleene introdujo las expresiones regulares con los operadores básicos para la unión, la concatenación y la cerradura de Kleene en la década de 1950, se han agregado muchas extensiones a las expresiones regulares para mejorar su habilidad al especificar los patrones de cadenas. Aquí mencionamos algunas extensiones rotacionales que se incorporaron por primera vez en las herramientas de Unix como **Lex**, que son muy útiles en la especificación de los analizadores léxicos. Las referencias a este capítulo contienen una explicación de algunas variantes de expresiones regulares en uso hoy en día.

1. *Una o más instancias.* El operador unario postfijo $^+$ representa la cerradura positivo de una expresión regular y su lenguaje. Es decir, si r es una expresión regular, entonces $(r)^+$ denota el lenguaje $(L(r))^+$. El operador $^+$ tiene la misma precedencia y asociatividad que el operador $*$. Dos leyes algebraicas útiles, $r^* = r^+|\epsilon$ y $r^+ = rr^* = r^*r$ relacionan la cerradura de Kleene y la cerradura positiva.
2. *Cero o una instancia.* El operador unario postfijo $?$ significa “cero o una ocurrencia”. Es decir, $r?$ es equivalente a $r|\epsilon$, o dicho de otra forma, $L(r?) = L(r) \cup \{\epsilon\}$. El operador $?$ tiene la misma precedencia y asociatividad que $*$ y $^+$.
3. *Clases de caracteres.* Una expresión regular $a_1|a_2|\dots|a_n$, en donde las a_i s son cada una símbolos del alfabeto, puede sustituirse mediante la abreviación $[a_1a_2\dots a_n]$. Lo que es más importante, cuando a_1, a_2, \dots, a_n forman una secuencia lógica, por ejemplo, letras mayúsculas, minúsculas o dígitos consecutivos, podemos sustituirlos por a_1-a_n ; es decir, sólo la primera y última separadas por un guión corto. Así, **[abc]** es la abreviación para **a|b|c**, y **[a-z]** lo es para **a|b|...|z**.

Ejemplo 3.7: Mediante estas abreviaciones, podemos rescribir la definición regular del ejemplo 3.5 como;

$$\begin{array}{lcl} letra_ & \rightarrow & [A-Za-z_] \\ digito & \rightarrow & [0-9] \\ id & \rightarrow & letra_ (letra_ | digito)^* \end{array}$$

La definición regular del ejemplo 3.6 también puede simplificarse:

$$\begin{array}{lcl} digito & \rightarrow & [0-9] \\ digitos & \rightarrow & digito^+ \\ numero & \rightarrow & digitos (. digitos)? (E [+-]? digitos)? \end{array}$$

3.3.6 Ejercicios para la sección 3.3

Ejercicio 3.3.1: Consulte los manuales de referencia del lenguaje para determinar (i) los conjuntos de caracteres que forman el alfabeto de entrada (excluyendo aquellos que sólo puedan aparecer en cadenas de caracteres o comentarios), (ii) la forma léxica de las constantes numéricas, y (iii) la forma léxica de los identificadores, para cada uno de los siguientes lenguajes: (a) C (b) C++ (c) C# (d) Fortran (e) Java (f) Lisp (g) SQL.

! Ejercicio 3.3.2: Describa los lenguajes denotados por las siguientes expresiones regulares:

- a) $a(a|b)^*a$.
- b) $((\epsilon|a)b^*)^*$.
- c) $(a|b)^*a(a|b)(a|b)$.
- d) $a^*ba^*ba^*ba^*$.
- !! e) $(aa|bb)^*((ab|ba)(aa|bb)^*(ab|ba)(aa|bb)^*)^*$.

Ejercicio 3.3.3: En una cadena de longitud n , ¿cuántos de los siguientes hay?

- a) Prefijos.
- b) Sufijos.
- c) Prefijos propios.
- ! d) Subcadenas.
- ! e) Subsecuencias.

Ejercicio 3.3.4: La mayoría de los lenguajes son *sensibles a mayúsculas y minúsculas*, por lo que las palabras clave sólo pueden escribirse de una forma, y las expresiones regulares que describen su lexema son muy simples. No obstante, algunos lenguajes como SQL son *insensibles a mayúsculas y minúsculas*, por lo que una palabra clave puede escribirse en minúsculas o en mayúsculas, o en cualquier mezcla de ambas. Por ende, la palabra clave `SELECT` de SQL también puede escribirse como `select`, `Select` o `sElEcT`. Muestre cómo escribir una expresión regular para una palabra clave en un lenguaje insensible a mayúsculas y minúsculas. Ilustre la idea escribiendo la expresión para “select” en SQL.

! Ejercicio 3.3.5: Escriba definiciones regulares para los siguientes lenguajes:

- a) Todas las cadenas de letras en minúsculas que contengan las cinco vocales en orden.
- b) Todas las cadenas de letras en minúsculas, en las que las letras se encuentren en orden lexicográfico ascendente.
- c) Comentarios, que consistan de una cadena rodeada por `/*` y `*/`, sin un `*/` entre ellos, a menos que se encierre entre dobles comillas (").

- !! d) Todas las cadenas de dígitos sin dígitos repetidos. *Sugerencia:* Pruebe este problema primero con unos cuantos dígitos, como $\{0, 1, 2\}$.
- !! e) Todas las cadenas de dígitos que tengan por lo menos un dígito repetido.
- !! f) Todas las cadenas de as y bs con un número par de as y un número impar de bs .
- g) El conjunto de movimientos de Ajedrez, en la notación informal, como $p-k4$ o $kbp \times qn$.
- !! h) Todas las cadenas de as y bs que no contengan la subcadena abb .
- i) Todas las cadenas de as y bs que no contengan la subsecuencia abb .

Ejercicio 3.3.6: Escriba clases de caracteres para los siguientes conjuntos de caracteres:

- a) Las primeras diez letras (hasta “j”), ya sea en mayúsculas o en minúsculas.
- b) Las consonantes en minúsculas.
- c) Los “dígitos” en un número hexadecimal (elija mayúsculas o minúsculas para los “dígitos” mayores a 9).
- d) Los caracteres que pueden aparecer al final de una oración legítima en inglés (por ejemplo, el signo de admiración).

Los siguientes ejercicios, hasta e incluyendo el ejercicio 3.3.10, tratan acerca de la notación de expresiones regulares extendidas de Lex (el generador de analizadores léxicos que veremos con detalle en la sección 3.5). La notación extendida se presenta en la figura 3.8.

Ejercicio 3.3.7: Observe que estas expresiones regulares proporcionan a todos los siguientes símbolos (*caracteres de operadores*) un significado especial:

`\ " . ^ $ [] * + ? { } | /`

Su significado especial debe desactivarse si se necesitan para representarse a sí mismos en una cadena de caracteres. Para ello, debemos colocar el carácter entre comillas, dentro de una cadena de longitud uno o más; por ejemplo, la expresión regular `"**"` coincide con la cadena `**`. También podemos obtener el significado literal de un carácter de operador si le anteponemos una barra diagonal inversa. Por ende, la expresión regular `**` también coincide con la cadena `**`. Escriba una expresión regular que coincida con la cadena `"\."`.

Ejercicio 3.3.8: En Lex, una *clase de carácter complementado* representa a cualquier carácter, excepto los que se listan en la clase de carácter. Denotamos a un carácter complementado mediante el uso de `^` como el primer carácter; este símbolo no forma en sí parte de la clase que se está complementando, a menos que se liste dentro de la misma clase. Así, `[^A-Za-z]` coincide con cualquier carácter que no sea una letra mayúscula o minúscula, y `[^\^]` representa a cualquier carácter excepto `^` (o nueva línea, ya que el carácter nueva línea no puede estar en ninguna clase de caracteres). Muestre que para cualquier expresión regular con clases de caracteres complementados, hay una expresión regular equivalente sin clases de caracteres complementados.

EXPRESIÓN	COINCIDE CON	EJEMPLO
c	un carácter c que no sea operador	a
$\backslash c$	el carácter c , literalmente	*
$"s"$	la cadena s , literalmente	"**"
$.$	cualquier carácter excepto nueva línea	a.*b
$^$	el inicio de una línea	^abc
$\$$	el final de una línea	abc\$
$[s]$	cualquiera de los caracteres en la cadena s	[abc]
$[^s]$	cualquier carácter que no esté en la cadena s	[^abc]
r^*	cero o más cadenas que coincidan con r	a*
r^+	una o más cadenas que coincidan con r	a+
$r^?$	cero o una r	a?
$r\{m, n\}$	entre m y n ocurrencias de r	a[1,5]
$r_1 r_2$	una r_1 seguida de una r_2	ab
$r_1 r_2$	una r_1 o una r_2	a b
(r)	igual que r	(a b)
r_1 / r_2	r_1 cuando va seguida de r_2	abc/123

Figura 3.8: Expresiones regulares de Lex

! Ejercicio 3.3.9: La expresión regular $r\{m, n\}$ coincide con las ocurrencias entre m y n del patrón r . Por ejemplo, a[1,5] coincide con una cadena de una a cinco as. Muestre que para cada expresión regular que contiene operadores de repetición de esta forma, hay una expresión regular equivalente sin operadores de repetición.

! Ejercicio 3.3.10: El operador $^$ coincide con el extremo izquierdo de una línea, y $\$$ coincide con el extremo derecho de una línea. El operador $^$ también se utiliza para introducir las clases de caracteres complementados, pero el contexto siempre deja en claro cuál es el significado deseado. Por ejemplo, $^[^aeiou]*\$$ coincide con cualquier línea completa que no contiene una vocal en minúscula.

- ¿Cómo podemos saber cuál es el significado deseado de $^$?
- ¿Podemos sustituir siempre una expresión regular, usando los operadores $^$ y $\$$, por una expresión equivalente que no utilice ninguno de estos operadores?

! Ejercicio 3.3.11: El comando del intérprete (shell) de UNIX sh utiliza los operadores de la figura 3.9 en expresiones de nombres de archivo para describir conjuntos de nombres de archivos. Por ejemplo, la expresión de nombre de archivo $*.o$ coincide con todos los nombres de archivo que terminen en .o; $orden1.?$ coincide con todos los nombres de archivo de la forma

EXPRESIÓN	COINCIDE CON	EJEMPLO
's'	la cadena <i>s</i> , literalmente	'\'
\c	el carácter <i>c</i> , literalmente	\'
*	cualquier cadena	*.o
?	cualquier carácter	orden1.?
[s]	cualquier carácter en <i>s</i>	orden1.[cso]

Figura 3.9: Expresiones de nombres de archivo utilizadas por el comando `sh` del intérprete de UNIX

`orden.c`, en donde *c* es cualquier carácter. Muestre cómo pueden sustituirse las expresiones de nombres de archivo de `sh` por expresiones regulares equivalentes, usando sólo los operadores básicos de unión, concatenación y cerradura.

! Ejercicio 3.3.12: SQL permite una forma rudimentaria de patrones, en los cuales dos caracteres tienen un significado especial: el guión bajo (`_`) representa a cualquier carácter y el signo de por ciento (`%`) representa a cualquier cadena de 0 o más caracteres. Además, el programador puede definir cualquier carácter, por decir *e*, para que sea el carácter de escape, de manera que si colocamos a *e* antes de *e* antes de `_`, `%` o cualquier *e*, obtenemos el carácter que va después de su significado literal. Muestre cómo expresar cualquier patrón de SQL como una expresión regular, dado que sabemos cuál es el carácter de escape.

3.4 Reconocimiento de tokens

En la sección anterior, aprendimos a expresar los patrones usando expresiones regulares. Ahora debemos estudiar cómo tomar todos los patrones para todos los tokens necesarios y construir una pieza de código para examinar la cadena de entrada y buscar un prefijo que sea un lexema que coincida con uno de esos patrones. En nuestra explicación haremos uso del siguiente bosquejo:

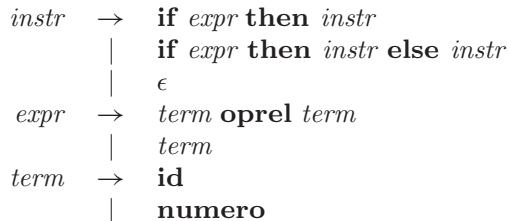


Figura 3.10: Una gramática para las instrucciones de bifurcación

Ejemplo 3.8: El fragmento de gramática de la figura 3.10 describe una forma simple de instrucciones de bifurcación y de expresiones condicionales. Esta sintaxis es similar a la del lenguaje Pascal porque `then` aparece en forma explícita después de las condiciones.

Para **oprel**, usamos los operadores de comparación de lenguajes como Pascal o SQL, en donde `=` es “es igual a” y `<>` es “no es igual a”, ya que presenta una estructura interesante de lexemas.

Las terminales de la gramática, que son **if**, **then**, **else**, **oprel**, **id** y **numero**, son los nombres de tokens en lo que al analizador léxico respecta. Los patrones para estos tokens se describen mediante el uso de definiciones regulares, como en la figura 3.11. Los patrones para **id** y **número** son similares a lo que vimos en el ejemplo 3.7.

<i>digito</i>	\rightarrow	$[0-9]$
<i>digitos</i>	\rightarrow	$digito^+$
<i>numero</i>	\rightarrow	$digitos (.\ digitos)? (E [+-]? digitos)?$
<i>letra</i>	\rightarrow	$[A-Za-z]$
<i>id</i>	\rightarrow	$letra (letra digito)^*$
<i>if</i>	\rightarrow	if
<i>then</i>	\rightarrow	then
<i>else</i>	\rightarrow	else
<i>oprel</i>	\rightarrow	$< > <= >= = <>$

Figura 3.11: Patrones para los tokens del ejemplo 3.8

Para este lenguaje, el analizador léxico reconocerá las palabras clave **if**, **then** y **else**, así como los lexemas que coinciden con los patrones para **oprel**, **id** y **numero**. Para simplificar las cosas, vamos a hacer la suposición común de que las palabras clave también son *palabras reservadas*; es decir, no son identificadores, aun cuando sus lexemas coinciden con el patrón para identificadores.

Además, asignaremos al analizador léxico el trabajo de eliminar el espacio en blanco, reconociendo el “token” **ws** definido por:

$$ws \rightarrow (\text{blanco} | \text{tab} | \text{nuevalinea})^+$$

Aquí, **blanco**, **tab** y **nuevalinea** son símbolos abstractos que utilizamos para expresar los caracteres ASCII de los mismos nombres. El token **ws** es distinto de los demás tokens porque cuando lo reconocemos, no lo regresamos al analizador sintáctico, sino que reiniciamos el analizador léxico a partir del carácter que va después del espacio en blanco. El siguiente token es el que se devuelve al analizador sintáctico.

Nuestro objetivo para el analizador léxico se resume en la figura 3.12. Esa tabla muestra, para cada lexema o familia de lexemas, qué nombre de token se devuelve al analizador sintáctico y qué valor de atributo se devuelve, como vimos en la sección 3.1.3. Observe que para los seis operadores relacionales, se utilizan las constantes simbólicas **LT**, **LE**, y demás como el valor del atributo, para poder indicar qué instancia del token **oprel** hemos encontrado. El operador específico que se encuentre influirá en el código que genere el compilador de salida. \square

LEXEMAS	NOMBRE DEL TOKEN	VALOR DEL ATRIBUTO
Cualquier <i>ws</i>	—	—
if	if	—
Then	then	—
else	else	—
Cualquier <i>id</i>	id	Apuntador a una entrada en la tabla
Cualquier <i>numero</i>	numero	Apuntador a una entrada en la tabla
<	oprel	LT
<=	oprel	LE
=	oprel	EQ
<>	oprel	NE
>	oprel	GT
>=	oprel	GE

Figura 3.12: Tokens, sus patrones y los valores de los atributos

3.4.1 Diagramas de transición de estados

Como paso intermedio en la construcción de un analizador léxico, primero convertimos los patrones en diagramas de flujo estilizados, a los cuales se les llama “diagramas de transición de estados”. En esta sección, realizaremos la conversión de los patrones de expresiones regulares a los diagramas de transición de estados en forma manual, pero en la sección 3.6 veremos que hay una forma mecánica de construir estos diagramas, a partir de colecciones de expresiones regulares.

Los *diagramas de transición de estados* tienen una colección de nodos o círculos, llamados *estados*. Cada estado representa una condición que podría ocurrir durante el proceso de explorar la entrada, buscando un lexema que coincida con uno de varios patrones. Podemos considerar un estado como un resumen de todo lo que debemos saber acerca de los caracteres que hemos visto entre el apuntador *inicioLexema* y el apuntador *avance* (como en la situación de la figura 3.3).

Las *líneas* se dirigen de un estado a otro del diagrama de transición de estados. Cada línea se *etiqueta* mediante un símbolo o conjunto de símbolos. Si nos encontramos en cierto estado *s*, y el siguiente símbolo de entrada es *a*, buscamos una línea que salga del estado *s* y esté etiquetado por *a* (y tal vez por otros símbolos también). Si encontramos dicha línea, avanzamos el apuntador *avance* y entramos al estado del diagrama de transición de estados al que nos lleva esa línea. Asumiremos que todos nuestros diagramas de transición de estados son *deterministas*, lo que significa que nunca hay más de una línea que sale de un estado dado, con un símbolo dado de entre sus etiquetas. A partir de la sección 3.5, relajaremos la condición del determinismo, facilitando en forma considerable la vida del diseñador de un analizador léxico, aunque esto se hará más difícil para el implementador. Algunas convenciones importantes de los diagramas de transición de estados son:

1. Se dice que ciertos estados son *de aceptación*, o *finales*. Estos estados indican que se ha encontrado un lexema, aunque el lexema actual tal vez no consista de todas las posiciones entre los apuntadores *inicioLexema* y *avance*. Siempre indicamos un estado de

aceptación mediante un círculo doble, y si hay que realizar una acción (por lo general, devolver un token y un valor de atributo al analizador sintáctico), la adjuntaremos al estado de aceptación.

2. Además, si es necesario retroceder el apuntador *avance* una posición (es decir, si el lexema no incluye el símbolo que nos llevó al estado de aceptación), entonces deberemos colocar de manera adicional un * cerca del estado de aceptación. En nuestro ejemplo, nunca es necesario retroceder a *avance* más de una posición, pero si lo fuera, podríamos adjuntar cualquier número de *s al estado de aceptación.
3. Un estado se designa como el *estado inicial*; esto se indica mediante una línea etiquetada como “inicio”, que no proviene de ninguna parte. El diagrama de transición siempre empieza en el estado inicial, antes de leer cualquier símbolo de entrada.

Ejemplo 3.9: La figura 3.13 es un diagrama de transición de estados que reconoce los lexemas que coinciden con el token **oprel**. Empezamos en el estado 0, el estado inicial. Si vemos < como el primer símbolo de entrada, entonces de entre los lexemas que coinciden con el patrón para **oprel** sólo podemos ver a <, <> o <=. Por lo tanto, pasamos al estado 1 y analizamos el siguiente carácter. Si es =, entonces reconocemos el lexema <=, pasamos al estado 2 y devolvemos el token **oprel** con el atributo LE, la constante simbólica que representa a este operador de comparación específico. Si en el estado 1 el siguiente carácter es >, entonces tenemos el lexema <> y pasamos al estado 3 para devolver una indicación de que se ha encontrado el operador “no es igual a”. En cualquier otro carácter, el lexema es <, y pasamos al estado 4 para devolver esa información. Sin embargo, observe que el estado 4 tiene un * para indicar que debemos retroceder la entrada una posición.

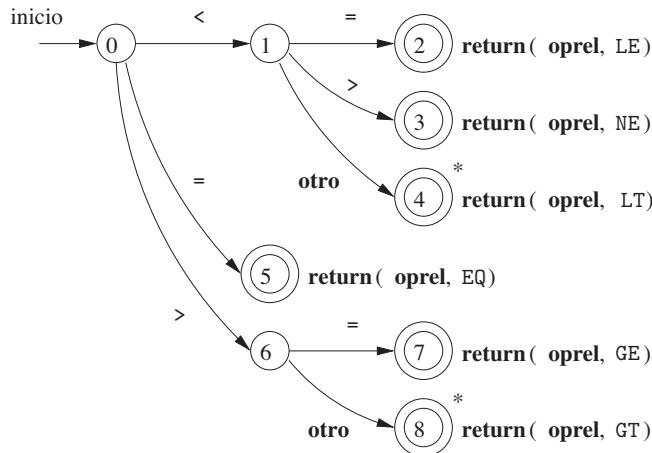


Figura 3.13: Diagrama de transición de estados para **oprel**

Por otro lado, si en el estado 0 el primer carácter que vemos es =, entonces este carácter debe ser el lexema. De inmediato devolvemos ese hecho desde el estado 5. La posibilidad restante es que el primer carácter sea >. Entonces, debemos pasar al estado 6 y decidir, en base

al siguiente carácter, si el lexema es \geq (si vemos a continuación el signo $=$), o sólo $>$ (con cualquier otro carácter). Observe que, si en el estado 0 vemos cualquier carácter además de $<$, $=$ o $>$, no es posible que estemos viendo un lexema `oprel`, por lo que no utilizaremos este diagrama de transición de estados. \square

3.4.2 Reconocimiento de las palabras reservadas y los identificadores

El reconocimiento de las palabras reservadas y los identificadores presenta un problema. Por lo general, las palabras clave como `if` o `then` son reservadas (como en nuestro bosquejo), por lo que no son identificadores, aun cuando lo *parecen*. Así, aunque por lo general usamos un diagrama de transición de estados como el de la figura 3.14 para buscar lexemas de identificadores, este diagrama también reconocerá las palabras clave `if`, `then` y `else` de nuestro bosquejo.

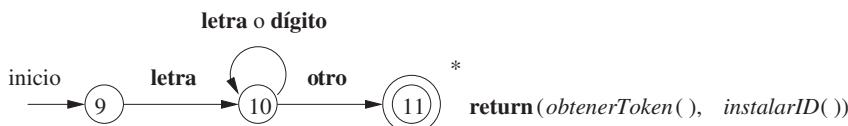


Figura 3.14: Un diagrama de transición de estados para identificadores (**id**) y palabras clave

Hay dos formas en las que podemos manejar las palabras reservadas que parecen identificadores:

1. Instalar las palabras reservadas en la tabla de símbolos desde el principio. Un campo de la entrada en la tabla de símbolos indica que estas cadenas nunca serán identificadores ordinarios, y nos dice qué token representan. Hemos supuesto que este método es el que se utiliza en la figura 3.14. Al encontrar un identificador, una llamada a `instalarID` lo coloca en la tabla de símbolos, si no se encuentra ahí todavía, y devuelve un apuntador a la entrada en la tabla de símbolos para el lexema que se encontró. Desde luego que cualquier identificador que no se encuentre en la tabla de símbolos durante el análisis léxico no puede ser una palabra reservada, por lo que su token es **id**. La función `obtenerToken` examina la entrada en la tabla de símbolos para el lexema encontrado, y devuelve el nombre de token que la tabla de símbolos indique que representa este lexema; ya sea **id** o uno de los tokens de palabra clave que se instaló en un principio en la tabla.
2. Crear diagramas de transición de estados separados para cada palabra clave; en la figura 3.15 se muestra un ejemplo para la palabra clave `then`. Observe que dicho diagrama de transición de estado consiste en estados que representan la situación después de ver cada letra sucesiva de la palabra clave, seguida de una prueba para un “no letra ni dígito”, es decir, cualquier carácter que no pueda ser la continuación de un identificador. Es necesario verificar que el identificador haya terminado, o de lo contrario devolveríamos el token `then` en situaciones en las que el token correcto era **id**, con un lexema como `thenextValue` que tenga a `then` como un prefijo propio. Si adoptamos este método, entonces debemos dar prioridad a los tokens, para que los tokens de palabra reservada se

reconozcan de preferencia en vez de **id**, cuando el lexema coincide con ambos patrones. No utilizaremos este método en nuestro ejemplo, que explica por qué los estados en la figura 3.15 no están enumerados.

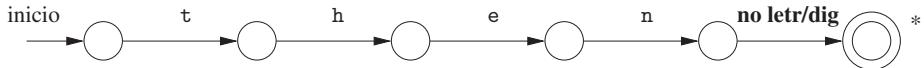


Figura 3.15: Diagrama de transición hipotético para la palabra clave **then**

3.4.3 Finalización del bosquejo

El diagrama de transición para los tokens **id** que vimos en la figura 3.14 tiene una estructura simple. Empezando en el estado 9, comprueba que el lexema empiece con una letra y que pase al estado 10 en caso de ser así. Permanecemos en el estado 10 siempre y cuando la entrada contenga letras y dígitos. Al momento de encontrar el primer carácter que no sea letra ni dígito, pasamos al estado 11 y aceptamos el lexema encontrado. Como el último carácter no forma parte del identificador, debemos retroceder la entrada una posición, y como vimos en la sección 3.4.2, introducimos lo que hemos encontrado en la tabla de símbolos y determinamos si tenemos una palabra clave o un verdadero identificador.

El diagrama de transición de estados para el token **numero** se muestra en la figura 3.16, y es hasta ahora el diagrama más complejo que hemos visto. Empezando en el estado 12, si vemos un dígito pasamos al estado 13. En ese estado podemos leer cualquier número de dígitos adicionales. No obstante, si vemos algo que no sea un dígito o un punto, hemos visto un número en forma de entero; 123 es un ejemplo. Para manejar ese caso pasamos al estado 20, en donde devolvemos el token **numero** y un apuntador a una tabla de constantes en donde se introduce el lexema encontrado. Esta mecánica no se muestra en el diagrama, pero es análoga a la forma en la que manejamos los identificadores.

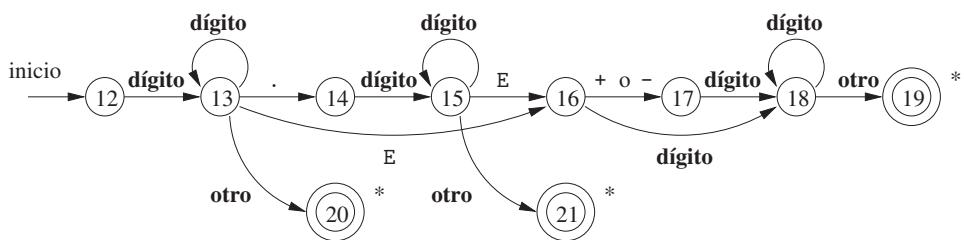


Figura 3.16: Un diagrama de transición para los números sin signo

Si en vez de ello vemos un punto en el estado 13, entonces tenemos una “fracción opcional”. Pasamos al estado 14, y buscamos uno o más dígitos adicionales; el estado 15 se utiliza para este fin. Si vemos una E, entonces tenemos un “exponente opcional”, cuyo reconocimiento es trabajo de los estados 16 a 19. Si en el estado 15 vemos algo que no sea una E o un dígito, entonces hemos llegado al final de la fracción, no hay exponente y devolvemos el lexema encontrado, mediante el estado 21.

El diagrama de transición de estados final, que se muestra en la figura 3.17, es para el espacio en blanco. En ese diagrama buscamos uno o más caracteres de “espacio en blanco”, representados por **delim** en ese diagrama; por lo general estos caracteres son los espacios, tabuladores, caracteres de nueva línea y tal vez otros caracteres que el diseño del lenguaje no considere como parte de algún token.

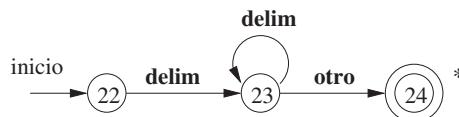


Figura 3.17: Un diagrama de transición para el espacio en blanco

Observe que, en el estado 24, hemos encontrado un bloque de caracteres de espacio en blanco consecutivos, seguidos de un carácter que no es espacio en blanco. Regresemos la entrada para que empiece en el carácter que no es espacio en blanco, pero no regresamos nada al analizador sintáctico, sino que debemos reiniciar el proceso del análisis léxico después del espacio en blanco.

3.4.4 Arquitectura de un analizador léxico basado en diagramas de transición de estados

Hay varias formas en las que pueden utilizarse los diagramas de transición de estados para construir un analizador léxico. Sin importar la estrategia general, cada estado se representa mediante una pieza de código. Podemos imaginar una variable **estado** que contiene el número del estado actual para un diagrama de transición de estados. Una instrucción **switch** con base en el valor de **estado** nos lleva al código para cada uno de los posibles estados, en donde encontramos la acción de ese estado. A menudo, el código para un estado es en sí una instrucción **switch** o una bifurcación de varias vías que determina el siguiente estado mediante el proceso de leer y examinar el siguiente carácter de entrada.

Ejemplo 3.10: En la figura 3.18 vemos un bosquejo de **obtenerOpRel()**, una función en C++ cuyo trabajo es simular el diagrama de transición de la figura 3.13 y devolver un objeto de tipo **TOKEN**; es decir, un par que consiste en el nombre del token (que debe ser **oprel** en este caso) y un valor de atributo (el código para uno de los seis operadores de comparación en este caso). Lo primero que hace **obtenerOpRel()** es crear un nuevo objeto **tokenRet** e inicializar su primer componente con **OPREL**, el código simbólico para el token **oprel**.

Podemos ver el comportamiento típico de un estado en el caso 0, el caso en donde el estado actual es 0. Una función llamada **sigCar()** obtiene el siguiente carácter de la entrada y lo asigna a la variable local **c**. Después verificamos si **c** es uno de los tres caracteres que esperamos encontrar, y realizamos la transición de estado que indica el diagrama de transición de la figura 3.13 en cada caso. Por ejemplo, si el siguiente carácter de entrada es **=**, pasamos al estado 5.

Si el siguiente carácter de entrada no es uno que pueda empezar un operador de comparación, entonces se hace una llamada a la función **falla()**. Lo que haga ésta dependerá de la

```

TOKEN obtenerOpRel()
{
    TOKEN tokenRet = new (OPREL);
    while(1) { /* repite el procesamiento de caracteres hasta que
                  ocurre un retorno o un fallo */
        switch(estado) {
            case 0: c = sigCar();
                if ( c == '<' ) estado = 1;
                else if ( c == '=' ) estado = 5;
                else if ( c == '>' ) estado = 6;
                else fallo(); /* el lexema no es un oprel */
                break;
            case 1: ...
            ...
            case 8: retractar();
                tokenRet.atributo = GT;
                return(tokenRet);
        }
    }
}

```

Figura 3.18: Bosquejo de la implementación del diagrama de transición **oprel**

estrategia global de recuperación de errores del analizador léxico. Debe reiniciar el apuntador `avance` para colocarlo en la misma posición que `inicioLexema`, de manera que pueda aplicarse otro diagrama de transición de estados al verdadero inicio de la entrada sin procesar. Entonces podría cambiar el valor de `estado` para que sea el estado inicial para otro diagrama de transición, el cual buscará otro token. De manera alternativa, si no hay otro diagrama de transición de estados que esté sin uso, `fallo()` podría iniciar una fase de corrección de errores que tratará de reparar la entrada y encontrar un lexema, como vimos en la sección 3.1.4.

También mostramos la acción para el estado 8 en la figura 3.18. Como el estado 8 lleva un `*`, debemos regresar el apuntador de entrada una posición (es decir, colocar a `c` de vuelta en el flujo de entrada). Esta tarea se lleva a cabo mediante la función `retractar()`. Como el estado 8 representa el reconocimiento del lexema `>=`, establecemos el segundo componente del objeto devuelto (el cual suponemos se llama `atributo`) con `GT`, el código para este operador. □

Para poner en perspectiva la simulación de un diagrama de transición de estados, vamos a considerar las formas en que el código como el de la figura 3.18 podría ajustarse a todo el analizador léxico.

1. Podríamos arreglar que los diagramas de transición para cada token se probaran en forma secuencial. Después, la función `fallo()` del ejemplo 3.10 restablece el apuntador `avance` e inicia el siguiente diagrama de transición de estados, cada vez que se le llama. Este método nos permite usar diagramas de transición de estados para las palabras clave individuales, como el que se sugiere en la figura 3.15. Sólo tenemos que usarlos antes que el diagrama para `id`, de manera que las palabras clave sean palabras reservadas.

2. Podríamos ejecutar los diversos diagramas de transición de estados “en paralelo”, alimentando el siguiente carácter de entrada a todos ellos y permitiendo que cada uno realice las transiciones requeridas. Si utilizamos esta estrategia, debemos tener cuidado de resolver el caso en el que un diagrama encuentra a un lexema que coincide con su patrón, mientras uno o varios de los otros diagramas pueden seguir procesando la entrada. La estrategia normal es tomar el prefijo más largo de la entrada que coincida con cualquier patrón. Esa regla nos permite por ejemplo, dar preferencia al identificador **siguiente** sobre la palabra clave **then**, o al operador \rightarrow sobre $-$.
3. El método preferido, y el que vamos a usar en las siguientes secciones, es combinar todos los diagramas de transición de estados en uno solo. Permitimos que el diagrama de transición de estados lea la entrada hasta que no haya un siguiente estado posible, y después tomamos el lexema más largo que haya coincidido con algún patrón, como dijimos en el punto (2) anterior. En nuestro bosquejo, esta combinación es fácil, ya que no puede haber dos tokens que empiecen con el mismo carácter; es decir, el primer carácter nos indica de inmediato el token que estamos buscando. Por ende, podríamos simplemente combinar los estados 0, 9, 12 y 22 en un estado inicial, dejando las demás transiciones intactas. No obstante, en general, el problema de combinar diagramas de transición de estados para varios tokens es más complejo, como veremos en breve.

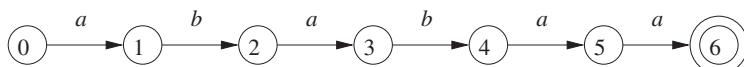
3.4.5 Ejercicios para la sección 3.4

Ejercicio 3.4.1: Proporcione los diagramas de transición de estados para reconocer los mismos lenguajes que en cada una de las expresiones regulares en el ejercicio 3.3.2.

Ejercicio 3.4.2: Proporcione los diagramas de transición para reconocer los mismos lenguajes que en cada una de las expresiones regulares en el ejercicio 3.3.5.

Los siguientes ejercicios, hasta el 3.4.12, presentan el algoritmo de Aho-Corasick para reconocer una colección de palabras clave en una cadena de texto, en un tiempo proporcional a la longitud del texto y la suma de la longitud de las palabras clave. Este algoritmo utiliza una forma especial de diagrama de transición, conocido como *trie*. Un trie es un diagrama de transición de estados con estructura de árbol y distintas etiquetas en las líneas que nos llevan de un nodo a sus hijos. Las hojas del trie representan las palabras clave reconocidas.

Knuth, Morris y Pratt presentaron un algoritmo para reconocer una palabra clave individual $b_1b_2 \dots b_n$ en una cadena de texto. Aquí, el trie es un diagrama de transición de estados con n estados, de 0 a n . El estado 0 es el estado inicial, y el estado n representa la aceptación, es decir, el descubrimiento de la palabra clave. De cada estado s , desde 0 hasta $n - 1$, hay una transición al estado $s + 1$, que se etiqueta mediante el símbolo b_{s+1} . Por ejemplo, el trie para la palabra clave **ababaa** es:



Para poder procesar las cadenas de texto con rapidez y buscar una palabra clave en esas cadenas, es útil definir, para la palabra clave $b_1b_2 \dots b_n$ y la posición s en esa palabra clave (correspondiente al estado s de su trie), una *función de fallo*, $f(s)$, que se calcula como en la

figura 3.19. El objetivo es que $b_1b_2 \dots b_{f(s)}$ sea el prefijo propio más largo de $b_1b_2 \dots b_s$, que también sea sufijo de $b_1b_2 \dots b_s$. La razón de la importancia de $f(s)$ es que, si tratamos de encontrar una coincidencia en una cadena de texto para $b_1b_2 \dots b_n$, y hemos relacionado las primeras s posiciones, pero después fallamos (es decir, que la siguiente posición de la cadena de texto no contenga a b_{s+1}), entonces $f(s)$ es el prefijo más largo de $b_1b_2 \dots b_n$ que podría quizás coincidir con la cadena de texto hasta el punto en el que nos encontramos. Desde luego que el siguiente carácter de la cadena de texto debe ser $b_{f(s)+1}$, o de lo contrario seguiríamos teniendo problemas y tendríamos que considerar un prefijo aún más corto, el cual será $b_{f(f(s))}$.

```

1)    $t = 0;$ 
2)    $f(1) = 0;$ 
3)   for ( $s = 1; s < n; s++$ ) {
4)       while ( $t > 0 \&\& b_{s+1} \neq b_{t+1}$ )  $t = f(t);$ 
5)       if ( $b_{s+1} == b_{t+1}$ ) {
6)            $t = t + 1;$ 
7)            $f(s + 1) = t;$ 
8)       }
}
else  $f(s + 1) = 0;$ 
}

```

Figura 3.19: Algoritmo para calcular la función de fallo para la palabra clave $b_1b_2 \dots b_n$

Como ejemplo, la función de fallo para el trie que construimos antes para **ababaa** es:

s	1	2	3	4	5	6
$f(s)$	0	0	1	2	3	1

Por ejemplo, los estados 3 y 1 representan a los prefijos **aba** y **a**, respectivamente. $f(3) = 1$ ya que **a** es el prefijo propio más largo de **aba**, que también es un sufijo de **aba**. Además, $f(2) = 0$, debido a que el prefijo propio más largo de **ab**, que también es un sufijo, es la cadena vacía.

Ejercicio 3.4.3: Construya la función de fallo para las siguientes cadenas:

- a) **abababaab.**
- b) **aaaaaa.**
- c) **abbaabb.**

! Ejercicio 3.4.4: Demuestre, por inducción en s , que el algoritmo de la figura 3.19 calcula en forma correcta la función de fallo.

!! Ejercicio 3.4.5: Muestre que la asignación $t = f(t)$ en la línea (4) de la figura 3.19 se ejecuta cuando mucho n veces. Muestre que, por lo tanto, todo el algoritmo requiere sólo un tiempo $O(n)$ para una palabra clave de longitud n .

Habiendo calculado la función de fallo para una palabra clave $b_1b_2 \dots b_n$, podemos explorar una cadena $a_1a_2 \dots a_m$ en un tiempo $O(m)$ para saber si hay una ocurrencia de la palabra clave en la cadena. El algoritmo, que se muestra en la figura 3.20, desliza la palabra clave a lo largo de la cadena, tratando de progresar en busca de la coincidencia del siguiente carácter de la palabra clave con el siguiente carácter de la cadena. Si no puede hacerlo después de relacionar s caracteres, entonces “desliza” la palabra clave a la derecha $s - f(s)$ posiciones, de manera que sólo los primeros $f(s)$ caracteres de la palabra clave se consideren como coincidencias con la cadena.

```

1)   $s = 0;$ 
2)  for ( $i = 1; i \leq m; i++$ ) {
3)      while ( $s > 0 \ \&\& a_i \neq b_{s+1}$ )  $s = f(s);$ 
4)      if ( $a_i == b_{s+1}$ )  $s = s + 1;$ 
5)      if ( $s == n$ ) return “si”;
6)  }
6)  return “no”;

```

Figura 3.20: El algoritmo KMP evalúa si la cadena $a_1a_2 \dots a_m$ contiene una palabra clave individual $b_1b_2 \dots b_n$ como una subcadena, en un tiempo $O(m + n)$

Ejercicio 3.4.6: Aplique el algoritmo KMP para evaluar si la palabra clave ababaa es una subcadena de:

- a) abababaab.
- b) abababbaa.

!! Ejercicio 3.4.7: Muestre que el algoritmo de la figura 3.20 nos indica en forma correcta si la palabra clave es una subcadena de la cadena dada. *Sugerencia:* proceda por inducción en i . Muestre que para todas las i , el valor de s después de la línea (4) es la longitud del prefijo más largo de la palabra clave que es un sufijo de $a_1a_2 \dots a_i$.

!! Ejercicio 3.4.8: Muestre que el algoritmo de la figura 3.20 se ejecuta en un tiempo $O(m + n)$, suponiendo que la función f ya se ha calculado y que sus valores se almacenaron en un arreglo indexado por s .

Ejercicio 3.4.9: Las *cadenas de Fibonacci* se definen de la siguiente manera:

1. $s_1 = b$.
2. $s_2 = a$.
3. $s_k = s_{k-1}s_{k-2}$ para $k > 2$.

Por ejemplo, $s_3 = ab$, $s_4 = aba$ y $s_5 = abaab$.

- a) ¿Cuál es la longitud de s_n ?

- b) Construya la función de fallo para s_6 .
- c) Construya la función de fallo para s_7 .
- !! d) Muestre que la función de fallo para cualquier s_n puede expresarse mediante $f(1) = f(2) = 0$, y que para $2 < j \leq |s_n|$, $f(j)$ es $j - |s_{k-1}|$, en donde k es el entero más largo, de forma que $|s_k| \leq j + 1$.
- !! e) En el algoritmo KMP, ¿cuál es el número más grande de aplicaciones consecutivas de la función de fallo, cuando tratamos de determinar si la palabra clave s_k aparece en la cadena de texto s_{k+1} ?

Aho y Corasick generalizaron el algoritmo KMP para reconocer cualquiera de un conjunto de palabras clave en una cadena de texto. En este caso el trie es un verdadero árbol, con ramificaciones desde la raíz. Hay un estado para cada cadena que sea un prefijo (no necesariamente propio) de cualquier palabra clave. El padre de un estado correspondiente a la cadena $b_1 b_2 \dots b_k$ es el estado que corresponde a $b_1 b_2 \dots b_{k-1}$. Un estado de aceptación si corresponde a una palabra clave completa. Por ejemplo, la figura 3.21 muestra el trie para las palabras clave **he**, **she**, **his** y **hers**.

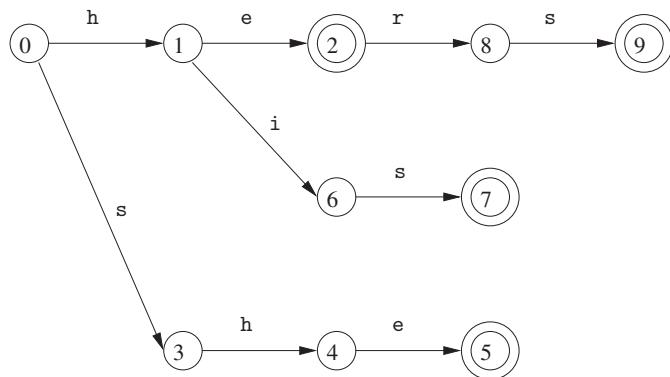


Figura 3.21: Trie para las palabras clave **he**, **she**, **his**, **hers**

La función de fallo para el trie general se define de la siguiente forma. Suponga que s es el estado que corresponde a la cadena $b_1 b_2 \dots b_n$. Entonces, $f(s)$ es el estado que corresponde al sufijo propio más largo de $b_1 b_2 \dots b_n$ que también es un prefijo de *alguna* palabra clave. Por ejemplo, la función de fallo para el trie de la figura 3.21 es:

s	1	2	3	4	5	6	7	8	9
$f(s)$	0	0	0	1	2	0	3	0	3

! **Ejercicio 3.4.10:** Modifique el algoritmo de la figura 3.19 para calcular la función de fallo para tries generales. *Sugerencia:* La principal diferencia es que no podemos simplificar la prueba de igualdad o desigualdad de b_{s+1} y b_{t+1} en las líneas (4) y (5) de la figura 3.19. En vez de

ello, desde cualquier estado puede haber varias transiciones sobre diversos caracteres, así como hay transiciones tanto en `e` como en `i` desde el estado 1 en la figura 3.21. Cualquiera de esas transiciones podría conducirnos a un estado que represente el sufijo más largo que sea también un prefijo.

Ejercicio 3.4.11: Construya los tries y calcule la función de fallo para los siguientes conjuntos de palabras clave:

- a) `aaa, abaaa y ababaaa.`
- b) `all, fall, fatal, llama y lame.`
- c) `pipe, pet, item, temper y perpetual.`

! Ejercicio 3.4.12: Muestre que su algoritmo del ejercicio 3.4.10 se sigue ejecutando en un tiempo que es lineal, en la suma de las longitudes de las palabras clave.

3.5 El generador de analizadores léxicos Lex

En esta sección presentaremos una herramienta conocida como `Lex`, o `Flex` en una implementación más reciente, que nos permite especificar un analizador léxico mediante la especificación de expresiones regulares para describir patrones de los tokens. La notación de entrada para la herramienta `Lex` se conoce como el *lenguaje Lex*, y la herramienta en sí es el *compilador Lex*. El compilador Lex transforma los patrones de entrada en un diagrama de transición y genera código, en un archivo llamado `lex.yy.c`, que simula este diagrama de transición. La mecánica de cómo ocurre esta traducción de expresiones regulares a diagramas de transición es el tema de las siguientes secciones; aquí sólo aprenderemos acerca del lenguaje Lex.

3.5.1 Uso de Lex

La figura 3.22 sugiere cómo se utiliza `Lex`. Un archivo de entrada, al que llamaremos `lex.1`, está escrito en el lenguaje Lex y describe el analizador léxico que se va a generar. El compilador Lex transforma a `lex.1` en un programa en C, en un archivo que siempre se llama `lex.yy.c`. El compilador de C compila este archivo en un archivo llamado `a.out`, como de costumbre. La salida del compilador de C es un analizador léxico funcional, que puede recibir un flujo de caracteres de entrada y producir una cadena de tokens.

El uso normal del programa compilado en C, denominado `a.out` en la figura 3.22, es como una subrutina del analizador sintáctico. Es una función en C que devuelve un entero, el cual representa un código para uno de los posibles nombres de cada token. El valor del atributo, ya sea otro código numérico, un apuntador a la tabla de símbolos, o nada, se coloca en una variable global llamada `yyval`,² la cual se comparte entre el analizador léxico y el analizador sintáctico, con lo cual se simplifica el proceso de devolver tanto el nombre como un valor de atributo de un token.

²A propósito, la `yy` que aparece en `yyval` y `lex.yy.c` se refiere al generador de analizadores sintácticos `Yacc`, que describiremos en la sección 4.9, el cual se utiliza, por lo regular, en conjunto con `Lex`.

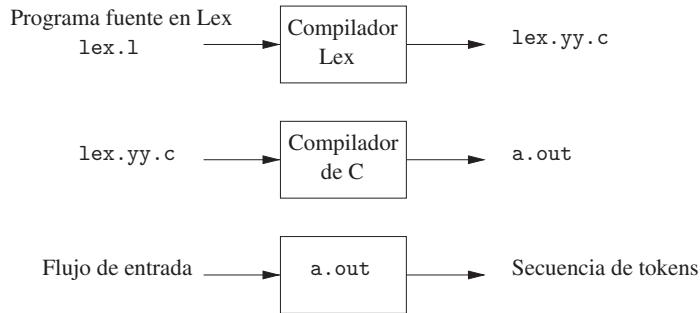


Figura 3.22: Creación de un analizador léxico con Lex

3.5.2 Estructura de los programas en Lex

Un programa en Lex tiene la siguiente forma:

```

declaraciones
%%
reglas de traducción
%%
funciones auxiliares
  
```

La sección de declaraciones incluye las declaraciones de variables, *constantes de manifiesto* (identificadores que se declaran para representar a una constante; por ejemplo, el nombre de un token) y definiciones regulares, en el estilo de la sección 3.3.4.

Cada una de las reglas de traducción tiene la siguiente forma:

Patrón { Acción }

Cada patrón es una expresión regular, la cual puede usar las definiciones regulares de la sección de declaraciones. Las acciones son fragmentos de código, por lo general, escritos en C, aunque se han creado muchas variantes de Lex que utilizan otros lenguajes.

La tercera sección contiene las funciones adicionales que se utilizan en las acciones. De manera alternativa, estas funciones pueden compilarse por separado y cargarse con el analizador léxico.

El analizador léxico que crea Lex trabaja en conjunto con el analizador sintáctico de la siguiente manera. Cuando el analizador sintáctico llama al analizador léxico, éste empieza a leer el resto de su entrada, un carácter a la vez, hasta que encuentra el prefijo más largo de la entrada que coincide con uno de los patrones P_i . Despues ejecuta la acción asociada A_i . Por lo general, A_i regresará al analizador sintáctico, pero si no lo hace (tal vez debido a que P_i describe espacio en blanco o comentarios), entonces el analizador léxico procede a buscar lexemas adicionales, hasta que una de las acciones correspondientes provoque un retorno al analizador sintáctico. El analizador léxico devuelve un solo valor, el nombre del token, al analizador sintáctico, pero utiliza la variable entera compartida `yylval` para pasarle información adicional sobre el lexema encontrado, si es necesario.

Ejemplo 3.11: La figura 3.23 es un programa en Lex que reconoce los tokens de la figura 3.12 y devuelve el token encontrado. Unas cuantas observaciones acerca de este código nos darán una idea de varias de las características importantes de Lex.

En la sección de declaraciones vemos un par de llaves especiales, `%{` y `%}`. Cualquier cosa dentro de ellas se copia directamente al archivo `lex.yy.c`, y no se trata como definición regular. Es común colocar ahí las definiciones de las constantes de manifiesto, usando instrucciones `#define` de C para asociar códigos enteros únicos con cada una de las constantes de manifiesto. En nuestro ejemplo, hemos listado en un comentario los nombres de las constantes de manifiesto `LT`, `IF`, etcétera, pero no las hemos mostrado definidas para ser enteros específicos.³

Además, en la sección de declaraciones hay una secuencia de definiciones regulares. Éstas utilizan la notación extendida para expresiones regulares que describimos en la sección 3.3.5. Las definiciones regulares que se utilizan en definiciones posteriores o en los patrones de las reglas de traducción, van rodeadas de llaves. Así, por ejemplo, `delim` se define como abreviación para la clase de caracteres que consiste en el espacio en blanco, el tabulador y el carácter de nueva línea; estos últimos dos se representan, como en todos los comandos de UNIX, por una barra diagonal inversa seguida de `t` o de `n`, respectivamente. Entonces, `ws` se define para que sea uno o más delimitadores, mediante la expresión regular `{delim}+`.

Observe que en la definición de `id` y de `numero` se utilizan paréntesis como meta símbolos de agrupamiento, por lo que no se representan a sí mismos. En contraste, la `E` en la definición de `numero` se representa a sí misma. Si deseamos usar uno de los meta símbolos de Lex, como cualquiera de los paréntesis, `+`, `*` o `?` para que se representen a sí mismos, podemos colocar una barra diagonal inversa antes de ellos. Por ejemplo, vemos `\.` en la definición de `numero` para representar el punto, ya que ese carácter es un meta símbolo que representa a “cualquier carácter”, como es costumbre en las expresiones regulares de UNIX.

En la sección de funciones auxiliares vemos dos de esas funciones, `instalarID()` e `instalarNum()`. Al igual que la porción de la sección de declaración que aparece entre `%{...%}`, todo lo que hay en la sección auxiliar se copia directamente al archivo `lex.yy.c`, pero puede usarse en las acciones.

Por último, vamos a examinar algunos de los patrones y reglas en la sección media de la figura 3.23. En primer lugar tenemos a `ws`, un identificador declarado en la primera sección, el cual tiene una acción vacía asociada. Si encontramos espacio en blanco, no regresamos al analizador sintáctico, sino que buscamos otro lexema. El segundo token tiene el patrón de expresión regular simple `if`. Si vemos las dos letras `if` en la entrada, y éstas no van seguidas de otra letra o dígito (que haría que el analizador léxico buscara un prefijo más largo de la entrada que coincida con el patrón para `id`), entonces el analizador léxico consume estas dos letras de la entrada y devuelve el nombre de token `IF`, es decir, el entero que la constante de manifiesto `IF` representa. Las palabras clave `then` y `else` se tratan de manera similar.

El quinto token tiene el patrón definido por `id`. Observe que, aunque las palabras clave como `if` coinciden con este patrón así como con un patrón anterior, Lex elige el patrón que aparezca

³Si se utiliza Lex junto con Yacc, entonces sería normal definir las constantes de manifiesto en el programa Yacc y usarlas sin definición en el programa Lex. Como `lex.yy.c` se compila con la salida de Yacc, por consiguiente las constantes estarían disponibles para las acciones en el programa en Lex.

```

%{
/* definiciones de las constantes de manifiesto
LT, LE, EQ, NE, GT, GE,
IF, THEN, ELSE, ID, NUMERO, OPREL */
%}

/* definiciones regulares */
delim      [ \t\n]
ws         {delim}+
letra      {A-Za-z}
digito     [0-9]
id         {letra}({letra}|{digito})*
numero    {digito}+(\.{digito}+)?(E[+-]?(digito)+)?

%%

{ws}        {/* no hay accion y no hay retorno */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}         {yyval = (int) instalarID(); return(ID);}
{numero}    {yyval = (int) instalarNum(); return(NUMERO);}
"<"         {yyval = LT; return(OPREL);}
"<="        {yyval = LE; return(OPREL);}
"="         {yyval = EQ; return(OPREL);}
"<>"        {yyval = NE; return(OPREL);}
">"         {yyval = GT; return(OPREL);}
">="        {yyval = GE; return(OPREL);}

%%

int instalarID() {/* funcion para instalar el lexema en
                     la tabla de simbolos y devolver un
                     apuntador a esto; yytext apunta al
                     primer caracter y yylent es
                     la longitud */
}

int instalarNum() {/* similar a instalarID, pero coloca las
                     constantes numericas en una tabla separada */}
}

```

Figura 3.23: Programa en Lex para los tokens de la figura 3.12

primero en la lista, en situaciones en las que el prefijo más largo coincide con dos o más patrones. La acción que se realiza cuando hay una coincidencia con *id* consiste en tres pasos:

1. Se hace una llamada a la función `instalarID()` para colocar el lexema encontrado en la tabla de símbolos.
2. Esta función devuelve un apuntador a la tabla de símbolos, el cual se coloca en la variable global `yyval`, en donde el analizador sintáctico o un componente posterior del compilador puedan usarla. Observe que `instalarID()` tiene dos variables disponibles, que el analizador léxico generado por `Lex` establece de manera automática:
 - (a) `yytext` es un apuntador al inicio del lexema, análogo a `inicioLexema` en la figura 3.3.
 - (b) `yylen` es la longitud del lexema encontrado.
3. El nombre de token `ID` se devuelve al analizador sintáctico.

La acción que se lleva a cabo cuando un lexema que coincide con el patrón *numero* es similar, sólo que se usa la función auxiliar `instalarNum()`. \square

3.5.3 Resolución de conflictos en Lex

Nos hemos referido a las dos reglas que utiliza `Lex` para decidir acerca del lexema apropiado a seleccionar, cuando varios prefijos de la entrada coinciden con uno o más patrones:

1. Preferir siempre un prefijo más largo a uno más corto.
2. Si el prefijo más largo posible coincide con dos o más patrones, preferir el patrón que se lista primero en el programa en `Lex`.

Ejemplo 3.12: La primera regla nos dice que debemos continuar leyendo letras y dígitos para buscar el prefijo más largo de estos caracteres y agruparlos como un identificador. También nos dice que debemos tratar a `<=` como un solo lexema, en vez de seleccionar `<` como un lexema y `=` como el siguiente lexema. La segunda regla hace a las palabras clave reservadas, si listamos las palabras clave antes que `id` en el programa. Por ejemplo, si se determina que `then` es el prefijo más largo de la entrada que coincide con algún patrón, y el patrón `then` va antes que `{id}`, como en la figura 3.23, entonces se devuelve el token `THEN`, en vez de `ID`. \square

3.5.4 El operador adelantado

`Lex` lee de manera automática un carácter adelante del último carácter que forma el lexema seleccionado, y después regresa la entrada para que sólo se consuma el propio lexema de la entrada. No obstante, algunas veces puede ser conveniente que cierto patrón coincida con la entrada, sólo cuando vaya seguido de ciertos caracteres más. De ser así, tal vez podamos utilizar la barra diagonal en un patrón para indicar el final de la parte del patrón que coincide

con el lexema. Lo que va después de / es un patrón adicional que se debe relacionar antes de poder decidir que vimos el token en cuestión, pero que lo que coincide con este segundo patrón no forma parte del lexema.

Ejemplo 3.13: En Fortran y en algunos otros lenguajes, las palabras clave no son reservadas. Esa situación crea problemas, como por ejemplo la siguiente instrucción:

```
IF(I,J) = 3
```

en donde IF es el nombre de un arreglo, no una palabra clave. Esta instrucción contrasta con las instrucciones de la forma

```
IF( condición ) THEN ...
```

en donde IF es una palabra clave. Por fortuna, podemos asegurarnos de que la palabra clave IF siempre vaya seguida de un paréntesis izquierdo, cierto texto (la condición) que puede contener paréntesis, un paréntesis derecho y una letra. Así, podríamos escribir una regla de Lex para la palabra clave IF, como:

```
IF / \(. * \) {letra}
```

Esta regla dice que el patrón con el que coincide el lexema es sólo las dos letras IF. La barra diagonal dice que le sigue un patrón adicional, pero que no coincide con el lexema. En este patrón, el primer carácter es el paréntesis izquierdo. Como ese carácter es un meta símbolo de Lex, debe ir precedido por una barra diagonal inversa para indicar que tiene su significado literal. El punto y el asterisco coinciden con “cualquier cadena sin un carácter de nueva línea”. Observe que el punto es un meta símbolo de Lex que significa “cualquier carácter excepto nueva línea”. Va seguido de un paréntesis derecho, de nuevo con una barra diagonal inversa para dar a ese carácter su significado literal. El patrón adicional va seguido por el símbolo *letra*, que es una definición regular que representa a la clase de caracteres de todas las letras.

Observe que, para que este patrón pueda ser a prueba de errores, debemos preprocessar la entrada para eliminar el espacio en blanco. No hay provisión en el patrón para el espacio en blanco, ni podemos tratar con la posibilidad de que la condición se extienda en varias líneas, ya que el punto no coincidirá con un carácter de nueva línea.

Por ejemplo, suponga que a este patrón se le pide que coincida con un prefijo de entrada:

```
IF(A<(B+C)*D)THEN...
```

los primeros dos caracteres coinciden con IF, el siguiente carácter coincide con \(<, los siguientes nueve caracteres coinciden con .*, y los siguientes dos coinciden con \) y *letra*. Observar el hecho de que el primer paréntesis derecho (después de C) no vaya seguido de una letra es irrelevante; sólo necesitamos encontrar alguna manera de hacer que la entrada coincida con el patrón. Concluimos que las letras IF constituyen el lexema, y que son una instancia del token **if**. □

3.5.5 Ejercicios para la sección 3.5

Ejercicio 3.5.1: Describa cómo hacer las siguientes modificaciones al programa en **Lex** de la figura 3.23:

- a) Agregar la palabra clave **while**.
- b) Cambiar los operadores de comparación para que sean los operadores en C de ese tipo.
- c) Permitir el guión bajo (_) como una letra adicional.
- ! d) Agregar un nuevo patrón con el token **STRING**. El patrón consiste en una doble comilla ("), cualquier cadena de caracteres y una doble comilla final. No obstante, si aparece una doble comilla en la cadena debe ser carácter de escape, para lo cual le anteponemos una barra diagonal inversa (\) y, por lo tanto, una barra diagonal inversa en la cadena debe representarse mediante dos barras diagonales inversas. El valor léxico es la cadena sin las dobles comillas circundantes, y sin las barras diagonales inversas que se usan como carácter de escape. Las cadenas deben instalarse en una tabla de cadenas.

Ejercicio 3.5.2: Escriba un programa en **Lex** que copie un archivo, sustituyendo cada secuencia no vacía de espacios en blanco por un solo espacio.

Ejercicio 3.5.3: Escriba un programa en **Lex** que copie un programa en C, sustituyendo cada instancia de la palabra clave **float** por **double**.

! Ejercicio 3.5.4: Escriba un programa en **Lex** que convierta un archivo a “Pig latín”. En específico, suponga que el archivo es una secuencia de palabras (grupos de letras) separadas por espacio en blanco. Cada vez que se encuentre con una palabra:

1. Si la primera letra es una consonante, desplácela hasta el final de la palabra y después agregue **ay**.
2. Si la primera letra es una vocal, sólo agregue **ay** al final de la palabra.

Todos los caracteres que no sean letras deben copiarse intactos a la salida.

! Ejercicio 3.5.5: En SQL, las palabras clave y los identificadores son sensibles a mayúsculas y minúsculas. Escriba un programa en **Lex** que reconozca las palabras clave **SELECT**, **FROM** y **WHERE** (en cualquier combinación de letras mayúsculas y minúsculas), y el token **ID**, que para los fines de este ejercicio puede considerar como cualquier secuencia de letras y dígitos, empezando con una letra. No tiene que instalar los identificadores en una tabla de símbolos, pero debe indicar cuál sería la diferencia en la función “install” de la que describimos para los identificadores sensibles a mayúsculas y minúsculas, como en la figura 3.23.

3.6 Autómatas finitos

Ahora vamos a descubrir cómo `Lex` convierte su programa de entrada en un analizador léxico. En el corazón de la transición se encuentra el formalismo conocido como *autómatas finitos*. En esencia, estos consisten en gráficos como los diagramas de transición de estados, con algunas diferencias:

1. Los autómatas finitos son *reconocedores*; sólo dicen “sí” o “no” en relación con cada posible cadena de entrada.
2. Los autómatas finitos pueden ser de dos tipos:
 - (a) Los *autómatas finitos no deterministas* (AFN) no tienen restricciones en cuanto a las etiquetas de sus líneas. Un símbolo puede etiquetar a varias líneas que surgen del mismo estado, y ϵ , la cadena vacía, es una posible etiqueta.
 - (b) Los *autómatas finitos deterministas* (AFD) tienen, para cada estado, y para cada símbolo de su alfabeto de entrada, exactamente una línea con ese símbolo que sale de ese estado.

Tanto los autómatas finitos deterministas como los no deterministas son capaces de reconocer los mismos lenguajes. De hecho, estos lenguajes son exactamente los mismos lenguajes, conocidos como *lenguajes regulares*, que pueden describir las expresiones regulares.⁴

3.6.1 Autómatas finitos no deterministas

Un *autómata finito no determinista* (AFN) consiste en:

1. Un conjunto finito de estados S .
2. Un conjunto de símbolos de entrada Σ , el *alfabeto de entrada*. Suponemos que ϵ , que representa a la cadena vacía, nunca será miembro de Σ .
3. Una *función de transición* que proporciona, para cada estado y para cada símbolo en $\Sigma \cup \{\epsilon\}$, un conjunto de *estados siguientes*.
4. Un estado s_0 de S , que se distingue como el *estado inicial*.
5. Un conjunto de estados F , un subconjunto de S , que se distinguen como los *estados aceptantes* (o *estados finales*).

Podemos representar un AFN o AFD mediante un *gráfico de transición*, en donde los nodos son estados y los flancos Indecidibles representan a la función de transición. Hay un flanco Indecidible a , que va del estado s al estado t si, y sólo si t es uno de los estados siguientes para el estado s y la entrada a . Este gráfico es muy parecido a un diagrama de transición, excepto que:

⁴Hay una pequeña laguna: según la definición que les dimos, las expresiones regulares no pueden describir el lenguaje vacío, ya que no es conveniente utilizar este patrón en la práctica. No obstante, los autómatas finitos *pueden* definir el lenguaje vacío. En teoría, \emptyset se trata como una expresión regular adicional, para el único fin de definir el lenguaje vacío.

- a) El mismo símbolo puede etiquetar flancos de un estado hacia varios estados distintos.
- b) Un flanco puede etiquetarse por ϵ , la cadena vacía, en vez de, o además de, los símbolos del alfabeto de entrada.

Ejemplo 3.14: El gráfico de transición para un AFN que reconoce el lenguaje de la expresión regular $(a|b)^*abb$ se muestra en la figura 3.24. Utilizaremos este ejemplo abstracto, que describe a todas las cadenas de as y bs que terminan en la cadena específica abb , a lo largo de esta sección. No obstante, es similar a las expresiones regulares que describen lenguajes de verdadero interés. Por ejemplo, una expresión que describe a todos los archivos cuyo nombre termina en $.o$ es **cualquiera** $^*.$ o , en donde **cualquiera** representa a cualquier carácter imprimible.

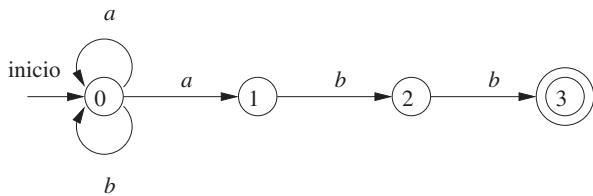


Figura 3.24: Un autómata finito no determinista

Siguiendo nuestra convención para los diagramas de transición, el doble círculo alrededor del estado 3 indica que este estado es de estados. Observe que la única forma de llegar del estado inicial 0 al estado de aceptación es seguir cierto camino que permanezca en el estado 0 durante cierto tiempo, y después vaya a los estados 1, 2 y 3 leyendo abb de la entrada. Por ende, las únicas cadenas que llegan al estado de aceptación son las que terminan en abb . \square

3.6.2 Tablas de transición

También podemos representar a un AFN mediante una *tabla de transición*, cuyas filas corresponden a los estados, y cuyas columnas corresponden a los símbolos de entrada y a ϵ . La entrada para un estado dado y la entrada es el valor de la función de transición que se aplica a esos argumentos. Si la función de transición no tiene información acerca de ese par estado-entrada, colocamos \emptyset en la tabla para ese estado.

Ejemplo 3.15: La tabla de transición para el AFN de la figura 3.24 se muestra en la figura 3.25. \square

La tabla de transición tiene la ventaja de que podemos encontrar con facilidad las transiciones en cierto estado y la entrada. Su desventaja es que ocupa mucho espacio, cuando el alfabeto de entrada es extenso, aunque muchos de los estados no tengan movimientos en la mayoría de los símbolos de entrada.

ESTADO	a	b	ϵ
0	{0, 1}	{0}	\emptyset
1	\emptyset	{2}	\emptyset
2	\emptyset	{3}	\emptyset
3	\emptyset	\emptyset	\emptyset

Figura 3.25: Tabla de transición para el AFN de la figura 3.24

3.6.3 Aceptación de las cadenas de entrada mediante los autómatas

Un AFN *acepta* la cadena de entrada x si, y sólo si hay algún camino en el grafo de transición, desde el estado inicial hasta uno de los estados de aceptación, de forma que los símbolos a lo largo del camino deletreen a x . Observe que las etiquetas ϵ a lo largo del camino se ignoran, ya que la cadena vacía no contribuye a la cadena que se construye a lo largo del camino.

Ejemplo 3.16: El AFN de la figura 3.24 acepta la cadena $aabb$. El camino etiquetado por $aabb$ del estado 0 al estado 3 que demuestra este hecho es:

$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

Observe que varios caminos etiquetadas por la misma cadena pueden conducir hacia estados distintos. Por ejemplo, el camino

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

es otro camino que parte del estado 0, etiquetada por la cadena $aabb$. Este camino conduce al estado 0, que no de aceptación. Sin embargo, recuerde que un AFN acepta a una cadena siempre y cuando haya *cierto* camino etiquetado por esa cadena, que conduzca del estado inicial a un estado de aceptación. Es irrelevante la existencia de otros caminos que conduzcan a un estado de no aceptación. \square

El *lenguaje definido* (o *aceptado*) por un AFN es el conjunto de cadenas que etiquetan cierto camino, del estado inicial a un estado de aceptación. Como mencionamos antes, el AFN de la figura 3.24 define el mismo lenguaje que la expresión regular $(a|b)^*abb$; es decir, todas las cadenas del alfabeto $\{a, b\}$ que terminen en abb . Podemos usar $L(A)$ para representar el lenguaje aceptado por el autómata A .

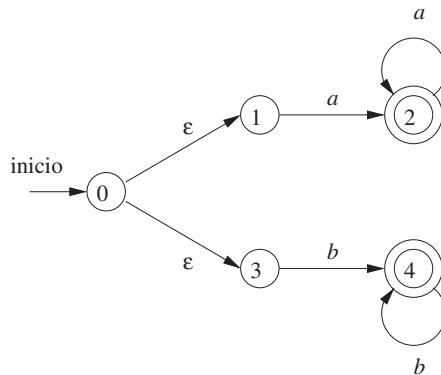
Ejemplo 3.17: La figura 3.26 es un AFN que acepta a $L(aa^*|bb^*)$. La cadena aaa se acepta debido a el camino.

$$0 \xrightarrow{\epsilon} 1 \xrightarrow{a} 2 \xrightarrow{a} 2 \xrightarrow{a} 2$$

Observe que las ϵ s “desaparecen” en una concatenación, por lo que la etiqueta del camino es aaa . \square

3.6.4 Autómatas finitos deterministas

Un *autómata finito determinista* (AFD) es un caso especial de un AFN, en donde:

Figura 3.26: AFN que acepta a $aa^*|bb^*$

1. No hay movimientos en la entrada ϵ .
2. Para cada estado s y cada símbolo de entrada a , hay exactamente una línea que surge de s , Indecidible como a .

Si utilizamos una tabla de transición para representar a un AFD, entonces cada entrada es un solo estado. Por ende, podemos representar a este estado sin las llaves que usamos para formar los conjuntos.

Mientras que el AFN es una representación abstracta de un algoritmo para reconocer las cadenas de cierto lenguaje, el AFD es un algoritmo simple y concreto para reconocer cadenas. Sin duda es afortunado que cada expresión regular y cada AFN puedan convertirse en un AFD que acepte el mismo lenguaje, ya que es el AFD el que en realidad implementamos o simulamos al construir analizadores léxicos. El siguiente algoritmo muestra cómo aplicar un AFD a una cadena.

Algoritmo 3.18: Simulación de un AFD.

ENTRADA: Una cadena de entrada x , que se termina con un carácter de fin de archivo **eof**. Un AFD D con el estado inicial s_0 , que acepta estados F , y la función de transición *mover*.

SALIDA: Responde “sí” en caso de que D acepte a x ; “no” en caso contrario.

MÉTODO: Aplicar el algoritmo de la figura 3.27 a la cadena de entrada x . La función *mover*(s, c) proporciona el estado para el cual hay un flanco desde el estado s sobre la entrada c . La función *sigCar* devuelve el siguiente carácter de la cadena de entrada x . \square

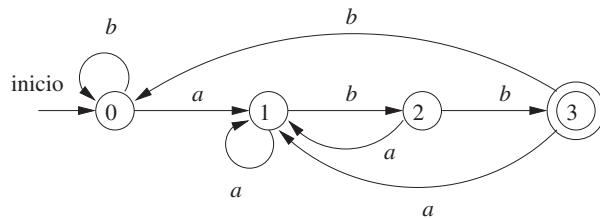
Ejemplo 3.19: En la figura 3.28 vemos el grafo de transición de un AFD que acepta el lenguaje $(a|b)^*abb$, el mismo que acepta el AFN de la figura 3.24. Dada la cadena de entrada $ababb$, este AFD introduce la secuencia de estados 0, 1, 2, 1, 2, 3 y devuelve “sí”. \square

```

 $s = s_0;$ 
 $c = sigCar();$ 
while (  $c \neq \text{eof}$  ) {
     $s = mover(s, c);$ 
     $c = sigCar();$ 
}
if (  $s$  está en  $F$  ) return "si";
else return "no";

```

Figura 3.27: Simulación de un AFD

Figura 3.28: AFD que acepta a $(a|b)^*abb$

3.6.5 Ejercicios para la sección 3.6

Ejercicio 3.6.1: La figura 3.19 en los ejercicios de la sección 3.4 calcula la función de fallo para el algoritmo KMP. Muestre cómo, dada esa función de fallo, podemos construir, a partir de una palabra clave $b_1b_2 \dots b_n$, un AFD de $n + 1$ estados que reconozca a $.*b_1b_2 \dots b_n$, en donde el punto representa a “cualquier carácter”. Además, este AFD puede construirse en un tiempo $O(n)$.

Ejercicio 3.6.2: Diseñe autómatas finitos (deterministas o no) para cada uno de los lenguajes del ejercicio 3.3.5.

Ejercicio 3.6.3: Para el AFN de la figura 3.29, indique todos los caminos etiquetadas como $aabb$. ¿El AFN acepta a $aabb$?

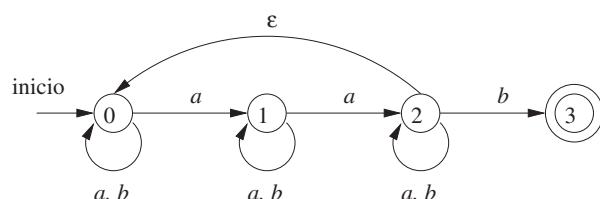


Figura 3.29: AFN para el ejercicio 3.6.3

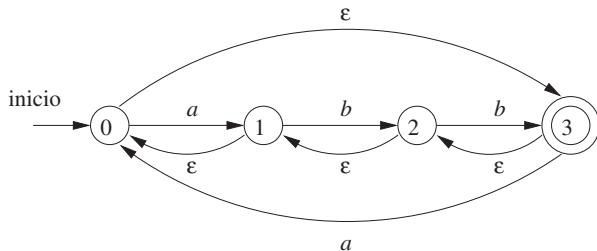


Figura 3.30: AFN para el ejercicio 3.6.4

Ejercicio 3.6.4: Repita el ejercicio 3.6.3 para el AFN de la figura 3.30.

Ejercicio 3.6.5: Proporcione las tablas de transición para el AFN de:

- El ejercicio 3.6.3.
- El ejercicio 3.6.4.
- La figura 3.26.

3.7 De las expresiones regulares a los autómatas

La expresión regular es la notación de elección para describir analizadores léxicos y demás software de procesamiento de patrones, como se vio en la sección 3.5. No obstante, la implementación de ese software requiere la simulación de un AFD, como en el algoritmo 3.18, o tal vez la simulación de un AFN. Como, por lo general, un AFN tiene la opción de moverse sobre un símbolo de entrada (como lo hace la figura 3.24, sobre la entrada a desde el estado 0) o sobre ϵ (como lo hace la figura 3.26 desde el estado 0), o incluso la opción de realizar una transición sobre ϵ o sobre un símbolo de entrada real, su simulación es menos simple que la de un AFD. Por ende, con frecuencia es importante convertir un AFN a un AFD que acepte el mismo lenguaje.

En esta sección veremos primero cómo convertir AFNs a AFDs. Después utilizaremos esta técnica, conocida como “la construcción de subconjuntos”, para producir un algoritmo útil para simular a los AFNs de forma directa, en situaciones (aparte del análisis léxico) en donde la conversión de AFN a AFD requiere más tiempo que la simulación directa. Después, mostraremos cómo convertir las expresiones regulares en AFNs, a partir de lo cual puede construir un AFD, si lo desea. Concluiremos con una discusión de las concesiones entre tiempo y espacio, inherentes en los diversos métodos para implementar expresiones regulares, y veremos cómo puede elegir el método apropiado para su aplicación.

3.7.1 Conversión de un AFN a AFD

La idea general de la construcción de subconjuntos es que cada estado del AFD construido corresponde a un conjunto de estados del AFN. Después de leer la entrada $a_1 a_2 \dots a_n$, el AFD

se encuentra en el estado que corresponde al conjunto de estados que el AFN puede alcanzar, desde su estado inicial, siguiendo los caminos etiquetados como $a_1a_2 \cdots a_n$.

Es posible que el número de estados del AFD sea exponencial en el número de estados del AFN, lo cual podría provocar dificultades al tratar de implementar este AFD. No obstante, parte del poder del método basado en autómatas para el análisis léxico es que para los lenguajes reales, el AFN y el AFD tienen aproximadamente el mismo número de estados, y no se ve el comportamiento exponencial.

Algoritmo 3.20: La *construcción de subconjuntos* de un AFD, a partir de un AFN.

ENTRADA: Un AFN N .

SALIDA: Un AFD D que acepta el mismo lenguaje que N .

MÉTODO: Nuestro algoritmo construye una tabla de transición $Dtran$ para D . Cada estado de D es un conjunto de estados del AFN, y construimos $Dtran$ para que D pueda simular “en paralelo” todos los posibles movimientos que N puede realizar sobre una cadena de entrada dada. Nuestro primer problema es manejar las transiciones ϵ de N en forma apropiada. En la figura 3.31 vemos las definiciones de varias funciones que describen cálculos básicos en los estados de N que son necesarios en el algoritmo. Observe que s es un estado individual de N , mientras que T es un conjunto de estados de N .

OPERACIÓN	DESCRIPCIÓN
ϵ - <i>cerradura</i> (s)	Conjunto de estados del AFN a los que se puede llegar desde el estado s del AFN, sólo en las transiciones ϵ .
ϵ - <i>cerradura</i> (T)	Conjunto de estados del AFN a los que se puede llegar desde cierto estado s del AFN en el conjunto T , sólo en las transiciones ϵ ; $= \bigcup_{s \in T} \epsilon$ - <i>cerradura</i> (s).
<i>mover</i> (T, a)	Conjunto de estados del AFN para los cuales hay una transición sobre el símbolo de entrada a , a partir de cierto estado s en T .

Figura 3.31: Operaciones sobre los estados del AFN

Debemos explorar esos conjuntos de estados en los que puede estar N después de ver cierta cadena de entrada. Como base, antes de leer el primer símbolo de entrada, N puede estar en cualquiera de los estados de ϵ -*cerradura*(s_0), en donde s_0 es su estado inicial. Para la inducción, suponga que N puede estar en el conjunto de estados T después de leer la cadena de entrada x . Si a continuación lee la entrada a , entonces N puede pasar de inmediato a cualquiera de los estados en *mover*(T, a). No obstante, después de leer a también podría realizar varias transiciones ϵ ; por lo tanto, N podría estar en cualquier estado de ϵ -*cerradura*(*mover*(T, a)) después de leer la entrada xa . Siguiendo estas ideas, la construcción del conjunto de estados de D , *Destados*, y su función de transición *Dtran*, se muestran en la figura 3.32.

El estado inicial de D es ϵ -*cerradura*(s_0), y los estados de aceptación de D son todos aquellos conjuntos de estados de N que incluyen cuando menos un estado de aceptación de N . Para

completar nuestra descripción de la construcción de subconjuntos, sólo necesitamos mostrar cómo al principio, ϵ -*cerradura*(s_0) es el único estado en *Destados*, y está sin marcar:

```

while ( hay un estado sin marcar  $T$  en Destados ) {
    marcar  $T$ ;
    for ( cada símbolo de entrada  $a$  ) {
         $U = \epsilon$ -cerradura(mover( $T, a$ ));
        if (  $U$  no está en Destados )
            agregar  $U$  como estado sin marcar a Destados;
             $Dtran[T, a] = U$ ;
    }
}

```

Figura 3.32: La construcción de subconjuntos

ϵ -*cerradura*(T) se calcula para cualquier conjunto de estados T del AFN. Este proceso, que se muestra en la figura 3.33, es una búsqueda simple y directa en un gráfo, a partir de un conjunto de estados. En este caso, imagine que sólo están disponibles las líneas Indecidibles como ϵ en el gráfico. \square

```

meter todos los estados de  $T$  en pila;
inicializar  $\epsilon$ -cerradura( $T$ ) con  $T$ ;
while ( pila no está vacía ) {
    sacar  $t$ , el elemento superior, de la pila;
    for ( cada estado  $u$  con un flanco de  $t$  a  $u$ , Indecidible como  $\epsilon$  )
        if (  $u$  no está en  $\epsilon$ -cerradura( $T$ ) ) {
            agregar  $u$  a  $\epsilon$ -cerradura( $T$ );
            meter  $u$  en la pila;
        }
}

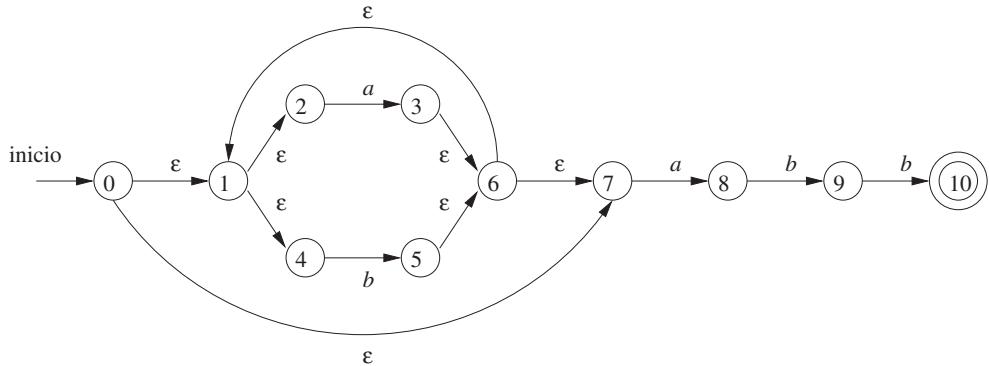
```

Figura 3.33: Cálculo de ϵ -*cerradura*(T)

Ejemplo 3.21: La figura 3.34 muestra a otro AFN que acepta a $(a|b)^*abb$; resulta que es el que vamos a construir directamente a partir de esta expresión regular en la sección 3.7. Vamos a aplicar el Algoritmo 3.20 a la figura 3.29.

El estado inicial A del AFD equivalente es ϵ -*cerradura*(0), o $A = \{0, 1, 2, 4, 7\}$, ya que éstos son los mismos estados a los que se puede llegar desde el estado 0, a través de un camino cuyas líneas tienen todos la etiqueta ϵ . Observe que un camino pueda tener cero líneas, por lo que se puede llegar al estado 0 desde sí mismo, mediante un camino etiquetada como ϵ .

El alfabeto de entrada es $\{a, b\}$. Por ende, nuestro primer paso es marcar A y calcular $Dtran[A, a] = \epsilon$ -*cerradura*(*mover*(A, a)) y $Dtran[A, b] = \epsilon$ -*cerradura*(*mover*(A, b)). De los estados 0, 1, 2, 4 y 7, sólo 2 y 7 tienen transiciones sobre a , hacia 3 y 8, respectivamente. Por ende, *mover*(A, a) = $\{3, 8\}$. Además, ϵ -*cerradura*($\{3, 8\}$) = $\{1, 2, 3, 4, 6, 7, 8\}$, por lo cual concluimos que:

Figura 3.34: El AFN N para $(a|b)^*abb$

$$Dtran[A, a] = \epsilon\text{-cerradura}(mover(A, a)) = \epsilon\text{-cerradura}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$$

Vamos a llamar a este conjunto B , de manera que $Dtran[A, a] = B$.

Ahora, debemos calcular $Dtran[A, b]$. De los estados en A sólo el 4 tiene una transición sobre b , y va al estado 5. Por ende,

$$Dtran[A, b] = \epsilon\text{-cerradura}(\{5\}) = \{1, 2, 4, 6, 7\}$$

Vamos a llamar al conjunto anterior C , de manera que $Dtran[A, b] = C$.

ESTADO DEL AFN	ESTADO DEL AFD	a	b
$\{0, 1, 2, 4, 7\}$	A	B	C
$\{1, 2, 3, 4, 6, 7, 8\}$	B	B	D
$\{1, 2, 4, 5, 6, 7\}$	C	B	C
$\{1, 2, 4, 5, 6, 7, 9\}$	D	B	E
$\{1, 2, 3, 5, 6, 7, 10\}$	E	B	C

Figura 3.35: Tabla de transición $Dtran$ para el AFD D

Si continuamos este proceso con los conjuntos desmarcados B y C , en un momento dado llegaremos a un punto en el que todos los estados del AFD estén marcados. Esta conclusión se garantiza, ya que “sólo” hay 2^{11} subconjuntos distintos de un conjunto de once estados de un AFN. Los cinco estados distintos del AFD que realmente construimos, sus correspondientes conjuntos de estados del AFN, y la tabla de transición para el AFD D se muestran en la figura 3.35, y el gráfico de transición para D está en la figura 3.36. El estado A es el estado inicial, y el estado E , que contiene el estado 10 del AFN, es el único estado de aceptación.

Observe que D tiene un estado más que el AFD de la figura 3.28 para el mismo lenguaje. Los estados A y C tienen la misma función de movimiento, por lo cual pueden combinarse. En la sección 3.9.6 hablaremos sobre la cuestión de reducir al mínimo el número de estados de un AFD. \square

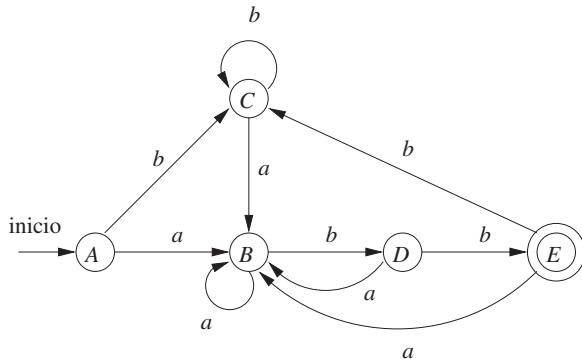


Figura 3.36: Resultado de aplicar la construcción de subconjuntos a la figura 3.34

3.7.2 Simulación de un AFN

Una estrategia que se ha utilizado en varios programas de edición de texto es la de construir un AFN a partir de una expresión regular, y después simular el AFN utilizando algo como una construcción de subconjuntos “sobre la marcha”. La simulación se describe a continuación.

Algoritmo 3.22: Simulación de un AFN.

ENTRADA: Una cadena de entrada x que termina con un carácter de fin de línea **eof**. Un AFN N con el estado inicial s_0 , que acepta estados F , y la función de transición *mover*.

SALIDA: Responde “sí” en caso de que M acepte a x ; “no” en caso contrario.

MÉTODO: El algoritmo mantiene un conjunto de estados actuales S , aquellos a los que se llega desde s_0 siguiendo un camino etiquetado por las entradas leídas hasta ese momento. Si c es el siguiente carácter de entrada leído por la función *sigCar()*, entonces primero calculamos $mover(S, c)$ y después cerramos ese conjunto usando ϵ -*cerradura()*. En la figura 3.37 se muestra un bosquejo del algoritmo. \square

```

1)    $S = \epsilon$ -cerradura( $s_0$ );
2)    $c = \text{sigCar}()$ ;
3)   while (  $c \neq \text{eof}$  ) {
4)        $S = \epsilon$ -cerradura(mover( $S, c$ ));
5)        $c = \text{sigCar}()$ ;
6)   }
7)   if (  $S \cap F \neq \emptyset$  ) return "sí";
8)   else return "no";

```

Figura 3.37: Simulación de un AFN

3.7.3 Eficiencia de la simulación de un AFN

Si se implementa con cuidado, el algoritmo 3.22 puede ser bastante eficiente. Como las ideas implicadas son útiles en una variedad de algoritmos similares que involucran la búsqueda de gráficos, veremos esta implementación con más detalle. Las estructuras de datos que necesitamos son:

1. Dos pilas, cada una de las cuales contiene un conjunto de estados del AFN. Una de estas pilas, *estadosAnt*, contiene el conjunto “actual” de estados; es decir, el valor de S del lado derecho de la línea (4) en la figura 3.37. La segunda, *estadosNuev*, contiene el “siguiente” conjunto de estados: S del lado izquierdo de la línea (4). Hay un paso que no se ve en el que, a medida que avanzamos por el ciclo de las líneas (3) a la (6), *estadosNuev* se transfiere a *estadosAnt*.
2. Un arreglo booleano *yaEstaEn*, indexado por los estados del AFN, para indicar cuáles estados ya están en *estadosNuev*. Aunque el arreglo y la pila contienen la misma información, es mucho más rápido interrogar a *yaEstaEn*[s] que buscar el estado s en la pila *estadosNuev*. Es por eficiencia que mantenemos ambas representaciones.
3. Un arreglo bidimensional *mover*[s, a] que contiene la tabla de transición del AFN. Las entradas en esta tabla, que son conjuntos de estados, se representan mediante listas enlazadas.

Para implementar la línea (1) de la figura 3.37, debemos establecer cada entrada en el arreglo *yaEstaEn* a FALSE, después para cada estado s en ϵ -*cerradura*(s_0), hay que meter s en *estadosAnt* y *yaEstaEn*[s] a TRUE. Esta operación sobre el estado s , y la implementación de la línea (4) también, se facilitan mediante una función a la que llamaremos *agregarEstado*(s). Esta función mete el estado s en *estadosNuev*, establece *yaEstaEn*[s] a TRUE, y se llama a sí misma en forma recursiva sobre los estados en *mover*[s, ϵ] para poder ampliar el cálculo de ϵ -*cerradura*(s). Sin embargo, para evitar duplicar el trabajo, debemos tener cuidado de nunca llamar a *agregarEstado* en un estado que ya se encuentre en la pila *estadosNuev*. La figura 3.38 muestra un bosquejo de esta función.

```

9)  agregarEstado( $s$ ) {
10)    meter  $s$  en estadosNuev;
11)    yaEstaEn[ $s$ ] = TRUE;
12)    for (  $t$  en mover[ $s, \epsilon$ ] )
13)      if ( !yaEstaEn( $t$ ) )
14)        agregarEstado( $t$ );
15)    }

```

Figura 3.38: Agregar un nuevo estado s , que sabemos no se encuentra en *estadosNuev*

Para implementar la línea (4) de la figura 3.37, analizamos cada estado s en *estadosAnt*. Primero buscamos el conjunto de estados *mover*[s, c], en donde c es la siguiente entrada, y para

cada uno de esos estados que no se encuentren ya en *estadosNuev*, le aplicamos *agregarEstado*. Observe que *agregarEstado* tiene el efecto de calcular ϵ -*cerradura* y de agregar todos esos estados a *estadosNuev* también, si no se encontraban ya agregados. Esta secuencia de pasos se resume en la figura 3.39.

```

16)  for ( s en estadosAnt ) {
17)      for ( t en mover[s, c] )
18)          if ( !yaEstaEn[t] )
19)              agregarEstado(t);
20)          sacar s de estadosAnt;
21)      }

22)  for ( s en estadosNuev ) {
23)      sacar s de estadosNuev;
24)      meter s a estadosAnt;
25)      yaEstaEn[s] = FALSE;
26)  }

```

Figura 3.39: Implementación del paso (4) de la figura 3.37

Ahora, suponga que el AFN *N* tiene *n* estados y *m* transiciones; es decir, *m* es la suma de todos los estados del número de símbolos (ϵ) sobre los cuales el estado tiene una transición de salida. Sin contar la llamada a *agregarEstado* en la línea (19) de la figura 3.39, el tiempo invertido en el ciclo de las líneas (16) a (21) es $O(n)$. Es decir, podemos recorrer el ciclo cuando mucho *n* veces, y cada paso del ciclo requiere un trabajo constante, excepto por el tiempo invertido en *agregarEstado*. Lo mismo se aplica al ciclo de las líneas (22) a la (26).

Durante una ejecución de la figura 3.39, es decir, del paso (4) de la figura 3.37, sólo es posible llamar una vez a *agregarEstado* sobre un estado dado. La razón es que, cada vez que llamamos a *agregarEstado*(*s*), establecemos *yaEstaEn*[*s*] a TRUE en la línea 11 de la figura 3.39. Una vez que *yaEstaEn*[*s*] es TRUE, las pruebas en la línea (13) de la figura 3.38, y la línea (18) de la figura 3.39 evitan otra llamada.

El tiempo invertido en una llamada a *agregarEstado*, exclusivo del tiempo invertido en las llamadas recursivas en la línea (14), es $O(1)$ para las líneas (10) y (11). Para las líneas (12) y (13), el tiempo depende de cuántas transiciones ϵ haya al salir del estado *s*. No conocemos este número para un estado dado, pero sabemos que hay cuando menos *m* transiciones en total, saliendo de todos los estados. Como resultado, el tiempo adicional invertido en las líneas (11) de todas las llamadas a *agregarEstado* durante una ejecución del código de la figura 3.39 es $O(m)$. El agregado para el resto de los pasos de *agregarEstado* es $O(n)$, ya que es una constante por llamada, y hay cuando menos *n* llamadas.

Concluimos que, si se implementa en forma apropiada, el tiempo para ejecutar la línea (4) de la figura 3.37 es $O(n + m)$. El resto del ciclo while de las líneas (3) a la (6) requiere de un tiempo $O(1)$ por iteración. Si la entrada *x* es de longitud *k*, entonces el trabajo total en ese ciclo es $O((k(n + m))$. La línea (1) de la figura 3.37 puede ejecutarse en un tiempo $O(n + m)$,

Notación O grande (Big-Oh)

Una expresión como $O(n)$ es una abreviación para “cuando menos ciertos tiempos n constantes”. Técnicamente, decimos que una función $f(n)$, tal vez el tiempo de ejecución de algún paso de un algoritmo, es $O(g(n))$ si hay constantes c y n_0 , de tal forma que cada vez que $n \geq n_0$, es cierto que $f(n) \leq cg(n)$. Un modismo útil es “ $O(1)$ ”, que significa “alguna constante”. El uso de esta *notación Big-Oh* nos permite evitar profundizar demasiado en los detalles acerca de lo que contamos como unidad de tiempo de ejecución, y aún así nos permite expresar la velocidad a la cual crece el tiempo de ejecución de un algoritmo.

ya que en esencia consta de los pasos de la figura 3.39, en donde *estadosAnt* sólo contienen el estado s_0 . Las líneas (2), (7) y (8) requieren de un tiempo $O(1)$ cada una. Por ende, el tiempo de ejecución del Algoritmo 3.22, si se implementa en forma apropiada, es $O((k(n + m))$. Es decir, el tiempo requerido es proporcional a la longitud de la entrada multiplicada por el tamaño (nodos más líneas) del gráfico de transición.

3.7.4 Construcción de un AFN a partir de una expresión regular

Ahora le proporcionaremos un algoritmo para convertir cualquier expresión regular en un AFN que defina el mismo lenguaje. El algoritmo está orientado a la sintaxis, ya que recorre en forma recursiva hacia arriba el árbol sintáctico para la expresión regular. Para cada subexpresión, el algoritmo construye un AFN con un solo estado aceptante.

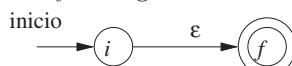
Algoritmo 3.23: El algoritmo de McNaughton-Yamada-Thompson para convertir una expresión regular en un AFN.

ENTRADA: Una expresión regular r sobre el alfabeto Σ .

SALIDA: Un AFN N que acepta a $L(r)$.

MÉTODO: Empezar por un análisis sintáctico de r para obtener las subexpresiones que la constituyen. Las reglas para construir un AFN consisten en reglas básicas para el manejo de subexpresiones sin operadores, y reglas inductivas para construir AFNs más largos, a partir de los AFNs para las subexpresiones inmediatas de una expresión dada.

BASE: Para la expresión ϵ , se construye el siguiente AFN:



Aquí, i es un nuevo estado, el estado inicial de este AFN, y f es otro nuevo estado, el estado aceptante para el AFN.

Para cualquier subexpresión a en Σ , se construye el siguiente AFN:



en donde otra vez i y f son nuevos estados, los estados inicial y de aceptación, respectivamente. Observe que en ambas construcciones básicas, construimos un AFN distinto, con nuevos estados, para cada ocurrencia de ϵ o alguna a como subexpresión de r .

INDUCCIÓN: Suponga que $N(s)$ y $N(t)$ son AFNs para las expresiones regulares s y t , respectivamente.

- a) Suponga que $r = s|t$. Entonces $N(r)$, el AFN para r , se construye como en la figura 3.40. Aquí, i y f son nuevos estados, los estados inicial y de aceptación de $N(r)$, respectivamente. Hay transiciones ϵ desde i hasta los estados iniciales de $N(s)$ y $N(t)$, y cada uno de los estados de aceptación tiene transiciones ϵ hacia el estado de aceptación f . Observe que los estados aceptantes de $N(s)$ y $N(t)$ no son de aceptación en $N(r)$. Como cualquier camino de i a f debe pasar por $N(s)$ o $N(t)$ exclusivamente, y como la etiqueta de ese camino no se modifica por el hecho de que las ϵ s salen de i o entran a f , concluimos que $N(r)$ acepta a $L(s) \cup L(t)$, que es lo mismo que $L(r)$. Es decir, la figura 3.40 es una construcción correcta para el operador de unión.

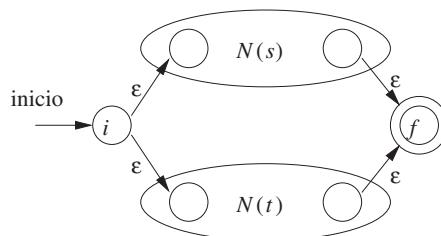


Figura 3.40: AFN para la unión de dos expresiones regulares

- b) Suponga que $r = st$. Entonces, construya $N(r)$ como en la figura 3.41. El estado inicial de $N(s)$ se convierte en el estado inicial de $N(r)$, y el estado de aceptación de $N(t)$ es el único estado de aceptación de $N(r)$. El estado de aceptación de $N(s)$ y el estado inicial de $N(t)$ se combinan en un solo estado, con todas las transiciones que entran o salen de cualquiera de esos estados. Un camino de i a f en la figura 3.41 debe pasar primero a través de $N(s)$ y, por lo tanto, su etiqueta empezará con alguna cadena en $L(s)$. Después, el camino continúa a través de $N(t)$, por lo que la etiqueta del camino termina con una cadena en $L(t)$. Como pronto argumentaremos, los estados de aceptación nunca tienen líneas de salida y los estados iniciales nunca tienen líneas de entrada, por lo que no es posible que un camino vuelva a entrar a $N(s)$ después de salir de este estado. Por ende, $N(r)$ acepta exactamente a $L(s)L(t)$, y es un AFN correcto para $r = st$.

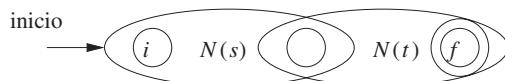


Figura 3.41: AFN para la concatenación de dos expresiones regulares

- c) Suponga que $r = s^*$. Entonces, para r construimos el AFN $N(r)$ que se muestra en la figura 3.42. Aquí, i y f son nuevos estados, el estado inicial y el único estado aceptante de $N(r)$. Para ir de i a f , podemos seguir el camino introducido, etiquetado como ϵ , el cual se ocupa de la única cadena en $L(s)^0$, podemos pasar al estado inicial de $N(s)$, a través de ese AFN, y después de su estado de aceptación regresar a su estado inicial, cero o más veces. Estas opciones permiten que $N(r)$ acepte a todas las cadenas en $L(s)^1, L(s)^2$, y así en lo sucesivo, de forma que el conjunto entero de cadenas aceptadas por $N(r)$ es $L(s^*)$.

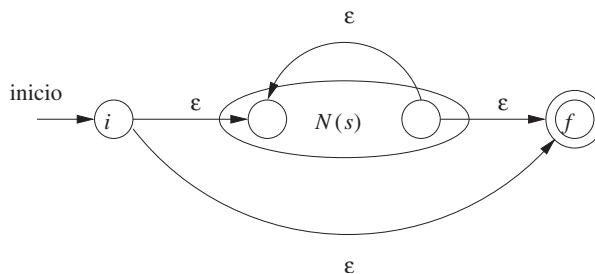


Figura 3.42: AFN para el cierre de una expresión regular

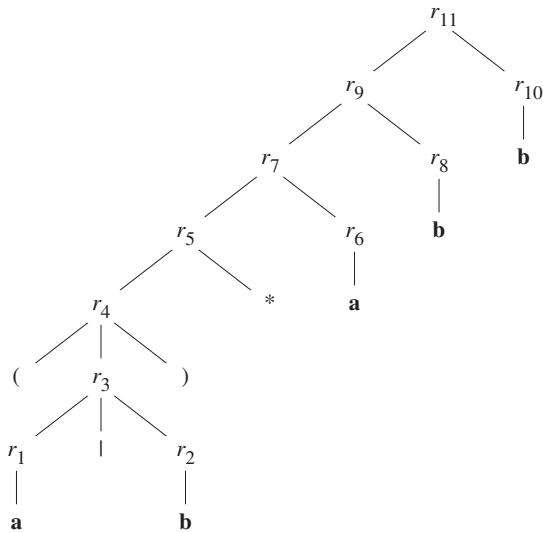
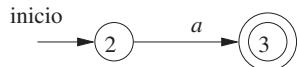
- d) Por último, suponga que $r = (s)$. Entonces, $L(r) = L(s)$, y podemos usar el AFN $N(s)$ como $N(r)$.

□

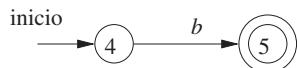
La descripción del método en el Algoritmo 3.23 contiene sugerencias de por qué la construcción inductiva funciona en la forma esperada. No proporcionaremos una prueba formal de su correcto funcionamiento, pero sí presentaremos varias propiedades de los AFNs construidos, además del importantísimo hecho de que $N(r)$ acepta el lenguaje $L(r)$. Estas propiedades son interesantes y útiles para realizar una prueba formal.

1. $N(r)$ tiene cuando menos el doble de estados, que equivalen a los operadores y operandos en r . Este enlace resulta del hecho de que cada paso del algoritmo crea cuando menos dos nuevos estados.
2. $N(r)$ tiene un estado inicial y un estado de aceptación. El estado de aceptación no tiene transiciones salientes, y el estado inicial no tiene transiciones entrantes.
3. Cada estado de $N(r)$ que no sea el estado de aceptación tiene una transición saliente en un símbolo en Σ , o dos transiciones salientes, ambas en ϵ .

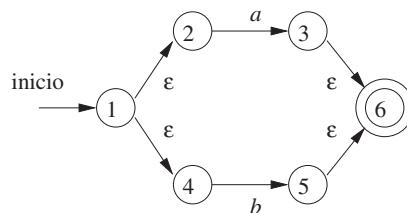
Ejemplo 3.24: Vamos a usar el Algoritmo 3.23 para construir un AFN para $r = (\mathbf{a}|\mathbf{b})^*\mathbf{a}\mathbf{b}$. La figura 3.43 muestra un árbol de análisis sintáctico para r , que es análogo a los árboles de análisis sintáctico construidos para las expresiones aritméticas en la sección 2.2.3. Para la subexpresión r_1 , la primera **a**, construimos el siguiente AFN:

Figura 3.43: Árbol de análisis sintáctico para $(a|b)^*abb$ 

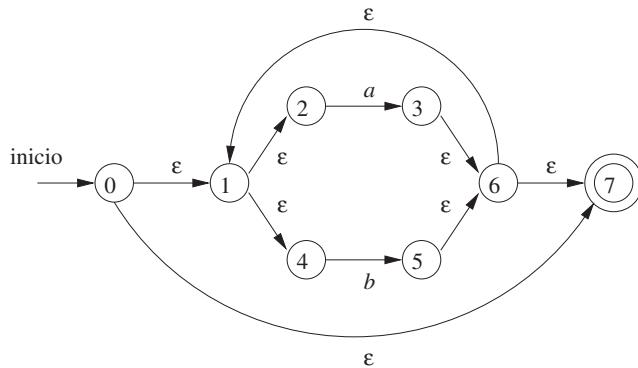
Hemos elegido los números de los estados para que sean consistentes con lo que sigue. Para r_2 construimos lo siguiente:



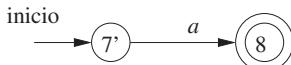
Ahora podemos combinar $N(r_1)$ y $N(r_2)$, mediante la construcción de la figura 3.40 para obtener el AFN para $r_3 = r_1|r_2$; este AFN se muestra en la figura 3.44.

Figura 3.44: AFN para r_3

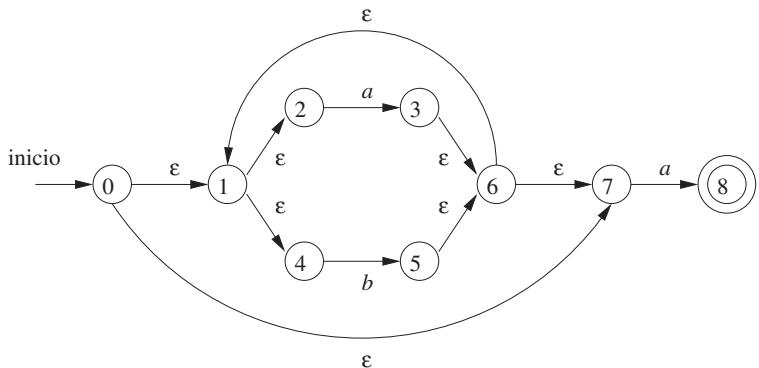
El AFN para $r_4 = (r_3)$ es el mismo que para r_3 . Entonces, el AFN para $r_5 = (r_3)^*$ es como se muestra en la figura 3.45. Hemos usado la construcción en la figura 3.42 para crear este AFN, a partir del AFN de la figura 3.44.

Figura 3.45: AFN para r_5

Ahora consideremos la subexpresión r_6 , que es otra **a**. Utilizamos la construcción básica para **a** otra vez, pero debemos usar nuevos estados. No se permite reutilizar el AFN que construimos para r_1 , aun cuando r_1 y r_6 son la misma expresión. El AFN para r_6 es:



Para obtener el AFN de $r_7 = r_5 \cdot r_6$, aplicamos la construcción de la figura 3.41. Combinamos los estados 7 y 7', con lo cual se produce el AFN de la figura 3.46. Si continuamos de esta forma con nuevos AFNs para las dos subexpresiones **b** llamadas r_8 y r_{10} , en un momento dado construiremos el AFN para $(\mathbf{a}|\mathbf{b})^* \mathbf{a} \mathbf{b} \mathbf{b}$ que vimos primero en la figura 3.34. \square

Figura 3.46: AFN para r_7

3.7.5 Eficiencia de los algoritmos de procesamiento de cadenas

Observamos que el Algoritmo 3.18 procesa una cadena x en un tiempo $O(|x|)$, mientras que en la sección 3.7.3 concluimos que podríamos simular un AFN en un tiempo proporcional al

producto de $|x|$ y el tamaño del gráfo de transición del AFN. Obviamente, es más rápido hacer un AFD que simular un AFN, por lo que podríamos preguntarnos por qué tendría sentido simular un AFN.

Una cuestión que puede favorecer a un AFN es que la construcción de los subconjuntos puede, en el peor caso, exponenciar el número de estados. Aunque en principio, el número de estados del AFD no influye en el tiempo de ejecución del Algoritmo 3.18, si el número de estados se vuelve tan grande que la tabla de transición no quepa en la memoria principal, entonces el verdadero tiempo de ejecución tendría que incluir la E/S en disco y, por ende, se elevaría de manera considerable.

Ejemplo 3.25: Considere la familia de lenguajes descritos por las expresiones regulares de la forma $L_n = (\mathbf{a}|\mathbf{b})^* \mathbf{a}(\mathbf{a}|\mathbf{b})^{n-1}$, es decir, cada lenguaje L_n consiste en cadenas de as y bs de tal forma que el n -ésimo carácter a la izquierda del extremo derecho contiene a a . Un AFN de $n + 1$ estados es fácil de construir. Permanece en su estado inicial bajo cualquier entrada, pero también tiene la opción, en la entrada a , de pasar al estado 1. Del estado 1 pasa al estado 2 en cualquier entrada, y así en lo sucesivo, hasta que en el estado n acepta. La figura 3.47 sugiere este AFN.

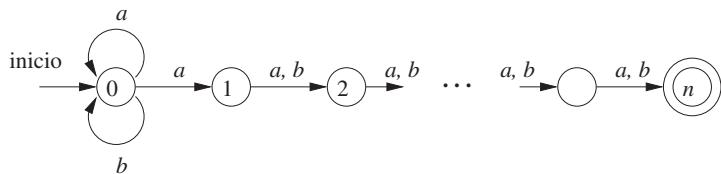


Figura 3.47: Un AFN que tiene mucho menos estados que el AFD equivalente más pequeño

No obstante, cualquier AFD para el lenguaje L_n debe tener cuando menos 2^n estados. No demostraremos este hecho, pero la idea es que si dos cadenas de longitud n pueden llevar al AFD al mismo estado, entonces podemos explotar la última posición en la que difieren las cadenas (y, por lo tanto, una debe tener a a y la otra a b) para continuarlas en forma idéntica, hasta que sean la misma en las últimas $n - 1$ posiciones. Entonces, el AFD estará en un estado en el que debe tanto aceptar como no aceptar. Por fortuna, como dijimos antes, es raro que en el análisis léxico se involucren patrones de este tipo, y no esperamos encontrar AFDs con números extravagantes de estados en la práctica. \square

Sin embargo, los generadores de analizadores léxicos y otros sistemas de procesamiento de cadenas empiezan a menudo con una expresión regular. Nos enfrentamos a la opción de convertir esta expresión en un AFN o AFD. El costo adicional de elegir un AFD es, por ende, el costo de ejecutar el Algoritmo 3.23 en el AFN (podríamos pasar directo de una expresión regular a un AFD, pero en esencia el trabajo es el mismo). Si el procesador de cadenas se va a ejecutar

muchas veces, como es el caso para el análisis léxico, entonces cualquier costo de convertir a un AFD es aceptable. No obstante, en otras aplicaciones de procesamiento de cadenas, como **Grez**, en donde el usuario especifica una expresión regular y uno de varios archivos en los que se va a buscar el patrón de esa expresión, puede ser más eficiente omitir el paso de construir un AFD, y simular el AFN directamente.

Vamos a considerar el costo de convertir una expresión regular r en un AFN mediante el Algoritmo 3.23. Un paso clave es construir el árbol de análisis sintáctico para r . En el capítulo 4 veremos varios métodos que son capaces de construir este árbol de análisis sintáctico en tiempo lineal, es decir, en un tiempo $O(|r|)$, en donde $|r|$ representa el *tamaño* de r : la suma del número de operadores y operandos en r . También es fácil verificar que cada una de las construcciones básica e inductiva del Algoritmo 3.23 requiera un tiempo constante, de manera que el tiempo completo invertido por la conversión a un AFN es $O(|r|)$.

Además, como observamos en la sección 3.7.4, el AFN que construimos tiene cuando mucho $|r|$ estados y $2|r|$ transiciones. Es decir, en términos del análisis en la sección 3.7.3, tenemos que $n \leq |r|$ y $m \leq 2|r|$. Así, la simulación de este AFN en una cadena de entrada x requiere un tiempo $O(|r| \times |x|)$. Este tiempo domina el tiempo requerido por la construcción del AFN, que es $O(|r|)$ y, por lo tanto, concluimos que es posible tomar una expresión regular r y la cadena x , e indicar si x está en $L(r)$ en un tiempo $O(|r| \times |x|)$.

El tiempo que requiere la construcción de subconjuntos depende en gran parte del número de estados que tenga el AFD resultante. Para empezar, observe que en la construcción de la figura 3.32, el paso clave, la construcción de un conjunto de estados U a partir de un conjunto de estados T y un símbolo de entrada a , es muy parecida a la construcción de un nuevo conjunto de estados a partir del antiguo conjunto de estados en la simulación de un AFN del Algoritmo 3.22. Ya hemos concluido que, si se implementa en forma apropiada, este paso requiere cuando mucho un tiempo proporcional al número de estados y transiciones del AFN.

Suponga que empezamos con una expresión regular r y la convertimos en un AFN. Este AFN tiene cuando mucho $|r|$ estados y $2|r|$ transiciones. Además, hay cuando mucho $|r|$ símbolos de entrada. Así, para cada estado construido del AFD, debemos construir cuando mucho $|r|$ nuevos estados, y cada uno requiere a lo más un tiempo $O(|r| + 2|r|)$. El tiempo para construir un AFD de s estados es, por consiguiente, $O(|r|^2s)$.

En el caso común en el que s es aproximadamente $|r|$, la construcción de subconjuntos requiere un tiempo $O(|r|^3)$. No obstante, en el peor caso como en el ejemplo 3.25, este tiempo es $O(|r|^22^{|r|})$. La figura 3.48 resume las opciones cuando recibimos una expresión regular r y deseamos producir un reconocedor que indique si una o más cadenas x están en $L(r)$.

AUTÓMATA	INICIAL	POR CADENA
AFN	$O(r)$	$O(r \times x)$
Caso típico del AFD	$O(r ^3)$	$O(x)$
Peor caso del AFD	$O(r ^22^{ r })$	$O(x)$

Figura 3.48: Costo inicial y costo por cadena de varios métodos para reconocer el lenguaje de una expresión regular

Si domina el costo por cadena, como es el caso cuando construimos un analizador léxico, es evidente que preferimos el AFD. No obstante, en los comandos como `grep`, en donde ejecutamos el autómata sólo sobre una cadena, por lo general, preferimos el AFN. No es sino hasta que $|x|$ se acerca a $|r|^3$ que empezamos a considerar la conversión a un AFD.

Sin embargo, hay una estrategia mixta que es casi tan buena como la mejor estrategia del AFN y AFD para cada expresión r y cadena x . Empezamos simulando el AFN, pero recordamos los conjuntos de estados del AFN (es decir, los estados del AFD) y sus transiciones, a medida que los calculamos. Antes de procesar el conjunto actual de estados del AFN y el símbolo de entrada actual, comprobamos si tenemos ya calculada esta transición, y utilizamos la información en caso de que así sea.

3.7.6 Ejercicios para la sección 3.7

Ejercicio 3.7.1: Convierta a AFDs los AFNs de:

- a) La figura 3.26.
- b) La figura 3.29.
- c) La figura 3.30.

Ejercicio 3.7.2: Use el Algoritmo 3.22 para simular los AFNs.

- a) Figura 3.29.
- b) Figura 3.30.

con la entrada $aabb$.

Ejercicio 3.7.3: Convierta las siguientes expresiones regulares en autómatas finitos deterministas, mediante los algoritmos 3.23 y 3.20:

- a) $(\mathbf{a}|\mathbf{b})^*$.
- b) $(\mathbf{a}^*|\mathbf{b}^*)^*$.
- c) $((\epsilon|\mathbf{a})\mathbf{b}^*)^*$.
- d) $(\mathbf{a}|\mathbf{b})^* \mathbf{a} \mathbf{b} \mathbf{b} (\mathbf{a}|\mathbf{b})^*$.

3.8 Diseño de un generador de analizadores léxicos

En esta sección aplicaremos las técnicas presentadas en la sección 3.7 para ver la arquitectura de un generador de analizadores léxicos como `Lex`. Hablaremos sobre dos métodos, basados en AFNs y AFDs; el último es en esencia la implementación de `Lex`.

3.8.1 La estructura del analizador generado

La figura 3.49 presenta las generalidades acerca de la arquitectura de un analizador léxico generado por **Lex**. El programa que sirve como analizador léxico incluye un programa fijo que simula a un autómata; en este punto dejamos abierta la decisión de si el autómata es determinista o no. El resto del analizador léxico consiste en componentes que se crean a partir del programa **Lex**, por el mismo **Lex**.

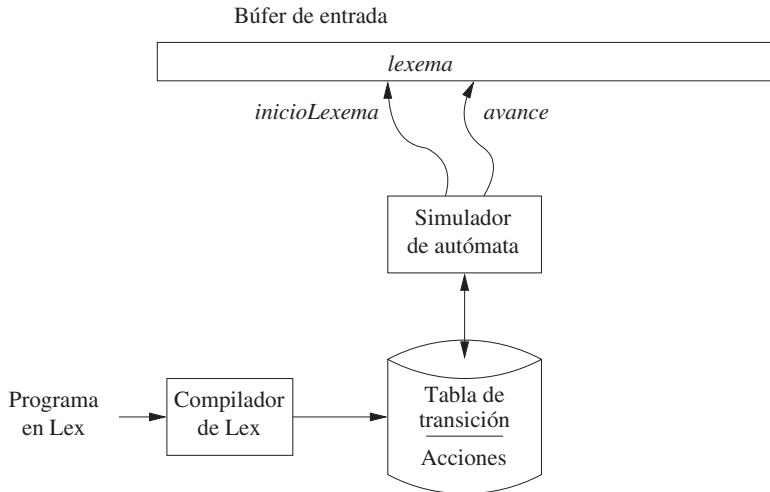


Figura 3.49: Un programa en **Lex** se convierte en una tabla de transición y en acciones, para que las utilice un simulador de autómatas finitos

Estos componentes son:

1. Una tabla de transición para el autómata.
2. Las funciones que se pasan directamente a través de **Lex** a la salida (vea la sección 3.5.2).
3. Las acciones del programa de entrada, que aparece como fragmentos de código que el simulador del autómata debe invocar en el momento apropiado.

Para construir el autómata, empezamos tomando cada patrón de expresión regular en el programa en **Lex** y lo convertimos, mediante el Algoritmo 3.23, en un AFN. Necesitamos un solo autómata que reconozca los lexemas que coinciden con alguno de los patrones en el programa, por lo que combinamos todos los AFNs en uno solo, introduciendo un nuevo estado inicial con transiciones ϵ hacia cada uno de los estados iniciales del N_i de los AFNs para el patrón p_i . Esta construcción se muestra en la figura 3.50.

Ejemplo 3.26: Vamos a ilustrar las ideas de esta sección con el siguiente ejemplo abstracto simple:

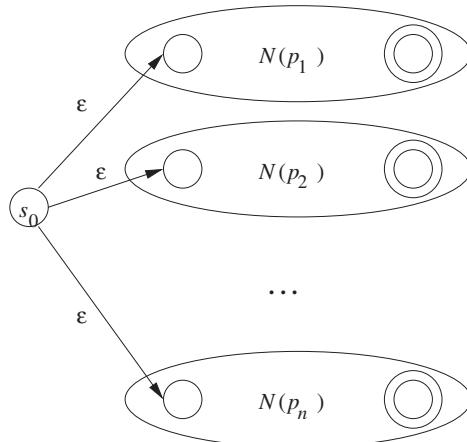


Figura 3.50: Un AFN construido a partir de un programa en Lex

a	{ acción A_1 para el patrón p_1 }
abb	{ acción A_2 para el patrón p_2 }
a*b⁺	{ acción A_3 para el patrón p_3 }

Observe que estos tres patrones presentan ciertos conflictos del tipo que describimos en la sección 3.5.3. En especial, la cadena *abb* coincide con el segundo y tercer patrones, pero vamos a considerarla como un lexema para el patrón p_2 , ya que el patrón se lista primero en el programa anterior en Lex. Entonces, las cadenas de entrada como $aabb\cdots$ tienen muchos prefijos que coinciden con el tercer patrón. La regla de Lex es tomar el más largo, por lo que continuamos leyendo *bs* hasta encontrarnos con otra *a*, en donde reportamos que el lexema consta de las *as* iniciales, seguidas de todas las *bs* que haya.

La figura 3.51 muestra tres AFNs que reconocen los tres patrones. El tercero es una simplificación de lo que saldría del Algoritmo 3.23. Así, la figura 3.52 muestra estos tres AFNs combinados en un solo AFN, mediante la adición del estado inicial 0 y de tres transiciones ϵ . \square

3.8.2 Coincidencia de patrones con base en los AFNs

Si el analizador léxico simula un AFN como el de la figura 3.52, entonces debe leer la entrada que empieza en el punto de su entrada, al cual nos hemos referido como *inicioLexema*. A medida que el apuntador llamado *avance* avanza hacia delante en la entrada, calcula el conjunto de estados en los que se encuentra en cada punto, siguiendo el Algoritmo 3.22.

En algún momento, la simulación del AFN llega a un punto en la entrada en donde no hay siguientes estados. En ese punto, no hay esperanza de que cualquier prefijo más largo de la entrada haga que el AFN llegue a un estado de aceptación; en vez de ello, el conjunto de estados siempre estará vacío. Por ende, estamos listos para decidir sobre el prefijo más largo que sea un lexema que coincide con cierto patrón.

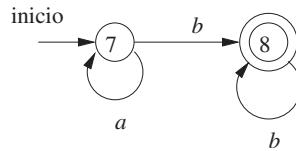
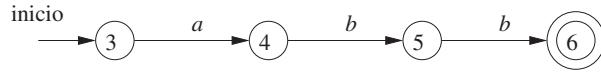
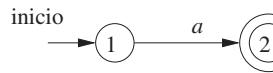
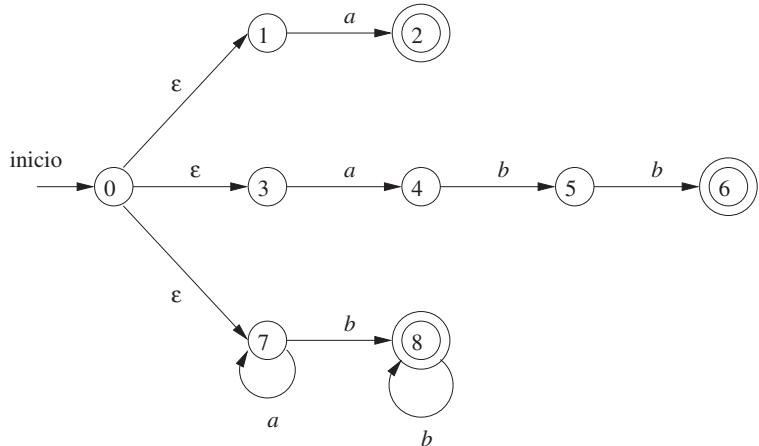
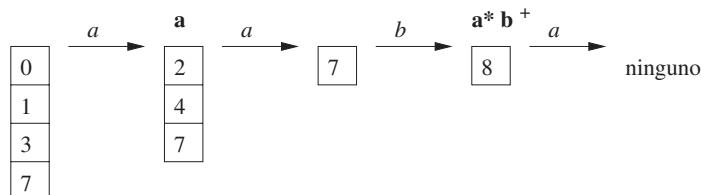
Figura 3.51: AFNs para a , abb y a^*b^+ 

Figura 3.52: AFN combinado

Figura 3.53: Secuencia de los conjuntos de estados que se introducen al procesar la entrada $aaba$

Buscamos hacia atrás en la secuencia de conjuntos de estados, hasta encontrar un conjunto que incluya uno o más estados de aceptación. Si hay varios estados de aceptación en ese conjunto, elegimos el que esté asociado con el primer patrón p_i en la lista del programa en **Lex**. Retrocedemos el apuntador *avance* hacia el final del lexema, y realizamos la acción A_i asociada con el patrón p_i .

Ejemplo 3.27: Suponga que tenemos los patrones del ejemplo 3.36 y que la entrada empieza con *aaba*. La figura 3.53 muestra los conjuntos de estados del AFN de la figura 3.52 que introducimos, empezando con ϵ -*cerradura* del estado inicial 0, el cual es $\{0, 1, 3, 7\}$, y procediendo a partir de ahí. Después de leer el cuarto símbolo de entrada, nos encontramos en un conjunto vacío de estados, ya que en la figura 3.52 no hay transiciones salientes del estado 8 en la entrada *a*.

Por ende, necesitamos retroceder para buscar un conjunto de estados que incluya un estado aceptante. Observe que, como se indica en la figura 3.53, después de leer *a* nos encontramos en un conjunto que incluye el estado 2 y, por lo tanto, indica que el patrón **a** tiene una coincidencia. No obstante, después de leer *aab* nos encontramos en el estado 8, el cual indica que se ha encontrado una coincidencia con **a***b⁺****; el prefijo *aab* es el prefijo más largo que nos lleva a un estado de aceptación. Por lo tanto, seleccionamos *aab* como el lexema y ejecutamos la acción A_3 , la cual debe incluir un regreso al analizador sintáctico, indicando que se ha encontrado el token cuyo patrón es $p_3 = \mathbf{a^*b^+}$. \square

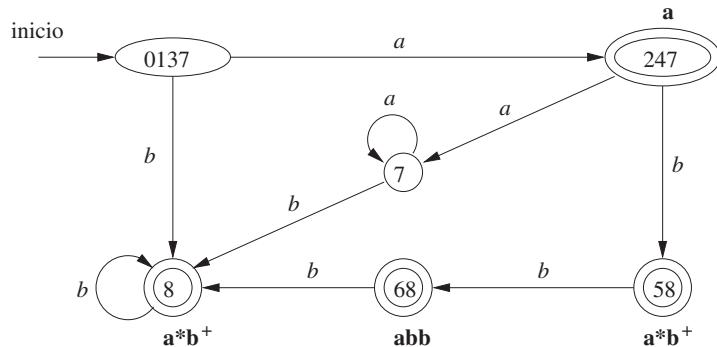
3.8.3 AFDs para analizadores léxicos

Otra arquitectura, que se asemeja a la salida de **Lex**, es convertir el AFN para todos los patrones en un AFD equivalente, mediante la construcción de subconjuntos del Algoritmo 3.20. Dentro de cada estado del AFD, si hay uno o más estados aceptantes del AFN, se determina el primer patrón cuyo estado aceptante se representa, y ese patrón se convierte en la salida del estado AFD.

Ejemplo 3.28: La figura 3.54 muestra un diagrama de transición de estado basado en el AFD que se construye mediante la construcción de subconjuntos del AFN en la figura 3.52. Los estados de aceptación se etiquetan mediante el patrón identificado por ese estado. Por ejemplo, el estado $\{6, 8\}$ tiene dos estados de aceptación, los cuales corresponden a los patrones **abb** y **a***b⁺****. Como el primer patrón se lista primero, ése es el patrón que se asocia con el estado $\{6, 8\}$. \square

Utilizamos el AFD en un analizador léxico en forma muy parecida al AFN. Simulamos el AFD hasta que en cierto punto no haya un estado siguiente (o hablando en sentido estricto, hasta que el siguiente estado sea \emptyset , el *estado muerto* que corresponde al conjunto vacío de estados del AFN). En ese punto, retrocedemos a través de la secuencia de estados que introdujimos y, tan pronto como nos encontramos con un estado de aceptación del AFD, realizamos la acción asociada con el patrón para ese estado.

Ejemplo 3.29: Suponga que el AFD de la figura 3.54 recibe la entrada *abba*. La secuencia de estados introducidos es 0137, 247, 58, 68, y en la *a* final no hay una transición que salga del estado 68. Por ende, consideramos la secuencia a partir del final, y en este caso, 68 en sí es un estado de aceptación que reporta el patrón $p_2 = \mathbf{abb}$. \square

Figura 3.54: Grafo de transición para un AFD que maneja los patrones **a**, **abb** y **a^*b^+**

3.8.4 Implementación del operador de preanálisis

En la sección 3.5.4 vimos que algunas veces es necesario el operador de preanálisis / de Lex en un patrón r_1/r_2 , ya que tal vez el patrón r_1 para un token específico deba describir cierto contexto r_2 a la izquierda, para poder identificar en forma correcta el lexema actual. Al convertir el patrón r_1/r_2 en un AFN, tratamos al / como si fuera ϵ , por lo que en realidad no buscamos un / en la entrada. No obstante, si el AFN reconoce un prefijo xy del búfer de entrada, de forma que coincida con esta expresión regular, el final del lexema no es en donde el AFN entró a su estado de aceptación. En vez de ello, el final ocurre cuando el AFN entra a un estado s tal que:

1. s tenga una transición ϵ en el / imaginario.
2. Hay un camino del estado inicial del AFN hasta el estado s , que deletrea a x .
3. Hay un camino del estado s al estado de aceptación que deletrea a y .
4. x es lo más largo posible para cualquier xy que cumpla con las condiciones 1-3.

Si sólo hay un estado de transición ϵ en el / imaginario en el AFN, entonces el final del lexema ocurre cuando se entra a este estado por última vez, como se ilustra en el siguiente ejemplo. Si el AFN tiene más de un estado de transición ϵ en el / imaginario, entonces el problema general de encontrar el estado s actual se dificulta mucho más.

Ejemplo 3.30: En la figura 3.55 se muestra un AFN para el patrón de la instrucción IF de Fortran con lectura adelantada del ejemplo 3.13. Observe que la transición ϵ del estado 2 al 3 representa al operador de aceptación. El estado 6 indica la presencia de la palabra clave IF. No obstante, para encontrar el lexema IF exploramos en retroceso hasta la última ocurrencia del estado 2, cada vez que se entra al estado 6. \square

Estados muertos en los AFDs

Técnicamente, el autómata en la figura 3.54 no es en sí un AFD. La razón es que un AFD tiene una transición proveniente de cada estado, en cada símbolo en su alfabeto de entrada. Aquí hemos omitido las transiciones que van al estado muerto \emptyset y, por lo tanto, hemos omitido las transiciones que van del estado muerto hacia sí mismo, en todas las entradas. Los ejemplos anteriores de conversión de AFN a AFD no tenían una forma de pasar del estado inicial a \emptyset , pero el AFN de la figura 3.52 sí.

No obstante, al construir un AFD para usarlo en un analizador léxico, es importante que tratemos al estado muerto de manera distinta, ya que debemos saber cuando no hay más posibilidad de reconocer un lexema más largo. Por ende, sugerimos siempre omitir las transiciones hacia el estado muerto y eliminar el estado muerto en sí. De hecho, el problema es mucho más difícil de lo que parece, ya que una construcción de AFN a AFD puede producir varios estados que no puedan llegar a un estado de aceptación, y debemos saber cuándo se ha llegado a cualquiera de estos estados. La sección 3.9.6 habla sobre cómo combinar todos estos estados en un estado muerto, de manera que sea más fácil identificarlos. También es interesante observar que si construimos un AFD a partir de una expresión regular que utilice los Algoritmos 3.20 y 3.23, entonces el AFD no tendrá ningún estado, aparte de \emptyset , que no pueda dirigirnos hacia un estado de aceptación.

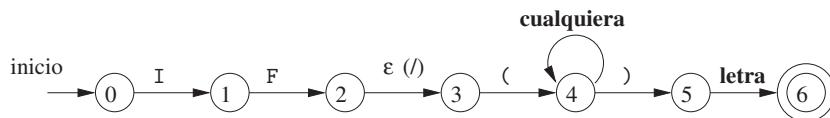


Figura 3.55: AFN que reconoce la palabra clave IF

3.8.5 Ejercicios para la sección 3.8

Ejercicio 3.8.1: Suponga que tenemos dos tokens: (1) la palabra clave `if` y (2) los identificadores, que son cadenas de letras distintas de `if`. Muestre:

- El AFN para estos tokens.
- El AFD para estos tokens.

Ejercicio 3.8.2: Repita el ejercicio 3.8.1 para los tokens que consistan en (1) la palabra clave `while`, (2) la palabra clave `when`, y (3) los identificadores compuestos de cadenas de letras y dígitos, empezando con una letra.

! Ejercicio 3.8.3: Suponga que vamos a revisar la definición de un AFD para permitir cero o una transición saliente de cada estado, en cada símbolo de entrada (en vez de que sea exactamente una transición, como en la definición estándar del AFD). De esta forma, algunas expre-

siones regulares tendrían “AFDs” más pequeños en comparación con la definición estándar de un AFD. Proporcione un ejemplo de una expresión regular así.

!! Ejercicio 3.8.4: Diseñe un algoritmo para reconocer los patrones de lectura por adelantado de Lex de la forma r_1 / r_2 , en donde r_1 y r_2 son expresiones regulares. Muestre cómo funciona su algoritmo en las siguientes entradas:

- $(abcd|abc)/d$
- $(a|ab)/ba$
- aa^*/a^*

3.9 Optimización de los buscadores por concordancia de patrones basados en AFD

En esa sección presentaremos tres algoritmos que se utilizan para implementar y optimizar buscadores por concordancia de patrones, construidos a partir de expresiones regulares.

1. El primer algoritmo es útil en un compilador de Lex, ya que construye un AFD directamente a partir de una expresión regular, sin construir un AFN intermedio. Además, el AFD resultante puede tener menos estados que el AFD que se construye mediante un AFN.
2. El segundo algoritmo disminuye al mínimo el número de estados de cualquier AFD, mediante la combinación de los estados que tienen el mismo comportamiento a futuro. El algoritmo en sí es bastante eficiente, pues se ejecuta en un tiempo $O(n \log n)$, en donde n es el número de estados del AFD.
3. El tercer algoritmo produce representaciones más compactas de las tablas de transición que la tabla estándar bidimensional.

3.9.1 Estados significativos de un AFN

Para empezar nuestra discusión acerca de cómo pasar directamente de una expresión regular a un AFD, primero debemos analizar con cuidado la construcción del AFN del algoritmo 3.23 y considerar los papeles que desempeñan varios estados. A un estado del AFN le llamamos *significativo* si tiene una transición de salida que no sea ϵ . Observe que la construcción de subconjuntos (Algoritmo 3.20) sólo utiliza los estados significativos en un conjunto T cuando calcula ϵ -*cerradura*($mover(T, a)$), el conjunto de estados a los que se puede llegar desde T con una entrada a . Es decir, el conjunto de estados $mover(s, a)$ no está vacío sólo si el estado s es importante. Durante la construcción de subconjuntos, pueden identificarse dos conjuntos de estados del AFN (que se tratan como si fueran el mismo conjunto) si:

1. Tienen los mismos estados significativos.
2. Ya sea que ambos tengan estados de aceptación, o ninguno.

Cuando el AFN se construye a partir de una expresión regular mediante el Algoritmo 3.23, podemos decir más acerca de los estados significativos. Los únicos estados significativos son los que se introducen como estados iniciales en la parte básica para la posición de un símbolo específico en la expresión regular. Es decir, cada estado significativo corresponde a un operando específico en la expresión regular.

El AFN construido sólo tiene un estado de aceptación, pero éste, que no tiene transiciones de salida, no es un estado significativo. Al concatenar un único marcador final $\#$ derecho con una expresión regular r , proporcionamos al estado de aceptación para r una transición sobre $\#$, con lo cual lo marcamos como un estado significativo del AFN para $(r)\#$. En otras palabras, al usar la expresión regular *aumentada* $(r)\#$, podemos olvidarnos de los estados de aceptación a medida que procede la construcción de subconjuntos; cuando se completa la construcción, cualquier estado con una transición sobre $\#$ debe ser un estado de aceptación.

Los estados significativos del AFN corresponden directamente a las posiciones en la expresión regular que contienen símbolos del alfabeto. Como pronto veremos, es conveniente presentar la expresión regular mediante su *árbol sintáctico*, en donde las hojas corresponden a los operandos y los nodos interiores corresponden a los operadores. A un nodo interior se le llama *nodo-concat*, *nodo-o* o *nodo-asterisco* si se etiqueta mediante el operador de concatenación (punto), el operador de unión $|$, o el operador $*$, respectivamente. Podemos construir un árbol sintáctico para una expresión regular al igual que como lo hicimos para las expresiones aritméticas en la sección 2.5.1.

Ejemplo 3.31: La figura 3.56 muestra el árbol sintáctico para la expresión regular de nuestro bosquejo. Los nodos-concat se representan mediante círculos.

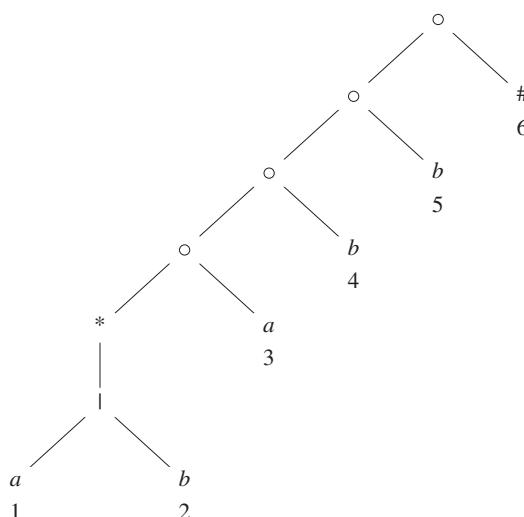


Figura 3.56: Árbol sintáctico para $(a|b)^*abb\#$

Las hojas en un árbol sintáctico se etiquetan mediante ϵ o mediante un símbolo del alfabeto. Para cada hoja que no se etiqueta como ϵ , le adjuntamos un entero único. Nos referimos a este entero como la *posición* de la hoja y también como una posición de su símbolo. Observe que un símbolo puede tener varias posiciones; por ejemplo, a tiene las posiciones 1 y 3 en la figura 3.56. Las posiciones en el árbol sintáctico corresponden a los estados significativos del AFN construido.

Ejemplo 3.32: La figura 3.57 muestra el AFN para la misma expresión regular que la figura 3.56, con los estados significativos enumerados y los demás estados representados por letras. Los estados enumerados en el AFN y las posiciones en el árbol sintáctico corresponden de una forma que pronto veremos. \square

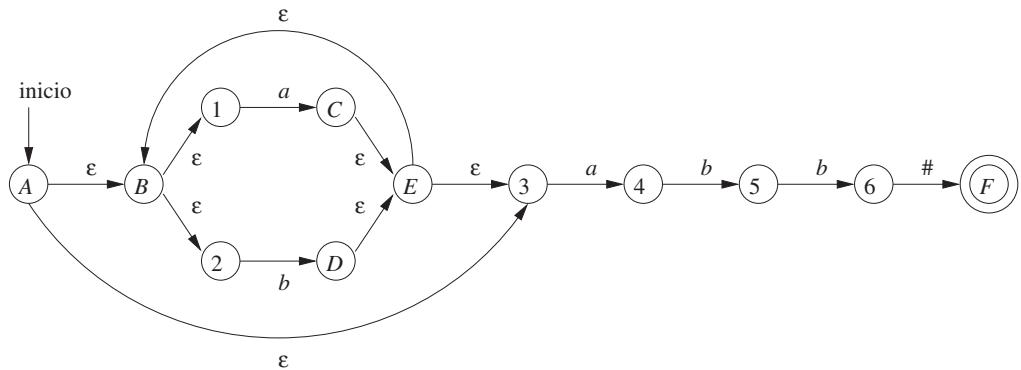


Figura 3.57: AFN construido por el Algoritmo 3.23 para $(a|b)^*abb\#$

3.9.2 Funciones calculadas a partir del árbol sintáctico

Para construir un AFD directamente a partir de una expresión regular, construimos su árbol sintáctico y después calculamos cuatro funciones: *anulable*, *primerapos*, *ultimapos* y *siguiente-apos*, las cuales se definen a continuación. Cada definición se refiere al árbol sintáctico para una expresión regular aumentada $(r)\#$ específica.

1. *anulable*(n) es verdadera para un nodo n del árbol sintáctico si, y sólo si, la subexpresión representada por n tiene a ϵ en su lenguaje. Es decir, la subexpresión puede “hacerse nula” o puede ser la cadena vacía, aun cuando pueda representar también a otras cadenas.
2. *primerapos*(n) es el conjunto de posiciones en el subárbol con raíz en n , que corresponde al primer símbolo de por lo menos una cadena en el lenguaje de la subexpresión con raíz en n .
3. *ultimapos*(n) es el conjunto de posiciones en el subárbol con raíz en n , que corresponde al último símbolo de por lo menos una cadena en el lenguaje de la subexpresión con raíz en n .

4. *siguiente**pos*(p), para una posición p , es el conjunto de posiciones q en todo el árbol sintáctico, de tal forma que haya cierta cadena $x = a_1 a_2 \cdots a_n$ en una $L((r)\#)$ tal que para cierta i , haya una forma de explicar la membresía de x en $L((r)\#)$, haciendo que a_i coincida con la posición p del árbol sintáctico y a_{i+1} con la posición q .

Ejemplo 3.33: Considere el nodo-concat n en la figura 3.56, que corresponde a la expresión $(\mathbf{a}|\mathbf{b})^*\mathbf{a}$. Alegamos que *anulable*(n) es falsa, ya que este nodo genera todas las cadenas de *as* y *bs* que terminan en una *a*; no genera ϵ . Por otro lado, el nodo-asterisco debajo de él puede hacerse nulo; genera ϵ junto con todas las demás cadenas de *as* y *bs*.

$\text{primerapos}(n) = \{1, 2, 3\}$. En una cadena generada en forma común, como *aa*, la primera posición de la cadena corresponde a la posición 1 del árbol, y en una cadena como *ba*, la primera posición de la cadena proviene de la posición 2 del árbol. No obstante, cuando la cadena generada por la expresión del nodo n es sólo *a*, entonces esta *a* proviene de la posición 3.

$\text{ultimapos}(n) = \{3\}$. Es decir, sin importar qué cadena se genere a partir de la expresión del nodo n , la última posición es la *a* que proviene de la posición 3 de la cadena.

*siguiente**pos* es más difícil de calcular, pero en breve veremos las reglas para hacerlo. He aquí un ejemplo del razonamiento: *siguiente**pos* (1) = {1, 2, 3}. Considere una cadena $\cdots ac\cdots$, en donde la *c* puede ser *a* o *b*, y la *a* proviene de la posición 1. Es decir, esta *a* es una de las que se generan mediante la **a** en la expresión $(\mathbf{a}|\mathbf{b})^*$. Esta *a* podría ir seguida de otra *a* o *b* que provenga de la misma subexpresión, en cuyo caso *c* proviene de la posición 1 o 2. También es posible que esta *a* sea la última en la cadena generada por $(\mathbf{a}|\mathbf{b})^*$, en cuyo caso el símbolo *c* debe ser la *a* que proviene de la posición 3. Por ende, 1, 2 y 3 son exactamente las posiciones que pueden seguir de la posición 1. \square

3.9.3 Cálculo de *anulable*, *primerapos* y *ultimapos*

Podemos calcular a *anulable*, *primerapos* y *ultimapos* mediante recursividad directa sobre la altura del árbol. Las reglas básicas e inductivas para *anulable* y *primerapos* se resumen en la figura 3.58. Las reglas para *ultimapos* son en esencia las mismas que para *primerapos*, pero las funciones de los hijos c_1 y c_2 deben intercambiarse en la regla para un nodo-concat.

Ejemplo 3.34: De todos los nodos en la figura 3.56, sólo el nodo-asterisco puede hacerse nulo. De la tabla de la figura 3.58 podemos observar que ninguna de las hojas puede hacerse nula, ya que todas ellas corresponden a operandos que no son ϵ . El nodo-o no puede hacerse nulo, ya que ninguno de sus hijos lo son. El nodo-asterisco puede hacerse nulo, ya que todos los nodos-asterisco pueden hacerse nulos. Por último, cada uno de los nodos-concatt, que tienen por lo menos un hijo que no puede hacerse nulo, no pueden hacerse nulos.

El cálculo de *primerapos* y *ultimapos* para cada uno de los nodos se muestra en la figura 3.59, con $\text{primerapos}(n)$ a la izquierda del nodo n , y $\text{ultimapos}(n)$ a su derecha. Cada una de las hojas sólo se tiene a sí misma para *primerapos* y *ultimapos*, según lo requerido en la regla para las hojas que no son ϵ en la figura 3.58. Para el nodo-o, tomamos la unión de *primerapos* en los hijos y hacemos lo mismo para *ultimapos*. La regla para el nodo-asterisco dice que debemos tomar el valor de *primerapos* o *ultimapos* en el único hijo de ese nodo.

NODO n	$anulable(n)$	$primerapos(n)$
Una hoja etiquetada como ϵ	true	\emptyset
Una hoja con la posición i	false	$\{i\}$
Un nodo-o $n = c_1 c_2$	$anulable(c_1)$ or $anulable(c_2)$	$primerapos(c_1) \cup primerapos(c_2)$
Un nodo-concat $n = c_1c_2$	$anulable(c_1)$ and $anulable(c_2)$	if ($anulable(c_1)$) $primerapos(c_1) \cup primerapos(c_2)$ else $primerapos(c_1)$
Un nodo-asterisco $n = c_1^*$	true	$primerapos(c_1)$

Figura 3.58: Reglas para calcular a *anulable* y *primerapos*

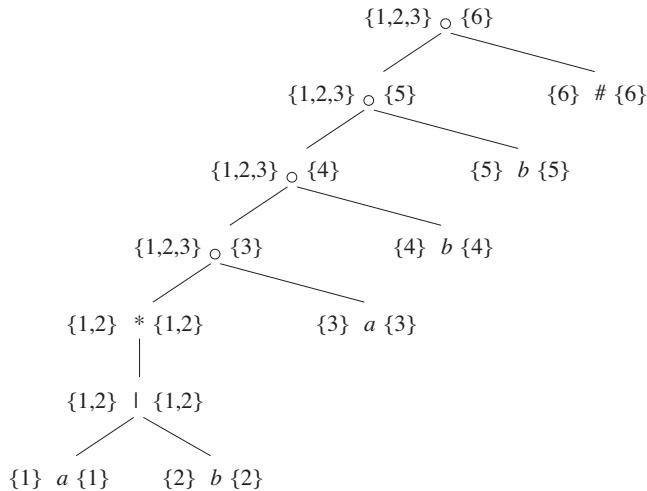
Ahora, considere el nodo-concat más inferior, al que llamaremos n . Para calcular $primerapos(n)$, primero consideramos si el operando izquierdo puede hacerse nulo, lo cual es cierto en este caso. Por lo tanto, $primerapos$ para n es la unión de $primerapos$ para cada uno de sus hijos; es decir, $\{1, 2\} \cup \{3\} = \{1, 2, 3\}$. La regla para $ultimapos$ no aparece en forma explícita en la figura 3.58, pero como dijimos antes, las reglas son las mismas que para $primerapos$, con los hijos intercambiados. Es decir, para calcular $ultimapos(n)$ debemos verificar si su hijo derecho (la hoja con la posición 3) puede hacerse nulo, lo cual no es cierto. Por lo tanto, $ultimapos(n)$ es el mismo que el nodo $ultimapos$ del hijo derecho, o $\{3\}$. \square

3.9.4 Cálculo de *siguiente* pos

Por último, necesitamos ver cómo calcular *siguiente* pos . Sólo hay dos formas en que podemos hacer que la posición de una expresión regular siga a otra.

1. Si n es un nodo-concat con el hijo izquierdo c_1 y con el hijo derecho c_2 , entonces para cada posición i en $ultimapos(c_1)$, todas las posiciones en $primerapos(c_2)$ se encuentran en $siguiente$ $pos(i)$.
2. Si n es un nodo-asterisco e i es una posición en $ultimapos(n)$, entonces todas las posiciones en $primerapos(n)$ se encuentran en $siguiente$ $pos(i)$.

Ejemplo 3.35: Vamos a continuar con nuestro bosquejo; recuerde que en la figura 3.59 calculamos $primerapos$ y $ultimapos$. La regla 1 para *siguiente* pos requiere que analicemos cada nodo-concat, y que coloquemos cada posición en $primerapos$ de su hijo derecho en *siguiente* pos , para cada posición en $ultimapos$ de su hijo izquierdo. Para el nodo-concat más inferior en la figura 3.59, esa regla indica que la posición 3 está en $siguiente$ $pos(1)$ y $siguiente$ $pos(2)$. El siguiente nodo-concat indica que la posición 4 está en $siguiente$ $pos(3)$, y los dos nodos-concat restantes nos dan la posición 5 en $siguiente$ $pos(4)$ y la 6 en $siguiente$ $pos(5)$.

Figura 3.59: *primerapos* y *ultimapos* para los nodos en el árbol sintáctico para $(a|b)^*abb\#$

También debemos aplicar la regla 2 al nodo-asterisco. Esa regla nos indica que las posiciones 1 y 2 están tanto en *siguientepos*(1) como en *siguientepos*(2), ya que tanto *primerapos* como *ultimapos* para este nodo son $\{1,2\}$. Los conjuntos completos *siguientepos* se resumen en la figura 3.60. \square

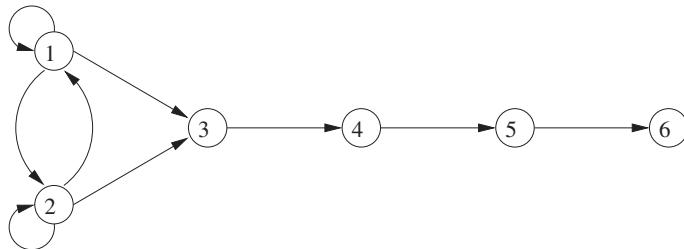
NODO n	$siguientepos(n)$
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	\emptyset

Figura 3.60: La función *siguientepos*

Podemos representar la función *siguientepos* mediante la creación de un gráfico dirigido, con un nodo para cada posición y un arco de la posición i a la posición j si y sólo si j se encuentra en *siguientepos*(i). La figura 3.61 muestra este gráfico para la función de la figura 3.60.

No debe sorprendernos el hecho de que el grafo para *siguientepos* sea casi un AFN sin transiciones ϵ para la expresión regular subyacente, y se convertiría en uno si:

1. Hacemos que todas las posiciones en *primerapos* de la raíz sean estados iniciales,
2. Etiquetamos cada arco de i a j mediante el símbolo en la posición i .

Figura 3.61: Gráfico dirigido para la función *siguientepos*

3. Hacemos que la posición asociada con el marcador final $\#$ sea el único estado de aceptación.

3.9.5 Conversión directa de una expresión regular a un AFD

Algoritmo 3.36: Construcción de un AFD a partir de una expresión regular r .

ENTRADA: Una expresión regular r .

SALIDA: Un AFD D que reconoce a $L(r)$.

MÉTODO:

1. Construir un árbol sintáctico T a partir de la expresión regular aumentada $(r)\#$.
2. Calcular *anulable*, *primerapos*, *ultimapos* y *siguientenpos* para T , mediante los métodos de las secciones 3.9.3 y 3.9.4.
3. Construir *Destados*, el conjunto de estados del AFD D , y *Dtran*, la función de transición para D , mediante el procedimiento de la figura 3.62. Los estados de D son estados de posiciones en T . Al principio, cada estado está “sin marca”, y un estado se “marca” justo antes de que consideremos sus transiciones de salida. El estado inicial de D es *primerapos*(n_0), en donde el nodo n_0 es la raíz de T . Los estados de aceptación son los que contienen la posición para el símbolo de marcador final $\#$.

□

Ejemplo 3.37: Ahora podemos unir los pasos de nuestro bosquejo y construir un AFD para la expresión regular $r = (\mathbf{a}|\mathbf{b})^* \mathbf{a} \mathbf{b} \mathbf{b}$. El árbol sintáctico para $(r)\#$ apareció en la figura 3.56. Ahí observamos que para este árbol, *anulable* es verdadera sólo para el nodo-asterisco, y exhibimos a *primerapos* y *ultimapos* en la figura 3.59. Los valores de *siguientenpos* aparecen en la figura 3.60.

El valor de *primerapos* para la raíz del árbol es $\{1, 2, 3\}$, por lo que este conjunto es el estado inicial de D . Llamemos a este conjunto de estados A . Debemos calcular $Dtran[A, a]$ y $Dtran[A, b]$. De entre las posiciones de A , la 1 y la 3 corresponden a a , mientras que la 2 corresponde a b . Por ende, $Dtran[A, a] = \text{siguientenpos}(1) \cup \text{siguientenpos}(3) = \{1, 2, 3, 4\}$ y

```

inicializar Destados para que contenga sólo el estado sin marcar primerapos( $n_0$ ),
en donde  $n_0$  es la raíz del árbol sintáctico  $T$  para  $(r)\#$ ;
while ( hay un estado sin marcar  $S$  en Destados ) {
    marcar  $S$ ;
    for ( cada símbolo de entrada  $a$  ) {
        dejar que  $U$  sea la unión de siguientepos( $p$ ) para todas las  $p$ 
        en  $S$  que correspondan a  $a$ ;
        if (  $U$  no está en Destados )
            agregar  $U$  como estado sin marcar a Destados;
             $Dtran[S, a] = U$ ;
    }
}

```

Figura 3.62: Construcción de un AFD directamente a partir de una expresión regular

$Dtran[A, b] = \text{siguientepos}(2) = \{1, 2, 3\}$. Este último es el estado A y, por lo tanto, no tiene que agregarse a *Destados*, pero el anterior, $B = \{1, 2, 3, 4\}$ es nuevo, por lo que lo agregamos a *Destados* y procedemos a calcular sus transiciones. El AFD completo se muestra en la figura 3.63. \square

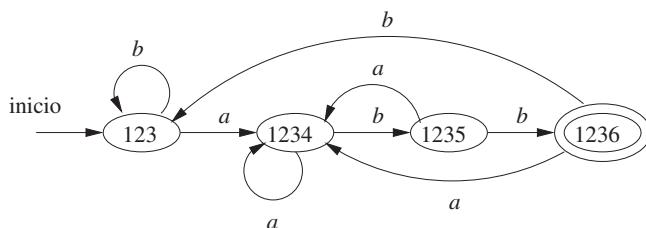


Figura 3.63: AFD construido a partir de la figura 3.57

3.9.6 Minimización del número de estados de un AFD

Puede haber muchos AFDs que reconozcan el mismo lenguaje. Por ejemplo, observe que los AFDs de las figuras 3.36 y 3.63 reconocen el lenguaje $L((\mathbf{a}|\mathbf{b})^*\mathbf{a}\mathbf{b}\mathbf{b})$. Estos autómatas no solo tienen estados con distintos nombres, sino que ni siquiera tienen el mismo número de estados. Si implementamos un analizador léxico como un AFD, por lo general, es preferible un AFD que tenga el menor número de estados posible, ya que cada estado requiere entradas en la tabla para describir al analizador léxico.

El asunto de los nombres de los estados es menor. Decimos que dos autómatas tienen *nombres de estados equivalentes* si uno puede transformarse en el otro con sólo cambiar los nombres de los estados. Las figuras 3.36 y 3.63 no tienen nombres de estados equivalentes. No obstante, hay una estrecha relación entre los estados de cada uno. Los estados A y C de la figura 3.36 son en realidad equivalentes, ya que ninguno es un estado de aceptación, y en cualquier entrada

transfieren hacia el mismo estado: hacia B en la entrada a y hacia C en la entrada b . Además, ambos estados A y C se comportan como el estado 123 de la figura 3.63. De igual forma, el estado B de la figura 3.36 se comporta como el estado 1234 de la figura 3.63, el estado D se comporta como el estado 1235, y el estado E se comporta como el estado 1236.

Como resultado siempre hay un AFD único con el número mínimo de estados (equivalentes) para cualquier lenguaje regular. Además, este AFD con el mínimo número de estados puede construirse a partir de cualquier AFD para el mismo lenguaje, mediante el agrupamiento de conjuntos con estados equivalentes. En el caso de $L((\mathbf{a}|\mathbf{b})^*\mathbf{a}\mathbf{b}\mathbf{b})$, la figura 3.63 es el AFD con el mínimo número de estados, y puede construirse partiendo los estados de la figura 3.36 de la siguiente manera: $\{A,C\}\{B\}\{D\}\{E\}$.

Para poder comprender el algoritmo para crear la partición de estados que convierte a cualquier AFD en su AFD equivalente con el mínimo número de estados, tenemos que ver cómo las cadenas de entrada diferencian un estado de otro. Decimos que la cadena x *diferencia* el estado s del estado t si sólo uno de los estados a los que se llega desde s y t , siguiendo el camino con la etiqueta x , es un estado de aceptación. El estado s puede *diferenciarse* del estado t si hay alguna cadena que los diferencie.

Ejemplo 3.38: La cadena vacía diferencia a cualquier estado de aceptación de cualquier estado de no aceptación. En la figura 3.36, la cadena bb diferencia el estado A del B , ya que bb lleva al estado A hacia un estado C de no aceptación, pero lleva a B al estado de aceptación E . \square

El algoritmo de minimización de estados funciona mediante el particionamiento de los estados de un AFD en grupos de estados que no puedan diferenciarse. Después, cada grupo se combina en un solo estado del AFD con el número mínimo de estados. El algoritmo funciona manteniendo una partición, cuyos grupos son conjuntos de estados que no se han diferenciado todavía, mientras que se sabe que dos estados cualesquiera de distintos grupos pueden diferenciarse. Cuando la partición no puede depurarse más mediante la descomposición de cualquier grupo en grupos más pequeños, tenemos el AFD con el mínimo número de estados.

Al principio, la partición consiste en dos grupos: los estados de aceptación y los de no aceptación. El paso fundamental es tomar cierto grupo de la partición actual, por decir $A = \{s_1, s_2, \dots, s_k\}$, y cierto símbolo de entrada a , y ver cómo puede usarse a para diferenciar algunos estados en el grupo A . Examinamos las transiciones que salen de cada estado s_1, s_2, \dots, s_k en la entrada a , y si los estados a los que se llega pertenecen a dos o más grupos de la partición actual, dividimos a A en una colección de grupos, para que s_i y s_j estén en el mismo grupo, si y sólo si pasan al mismo grupo en la entrada a . Repetimos este proceso de dividir grupos, hasta que ninguno de los grupos pueda dividirse más con ningún símbolo de entrada. En el siguiente algoritmo se formaliza esta idea.

Algoritmo 3.39: Minimización del número de estados de un AFD.

ENTRADA: Un AFD D con un conjunto de estados S , el alfabeto de entrada Σ , el estado inicial s_0 y el conjunto de estados de aceptación F .

SALIDA: Un AFD D' , que acepta el mismo lenguaje que D y tiene el menor número de estados posible.

Por qué funciona el algoritmo de minimización de estados

Debemos demostrar dos cosas: que los estados que permanecen en el mismo grupo en Π_{final} no pueden diferenciarse por ninguna cadena, y que los estados que terminan en grupos distintos sí pueden hacerlo. La primera es una inducción sobre i , la cual nos dice que si después de la i -ésima iteración del paso (2) del Algoritmo 3.39, s y t se encuentran en el mismo grupo, entonces no hay una cadena de longitud i o menor que pueda diferenciarlos. Dejaremos los detalles de esta inducción para que usted los deduzca.

La segunda es una inducción sobre i , la cual nos dice que si los estados s y t se colocan en distintos grupos en la i -ésima iteración del paso (2), entonces hay una cadena que puede diferenciarlos. La base, cuando s y t se colocan en grupos distintos de la partición inicial, es simple: uno debe ser de aceptación y el otro no, para que ϵ los diferencie. Para la inducción, debe haber una entrada a y estados p y q , de tal forma que s y t vayan a los estados p y q , respectivamente, con la entrada a . Además, p y q ya deben haberse colocado en grupos distintos. Entonces, por la hipótesis inductiva, hay alguna cadena x que diferencia a p de q . Por lo tanto, ax diferencia a s de t .

MÉTODO:

1. Empezar con una partición inicial Π con dos grupos, F y $S - F$, los estados de aceptación y de no aceptación de D .
2. Aplicar el procedimiento de la figura 3.64 para construir una nueva partición Π_{nueva} .

al principio, dejar que $\Pi_{\text{nueva}} = \Pi$;

for (cada grupo G de Π) {

 particionar G en subgrupos, de forma que dos estados s y t

 se encuentren en el mismo subgrupo, si y sólo si para todos

 los símbolos de entrada a , los estados s y t tienen transiciones

 sobre a hacia estados en el mismo grupo de Π ;

 /* en el peor caso, un estado estará en un subgrupo por sí solo */

 sustituir G en Π_{nueva} por el conjunto de todos los subgrupos formados;

}

Figura 3.64: Construcción de Π_{nueva}

3. Si $\Pi_{\text{nueva}} = \Pi$, dejar que $\Pi_{\text{final}} = \Pi$ y continuar con el paso (4). De no ser así, repetir el paso (2) con Π_{nueva} en vez de Π .
4. Elegir un estado en cada grupo de Π_{final} como el *representante* para ese grupo. Los representantes serán los estados del AFD D' con el mínimo número de estados. Los demás componentes de D' se construyen de la siguiente manera:

Eliminación del estado muerto

Algunas veces, el algoritmo de minimización produce un AFD con un estado muerto; uno que no es aceptante y que se transfiere hacia sí mismo en cada símbolo de entrada. Técnicamente, este estado es necesario, ya que un AFD debe tener una transición proveniente de cualquier estado con cada símbolo. No obstante y como vimos en la sección 3.8.3, con frecuencia es conveniente saber cuando no hay posibilidad de aceptación, para poder determinar que se ha visto el lexema apropiado. Por ende, tal vez sea conveniente eliminar el estado muerto y utilizar un autómata que omita algunas transiciones. Este autómata tiene un estado menos que el AFD con el mínimo número de estados, pero hablando en sentido estricto no es un AFD, debido a las transiciones que faltan hacia el estado muerto.

- (a) El estado inicial de D' es el representante del grupo que contiene el estado inicial de D .
- (b) Los estados de aceptación de D' son los representantes de los grupos que contienen un estado de aceptación de D . Observe que cada grupo contiene sólo estados de aceptación o sólo estados de no aceptación, ya que empezamos separando esas dos clases de estados, y el procedimiento de la figura 3.64 siempre forma nuevos grupos que son subgrupos de los grupos que se construyeron previamente.
- (c) Dejar que s sea el representante de algún grupo G de Π_{final} , y dejar que la transición de D , desde s con la entrada a , sea hacia el estado t . Dejar que r sea el representante del grupo H de t . Después en D' , hay una transición desde s hacia r con la entrada a . Observe que en D , cada estado en el grupo G debe ir hacia algún estado del grupo H con la entrada a , o de lo contrario, el grupo G se hubiera dividido de acuerdo a la figura 3.64.

□

Ejemplo 3.40: Vamos a reconsiderar el AFD de la figura 3.36. La partición inicial consiste en los dos grupos $\{A, B, C, D\} \{E\}$ que son, respectivamente, los estados de no aceptación y los estados de aceptación. Para construir Π_{nueva} , el procedimiento de la figura 3.64 considera ambos grupos y recibe como entrada a y b . El grupo $\{E\}$ no puede dividirse, ya que sólo tiene un estado, por lo cual $\{E\}$ permanecerá intacto en Π_{nueva} .

El otro grupo $\{A, B, C, D\}$ puede dividirse, por lo que debemos considerar el efecto de cada símbolo de entrada. Con la entrada a , cada uno de estos estados pasa al estado B , por lo que no hay forma de diferenciarlos mediante cadenas que empiecen con a . Con la entrada b , los estados A, B y C pasan a los miembros del grupo $\{A, B, C, D\}$, mientras que el estado D pasa a E , un miembro de otro grupo. Por ende, en Π_{nueva} , el grupo $\{A, B, C, D\}$ se divide en $\{A, B, C\} \{D\}$, y Π_{nueva} para esta ronda es $\{A, B, C\} \{D\} \{E\}$.

En la siguiente ronda, podemos dividir a $\{A, B, C\}$ en $\{A, C\}\{B\}$, ya que cada uno de los estados A y C pasan a un miembro de $\{A, B, C\}$ con la entrada b , mientras que B pasa a un miembro de otro grupo, $\{D\}$. Así, después de la segunda iteración, $\Pi_{\text{nueva}} = \{A, C\}\{B\}\{D\}\{E\}$. Para la tercera iteración, no podemos dividir el único grupo restante con más de un estado, ya que A y C van al mismo estado (y, por lo tanto, al mismo grupo) con cada entrada. Concluimos que $\Pi_{\text{final}} = \{A, C\}\{B\}\{D\}\{E\}$.

Ahora vamos a construir el AFD con el mínimo número de estados. Tiene cuatro estados, los cuales corresponden a los cuatro grupos de Π_{final} , y vamos a elegir a A, B, D y E como los representantes de estos grupos. El estado inicial es A y el único estado aceptante es E . La figura 3.65 muestra la función de transición para el AFD. Por ejemplo, la transición desde el estado E con la entrada b es hacia A , ya que en el AFD original, E pasa a C con la entrada b , y A es el representante del grupo de C . Por la misma razón, la transición con b desde el estado A pasa al mismo estado A , mientras que todas las demás transiciones son como en la figura 3.36. \square

ESTADO	a	b
A	B	A
B	B	D
D	B	E
E	B	A

Figura 3.65: Tabla de transición de un AFD con el mínimo número de estados

3.9.7 Minimización de estados en los analizadores léxicos

Para aplicar el procedimiento de minimización de estados a los AFDs generados en la sección 3.8.3, debemos empezar el Algoritmo 3.39 con la partición que agrupa en conjunto a todos los estados que reconocen a un token específico, y que también coloca en un grupo a todos los estados que no indican ningún token. Para hacer más claro esto, vamos a ver un ejemplo.

Ejemplo 3.41: Para el AFD de la figura 3.54, la partición inicial es:

$$\{0137, 7\}\{247\}\{8, 58\}\{7\}\{68\}\{\emptyset\}$$

Es decir, los estados 0137 y 7 deben estar juntos, ya que ninguno de ellos anuncia un token. Los estados 8 y 58 deben estar juntos, ya que ambos anuncian el token $\mathbf{a}^*\mathbf{b}^+$. Observe que hemos agregado un estado muerto \emptyset , el cual suponemos tiene transiciones hacia sí mismo con las entradas a y b . El estado muerto también es el destino de las transiciones faltantes con a desde los estados 8, 58 y 68.

Debemos separar a 0137 de 7, ya que pasan a distintos grupos con la entrada a . También separamos a 8 de 58, ya que pasan a distintos grupos con b . Por ende, todos los estados se encuentran en grupos por sí solos, y la figura 3.54 es el AFD con el mínimo número de estados que reconoce a sus tres tokens. Recuerde que un AFD que sirve como analizador léxico, por lo general, elimina el estado muerto, mientras que tratamos a las transiciones faltantes como una señal para finalizar el reconocimiento de tokens. \square

3.9.8 Intercambio de tiempo por espacio en la simulación de un AFD

La manera más simple y rápida de representar la función de transición de un AFD es una tabla bidimensional indexada por estados y caracteres. Dado un estado y el siguiente carácter de entrada, accedemos al arreglo para encontrar el siguiente estado y cualquier acción especial que debemos tomar; por ejemplo, devolver un token al analizador sintáctico. Como un analizador léxico ordinario tiene varios cientos de estados en su AFD e involucra al alfabeto ASCII de 128 caracteres de entrada, el arreglo consume menos de un megabyte.

No obstante, los compiladores también aparecen en dispositivos muy pequeños, en donde hasta un megabyte de memoria podría ser demasiado. Para tales situaciones, existen muchos métodos que podemos usar para compactar la tabla de transición. Por ejemplo, podemos representar cada estado mediante una lista de transiciones (es decir, pares carácter-estado) que se terminen mediante un estado predeterminado, el cual debe elegirse para cualquier carácter de entrada que no se encuentre en la lista. Si elegimos como predeterminado el siguiente estado que ocurra con más frecuencia, a menudo podemos reducir la cantidad de almacenamiento necesario por un factor extenso.

Hay una estructura de datos más sutil que nos permite combinar la velocidad del acceso a los arreglos con la compresión de listas con valores predeterminados. Podemos considerar esta estructura como cuatro arreglos, según lo sugerido en la figura 3.66.⁵ El arreglo *base* se utiliza para determinar la ubicación base de las entradas para el estado *s*, que se encuentran en los arreglos *siguiente* y *comprobacion*. El arreglo *predeterminado* se utiliza para determinar una ubicación base alternativa, si el arreglo *comprobacion* nos indica que el que proporciona *base*[*s*] es inválido.

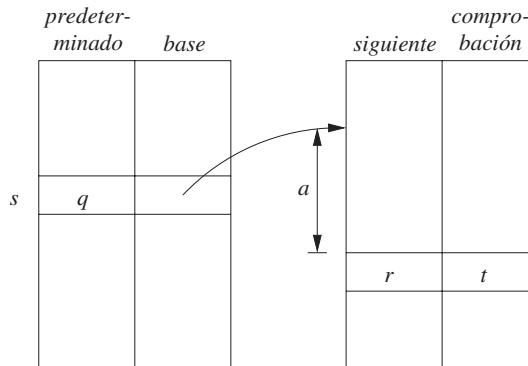


Figura 3.66: Estructura de datos para representar tablas de transición

Para calcular *siguienteEstado*(*s*, *a*), la transición para el estado *s* con la entrada *a*, examinamos las entradas *siguiente* y *comprobacion* en la ubicación *l* = *base*[*s*] + *a*, en donde el carácter *a* se trata como entero, supuestamente en el rango de 0 a 127. Si *comprobacion*[*l*] = *s*, entonces

⁵En la práctica, habría otro arreglo indexado por estados, para proporcionar la acción asociada con ese estado, si lo hay.

esta entrada es válida y el siguiente estado para el estado s con la entrada a es $siguiente[l]$. Si $comprobacion[l] \neq s$, entonces determinamos otro estado $t = predeterminado[s]$ y repetimos el proceso, como si t fuera el estado actual. De manera más formal, la función $siguienteEstado$ se define así:

```
int siguienteEstado(s, a) {
    if ( comprobacion[base[s]+a] = s ) return siguiente[base[s] + a];
    else return siguienteEstado(predeterminado[s], a);
}
```

El uso que se pretende de la estructura de la figura 3.66 es acortar los arreglos *siguiente-comprobacion*, aprovechando las similitudes entre los estados. Por ejemplo, el estado t , el predeterminado para el estado s , podría ser el estado que dice “estamos trabajando con un identificador”, al igual que el estado 10 en la figura 3.14. Tal vez se entre al estado s después de ver las letras **th**, que son un prefijo de la palabra clave **then**, así como también podrían ser el prefijo de algún lexema para un identificador. Con el carácter de entrada **e**, debemos pasar del estado s a un estado especial que recuerde que hemos visto **the**, pero en caso contrario, el estado s se comporta de igual forma que t . Por ende, a $comprobacion[base[s]+e]$ le asignamos s (para confirmar que esta entrada es válida para s) y a $siguiente[base[s]+e]$ le asignamos el estado que recuerda a **the**. Además, a $predeterminado[s]$ se le asigna t .

Aunque tal vez no podamos elegir los valores de *base* de forma que no haya entradas en *siguiente-comprobacion* sin utilizar, la experiencia ha demostrado que la estrategia simple de asignar valores *base* a los estados en turno, y asignar a cada valor de *base[s]* el entero más bajo, de manera que las entradas especiales para el estado s no estén ya ocupadas, utiliza un poco más de espacio que el mínimo posible.

3.9.9 Ejercicios para la sección 3.9

Ejercicio 3.9.1: Extienda la tabla de la figura 3.58 para que incluya los operadores (a) ? y (b) +.

Ejercicio 3.9.2: Use el Algoritmo 3.36 para convertir las expresiones regulares del ejercicio 3.7.3 directamente en autómatas finitos deterministas.

! **Ejercicio 3.9.3:** Podemos demostrar que dos expresiones regulares son equivalentes si mostramos que sus AFDs con el mínimo número de estados son los mismos si se cambia el nombre a los estados. Demuestre de esta forma que las siguientes expresiones regulares: $(a|b)^*$, $(a^*|b^*)^*$ y $((\epsilon|a)b^*)^*$ son todas equivalentes. *Nota:* Tal vez haya construido los AFDs para estas expresiones, al responder al ejercicio 3.7.3.

! **Ejercicio 3.9.4:** Construya los AFDs con el mínimo número de estados para las siguientes expresiones regulares:

- $(a|b)^*a(a|b)$.
- $(a|b)^*a(a|b)(a|b)$.
- $(a|b)^*a(a|b)(a|b)(a|b)$.

¿Puede ver un patrón?

!! **Ejercicio 3.9.5:** Para hacer uso formal de la afirmación informal del ejemplo 3.25, muestre que cualquier autómata finito determinista para la siguiente expresión regular:

$$(a|b)^*a(a|b)(a|b)\cdots(a|b)$$

en donde $(a|b)$ aparece $n - 1$ veces al final, debe tener por lo menos 2^n estados. *Sugerencia:* Observe el patrón en el ejercicio 3.9.4. ¿Qué condición con relación al historial de entradas representa cada estado?

3.10 Resumen del capítulo 3

- ◆ *Tokens.* El analizador léxico explora el programa fuente y produce como salida una secuencia de tokens, los cuales, por lo general, se pasan al analizador sintáctico, uno a la vez. Algunos tokens pueden consistir sólo de un nombre de token, mientras que otros también pueden tener un valor léxico asociado, el cual proporciona información acerca de la instancia específica del token que se ha encontrado en la entrada.
- ◆ *Lexemas.* Cada vez que el analizador léxico devuelve un token al analizador sintáctico, tiene un lexema asociado: la secuencia de caracteres de entrada que representa el token.
- ◆ *Uso de búferes.* Como a menudo es necesario explorar por adelantado sobre la entrada, para poder ver en dónde termina el siguiente lexema, es necesario que el analizador léxico utilice búferes en la entrada. El uso de un par de búferes en forma cíclica y la terminación del contenido de cada búfer con un centinela que avise al llegar a su final, son dos técnicas que aceleran el proceso de escaneo de la entrada.
- ◆ *Patrones.* Cada token tiene un patrón que describe cuáles son las secuencias de caracteres que pueden formar los lexemas correspondientes a ese token. Al conjunto de palabras, o cadenas de caracteres, que coinciden con un patrón dado se le conoce como lenguaje.
- ◆ *Expresiones regulares.* Estas expresiones se utilizan con frecuencia para describir los patrones. Las expresiones regulares se crean a partir de caracteres individuales, mediante el operador de unión, de concatenación y el cierre de Kleene, o el operador “cualquier número de”.
- ◆ *Definiciones regulares.* Las colecciones complejas de lenguajes, como los patrones que describen a los tokens de un lenguaje de programación, se definen a menudo mediante una definición regular, la cual es una secuencia de instrucciones, en las que cada una de ellas define a una variable que representa a cierta expresión regular. La expresión regular para una variable puede utilizar las variables definidas con anterioridad en su expresión regular.
- ◆ *Notación de expresión regular extendida.* Puede aparecer una variedad de operadores adicionales como abreviaciones en las expresiones regulares, para facilitar la acción de expresar los patrones. Algunos ejemplos incluyen el operador + (uno o más de), ? (cero

o uno de), y las clases de caracteres (la unión de las cadenas, en donde cada una consiste en uno de los caracteres).

- ◆ *Diagramas de transición de estados.* A menudo, el comportamiento de un analizador léxico puede describirse mediante un diagrama de transición de estados. Estos diagramas tienen estados, cada uno de los cuales representa algo acerca del historial de los caracteres vistos durante el escaneo actual, en busca de un lexema que coincida con uno de los posibles patrones. Hay flechas, o transiciones, de un estado a otro, cada una de las cuales indica los posibles siguientes caracteres de entrada, que pueden hacer que el analizador léxico realice ese cambio de estado.
- ◆ *Autómatas finitos.* Son una formalización de los diagramas de transición de estados, los cuales incluyen una designación de un estado inicial y uno o más estados de aceptación, así como el conjunto de estados, caracteres de entrada y transiciones entre los estados. Los estados aceptantes indican que se ha encontrado el lexema para cierto token. A diferencia de los diagramas de transición de estados, los autómatas finitos pueden realizar transiciones sobre una entrada vacía, así como sobre los caracteres de entrada.
- ◆ *Autómatas finitos deterministas.* Un AFD es un tipo especial de autómata finito, el cual tiene exactamente una transición saliente de cada estado, para cada símbolo de entrada. Tampoco se permiten las transiciones sobre una entrada vacía. El AFD se puede simular con facilidad, además de que realiza una buena implementación de un analizador léxico, similar a un diagrama de transición.
- ◆ *Autómatas finitos no deterministas.* A los autómatas que no son AFDs se les conoce como no deterministas. A menudo, los AFNs son más fáciles de diseñar que los AFDs. Otra posible arquitectura para un analizador léxico es tabular todos los estados en los que pueden encontrarse los AFNs para cada uno de los posibles patrones, a medida que exploramos los caracteres de entrada.
- ◆ *Conversión entre representaciones de patrones.* Es posible convertir cualquier expresión regular en un AFN de un tamaño aproximado, que reconozca el mismo lenguaje que define la expresión regular. Además, cualquier AFN puede convertirse en un AFD para el mismo patrón, aunque en el peor de los casos (que nunca se presenta en los lenguajes de programación comunes), el tamaño del autómata puede crecer en forma exponencial. También es posible convertir cualquier autómata finito no determinista o determinista en una expresión regular que defina el mismo lenguaje reconocido por el autómata finito.
- ◆ *Lex.* Hay una familia de sistemas de software, incluyendo a **Lex** y **Flex**, que son generadores de analizadores léxicos. El usuario especifica los patrones para los tokens, usando una notación de expresiones regulares extendidas. **Lex** convierte estas expresiones en un analizador léxico, el cual en esencia es un autómata finito determinista que reconoce cualquiera de los patrones.
- ◆ *Minimización de autómatas finitos.* Para cada AFD, hay un AFD con el mínimo número de estados que acepta el mismo lenguaje. Además, el AFD con el mínimo número de estados para un lenguaje dado es único, excepto por los nombres que se da a los diversos estados.

3.11 Referencias para el capítulo 3

En la década de 1950, Kleene desarrolló por primera vez las expresiones regulares [9]. Kleene estaba interesado en describir los eventos que podrían representarse por el modelo de autómatas finitos de actividad neuronal de McCullough y Pitts [12]. Desde entonces, las expresiones regulares y los autómatas finitos se han utilizado en gran medida en la ciencia computacional.

Desde el comienzo, se han usado las expresiones regulares en diversas formas en muchas herramientas populares de Unix, como `awk`, `ed`, `egrep`, `grep`, `lex`, `sed`, `sh` y `vi`. Los documentos de los estándares IEEE 1003 e ISO/IEC 9945 para la Interfaz de sistemas operativos portables (POSIX) definen las expresiones regulares extendidas de POSIX, que son similares a las expresiones regulares originales de Unix, con algunas excepciones como las representaciones de nemónicos para las clases de caracteres. Muchos lenguajes de secuencias de comandos como Perl, Python y Tcl han adoptado las expresiones regulares, pero a menudo con extensiones incompatibles.

El conocido modelo de autómata finito y la minimización de autómatas finitos, como se vieron en el Algoritmo 3.39, provienen de Huffman [6] y Moore [14]. Rabin y Scott [15] propusieron por primera vez los autómatas finitos no deterministas; la construcción de subconjuntos del Algoritmo 3.20, que muestran la equivalencia de los autómatas finitos deterministas y no deterministas, provienen de ahí.

McNaughton y Yamada [13] fueron los primeros en proporcionar un algoritmo para convertir las expresiones regulares de manera directa en autómatas finitos deterministas. Aho utilizó por primera vez el algoritmo 3.36 descrito en la sección 3.9 para crear la herramienta para relacionar expresiones regulares, conocida como `egrep`. Este algoritmo también se utilizó en las rutinas para relacionar patrones de expresiones regulares en `awk` [3]. El método de utilizar autómatas no deterministas como intermediarios se debe a Thompson [17]. Este último artículo también contiene el algoritmo para la simulación directa de autómatas finitos no deterministas (Algoritmo 3.22), que Thompson utilizó en el editor de texto QED.

Lesk desarrolló la primera versión de `Lex`; después Lesk y Schmidt crearon una segunda versión, utilizando el Algoritmo 3.36 [10]. Después de ello, se han implementado muchas variantes de `Lex`. La versión de GNU llamada `Flex` puede descargarse, junto con su documentación, en [4]. Las versiones populares de `Lex` en Java incluyen a `JFlex` [7] y `JLex` [8].

El algoritmo de KMP, que describimos en los ejercicios para la sección 3.4, justo antes del ejercicio 3.4.3, es de [11]. Su generalización para muchas palabras clave aparece en [2]; Aho lo utilizó en la primera implementación de la herramienta `fgrep` de Unix.

La teoría de los autómatas finitos y las expresiones regulares se cubre en [5]. En [1] hay una encuesta de técnicas para relacionar cadenas.

1. Aho, A. V., “Algorithms for finding patterns in strings”, en *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), Vol. A, Cap. 5, MIT Press, Cambridge, 1990.
2. Aho, A. V. y M. J. Corasick, “Efficient string matching: an aid to bibliographic search”, *Comm. ACM* 18:6 (1975), pp. 333-340.
3. Aho, A. V., B. W. Kernighan y P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, Boston, MA, 1988.

4. Página inicial de Flex, <http://www.gnu.org/software/flex/>, Fundación de software libre.
5. Hopcroft, J. E., R. Motwani y J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Boston MA, 2006.
6. Huffman, D. A., “The synthesis of sequential machines”, *J. Franklin Inst.* **257** (1954), pp. 3-4, 161, 190, 275-303.
7. Página inicial de JFlex, <http://jflex.de/> .
8. <http://www.cs.princeton.edu/~appel/modern/java/JLex>.
9. Kleene, S. C., “Representation of events in nerve nets”, en [16], pp. 3-40.
10. Lesk, M. E., “Lex – a lexical analyzer generator”, Computing Science Tech. Reporte 39, Bell Laboratories, Murray Hill, NJ, 1975. Un documento similar con el mismo nombre del título, pero con E. Schmidt como coautor, aparece en el volumen 2 de *Unix Programmer’s Manual*, Bell Laboratories, Murray Hill NJ, 1975; vea <http://dinosaur.compilers.net/lex/index.html>.
11. Knuth, D. E., J. H. Morris y V. R. Pratt, “Fast pattern matching in strings”, *SIAM J. Computing* **6**:2 (1977), pp. 323-350.
12. McCullough, W. S. y W. Pitts, “A logical calculus of the ideas immanent in nervous activity”, *Bull. Math. Biophysics* **5** (1943), pp. 115-133.
13. McNaughton, R. y H. Yamada, “Regular expressions and state graphs for automata”, *IRE Trans. on Electronic Computers* **EC-9**:1 (1960), pp. 38-47.
14. Moore, E. F., “Gedanken experiments on sequential machines”, en [16], pp. 129-153.
15. Rabin, M. O. y D. Scott, “Finite automata and their decision problems”, *IBM J. Res. and Devel.* **3**:2 (1959), pp. 114-125.
16. Shannon, C. y J. McCarthy (eds.), *Automata Studies*, Princeton Univ. Press, 1956.
17. Thompson, K., “Regular expression search algorithm”, *Comm. ACM* **11**:6 (1968), pp. 419-422.

Capítulo 4

Análisis sintáctico

Este capítulo está dedicado a los métodos de análisis sintáctico que se utilizan, por lo general, en los compiladores. Primero presentaremos los conceptos básicos, después las técnicas adecuadas para la implementación manual y, por último, los algoritmos que se han utilizado en las herramientas automatizadas. Debido a que los programas pueden contener errores sintácticos, hablaremos sobre las extensiones de los métodos de análisis sintáctico para recuperarse de los errores comunes.

Por diseño, cada lenguaje de programación tiene reglas precisas, las cuales prescriben la estructura sintáctica de los programas bien formados. Por ejemplo, en C un programa está compuesto de funciones, una función de declaraciones e instrucciones, una instrucción de expresiones, y así sucesivamente. La sintaxis de las construcciones de un lenguaje de programación puede especificarse mediante gramáticas libres de contexto o la notación BNF (Forma de Backus-Naur), que se presentó en la sección 2.2. Las gramáticas ofrecen beneficios considerables, tanto para los diseñadores de lenguajes como para los escritores de compiladores.

- Una gramática proporciona una especificación sintáctica precisa, pero fácil de entender, de un lenguaje de programación.
- A partir de ciertas clases de gramáticas, podemos construir de manera automática un analizador sintáctico eficiente que determine la estructura sintáctica de un programa fuente. Como beneficio colateral, el proceso de construcción del analizador sintáctico puede revelar ambigüedades sintácticas y puntos problemáticos que podrían haberse pasado por alto durante la fase inicial del diseño del lenguaje.
- La estructura impartida a un lenguaje mediante una gramática diseñada en forma apropiada es útil para traducir los programas fuente en código objeto correcto, y para detectar errores.
- Una gramática permite que un lenguaje evolucione o se desarrolle en forma iterativa, agregando nuevas construcciones para realizar nuevas tareas. Estas nuevas construcciones pueden integrarse con más facilidad en una implementación que siga la estructura gramatical del lenguaje.

4.1 Introducción

En esta sección, examinaremos la forma en que el analizador sintáctico se acomoda en un compilador ordinario. Después analizaremos las gramáticas comunes para las expresiones aritméticas. Las gramáticas para las expresiones son suficientes para ilustrar la esencia del análisis sintáctico, ya que dichas técnicas de análisis para las expresiones se transfieren a la mayoría de las construcciones de programación. Esta sección termina con una explicación sobre el manejo de errores, ya que el analizador sintáctico debe responder de manera adecuada para descubrir que su entrada no puede ser generada por su gramática.

4.1.1 La función del analizador sintáctico

En nuestro modelo de compilador, el analizador sintáctico obtiene una cadena de tokens del analizador léxico, como se muestra en la figura 4.1, y verifica que la cadena de nombres de los tokens pueda generarse mediante la gramática para el lenguaje fuente. Esperamos que el analizador sintáctico reporte cualquier error sintáctico en forma inteligible y que se recupere de los errores que ocurren con frecuencia para seguir procesando el resto del programa. De manera conceptual, para los programas bien formados, el analizador sintáctico construye un árbol de análisis sintáctico y lo pasa al resto del compilador para que lo siga procesando. De hecho, el árbol de análisis sintáctico no necesita construirse en forma explícita, ya que las acciones de comprobación y traducción pueden intercalarse con el análisis sintáctico, como veremos más adelante. Por ende, el analizador sintáctico y el resto de la interfaz de usuario podrían implementarse sin problemas mediante un solo módulo.

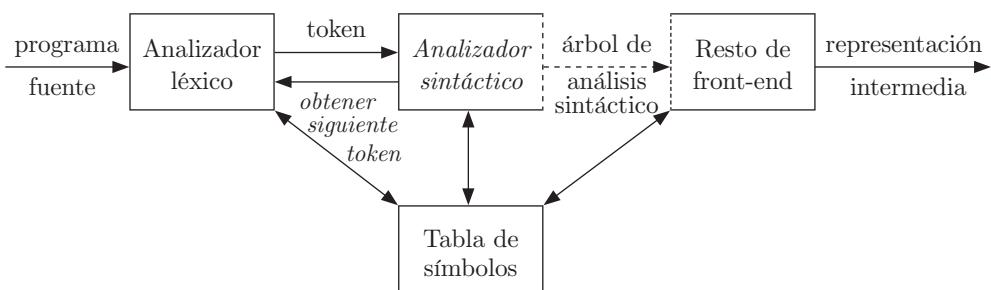


Figura 4.1: Posición del analizador sintáctico en el modelo del compilador

Existen tres tipos generales de analizadores para las gramáticas: universales, descendentes y ascendentes. Los métodos universales de análisis sintáctico como el algoritmo de Cocke-Younger-Kasami y el algoritmo de Earley pueden analizar cualquier gramática (vea las notas bibliográficas). Sin embargo, estos métodos generales son demasiado ineficientes como para usarse en la producción de compiladores.

Los métodos que se utilizan, por lo regular, en los compiladores pueden clasificarse como descendentes o ascendentes. Según sus nombres, los métodos descendentes construyen árboles de análisis sintáctico de la parte superior (raíz) a la parte inferior (hojas), mientras que los métodos ascendentes empiezan de las hojas y avanzan hasta la raíz. En cualquier caso, la entrada al analizador se explora de izquierda a derecha, un símbolo a la vez.

Los métodos descendentes y ascendentes más eficientes sólo funcionan para subclases de gramáticas, pero varias de estas clases, en especial las gramáticas LL y LR, son lo bastante expresivas como para describir la mayoría de las construcciones sintácticas en los lenguajes de programación modernos. Los analizadores sintácticos que se implementan en forma manual utilizan con frecuencia gramáticas LL; por ejemplo, el método de análisis sintáctico predictivo de la sección 2.4.2 funciona para las gramáticas LL. Los analizadores para la clase más extensa de gramáticas LR, por lo general, se construyen mediante herramientas automatizadas.

En este capítulo vamos a suponer que la salida del analizador sintáctico es cierta representación del árbol de análisis sintáctico para el flujo de tokens que proviene del analizador léxico. En la práctica, hay una variedad de tareas que podrían realizarse durante el análisis sintáctico, como la recolección de información de varios tokens para colocarla en la tabla de símbolos, la realización de la comprobación de tipos y otros tipos de análisis semántico, así como la generación de código intermedio. Hemos agrupado todas estas actividades en el cuadro con el título “Resto del front-end” de la figura 4.1. En los siguientes capítulos cubriremos con detalle estas actividades.

4.1.2 Representación de gramáticas

Algunas de las gramáticas que examinaremos en este capítulo se presentan para facilitar la representación de gramáticas. Las construcciones que empiezan con palabras clave como **while** o **int** son muy fáciles de analizar, ya que la palabra clave guía la elección de la producción gramatical que debe aplicarse para hacer que coincida con la entrada. Por lo tanto, nos concentraremos en las expresiones, que representan un reto debido a la asociatividad y la precedencia de operadores.

La asociatividad y la precedencia se resuelvan en la siguiente gramática, que es similar a las que utilizamos en el capítulo 2 para describir expresiones, términos y factores. E representa a las expresiones que consisten en términos separados por los signos $+$, T representa a los términos que consisten en factores separados por los signos $*$, y F representa a los factores que pueden ser expresiones entre paréntesis o identificadores:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned} \tag{4.1}$$

La gramática para expresiones (4.1) pertenece a la clase de gramáticas LR que son adecuadas para el análisis sintáctico ascendentes. Esta gramática puede adaptarse para manejar operadores adicionales y niveles adicionales de precedencia. Sin embargo, no puede usarse para el análisis sintáctico descendente, ya que es recursiva por la izquierda.

La siguiente variante no recursiva por la izquierda de la gramática de expresiones (4.1) se utilizará para el análisis sintáctico descendente:

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \epsilon \\ T &\rightarrow F T' \\ T' &\rightarrow * F T' \mid \epsilon \\ F &\rightarrow (E) \mid \text{id} \end{aligned} \tag{4.2}$$

La siguiente gramática trata a los signos + y * de igual forma, de manera que sirve para ilustrar las técnicas para el manejo de ambigüedades durante el análisis sintáctico:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id} \quad (4.3)$$

Aquí, E representa a las expresiones de todo tipo. La gramática (4.3) permite más de un árbol de análisis sintáctico para las expresiones como $a + b * c$.

4.1.3 Manejo de los errores sintácticos

El resto de esta sección considera la naturaleza de los errores sintácticos y las estrategias generales para recuperarse de ellos. Dos de estas estrategias, conocidas como recuperaciones en modo de pánico y a nivel de frase, se describirán con más detalle junto con los métodos específicos de análisis sintáctico.

Si un compilador tuviera que procesar sólo programas correctos, su diseño e implementación se simplificaría en forma considerable. No obstante, se espera que un compilador ayude al programador a localizar y rastrear los errores que, de manera inevitable, se infiltran en los programas, a pesar de los mejores esfuerzos del programador. Aunque parezca increíble, son pocos los lenguajes que se diseñan teniendo en mente el manejo de errores, aun cuando éstos son tan comunes. Nuestra civilización sería radicalmente distinta si los lenguajes hablados tuvieran los mismos requerimientos en cuanto a precisión sintáctica que los lenguajes de computadora. La mayoría de las especificaciones de los lenguajes de programación no describen la forma en que un compilador debe responder a los errores; el manejo de los mismos es responsabilidad del diseñador del compilador. La planeación del manejo de los errores desde el principio puede simplificar la estructura de un compilador y mejorar su capacidad para manejar los errores.

Los errores de programación comunes pueden ocurrir en muchos niveles distintos.

- Los errores *léxicos* incluyen la escritura incorrecta de los identificadores, las palabras clave o los operadores; por ejemplo, el uso de un identificador `tamanioElipce` en vez de `tamanioElipse`, y la omisión de comillas alrededor del texto que se debe interpretar como una cadena.
- Los errores *sintácticos* incluyen la colocación incorrecta de los signos de punto y coma, además de llaves adicionales o faltantes; es decir, “{” o “}”. Como otro ejemplo, en C o Java, la aparición de una instrucción `case` sin una instrucción `switch` que la encierre es un error sintáctico (sin embargo, por lo general, esta situación la acepta el analizador sintáctico y se atrapa más adelante en el procesamiento, cuando el compilador intenta generar código).
- Los errores *semánticos* incluyen los conflictos de tipos entre los operadores y los operandos. Un ejemplo es una instrucción `return` en un método de Java, con el tipo de resultado `void`.
- Los errores *lógicos* pueden ser cualquier cosa, desde un razonamiento incorrecto del programador en el uso (en un programa en C) del operador de asignación `=`, en vez del operador de comparación `==`. El programa que contenga `=` puede estar bien formado; sin embargo, tal vez no refleje la intención del programador.

La precisión de los métodos de análisis sintáctico permite detectar los errores sintácticos con mucha eficiencia. Varios métodos de análisis sintáctico, como los métodos LL y LR, detectan

un error lo más pronto posible; es decir, cuando el flujo de tokens que proviene del analizador léxico no puede seguirse analizando de acuerdo con la gramática para el lenguaje. Dicho en forma más precisa, tienen la *propiedad de prefijo viable*, lo cual significa que detectan la ocurrencia de un error tan pronto como ven un prefijo de la entrada que no puede completarse para formar una cadena válida en el lenguaje.

Otra de las razones para enfatizar la recuperación de los errores durante el análisis sintáctico es que muchos errores parecen ser sintácticos, sea cual fuere su causa, y se exponen cuando el análisis sintáctico no puede continuar. Algunos errores semánticos, como los conflictos entre los tipos, también pueden detectarse con eficiencia; sin embargo, la detección precisa de los errores semánticos y lógicos en tiempo de compilación es, por lo general, una tarea difícil.

El mango de errores en un analizador sintáctico tiene objetivos que son simples de declarar, pero difíciles de llevar a cabo:

- Reportar la presencia de errores con claridad y precisión.
- Recuperarse de cada error lo bastante rápido como para poder detectar los errores siguientes.
- Agregar una sobrecarga mínima al procesamiento de los programas correctos.

Por fortuna, los errores comunes son simples, y a menudo basta con un mecanismo simple para su manejo.

¿De qué manera un mango de errores debe reportar la presencia de un error? Como mínimo, debe reportar el lugar en el programa fuente en donde se detectó un error, ya que hay una buena probabilidad de que éste en sí haya ocurrido en uno de los pocos tokens anteriores. Una estrategia común es imprimir la línea del problema con un apuntador a la posición en la que se detectó el error.

4.1.4 Estrategias para recuperarse de los errores

Una vez que se detecta un error, ¿cómo debe recuperarse el analizador sintáctico? Aunque no hay una estrategia que haya demostrado ser aceptable en forma universal, algunos métodos pueden aplicarse en muchas situaciones. El método más simple es que el analizador sintáctico termine con un mensaje de error informativo cuando detecte el primer error. A menudo se descubren errores adicionales si el analizador sintáctico puede restaurarse a sí mismo, a un estado en el que pueda continuar el procesamiento de la entrada, con esperanzas razonables de que un mayor procesamiento proporcione información útil para el diagnóstico. Si los errores se apilan, es mejor para el compilador desistir después de exceder cierto límite de errores, que producir una molesta avalancha de errores “falsos”.

El resto de esta sección se dedica a las siguientes estrategias de recuperación de los errores: modo de pánico, nivel de frase, producciones de errores y corrección global.

Recuperación en modo de pánico

Con este método, al describir un error el analizador sintáctico descarta los símbolos de entrada, uno a la vez, hasta encontrar un conjunto designado de *tokens de sincronización*. Por lo general, los tokens de sincronización son delimitadores como el punto y coma o }, cuya función en

el programa fuente es clara y sin ambigüedades. El diseñador del compilador debe seleccionar los tokens de sincronización apropiados para el lenguaje fuente. Aunque la corrección en modo de pánico a menudo omite una cantidad considerable de entrada sin verificar errores adicionales, tiene la ventaja de ser simple y, a diferencia de ciertos métodos que consideraremos más adelante, se garantiza que no entrará en un ciclo infinito.

Recuperación a nivel de frase

Al descubrir un error, un analizador sintáctico puede realizar una corrección local sobre la entrada restante; es decir, puede sustituir un prefijo de la entrada restante por alguna cadena que le permita continuar. Una corrección local común es sustituir una coma por un punto y coma, eliminar un punto y coma extraño o insertar un punto y coma faltante. La elección de la corrección local se deja al diseñador del compilador. Desde luego que debemos tener cuidado de elegir sustituciones que no nos lleven hacia ciclos infinitos, como sería, por ejemplo, si siempre insertáramos algo en la entrada adelante del símbolo de entrada actual.

La sustitución a nivel de frase se ha utilizado en varios compiladores que reparan los errores, ya que puede corregir cualquier cadena de entrada. Su desventaja principal es la dificultad que tiene para arreglárselas con situaciones en las que el error actual ocurre antes del punto de detección.

Producciones de errores

Al anticipar los errores comunes que podríamos encontrar, podemos aumentar la gramática para el lenguaje, con producciones que generen las construcciones erróneas. Un analizador sintáctico construido a partir de una gramática aumentada por estas producciones de errores detecta los errores anticipados cuando se utiliza una producción de error durante el análisis sintáctico. Así, el analizador sintáctico puede generar diagnósticos de error apropiados sobre la construcción errónea que se haya reconocido en la entrada.

Corrección global

Lo ideal sería que un compilador hiciera la menor cantidad de cambios en el procesamiento de una cadena de entrada incorrecta. Hay algoritmos para elegir una secuencia mínima de cambios, para obtener una corrección con el menor costo a nivel global. Dada una cadena de entrada incorrecta x y una gramática G , estos algoritmos buscarán un árbol de análisis sintáctico para una cadena y relacionada, de tal forma que el número de inserciones, eliminaciones y modificaciones de los tokens requeridos para transformar a x en y sea lo más pequeño posible. Por desgracia, estos métodos son en general demasiado costosos para implementarlos en términos de tiempo y espacio, por lo cual estas técnicas sólo son de interés teórico en estos momentos.

Hay que observar que un programa casi correcto tal vez no sea lo que el programador tenía en mente. Sin embargo, la noción de la corrección con el menor costo proporciona una norma para evaluar las técnicas de recuperación de los errores, la cual se ha utilizado para buscar cadenas de sustitución óptimas para la recuperación a nivel de frase.

4.2 Gramáticas libres de contexto

En la sección 2.2 se presentaron las gramáticas para describir en forma sistemática la sintaxis de las construcciones de un lenguaje de programación, como las expresiones y las instrucciones. Si utilizamos una variable sintáctica *instr* para denotar las instrucciones, y una variable *expr* para denotar las expresiones, la siguiente producción:

$$\text{instr} \rightarrow \text{if} (\text{expr}) \text{instr} \text{else} \text{instr} \quad (4.4)$$

especifica la estructura de esta forma de instrucción condicional. Entonces, otras producciones definen con precisión lo que es una *expr* y qué más puede ser una *instr*.

En esta sección repasaremos la definición de una gramática libre de contexto y presentaremos la terminología para hablar acerca del análisis sintáctico. En especial, la noción de derivaciones es muy útil para hablar sobre el orden en el que se aplican las producciones durante el análisis sintáctico.

4.2.1 La definición formal de una gramática libre de contexto

En la sección 2.2 vimos que una gramática libre de contexto (o simplemente gramática) consiste en terminales, no terminales, un símbolo inicial y producciones.

1. Los *terminales* son los símbolos básicos a partir de los cuales se forman las cadenas. El término “nombre de token” es un sinónimo de “terminal”; con frecuencia usaremos la palabra “token” en vez de terminal, cuando esté claro que estamos hablando sólo sobre el nombre del token. Asumimos que las terminales son los primeros componentes de los tokens que produce el analizador léxico. En (4.4), los terminales son las palabras reservadas **if** y **else**, y los símbolos “(” y “)”.
2. Los *no terminales* son variables sintácticas que denotan conjuntos de cadenas. En (4.4), *instr* y *expr* son no terminales. Los conjuntos de cadenas denotados por los no terminales ayudan a definir el lenguaje generado por la gramática. Los no terminales imponen una estructura jerárquica sobre el lenguaje, que representa la clave para el análisis sintáctico y la traducción.
3. En una gramática, un no terminal se distingue como el *símbolo inicial*, y el conjunto de cadenas que denota es el lenguaje generado por la gramática. Por convención, las producciones para el símbolo inicial se listan primero.
4. Las producciones de una gramática especifican la forma en que pueden combinarse los terminales y los no terminales para formar cadenas. Cada *producción* consiste en:
 - (a) Un no terminal, conocido como *encabezado* o *lado izquierdo* de la producción; esta producción define algunas de las cadenas denotadas por el encabezado.
 - (b) El símbolo \rightarrow . Algunas veces se ha utilizado $::=$ en vez de la flecha.
 - (c) Un *cuerpo* o *lado derecho*, que consiste en cero o más terminales y no terminales. Los componentes del cuerpo describen una forma en que pueden construirse las cadenas del no terminal en el encabezado.

Ejemplo 4.5: La gramática en la figura 4.2 define expresiones aritméticas simples. En esta gramática, los símbolos de los terminales son:

id + - * / ()

Los símbolos de los no terminales son *expresión*, *term* y *factor*, y *expresión* es el símbolo inicial. \square

$$\begin{aligned}
 \text{expresión} &\rightarrow \text{expresión} + \text{term} \\
 \text{expresión} &\rightarrow \text{expresión} - \text{term} \\
 \text{expresión} &\rightarrow \text{term} \\
 \text{term} &\rightarrow \text{term} * \text{factor} \\
 \text{term} &\rightarrow \text{term} / \text{factor} \\
 \text{term} &\rightarrow \text{factor} \\
 \text{factor} &\rightarrow (\text{expresión}) \\
 \text{factor} &\rightarrow \text{id}
 \end{aligned}$$

Figura 4.2: Gramática para las expresiones aritméticas simples

4.2.2 Convenciones de notación

Para evitar siempre tener que decir que “éstos son los terminales”, “éstos son los no terminales”, etcétera, utilizaremos las siguientes convenciones de notación para las gramáticas durante el resto de este libro:

1. Estos símbolos son terminales:
 - (a) Las primeras letras minúsculas del alfabeto, como *a*, *b*, *c*.
 - (b) Los símbolos de operadores como +, *, etcétera.
 - (c) Los símbolos de puntuación como paréntesis, coma, etcétera.
 - (d) Los dígitos 0, 1, ..., 9.
 - (e) Las cadenas en negrita como **id** o **if**, cada una de las cuales representa un solo símbolo terminal.
2. Estos símbolos son no terminales:
 - (a) Las primeras letras mayúsculas del alfabeto, como *A*, *B*, *C*.
 - (b) La letra *S* que, al aparecer es, por lo general, el símbolo inicial.
 - (c) Los nombres en cursiva y minúsculas, como *expr* o *instr*.
 - (d) Al hablar sobre las construcciones de programación, las letras mayúsculas pueden utilizarse para representar no terminales. Por ejemplo, los no terminales para las expresiones, los términos y los factores se representan a menudo mediante *E*, *T* y *F*, respectivamente.

3. Las últimas letras mayúsculas del alfabeto, como X, Y, Z , representan *símbolos gramaticales*; es decir, pueden ser no terminales o terminales.
4. Las últimas letras minúsculas del alfabeto, como u, v, \dots, z , representan cadenas de terminales (posiblemente vacías).
5. Las letras griegas minúsculas α, β, γ , por ejemplo, representan cadenas (posiblemente vacías) de símbolos gramaticales. Por ende, una producción genérica puede escribirse como $A \rightarrow \alpha$, en donde A es el encabezado y α el cuerpo.
6. Un conjunto de producciones $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_k$ con un encabezado común A (las llamaremos *producciones A*), puede escribirse como $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_k$. A $\alpha_1, \alpha_2, \dots, \alpha_k$ les llamamos las *alternativas* para A .
7. A menos que se indique lo contrario, el encabezado de la primera producción es el símbolo inicial.

Ejemplo 4.6: Mediante estas convenciones, la gramática del ejemplo 4.5 puede rescribirse en forma concisa como:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid E - T \mid T \\ T & \rightarrow & T * F \mid T / F \mid F \\ F & \rightarrow & (E) \mid \text{id} \end{array}$$

Las convenciones de notación nos indican que E, T y F son no terminales, y E es el símbolo inicial. El resto de los símbolos son terminales. \square

4.2.3 Derivaciones

La construcción de un árbol de análisis sintáctico puede hacerse precisa si tomamos una vista derivacional, en la cual las producciones se tratan como reglas de rescritura. Empezando con el símbolo inicial, cada paso de rescritura sustituye a un no terminal por el cuerpo de una de sus producciones. Esta vista derivacional corresponde a la construcción descendente de un árbol de análisis sintáctico, pero la precisión que ofrecen las derivaciones será muy útil cuando hablamos del análisis sintáctico ascendente. Como veremos, el análisis sintáctico ascendente se relaciona con una clase de derivaciones conocidas como derivaciones de “más a la derecha”, en donde el no terminal por la derecha se rescribe en cada paso.

Por ejemplo, considere la siguiente gramática, con un solo no terminal E , la cual agrega una producción $E \rightarrow -E$ a la gramática (4.3):

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid \text{id} \quad (4.7)$$

La producción $E \rightarrow -E$ significa que si E denota una expresión, entonces $-E$ debe también denotar una expresión. La sustitución de una sola E por $-E$ se describirá escribiendo lo siguiente:

$$E \Rightarrow -E$$

lo cual se lee como “ E deriva a $-E$ ”. La producción $E \rightarrow (E)$ puede aplicarse para sustituir cualquier instancia de E en cualquier cadena de símbolos gramaticales por (E) ; por ejemplo, $E * E \Rightarrow (E) * E$ o $E * E \Rightarrow E * (E)$. Podemos tomar una sola E y aplicar producciones en forma repetida y en cualquier orden para obtener una secuencia de sustituciones. Por ejemplo,

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$

A dicha secuencia de sustituciones la llamamos una *derivación* de $-(\mathbf{id})$ a partir de E . Esta derivación proporciona la prueba de que la cadena $-(\mathbf{id})$ es una instancia específica de una expresión.

Para una definición general de la derivación, considere un no terminal A en la mitad de una secuencia de símbolos gramaticales, como en $\alpha A \beta$, en donde α y β son cadenas arbitrarias de símbolos gramaticales. Suponga que $A \rightarrow \gamma$ es una producción. Entonces, escribimos $\alpha A \beta \Rightarrow \alpha \gamma \beta$. El símbolo \Rightarrow significa, “se deriva en un paso”. Cuando una secuencia de pasos de derivación $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$ se describe como α_1 a α_n , decimos que α_1 *deriva* a α_n . Con frecuencia es conveniente poder decir, “deriva en cero o más pasos”. Para este fin, podemos usar el símbolo $\stackrel{*}{\Rightarrow}$. Así,

1. $\alpha \stackrel{*}{\Rightarrow} \alpha$, para cualquier cadena α .
2. Si $\alpha \stackrel{*}{\Rightarrow} \beta$ y $\beta \Rightarrow \gamma$, entonces $\alpha \stackrel{*}{\Rightarrow} \gamma$.

De igual forma, $\stackrel{\pm}{\Rightarrow}$ significa “deriva en uno o más pasos”.

Si $S \stackrel{*}{\Rightarrow} \alpha$, en donde S es el símbolo inicial de una gramática G , decimos que α es una *forma de frase* de G . Observe que una forma de frase puede contener tanto terminales como no terminales, y puede estar vacía. Un *enunciado* de G es una forma de frase sin símbolos no terminales. El *lenguaje generado* por una gramática es su conjunto de oraciones. Por ende, una cadena de terminales w está en $L(G)$, el lenguaje generado por G , si y sólo si w es un enunciado de G (o $S \stackrel{*}{\Rightarrow} w$). Un lenguaje que puede generarse mediante una gramática se considera un *lenguaje libre de contexto*. Si dos gramáticas generan el mismo lenguaje, se consideran como *equivalentes*.

La cadena $-(\mathbf{id} + \mathbf{id})$ es un enunciado de la gramática (4.7), ya que hay una derivación

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\mathbf{id} + E) \Rightarrow -(\mathbf{id} + \mathbf{id}) \quad (4.8)$$

Las cadenas E , $-E$, $-(E)$, \dots , $-(\mathbf{id} + \mathbf{id})$ son todas formas de frases de esta gramática. Escribimos $E \stackrel{*}{\Rightarrow} -(\mathbf{id} + \mathbf{id})$ para indicar que $-(\mathbf{id} + \mathbf{id})$ puede derivarse de E .

En cada paso de una derivación, hay dos elecciones por hacer. Debemos elegir qué no terminal debemos sustituir, y habiendo realizado esta elección, debemos elegir una producción con ese no terminal como encabezado. Por ejemplo, la siguiente derivación alternativa de $-(\mathbf{id} + \mathbf{id})$ difiere de la derivación (4.8) en los últimos dos pasos:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + \mathbf{id}) \Rightarrow -(\mathbf{id} + \mathbf{id}) \quad (4.9)$$

Cada no terminal se sustituye por el mismo cuerpo en las dos derivaciones, pero el orden de las sustituciones es distinto.

Para comprender la forma en que trabajan los analizadores sintácticos, debemos considerar las derivaciones en las que el no terminal que se va a sustituir en cada paso se elige de la siguiente manera:

1. En las derivaciones *por la izquierda*, siempre se elige el no terminal por la izquierda en cada de frase. Si $\alpha \Rightarrow \beta$ es un paso en el que se sustituye el no terminal por la izquierda en α , escribimos $\alpha \xrightarrow{lm} \beta$.
2. En las derivaciones *por la derecha*, siempre se elige el no terminal por la derecha; en este caso escribimos $\alpha \xrightarrow{rm} \beta$.

La derivación (4.8) es por la izquierda, por lo que puede rescribirse de la siguiente manera:

$$E \xrightarrow{lm} -E \xrightarrow{lm} -(E) \xrightarrow{lm} -(E + E) \xrightarrow{lm} -(\mathbf{id} + E) \xrightarrow{lm} -(\mathbf{id} + \mathbf{id})$$

Observe que (4.9) es una derivación por la derecha.

Si utilizamos nuestras convenciones de notación, cada paso por la izquierda puede escribirse como $wA\gamma \xrightarrow{lm} w\delta\gamma$, en donde w consiste sólo de terminales, $A \rightarrow \delta$ es la producción que se aplica, y γ es una cadena de símbolos gramaticales. Para enfatizar que α deriva a β mediante una derivación por la izquierda, escribimos $\alpha \xrightarrow{lm} \beta$. Si $S \xrightarrow{lm} \alpha$, decimos que α es una *forma de frase izquierda* de la gramática en cuestión.

Las análogas definiciones son válidas para las derivaciones por la derecha. A estas derivaciones se les conoce algunas veces como derivaciones *canónicas*.

4.2.4 Árboles de análisis sintáctico y derivaciones

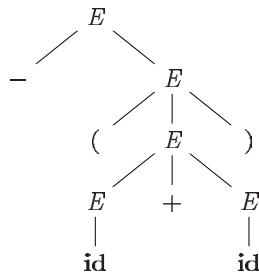
Un árbol de análisis sintáctico es una representación gráfica de una derivación que filtra el orden en el que se aplican las producciones para sustituir los no terminales. Cada nodo interior de un árbol de análisis sintáctico representa la aplicación de una producción. El nodo interior se etiqueta con el no terminal A en el encabezado de la producción; los hijos del nodo se etiquetan, de izquierda a derecha, mediante los símbolos en el cuerpo de la producción por la que se sustituyó esta A durante la derivación.

Por ejemplo, el árbol de análisis sintáctico para $-(\mathbf{id} + \mathbf{id})$ en la figura 4.3 resulta de la derivación (4.8), así como de la derivación (4.9).

Las hojas de un árbol de análisis sintáctico se etiquetan mediante no terminales o terminales y, leídas de izquierda a derecha, constituyen una forma de frase, a la cual se le llama *producto* o *frontera* del árbol.

Para ver la relación entre las derivaciones y los árboles de análisis sintáctico, considere cualquier derivación $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$, en donde α_1 es un sólo no terminal A . Para cada forma de frase α_i en la derivación, podemos construir un árbol de análisis sintáctico cuyo producto sea α_i . El proceso es una inducción sobre i .

BASE: El árbol para $\alpha_1 = A$ es un solo nodo, etiquetado como A .

Figura 4.3: Árbol de análisis sintáctico para $-(\text{id} + \text{id})$

INDUCCIÓN: Suponga que ya hemos construido un árbol de análisis sintáctico con el producto $\alpha_{i-1} = X_1 X_2 \cdots X_k$ (tenga en cuenta que, de acuerdo a nuestras convenciones de notación, cada símbolo gramatical X_i es un no terminal o un terminal). Suponga que α_i se deriva de α_{i-1} al sustituir X_j , un no terminal, por $\beta = Y_1 Y_2 \cdots Y_m$. Es decir, en el i -ésimo paso de la derivación, la producción $X_j \rightarrow \beta$ se aplica a α_{i-1} para derivar $\alpha_i = X_1 X_2 \cdots X_{j-1} \beta X_{j+1} \cdots X_k$.

Para modelar este paso de la derivación, buscamos la j -ésima hoja, partiendo de la izquierda en el árbol de análisis sintáctico actual. Esta hoja se etiqueta como X_j . A esta hoja le damos m hijos, etiquetados Y_1, Y_2, \dots, Y_m , partiendo de la izquierda. Como caso especial, si $m = 0$ entonces $\beta = \epsilon$, y proporcionamos a la j -ésima hoja un hijo etiquetado como ϵ .

Ejemplo 4.10: La secuencia de árboles de análisis sintáctico que se construyen a partir de la derivación (4.8) se muestra en la figura 4.4. En el primer paso de la derivación, $E \Rightarrow -E$. Para modelar este paso, se agregan dos hijos, etiquetados como $-$ y E , a la raíz E del árbol inicial. El resultado es el segundo árbol.

En el segundo paso de la derivación, $-E \Rightarrow -(E)$. Por consiguiente, agregamos tres hijos, etiquetados como $($, E y $)$, al nodo hoja etiquetado como E del segundo árbol, para obtener el tercer árbol con coséchale producto $-(E)$. Si continuamos de esta forma, obtenemos el árbol de análisis sintáctico completo como el sexto árbol. \square

Como un árbol de análisis sintáctico ignora las variaciones en el orden en el que se sustituyen los símbolos en las formas de las oraciones, hay una relación de varios a uno entre las derivaciones y los árboles de análisis sintáctico. Por ejemplo, ambas derivaciones (4.8) y (4.9) se asocian con el mismo árbol de análisis sintáctico final de la figura 4.4.

En lo que sigue, realizaremos con frecuencia el análisis sintáctico produciendo una derivación por la izquierda o por la derecha, ya que hay una relación de uno a uno entre los árboles de análisis sintáctico y este tipo de derivaciones. Tanto las derivaciones por la izquierda como las de por la derecha eligen un orden específico para sustituir símbolos en las formas de las oraciones, por lo que también filtran las variaciones en orden. No es difícil mostrar que todos los árboles sintácticos tienen asociadas una derivación única por la izquierda y una derivación única por la derecha.

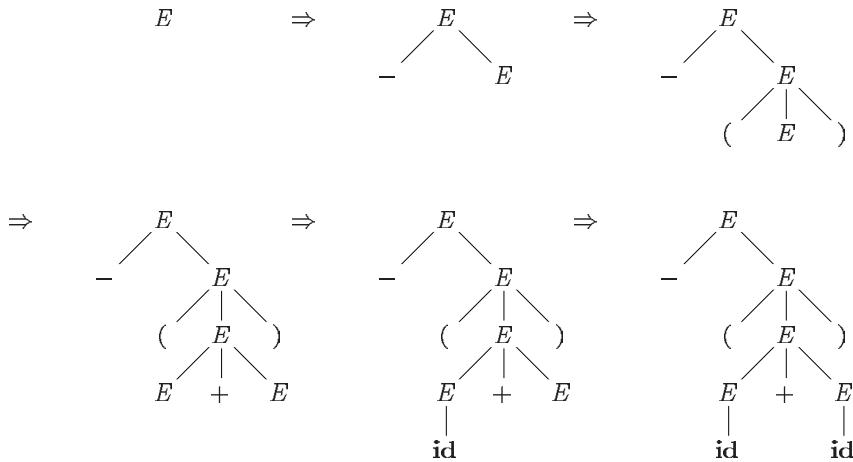


Figura 4.4: Secuencia de árboles de análisis sintáctico para la derivación (4.8)

4.2.5 Ambigüedad

En la sección 2.2.4 vimos que una gramática que produce más de un árbol de análisis sintáctico para cierto enunciado es *ambiguo*. Dicho de otra forma, una gramática ambigua es aquella que produce más de una derivación por la izquierda, o más de una derivación por la derecha para el mismo enunciado.

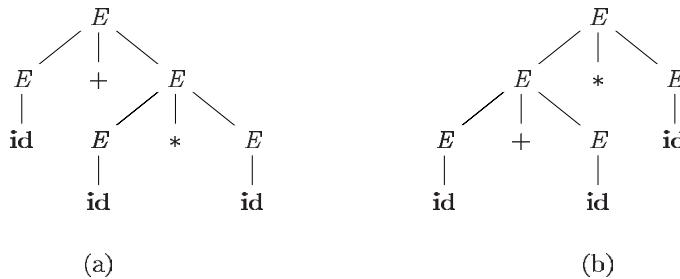
Ejemplo 4.11: La gramática de expresiones aritméticas (4.3) permite dos derivaciones por la izquierda distintas para el enunciado $id + id * id$:

$$\begin{array}{ll}
 E \Rightarrow E + E & E \Rightarrow E * E \\
 \Rightarrow id + E & \Rightarrow E + E * E \\
 \Rightarrow id + E * E & \Rightarrow id + E * E \\
 \Rightarrow id + id * E & \Rightarrow id + id * E \\
 \Rightarrow id + id * id & \Rightarrow id + id * id
 \end{array}$$

Los árboles de análisis sintáctico correspondientes aparecen en la figura 4.5.

Observe que el árbol de análisis sintáctico de la figura 4.5(a) refleja la precedencia que se asume comúnmente para $+$ y $*$, mientras que el árbol de la figura 4.5(b) no. Es decir, lo común es tratar al operador $*$ teniendo mayor precedencia que $+$, en forma correspondiente al hecho de que, por lo general, evaluamos la expresión $a + b * c$ como $a + (b * c)$, en vez de hacerlo como $(a + b) * c$. \square

Para la mayoría de los analizadores sintácticos, es conveniente que la gramática no tenga ambigüedades, ya que de lo contrario, no podemos determinar en forma única qué árbol de análisis sintáctico seleccionar para un enunciado. En otros casos, es conveniente usar gramáticas ambiguas elegidas con cuidado, junto con *reglas para eliminar la ambigüedad*, las cuales “descartan” los árboles sintácticos no deseados, dejando sólo un árbol para cada enunciado.

Figura 4.5: Dos árboles de análisis sintáctico para **id+id*id**

4.2.6 Verificación del lenguaje generado por una gramática

Aunque los diseñadores de compiladores muy raras veces lo hacen para una gramática de lenguaje de programación completa, es útil poder razonar que un conjunto dado de producciones genera un lenguaje específico. Las construcciones problemáticas pueden estudiarse mediante la escritura de una gramática abstracta y concisa, y estudiando el lenguaje que genera. A continuación vamos a construir una gramática de este tipo, para instrucciones condicionales.

Una prueba de que una gramática G genera un lenguaje L consta de dos partes: mostrar que todas las cadenas generadas por G están en L y, de manera inversa, que todas las cadenas en L pueden generarse sin duda mediante G .

Ejemplo 4.12: Considere la siguiente gramática:

$$S \rightarrow (S) S | \epsilon \quad (4.13)$$

Tal vez no sea evidente desde un principio, pero esta gramática simple genera todas las cadenas de paréntesis balanceados, y sólo ese tipo de cadenas. Para ver por qué, primero mostraremos que todas las frases que se derivan de S son balanceadas, y después que todas las cadenas balanceadas se derivan de S . Para mostrar que todas las frases que pueden derivarse de S son balanceadas, utilizaremos una prueba inductiva sobre el número de pasos n en una derivación.

BASE: La base es $n = 1$. La única cadena de terminales que puede derivarse de S en un paso es la cadena vacía, que sin duda está balanceada.

INDUCCIÓN: Ahora suponga que todas las derivaciones de menos de n pasos producen frases balanceadas, y considere una derivación por la izquierda, con n pasos exactamente. Dicha derivación debe ser de la siguiente forma:

$$S \xrightarrow{tm} (S)S \xrightarrow{tm} (x)S \xrightarrow{tm} (x)y$$

Las derivaciones de x y y que provienen de S requieren menos de n pasos, por lo que en base a la hipótesis inductiva, x y y están balanceadas. Por lo tanto, la cadena $(x)y$ debe ser balanceada. Es decir, tiene un número equivalente de paréntesis izquierdos y derechos, y cada prefijo tiene, por lo menos, la misma cantidad de paréntesis izquierdos que derechos.

Habiendo demostrado entonces que cualquier cadena que se deriva de S está balanceada, debemos ahora mostrar que todas las cadenas balanceadas se derivan de S . Para ello, utilizaremos la inducción sobre la longitud de una cadena.

BASE: Si la cadena es de longitud 0, debe ser ϵ , la cual está balanceada.

INDUCCIÓN: Primero, observe que todas las cadenas balanceadas tienen longitud uniforme. Suponga que todas las cadenas balanceadas de una longitud menor a $2n$ se derivan de S , y considere una cadena balanceada w de longitud $2n$, $n \geq 1$. Sin duda, w empieza con un paréntesis izquierdo. Hagamos que (x) sea el prefijo no vacío más corto de w , que tenga el mismo número de paréntesis izquierdos y derechos. Así, w puede escribirse como $w = (x)y$, en donde x y y están balanceadas. Como x y y tienen una longitud menor a $2n$, pueden derivarse de S mediante la hipótesis inductiva. Por ende, podemos buscar una derivación de la siguiente forma:

$$S \Rightarrow (S)S \xrightarrow{*} (x)S \xrightarrow{*} (x)y$$

con lo cual demostramos que $w = (x)y$ también puede derivarse de S . \square

4.2.7 Comparación entre gramáticas libres de contexto y expresiones regulares

Antes de dejar esta sección sobre las gramáticas y sus propiedades, establecemos que las gramáticas son una notación más poderosa que las expresiones regulares. Cada construcción que puede describirse mediante una expresión regular puede describirse mediante una gramática, pero no al revés. De manera alternativa, cada lenguaje regular es un lenguaje libre de contexto, pero no al revés.

Por ejemplo, la expresión regular $(a|b)^*abb$ y la siguiente gramática:

$$\begin{array}{lcl} A_0 & \rightarrow & aA_0 \mid bA_0 \mid aA_1 \\ A_1 & \rightarrow & bA_2 \\ A_2 & \rightarrow & bA_3 \\ A_3 & \rightarrow & \epsilon \end{array}$$

describen el mismo lenguaje, el conjunto de cadenas de as y bs que terminan en abb .

Podemos construir de manera mecánica una gramática para reconocer el mismo lenguaje que un autómata finito no determinista (AFN). La gramática anterior se construyó a partir del AFN de la figura 3.24, mediante la siguiente construcción:

1. Para cada estado i del AFN, crear un no terminal A_i .
2. Si el estado i tiene una transición al estado j con la entrada a , agregar la producción $A_i \rightarrow aA_j$. Si el estado i pasa al estado j con la entrada ϵ , agregar la producción $A_i \rightarrow A_j$.
3. Si i es un estado de aceptación, agregar $A_i \rightarrow \epsilon$.
4. Si i es el estado inicial, hacer que A_i sea el símbolo inicial de la gramática.

Por otra parte, el lenguaje $L = \{a^n b^n \mid n \geq 1\}$ con un número equivalente de *as* y *bs* es un ejemplo de prototipo de un lenguaje que puede describirse mediante una gramática, pero no mediante una expresión regular. Para ver por qué, suponga que L es el lenguaje definido por alguna expresión regular. Construiríamos un AFD D con un número finito de estados, por decir k , para aceptar a L . Como D sólo tiene k estados, para una entrada que empieza con más de k *as*, D debe entrar a cierto estado dos veces, por decir s_i , como en la figura 4.6. Suponga que la ruta de s_i de vuelta a sí mismo se etiqueta con una secuencia a^{j-i} . Como $a^i b^i$ está en el lenguaje, debe haber una ruta etiquetada como b^i desde s_i hasta un estado de aceptación f . Pero, entonces también hay una ruta que sale desde el estado s_0 y pasa a través de s_i para llegar a f , etiquetada como $a^i b^i$, como se muestra en la figura 4.6. Por ende, D también acepta a $a^i b^i$, que no está en el lenguaje, lo cual contradice la suposición de que L es el lenguaje aceptado por D .

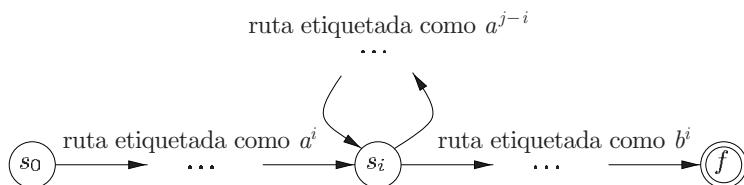


Figura 4.6: Un AFD D que acepta a $a^i b^i$ y a $a^j b^i$

En lenguaje coloquial, decimos que “los autómatas finitos no pueden contar”, lo cual significa que un autómata finito no puede aceptar un lenguaje como $\{a^n b^n \mid n \geq 1\}$, que requiera que el autómata lleve la cuenta del número de *as* antes de ver las *bs*. De igual forma, “una gramática puede contar dos elementos pero no tres”, como veremos cuando hablemos sobre las construcciones de lenguajes que no son libres de contexto en la sección 4.3.5.

4.2.8 Ejercicios para la sección 4.2

Ejercicio 4.2.1: Considere la siguiente gramática libre de contexto:

$$S \rightarrow S S + \mid S S * \mid a$$

y la cadena $aa + a*$.

- Proporcione una derivación por la izquierda para la cadena.
- Proporcione una derivación por la derecha para la cadena.
- Proporcione un árbol de análisis sintáctico para la cadena.
- ¿La gramática es ambigua o no? Justifique su respuesta.
- Describa el lenguaje generado por esta gramática.

Ejercicio 4.2.2: Repita el ejercicio 4.2.1 para cada una de las siguientes gramáticas y cadenas:

- a) $S \rightarrow 0 S 1 \mid 0 1$ con la cadena 000111.
- b) $S \rightarrow + S S \mid * S S \mid a$ con la cadena $+ * aaa$.
- ! c) $S \rightarrow S (S) S \mid \epsilon$ con la cadena $((())$.
- ! d) $S \rightarrow S + S \mid S S \mid (S) \mid S * \mid a$ con la cadena $(a + a) * a$.
- ! e) $S \rightarrow (L) \mid a$ y $L \rightarrow L, S \mid S$ con la cadena $((a, a), a, (a))$.
- !! f) $S \rightarrow a S b S \mid b S a S \mid \epsilon$ con la cadena $aabbab$.
- ! g) La siguiente gramática para las expresiones booleanas:

$$\begin{array}{lcl} bexpr & \rightarrow & bexpr \text{ or } bterm \mid bterm \\ bterm & \rightarrow & bterm \text{ and } bfactor \mid bfactor \\ bfactor & \rightarrow & \text{not } bfactor \mid (bexpr) \mid \text{true} \mid \text{false} \end{array}$$

Ejercicio 4.2.3: Diseñe gramáticas para los siguientes lenguajes:

- a) El conjunto de todas las cadenas de 0s y 1s, de tal forma que justo antes de cada 0 vaya por lo menos un 1.
- ! b) El conjunto de todas las cadenas de 0s y 1s que sean *palíndromos*; es decir, que la cadena se lea igual al derecho y al revés.
- ! c) El conjunto de todas las cadenas de 0s y 1s con un número igual de 0s y 1s.
- !! d) El conjunto de todas las cadenas de 0s y 1s con un número desigual de 0s y 1s.
- ! e) El conjunto de todas las cadenas de 0s y 1s en donde 011 no aparece como una subcadena.
- !! f) El conjunto de todas las cadenas de 0s y 1s de la forma xy , en donde $x \neq y$, y x y y tienen la misma longitud.

! Ejercicio 4.2.4: Hay una notación de gramática extendida de uso común. En esta notación, los corchetes y las llaves en los cuerpos de las producciones son meta símbolos (como \rightarrow o $|$) con los siguientes significados:

- i) Los corchetes alrededor de un símbolo o símbolos gramaticales denota que estas construcciones son opcionales. Por ende, la producción $A \rightarrow X [Y] Z$ tiene el mismo efecto que las dos producciones $A \rightarrow X Y Z$ y $A \rightarrow X Z$.
- ii) Las llaves alrededor de un símbolo o símbolos gramaticales indican que estos símbolos pueden repetirse cualquier número de veces, incluyendo cero. Por ende, $A \rightarrow X \{ Y Z \}$ tiene el mismo efecto que la secuencia infinita de producciones $A \rightarrow X$, $A \rightarrow X Y Z$, $A \rightarrow X Y Z Y Z$, y así sucesivamente.

Muestre que estas dos extensiones no agregan potencia a las gramáticas; es decir, cualquier lenguaje que pueda generarse mediante una gramática con estas extensiones, podrá generarse mediante una gramática sin las extensiones.

Ejercicio 4.2.5: Use las llaves descritas en el ejercicio 4.2.4 para simplificar la siguiente gramática para los bloques de instrucciones y las instrucciones condicionales:

$$\begin{array}{lcl} instr & \rightarrow & \mathbf{if} \ expr \mathbf{then} \ instr \mathbf{else} \ instr \\ & | & \mathbf{if} \ instr \mathbf{then} \ instr \\ & | & \mathbf{begin} \ listaInstr \ \mathbf{end} \\ listaInstr & \rightarrow & instr \ ; \ listaInstr \ | \ instr \end{array}$$

! **Ejercicio 4.2.6:** Exienda la idea del ejercicio 4.2.4 para permitir cualquier expresión regular de símbolos gramaticales en el cuerpo de una producción. Muestre que esta extensión no permite que las gramáticas definan nuevos lenguajes.

! **Ejercicio 4.2.7:** Un símbolo gramatical X (terminal o no terminal) es *inútil* si no hay derivación de la forma $S \xrightarrow{*} wXy \xrightarrow{*} wxy$. Es decir, X nunca podrá aparecer en la derivación de un enunciado.

- Proporcione un algoritmo para eliminar de una gramática todas las producciones que contengan símbolos inútiles.
- Aplique su algoritmo a la siguiente gramática:

$$\begin{array}{lcl} S & \rightarrow & 0 \ | \ A \\ A & \rightarrow & AB \\ B & \rightarrow & 1 \end{array}$$

Ejercicio 4.2.8: La gramática en la figura 4.7 genera declaraciones para un solo identificador numérico; estas declaraciones involucran a cuatro propiedades distintas e independientes de números.

$$\begin{array}{lcl} instr & \rightarrow & \mathbf{declare} \ id \ listaOpciones \\ listaOpciones & \rightarrow & listaOpciones \ opcion \ | \ \epsilon \\ opcion & \rightarrow & modo \ | \ escala \ | \ precision \ | \ base \\ modo & \rightarrow & \mathbf{real} \ | \ \mathbf{complex} \\ escala & \rightarrow & \mathbf{fixed} \ | \ \mathbf{floating} \\ precision & \rightarrow & \mathbf{single} \ | \ \mathbf{double} \\ base & \rightarrow & \mathbf{binary} \ | \ \mathbf{decimal} \end{array}$$

Figura 4.7: Una gramática para declaraciones con varios atributos

- Generalice la gramática de la figura 4.7, permitiendo n opciones A_i , para algunas n fijas y para $i = 1, 2, \dots, n$, en donde A_i puede ser a_i o b_i . Su gramática deberá usar sólo $O(n)$ símbolos gramaticales y tener una longitud total de producciones igual a $O(n)$.

- ! b) La gramática de la figura 4.7 y su generalización en la parte (a) permite declaraciones que son contradictorias o redundantes, tales como:

```
declare foo real fixed real floating
```

Podríamos insistir en que la sintaxis del lenguaje prohíbe dichas declaraciones; es decir, cada declaración generada por la gramática tiene exactamente un valor para cada una de las n opciones. Si lo hacemos, entonces para cualquier n fija hay sólo un número finito de declaraciones legales. Por ende, el lenguaje de declaraciones legales tiene una gramática (y también una expresión regular), al igual que cualquier lenguaje finito. La gramática obvia, en la cual el símbolo inicial tiene una producción para cada declaración legal, tiene $n!$ producciones y una longitud de producciones total de $O(n \times n!)$. Hay que esforzarse más: una longitud de producciones total que sea $O(n2^n)$.

- !! c) Muestre que cualquier gramática para la parte (b) debe tener una longitud de producciones total de por lo menos 2^n .
- d) ¿Qué dice la parte (c) acerca de la viabilidad de imponer la no redundancia y la no contradicción entre las opciones en las declaraciones, a través de la sintaxis del lenguaje de programación?

4.3 Escritura de una gramática

Las gramáticas son capaces de describir casi la mayoría de la sintaxis de los lenguajes de programación. Por ejemplo, el requerimiento de que los identificadores deben declararse antes de usarse, no puede describirse mediante una gramática libre de contexto. Por lo tanto, las secuencias de los tokens que acepta un analizador sintáctico forman un superconjunto del lenguaje de programación; las fases siguientes del compilador deben analizar la salida del analizador sintáctico, para asegurar que cumpla con las reglas que no verifica el analizador sintáctico.

Esta sección empieza con una discusión acerca de cómo dividir el trabajo entre un analizador léxico y un analizador sintáctico. Después consideraremos varias transformaciones que podrían aplicarse para obtener una gramática más adecuada para el análisis sintáctico. Una técnica puede eliminar la ambigüedad en la gramática, y las otras (eliminación de recursividad por la izquierda y factorización por la izquierda) son útiles para rescribir las gramáticas, de manera que sean adecuadas para el análisis sintáctico descendente. Concluiremos esta sección considerando algunas construcciones de los lenguajes de programación que ninguna gramática puede describir.

4.3.1 Comparación entre análisis léxico y análisis sintáctico

Como observamos en la sección 4.2.7, todo lo que puede describirse mediante una expresión regular también puede describirse mediante una gramática. Por lo tanto, sería razonable preguntar: “¿Por qué usar expresiones regulares para definir la sintaxis léxica de un lenguaje?” Existen varias razones.

1. Al separar la estructura sintáctica de un lenguaje en partes léxicas y no léxicas, se proporciona una manera conveniente de colocar en módulos la interfaz de usuario de un compilador en dos componentes de un tamaño manejable.
2. Las reglas léxicas de un lenguaje son con frecuencia bastante simples, y para describirlas no necesitamos una notación tan poderosa como las gramáticas.
3. Por lo general, las expresiones regulares proporcionan una notación más concisa y fácil de entender para los tokens, en comparación con las gramáticas.
4. Pueden construirse analizadores léxicos más eficientes en forma automática a partir de expresiones regulares, en comparación con las gramáticas arbitrarias.

No hay lineamientos firmes en lo que se debe poner en las reglas léxicas, en contraste a las reglas sintácticas. Las expresiones regulares son muy útiles para describir la estructura de las construcciones como los identificadores, las constantes, las palabras reservadas y el espacio en blanco. Por otro lado, las gramáticas son muy útiles para describir estructuras anidadas, como los paréntesis balanceados, las instrucciones begin-end relacionadas, las instrucciones if-then-else correspondientes, etcétera. Estas estructuras anidadas no pueden describirse mediante las expresiones regulares.

4.3.2 Eliminación de la ambigüedad

Algunas veces, una gramática ambigua puede describirse para eliminar la ambigüedad. Como ejemplo, vamos a eliminar la ambigüedad de la siguiente gramática del “else colgante”:

$$\begin{array}{lcl}
 \text{instr} & \rightarrow & \text{if } \text{expr} \text{ then } \text{instr} \\
 & | & \text{if } \text{expr} \text{ then } \text{instr} \text{ else } \text{instr} \\
 & | & \text{otra}
 \end{array} \tag{4.14}$$

Aquí, “**otra**” representa a cualquier otra instrucción. De acuerdo con esta gramática, la siguiente instrucción condicional compuesta:

if E_1 **then** S_1 **else** **if** E_2 **then** S_2 **else** S_3

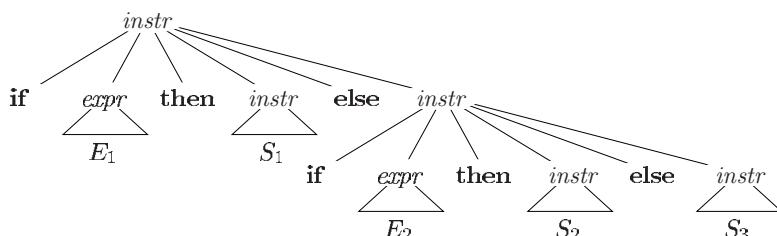


Figura 4.8: Árbol de análisis sintáctico para una instrucción condicional

tiene el árbol de análisis sintáctico que se muestra en la figura 4.8.¹ La gramática (4.14) es ambigua, ya que la cadena

$$\mathbf{if} \ E_1 \ \mathbf{then} \ \mathbf{if} \ E_2 \ \mathbf{then} \ S_1 \ \mathbf{else} \ S_2 \quad (4.15)$$

tiene los dos árboles de análisis sintáctico que se muestran en la figura 4.9.

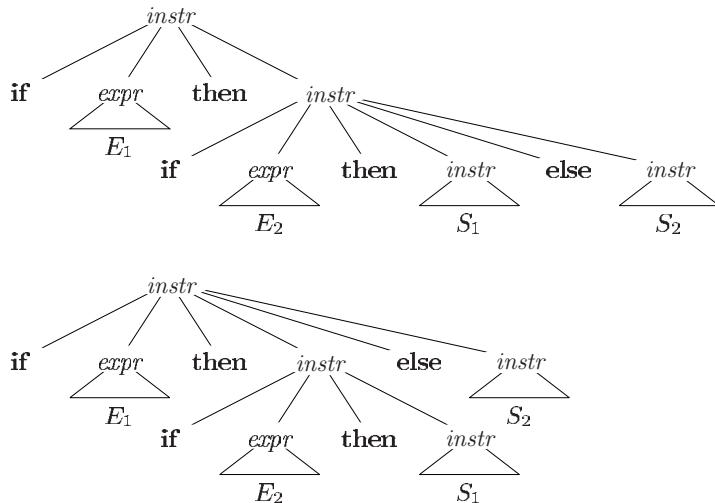


Figura 4.9: Dos árboles de análisis sintáctico para un enunciado ambiguo

En todos los lenguajes de programación con instrucciones condicionales de esta forma, se prefiere el primer árbol de análisis sintáctico. La regla general es, “Relacionar cada **else** con el **then** más cercano que no esté relacionado”.² Esta regla para eliminar ambigüedad puede, en teoría, incorporarse directamente en una gramática, pero en la práctica raras veces se integra a las producciones.

Ejemplo 4.16: Podemos rescribir la gramática del **else** colgante (4.14) como la siguiente gramática sin ambigüedades. La idea es que una instrucción que aparece entre un **then** y un **else** debe estar “relacionada”; es decir, la instrucción interior no debe terminar con un **then** sin relacionar o abierto. Una instrucción relacionada es una instrucción **if-then-else** que no contiene instrucciones abiertas, o es cualquier otro tipo de instrucción incondicional. Por ende, podemos usar la gramática de la figura 4.10. Esta gramática genera las mismas cadenas que la gramática del **else** colgante (4.14), pero sólo permite un análisis sintáctico para la cadena (4.15); en específico, el que asocia a cada **else** con la instrucción **then** más cercana que no haya estado relacionada antes. □

¹Los subíndices en E y S son sólo para diferenciar las distintas ocurrencias del mismo no terminal, por lo cual no implican no terminales distintos.

²Hay que tener en cuenta que C y sus derivados se incluyen en esta clase. Aun cuando la familia de lenguajes C no utiliza la palabra clave **then**, su función se representa mediante el paréntesis de cierre para la condición que va después de **if**.

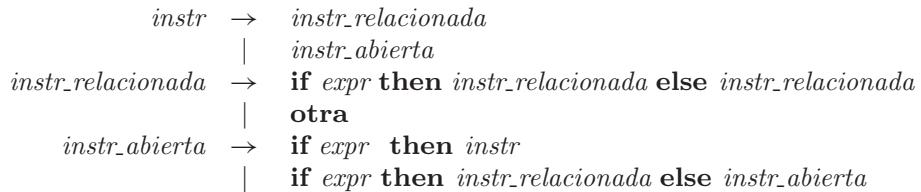


Figura 4.10: Gramática sin ambigüedades para las instrucciones if-then-else

4.3.3 Eliminación de la recursividad por la izquierda

Una gramática es *recursiva por la izquierda* si tiene una terminal A tal que haya una derivación $A \stackrel{*}{\Rightarrow} A\alpha$ para cierta cadena α . Los métodos de análisis sintáctico descendentes no pueden manejar las gramáticas recursivas por la izquierda, por lo que se necesita una transformación para eliminar la recursividad por la izquierda. En la sección 2.4.5 hablamos sobre la *recursividad inmediata por la izquierda*, en donde hay una producción de la forma $A \rightarrow A\alpha$. Aquí estudiaremos el caso general. En la sección 2.4.5, mostramos cómo el par recursivo por la izquierda de producciones $A \rightarrow A\alpha \mid \beta$ podía sustituirse mediante las siguientes producciones no recursivas por la izquierda:

$$\begin{aligned}
 A &\rightarrow \beta A' \\
 A' &\rightarrow \alpha A' \mid \epsilon
 \end{aligned}$$

sin cambiar las cadenas que se derivan de A . Esta regla por sí sola basta para muchas gramáticas.

Ejemplo 4.17: La gramática de expresiones no recursivas por la izquierda (4.2), que se repite a continuación:

$$\begin{aligned}
 E &\rightarrow T E' \\
 E' &\rightarrow + T E' \\
 T &\rightarrow F T' \\
 T' &\rightarrow * F T' \\
 F &\rightarrow (E) \mid \text{id}
 \end{aligned}$$

se obtiene mediante la eliminación de la recursividad inmediata por la izquierda de la gramática de expresiones (4.1). El par recursivo por la izquierda de las producciones $E \rightarrow E + T \mid T$ se sustituye mediante $E \rightarrow T E'$ y $E' \rightarrow + T E' \mid \epsilon$. Las nuevas producciones para T y T' se obtienen de manera similar, eliminando la recursividad inmediata por la izquierda. \square

La recursividad inmediata por la izquierda puede eliminarse mediante la siguiente técnica, que funciona para cualquier número de producciones A . En primer lugar, se agrupan las producciones de la siguiente manera:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

en donde ninguna β_i termina con una A . Después, se sustituyen las producciones A mediante lo siguiente:

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A' \\ A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

El no terminal A genera las mismas cadenas que antes, pero ya no es recursiva por la izquierda. Este procedimiento elimina toda la recursividad por la izquierda de las producciones A y A' (siempre y cuando ninguna α_i sea ϵ), pero no elimina la recursividad por la izquierda que incluye a las derivaciones de dos o más pasos. Por ejemplo, considere la siguiente gramática:

$$\begin{aligned} S &\rightarrow A \ a \mid b \\ A &\rightarrow A \ c \mid S \ d \mid \epsilon \end{aligned} \tag{4.18}$$

El no terminal S es recursiva por la izquierda, ya que $S \Rightarrow Aa \Rightarrow Sda$, pero no es inmediatamente recursiva por la izquierda.

El Algoritmo 4.19, que se muestra a continuación, elimina en forma sistemática la recursividad por la izquierda de una gramática. Se garantiza que funciona si la gramática no tiene ciclos (derivaciones de la forma $A \xrightarrow{*} A$) o producciones ϵ (producciones de la forma $A \rightarrow \epsilon$). Los ciclos pueden eliminarse en forma sistemática de una gramática, al igual que las producciones ϵ (vea los ejercicios 4.4.6 y 4.4.7).

Algoritmo 4.19: Eliminación de la recursividad por la izquierda.

ENTRADA: La gramática G sin ciclos ni producciones ϵ .

SALIDA: Una gramática equivalente sin recursividad por la izquierda.

MÉTODO: Aplicar el algoritmo de la figura 4.11 a G . Observe que la gramática no recursiva por la izquierda resultante puede tener producciones ϵ . \square

- 1) ordenar los no terminales de cierta forma A_1, A_2, \dots, A_n .
- 2) **for** (cada i de 1 a n) {
- 3) **for** (cada j de 1 a $i - 1$) {
- 4) sustituir cada producción de la forma $A_i \rightarrow A_j \gamma$ por las producciones $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$, en donde $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ sean todas producciones A_j actuales
- 5) }
- 6) eliminar la recursividad inmediata por la izquierda entre las producciones A_i
- 7) }

Figura 4.11: Algoritmo para eliminar la recursividad por la izquierda de una gramática

El procedimiento en la figura 4.11 funciona de la siguiente manera. En la primera iteración para $i = 1$, el ciclo for externo de las líneas (2) a la (7) elimina cualquier recursividad inmediata por la izquierda entre las producciones A_1 . Cualquier producción A_1 restante de la forma $A_1 \rightarrow A_l \alpha$ debe, por lo tanto, tener $l > 1$. Después de la $i-1$ -ésima iteración del ciclo for externo, todas las no terminales A_k , en donde $k < i$, se “limpian”; es decir, cualquier producción $A_k \rightarrow A_l \alpha$ debe tener $l > k$. Como resultado, en la i -ésima iteración, el ciclo interno de las líneas (3) a la (5) eleva en forma progresiva el límite inferior en cualquier producción $A_i \rightarrow A_m \alpha$, hasta tener $m \geq i$.

Después, la eliminación de la recursividad inmediata por la izquierda para las producciones A_i en la línea (6) obliga a que m sea mayor que i .

Ejemplo 4.20: Vamos a aplicar el Algoritmo 4.19 a la gramática (4.18). Técnicamente, no se garantiza que el algoritmo vaya a funcionar debido a la producción ϵ , pero en este caso, la producción $A \rightarrow \epsilon$ resulta ser inofensiva.

Ordenamos los no terminales S, A . No hay recursividad inmediata por la izquierda entre las producciones S , por lo que no ocurre nada durante el ciclo externo para $i = 1$. Para $i = 2$, sustituimos la S en $A \rightarrow S d$ para obtener las siguientes producciones A .

$$A \rightarrow A c \mid A a d \mid b d \mid \epsilon$$

Al eliminar la recursividad inmediata por la izquierda entre las producciones A produce la siguiente gramática:

$$\begin{aligned} S &\rightarrow A a \mid b \\ A &\rightarrow b d A' \mid A' \\ A' &\rightarrow c A' \mid a d A' \mid \epsilon \end{aligned}$$

□

4.3.4 Factorización por la izquierda

La factorización por la izquierda es una transformación gramatical, útil para producir una gramática adecuada para el análisis sintáctico predictivo, o descendente. Cuando la elección entre dos producciones A alternativas no está clara, tal vez podamos rescribir las producciones para diferir la decisión hasta haber visto la suficiente entrada como para poder realizar la elección correcta.

Por ejemplo, si tenemos las siguientes dos producciones:

$$\begin{aligned} \text{instr} &\rightarrow \text{if } \text{expr} \text{ then } \text{instr} \text{ else } \text{instr} \\ &\mid \text{if } \text{expr} \text{ then } \text{instr} \end{aligned}$$

al ver la entrada **if**, no podemos saber de inmediato qué producción elegir para expandir instr . En general, si $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ son dos producciones A , y la entrada empieza con una cadena no vacía derivada de α , no sabemos si debemos expandir A a $\alpha\beta_1$ o a $\alpha\beta_2$. No obstante, podemos diferir la decisión si expandimos A a $\alpha A'$. Así, después de ver la entrada derivada de α , expandimos A' a β_1 o a β_2 . Es decir, si se factorizan por la izquierda, las producciones originales se convierten en:

$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 \mid \beta_2 \end{aligned}$$

Algoritmo 4.21: Factorización por la izquierda de una gramática.

ENTRADA: La gramática G .

SALIDA: Una gramática equivalente factorizada por la izquierda.

MÉTODO: Para cada no terminal A , encontrar el prefijo α más largo que sea común para una o más de sus alternativas. Si $\alpha \neq \epsilon$ (es decir, si hay un prefijo común no trivial), se sustituyen todas las producciones A , $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$, en donde γ representa a todas las alternativas que no empiezan con α , mediante lo siguiente:

$$\begin{aligned} A &\rightarrow \alpha A' \mid \gamma \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{aligned}$$

Aquí, A' es un no terminal nuevo. Se aplica esta transformación en forma repetida hasta que no haya dos alternativas para un no terminal que tengan un prefijo común. \square

Ejemplo 4.22: La siguiente gramática abstrae el problema del “else colgante”:

$$\begin{aligned} S &\rightarrow i E t S \mid i E t S e S \mid a \\ E &\rightarrow b \end{aligned} \tag{4.23}$$

Aquí, i , t y e representan a **if**, **then** y **else**; E y S representan “expresión condicional” e “instrucción”. Si se factoriza a la izquierda, esta gramática se convierte en:

$$\begin{aligned} S &\rightarrow i E t S S' \mid a \\ S' &\rightarrow e S \mid \epsilon \\ E &\rightarrow b \end{aligned} \tag{4.24}$$

Así, podemos expandir S a $iEtSS'$ con la entrada i , y esperar hasta que se haya visto $iEtS$ para decidir si se va a expandir S' a eS o a ϵ . Desde luego que estas gramáticas son ambiguas, y con la entrada e no quedará claro qué alternativa debe elegirse para S' . El ejemplo 4.33 habla sobre cómo salir de este dilema. \square

4.3.5 Construcciones de lenguajes que no son libres de contexto

Algunas construcciones sintácticas que se encuentran en los lenguajes de programación ordinarios no pueden especificarse sólo mediante el uso de gramáticas. Aquí consideraremos dos de estas construcciones, usando lenguajes abstractos simples para ilustrar las dificultades.

Ejemplo 4.25: El lenguaje en este ejemplo abstrae el problema de comprobar que se declaren los identificadores antes de poder usarlos en un programa. El lenguaje consiste en cadenas de la forma wcw , en donde la primera w representa la declaración de un identificador w , c representa un fragmento intermedio del programa, y la segunda w representa el uso del identificador.

El lenguaje abstracto es $L_1 = \{wcw \mid w \text{ está en } (\mathbf{a}|\mathbf{b})^*\}$. L_1 consiste en todas las palabras compuestas de una cadena repetida de as y bs separadas por c , como $aabcaab$. Aunque no lo vamos a demostrar aquí, la característica de no ser libre de contexto de L_1 implica directamente que los lenguajes de programación como C y Java no sean libres de contexto, los cuales requieren la declaración de los identificadores antes de usarlos, además de permitir identificadores de longitud arbitraria.

Por esta razón, una gramática para C o Java no hace diferencias entre los identificadores que son cadenas distintas de caracteres. En vez de ello, todos los identificadores se representan

mediante un token como **id** en la gramática. En un compilador para dicho lenguaje, la fase de análisis semántico comprueba que los identificadores se declaren antes de usarse. \square

Ejemplo 4.26: El lenguaje, que no es independiente del contexto en este ejemplo, abstrae el problema de comprobar que el número de parámetros formales en la declaración de una función coincide con el número de parámetros actuales en un uso de la función. El lenguaje consiste en cadenas de la forma $a^n b^m c^n d^m$. (Recuerde que a^n significa a escrita n veces). Aquí, a^n y b^m podrían representar las listas de parámetros formales de dos funciones declaradas para tener n y m argumentos, respectivamente, mientras que c^n y d^m representan las listas de los parámetros actuales en las llamadas a estas dos funciones.

El lenguaje abstracto es $L_2 = \{a^n b^m c^n d^m \mid n \geq 1 \text{ y } m \geq 1\}$. Es decir, L_2 consiste de cadenas en el lenguaje generado por la expresión regular **a*b*c*d***, de tal forma que el número de *as* y *cs* y de *bs* y *ds* sea igual. Este lenguaje no es independiente del contexto.

De nuevo, la sintaxis común de las declaraciones de las funciones y los usos no se responsabiliza de contar el número de parámetros. Por ejemplo, la llamada a una función en un lenguaje similar a C podría especificarse de la siguiente manera:

$$\begin{array}{lcl} instr & \rightarrow & \mathbf{id} \ (lista_expr) \\ lista_expr & \rightarrow & lista_expr, \ expr \\ & | & expr \end{array}$$

con producciones adecuadas para *expr*. La comprobación de que el número de parámetros en una llamada sea correcto se realiza por lo general durante la fase del análisis semántico. \square

4.3.6 Ejercicios para la sección 4.3

Ejercicio 4.3.1: La siguiente gramática es para expresiones regulares sobre los símbolos *a* y *b* solamente, usando + en vez de | para la unión, con lo cual se evita el conflicto con el uso de la barra vertical como un meta símbolo en las gramáticas:

$$\begin{array}{lcl} rexpr & \rightarrow & rexpr + rterm \mid rterm \\ rterm & \rightarrow & rterm \ rfactor \mid rfactor \\ rfactor & \rightarrow & rfactor \ * \mid rprimario \\ rprimario & \rightarrow & \mathbf{a} \mid \mathbf{b} \end{array}$$

- Factorice esta gramática por la izquierda.
- ¿La factorización por la izquierda hace a la gramática adecuada para el análisis sintáctico descendente?
- Además de la factorización por la izquierda, elimine la recursividad por la izquierda de la gramática original.
- ¿La gramática resultante es adecuada para el análisis sintáctico descendente?

Ejercicio 4.3.2: Repita el ejercicio 4.3.1 con las siguientes gramáticas:

- La gramática del ejercicio 4.2.1.
- La gramática del ejercicio 4.2.2(a).

- c) La gramática del ejercicio 4.2.2(c).
- d) La gramática del ejercicio 4.2.2(e).
- e) La gramática del ejercicio 4.2.2(g).

! Ejercicio 4.3.3: Se propone la siguiente gramática para eliminar la “ambigüedad del else colgante”, descrita en la sección 4.3.2:

$$\begin{array}{lcl}
 \text{instr} & \rightarrow & \text{if } \text{expr} \text{ then } \text{instr} \\
 & | & \text{instrRelacionada} \\
 \text{instrRelacionada} & \rightarrow & \text{if } \text{expr} \text{ then } \text{instrRelacionada} \text{ else } \text{instr} \\
 & | & \text{otra}
 \end{array}$$

Muestre que esta gramática sigue siendo ambigua.

4.4 Análisis sintáctico descendente

El análisis sintáctico descendente puede verse como el problema de construir un árbol de análisis sintáctico para la cadena de entrada, partiendo desde la raíz y creando los nodos del árbol de análisis sintáctico en preorden (profundidad primero, como vimos en la sección 2.3.4). De manera equivalente, podemos considerar el análisis sintáctico descendente como la búsqueda de una derivación por la izquierda para una cadena de entrada.

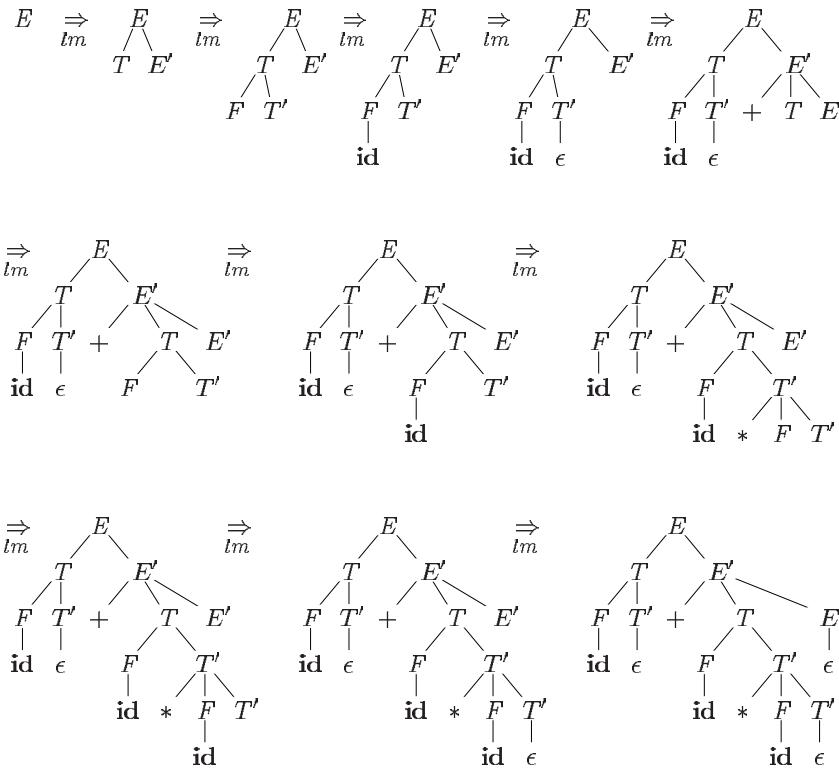
Ejemplo 4.27: La secuencia de árboles de análisis sintáctico en la figura 4.12 para la entrada **id+id*id** es un análisis sintáctico descendente, de acuerdo con la gramática (4.2), que repetimos a continuación:

$$\begin{array}{lcl}
 E & \rightarrow & T \ E' \\
 E' & \rightarrow & + \ T \ E' \ | \ \epsilon \\
 T & \rightarrow & F \ T' \\
 T' & \rightarrow & * \ F \ T' \ | \ \epsilon \\
 F & \rightarrow & (\ E) \ | \ \text{id}
 \end{array} \tag{4.28}$$

Esta secuencia de árboles corresponde a una derivación por la izquierda de la entrada. \square

En cada paso de un análisis sintáctico descendente, el problema clave es el de determinar la producción que debe aplicarse para un no terminal, por decir A . Una vez que se elige una producción A , el resto del proceso de análisis sintáctico consiste en “relacionar” los símbolos terminales en el cuerpo de la producción con la cadena de entrada.

Esta sección empieza con una forma general del análisis sintáctico descendente, conocida como análisis sintáctico de descenso recursivo, la cual puede requerir de un rastreo hacia atrás para encontrar la producción A correcta que debe aplicarse. La sección 2.4.2 introdujo el análisis sintáctico predictivo, un caso especial de análisis sintáctico de descenso recursivo, en donde no se requiere un rastreo hacia atrás. El análisis sintáctico predictivo elige la producción A correcta mediante un análisis por adelantado de la entrada, en donde se ve un número fijo de símbolos adelantados; por lo general, sólo necesitamos ver un símbolo por adelantado (es decir, el siguiente símbolo de entrada).

Figura 4.12: Análisis sintáctico descendente para **id+id*id**

Por ejemplo, considere el análisis sintáctico descendente en la figura 4.12, en la cual se construye un árbol con dos nodos etiquetados como E' . En el primer nodo E' (en preorden), se elige la producción $E' \rightarrow +TE'$; en el segundo nodo E' , se elige la producción $E' \rightarrow \epsilon$. Un analizador sintáctico predictivo puede elegir una de las producciones E' mediante el análisis del siguiente símbolo de entrada.

A la clase de gramáticas para las cuales podemos construir analizadores sintácticos predictivos que analicen k símbolos por adelantado en la entrada, se le conoce algunas veces como la clase $LL(k)$. En la sección 4.4.3 hablaremos sobre la clase $LL(1)$, pero presentaremos antes ciertos cálculos, llamados PRIMERO y SIGUIENTE, en la sección 4.4.2. A partir de los conjuntos PRIMERO y SIGUIENTE para una gramática, construiremos “tablas de análisis sintáctico predictivo”, las cuales hacen explícita la elección de la producción durante el análisis sintáctico descendente. Estos conjuntos también son útiles durante el análisis sintáctico ascendente.

En la sección 4.4.4 proporcionaremos un algoritmo de análisis sintáctico no recursivo que mantiene una pila en forma explícita, en vez de hacerlo en forma implícita mediante llamadas recursivas. Por último, en la sección 4.4.5 hablaremos sobre la recuperación de errores durante el análisis sintáctico descendente.

4.4.1 Análisis sintáctico de descenso recursivo

```

void A() {
1)      Elegir una producción  $A$ ,  $A \rightarrow X_1 X_2 \dots X_k$ ;
2)      for (  $i = 1$  a  $k$  ) {
3)          if (  $X_i$  es un no terminal )
4)              llamar al procedimiento  $X_i()$ ;
5)          else if (  $X_i$  es igual al símbolo de entrada actual  $a$  )
6)              avanzar la entrada hasta el siguiente símbolo;
7)          else /* ha ocurrido un error */;
}
}

```

Figura 4.13: Un procedimiento ordinario para un no terminal en un analizador sintáctico descendente

Un programa de análisis sintáctico de descenso recursivo consiste en un conjunto de procedimientos, uno para cada no terminal. La ejecución empieza con el procedimiento para el símbolo inicial, que se detiene y anuncia que tuvo éxito si el cuerpo de su procedimiento explora toda la cadena completa de entrada. En la figura 4.13 aparece el seudocódigo para un no terminal común. Observe que este seudocódigo es no determinista, ya que empieza eligiendo la producción A que debe aplicar de una forma no especificada.

El descenso recursivo general puede requerir de un rastreo hacia atrás; es decir, tal vez requiera exploraciones repetidas sobre la entrada. Sin embargo, raras veces se necesita el rastreo hacia atrás para analizar las construcciones de un lenguaje de programación, por lo que los analizadores sintácticos con éste no se ven con frecuencia. Incluso para situaciones como el análisis sintáctico de un lenguaje natural, el rastreo hacia atrás no es muy eficiente, por lo cual se prefieren métodos tabulares como el algoritmo de programación dinámico del ejercicio 4.4.9, o el método de Earley (vea las notas bibliográficas).

Para permitir el rastreo hacia atrás, hay que modificar el código de la figura 4.13. En primer lugar, no podemos elegir una producción A única en la línea (1), por lo que debemos probar cada una de las diversas producciones en cierto orden. Después, el fallo en la línea (7) no es definitivo, sino que sólo sugiere que necesitamos regresar a la línea (1) y probar otra producción A . Sólo si no hay más producciones A para probar es cuando declaramos que se ha encontrado un error en la entrada. Para poder probar otra producción A , debemos restablecer el apuntador de entrada a la posición en la que se encontraba cuando llegamos por primera vez a la línea (1). Es decir, se requiere una variable local para almacenar este apuntador de entrada, para un uso futuro.

Ejemplo 4.29: Considere la siguiente gramática:

$$\begin{array}{lcl}
S & \rightarrow & c A d \\
A & \rightarrow & a b \mid a
\end{array}$$

Para construir un árbol de análisis sintáctico descendente para la cadena de entrada $w = cad$, empezamos con un árbol que consiste en un solo nodo etiquetado como S , y el apuntador de entrada apunta a c , el primer símbolo de w . S sólo tiene una producción, por lo que la utilizamos

para expandir S y obtener el árbol de la figura 4.14(a). La hoja por la izquierda, etiquetada como c , coincide con el primer símbolo de la entrada w , por lo que avanzamos el apuntador de entrada hasta a , el segundo símbolo de w , y consideramos la siguiente hoja, etiquetada como A .

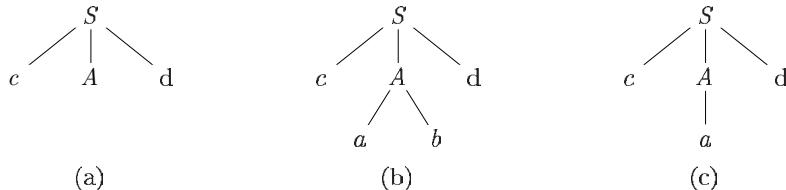


Figura 4.14: Los pasos de un análisis sintáctico descendente

Ahora expandimos A mediante la primera alternativa $A \rightarrow a b$ para obtener el árbol de la figura 4.14(b). Tenemos una coincidencia para el segundo símbolo de entrada a , por lo que avanzamos el apuntador de entrada hasta d , el tercer símbolo de entrada, y comparamos a d con la siguiente hoja, etiquetada con b . Como b no coincide con d , reportamos un error y regresamos a A para ver si hay otra alternativa para A que no hayamos probado y que pueda producir una coincidencia.

Al regresar a A , debemos restablecer el apuntador de entrada a la posición 2, la posición que tenía cuando llegamos por primera vez a A , lo cual significa que el procedimiento para A debe almacenar el apuntador de entrada en una variable local.

La segunda alternativa para A produce el árbol de la figura 4.14(c). La hoja a coincide con el segundo símbolo de w y la hoja d coincide con el tercer símbolo. Como hemos producido un árbol de análisis sintáctico para w , nos detenemos y anunciamos que se completó el análisis sintáctico con éxito. \square

Una gramática recursiva por la izquierda puede hacer que un analizador sintáctico de descenso recursivo, incluso uno con rastreo hacia atrás, entre en un ciclo infinito. Es decir, al tratar de expandir una no terminal A , podríamos en un momento dado encontrarnos tratando otra vez de expandir a A , sin haber consumido ningún símbolo de la entrada.

4.4.2 PRIMERO y SIGUIENTE

La construcción de los analizadores sintácticos descendentes y ascendentes es auxiliada por dos funciones, PRIMERO y SIGUIENTE, asociadas con la gramática G . Durante el análisis sintáctico descendente, PRIMERO y SIGUIENTE nos permiten elegir la producción que vamos a aplicar, con base en el siguiente símbolo de entrada. Durante la recuperación de errores en modo de pánico, los conjuntos de tokens que produce SIGUIENTE pueden usarse como tokens de sincronización.

Definimos a $PRIMERO(\alpha)$, en donde α es cualquier cadena de símbolos gramaticales, como el conjunto de terminales que empiezan las cadenas derivadas a partir de α . Si $\alpha \xrightarrow{lm} \epsilon$, entonces ϵ también se encuentra en $PRIMERO(\alpha)$. Por ejemplo, en la figura 4.15, $A \xrightarrow{lm} c\gamma$, por lo que c está en $PRIMERO(A)$.

Para una vista previa de cómo usar PRIMERO durante el análisis sintáctico predictivo, considere dos producciones A , $A \rightarrow \alpha \mid \beta$, en donde $PRIMERO(\alpha)$ y $PRIMERO(\beta)$ son conjuntos

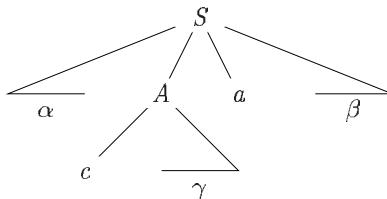


Figura 4.15: El terminal c está en $\text{PRIMERO}(A)$ y a está en $\text{SIGUIENTE}(A)$

separados. Entonces, podemos elegir una de estas producciones A si analizamos el siguiente símbolo de entrada a , ya que a puede estar a lo más en $\text{PRIMERO}(\alpha)$ o en $\text{PRIMERO}(\beta)$, pero no en ambos. Por ejemplo, si a está en $\text{PRIMERO}(\beta)$, elegimos la producción $A \rightarrow \beta$. Exploraremos esta idea en la sección 4.4.3, cuando definamos las gramáticas LL(1).

Definimos a $\text{SIGUIENTE}(A)$, para el no terminal A , como el conjunto de terminales a que pueden aparecer de inmediato a la derecha de A en cierta forma de frase; es decir, el conjunto de terminales A de tal forma que exista una derivación de la forma $S \xrightarrow{*} \alpha A a \beta$, para algunas α y β , como en la figura 4.15. Observe que pudieron haber aparecido símbolos entre A y a , en algún momento durante la derivación, pero si es así, derivaron ϵ y desaparecieron. Además, si A puede ser el símbolo por la derecha en cierta forma de frase, entonces $\$$ está en $\text{SIGUIENTE}(A)$; recuerde que $\$$ es un símbolo “delimitador” especial, el cual se supone que no es un símbolo de ninguna gramática.

Para calcular $\text{PRIMERO}(X)$ para todos los símbolos gramaticales X , aplicamos las siguientes reglas hasta que no puedan agregarse más terminales o ϵ a ningún conjunto PRIMERO .

1. Si X es un terminal, entonces $\text{PRIMERO}(X) = \{X\}$.
2. Si X es un no terminal y $X \rightarrow Y_1 Y_2 \dots Y_k$ es una producción para cierta $k \geq 1$, entonces se coloca a en $\text{PRIMERO}(X)$ si para cierta i , a está en $\text{PRIMERO}(Y_i)$, y ϵ está en todas las funciones $\text{PRIMERO}(Y_1), \dots, \text{PRIMERO}(Y_{i-1})$; es decir, $Y_1 \dots Y_{i-1} \xrightarrow{*} \epsilon$. Si ϵ está en $\text{PRIMERO}(Y_j)$ para todas las $j = 1, 2, \dots, k$, entonces se agrega ϵ a $\text{PRIMERO}(X)$. Por ejemplo, todo lo que hay en $\text{PRIMERO}(Y_1)$ se encuentra sin duda en $\text{PRIMERO}(X)$. Si Y_1 no deriva a ϵ , entonces no agregamos nada más a $\text{PRIMERO}(X)$, pero si $Y_1 \xrightarrow{*} \epsilon$, entonces agregamos $\text{PRIMERO}(Y_2)$, y así sucesivamente.
3. Si $X \rightarrow \epsilon$ es una producción, entonces se agrega ϵ a $\text{PRIMERO}(X)$.

Ahora, podemos calcular PRIMERO para cualquier cadena $X_1 X_2 \dots X_n$ de la siguiente manera. Se agregan a $\text{PRIMERO}(X_1 X_2 \dots X_n)$ todos los símbolos que no sean ϵ de $\text{PRIMERO}(X_1)$. También se agregan los símbolos que no sean ϵ de $\text{PRIMERO}(X_2)$, si ϵ está en $\text{PRIMERO}(X_1)$; los símbolos que no sean ϵ de $\text{PRIMERO}(X_3)$, si ϵ está en $\text{PRIMERO}(X_1)$ y $\text{PRIMERO}(X_2)$; y así sucesivamente. Por último, se agrega ϵ a $\text{PRIMERO}(X_1 X_2 \dots X_n)$ si, para todas las i , ϵ se encuentra en $\text{PRIMERO}(X_i)$.

Para calcular $\text{SIGUIENTE}(A)$ para todas las no terminales A , se aplican las siguientes reglas hasta que no pueda agregarse nada a cualquier conjunto SIGUIENTE .

1. Colocar $\$$ en $\text{SIGUIENTE}(S)$, en donde S es el símbolo inicial y $\$$ es el delimitador derecho de la entrada.

2. Si hay una producción $A \rightarrow \alpha B \beta$, entonces todo lo que hay en $\text{PRIMERO}(\beta)$ excepto ϵ está en $\text{SIGUIENTE}(B)$.
3. Si hay una producción $A \rightarrow \alpha B$, o una producción $A \rightarrow \alpha B \beta$, en donde $\text{PRIMERO}(\beta)$ contiene a ϵ , entonces todo lo que hay en $\text{SIGUIENTE}(A)$ está en $\text{SIGUIENTE}(B)$.

Ejemplo 4.30: Considere de nuevo la gramática no recursiva por la izquierda (4.28). Entonces:

1. $\text{PRIMERO}(F) = \text{PRIMERO}(T) = \text{PRIMERO}(E) = \{(), \mathbf{id}\}$. Para ver por qué, observe que las dos producciones para F tienen producciones que empiezan con estos dos símbolos terminales, **id** y el paréntesis izquierdo. T sólo tiene una producción y empieza con F . Como F no deriva a ϵ , $\text{PRIMERO}(T)$ debe ser igual que $\text{PRIMERO}(F)$. El mismo argumento se cumple con $\text{PRIMERO}(E)$.
2. $\text{PRIMERO}(E') = \{+, \epsilon\}$. La razón es que una de las dos producciones para E' tiene un cuerpo que empieza con el terminal $+$, y la otra es ϵ . Cada vez que un no terminal deriva a ϵ , colocamos a ϵ en PRIMERO para ese no terminal.
3. $\text{PRIMERO}(T') = \{*, \epsilon\}$. El razonamiento es análogo al de $\text{PRIMERO}(E')$.
4. $\text{SIGUIENTE}(E) = \text{SIGUIENTE}(E') = \{(), \$\}$. Como E es el símbolo inicial, $\text{SIGUIENTE}(E)$ debe contener $\$$. El cuerpo de la producción (E) explica por qué el paréntesis derecho está en $\text{SIGUIENTE}(E)$. Para E' , observe que esta no terminal sólo aparece en los extremos de los cuerpos de las producciones E . Por ende, $\text{SIGUIENTE}(E')$ debe ser el mismo que $\text{SIGUIENTE}(E)$.
5. $\text{SIGUIENTE}(T) = \text{SIGUIENTE}(T') = \{+, (), \$\}$. Observe que T aparece en las producciones sólo seguido por E' . Por lo tanto, todo lo que esté en $\text{PRIMERO}(E')$, excepto ϵ , debe estar en $\text{SIGUIENTE}(T)$; eso explica el símbolo $+$. No obstante, como $\text{PRIMERO}(E')$ contiene a ϵ (es decir, $E' \xrightarrow{*} \epsilon$), y E' es la cadena completa que va después de T en los cuerpos de las producciones E , todo lo que hay en $\text{SIGUIENTE}(E)$ también debe estar en $\text{SIGUIENTE}(T)$. Eso explica los símbolos $\$$ y el paréntesis derecho. En cuanto a T' , como aparece sólo en los extremos de las producciones T , debe ser que $\text{SIGUIENTE}(T') = \text{SIGUIENTE}(T)$.
6. $\text{SIGUIENTE}(F) = \{+, *, (), \$\}$. El razonamiento es análogo al de T en el punto (5).

□

4.4.3 Gramáticas LL(1)

Los analizadores sintácticos predictivos, es decir, los analizadores sintácticos de descenso recursivo que no necesitan rastreo hacia atrás, pueden construirse para una clase de gramáticas llamadas LL(1). La primera “L” en LL(1) es para explorar la entrada de izquierda a derecha (por left en inglés), la segunda “L” para producir una derivación por la izquierda, y el “1” para usar un símbolo de entrada de anticipación en cada paso, para tomar las decisiones de acción del análisis sintáctico.

Diagramas de transición para analizadores sintácticos predictivos

Los diagramas de transición son útiles para visualizar los analizadores sintácticos predictivos. Por ejemplo, los diagramas de transición para las no terminales E y E' de la gramática (4.28) aparecen en la figura 4.16(a). Para construir el diagrama de transición a partir de una gramática, primero hay que eliminar la recursividad por la izquierda y después factorizar la gramática por la izquierda. Entonces, para cada no terminal A ,

1. Se crea un estado inicial y un estado final (retorno).
2. Para cada producción $A \rightarrow X_1 X_2 \dots X_k$, se crea una ruta desde el estado inicial hasta el estado final, con los flancos etiquetados como X_1, X_2, \dots, X_k . Si $A \rightarrow \epsilon$, la ruta es una línea que se etiqueta como ϵ .

Los diagramas de transición para los analizadores sintácticos predictivos difieren de los diagramas para los analizadores léxicos. Los analizadores sintácticos tienen un diagrama para cada no terminal. Las etiquetas de las líneas pueden ser tokens o no terminales. Una transición sobre un token (terminal) significa que tomamos esa transición si ese token es el siguiente símbolo de entrada. Una transición sobre un no terminal A es una llamada al procedimiento para A .

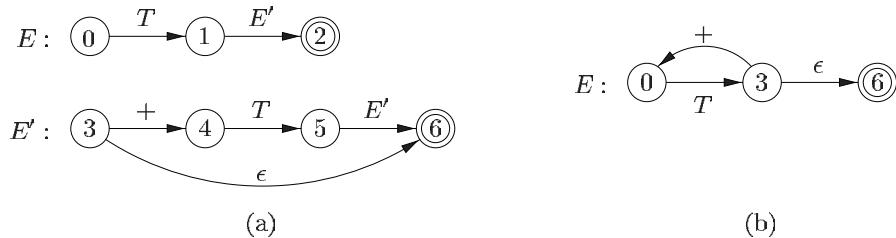
Con una gramática LL(1), la ambigüedad acerca de si se debe tomar o no una línea ϵ puede resolverse si hacemos que las transiciones ϵ sean la opción predeterminada.

Los diagramas de transición pueden simplificarse, siempre y cuando se preserve la secuencia de símbolos gramaticales a lo largo de las rutas. También podemos sustituir el diagrama para un no terminal A , en vez de una línea etiquetada como A . Los diagramas en las figuras 4.16(a) y (b) son equivalentes: si trazamos rutas de E a un estado de aceptación y lo sustituimos por E' , entonces, en ambos conjuntos de diagramas, los símbolos gramaticales a lo largo de las rutas forman cadenas del tipo $T + T + \dots + T$. El diagrama en (b) puede obtenerse a partir de (a) mediante transformaciones semejantes a las de la sección 2.5.4, en donde utilizamos la eliminación de recursividad de la parte final y la sustitución de los cuerpos de los procedimientos para optimizar el procedimiento en un no terminal.

La clase de gramáticas LL(1) es lo bastante robusta como para cubrir la mayoría de las construcciones de programación, aunque hay que tener cuidado al escribir una gramática adecuada para el lenguaje fuente. Por ejemplo, ninguna gramática recursiva por la izquierda o ambigua puede ser LL(1).

Una gramática G es LL(1) si, y sólo si cada vez que $A \rightarrow \alpha \mid \beta$ son dos producciones distintas de G , se aplican las siguientes condiciones:

1. Para el no terminal a , tanto α como β derivan cadenas que empiecen con a .
2. A lo más, sólo α o β puede derivar la cadena vacía.
3. Si $\beta \xrightarrow{*} \epsilon$, entonces α no deriva a ninguna cadena que empiece con una terminal en SIGUIENTE(A). De igual forma, si $\alpha \xrightarrow{*} \epsilon$, entonces β no deriva a ninguna cadena que empiece con una terminal en SIGUIENTE(A).

Figura 4.16: Diagramas de transición para los no terminales E y E' de la gramática 4.28

Las primeras dos condiciones son equivalentes para la instrucción que establece que $\text{PRIMERO}(\alpha)$ y $\text{PRIMERO}(\beta)$ son conjuntos separados. La tercera condición equivale a decir que si ϵ está en $\text{PRIMERO}(\beta)$, entonces $\text{PRIMERO}(\alpha)$ y $\text{SIGUIENTE}(\beta)$ son conjuntos separados, y de igual forma si ϵ está en $\text{PRIMERO}(\alpha)$.

Pueden construirse analizadores sintácticos predictivos para las gramáticas LL(1), ya que puede seleccionarse la producción apropiada a aplicar para una no terminal con sólo analizar el símbolo de entrada actual. Los constructores del flujo de control, con sus palabras clave distintivas, por lo general, cumplen con las restricciones de LL(1). Por ejemplo, si tenemos las siguientes producciones:

$$\begin{array}{lcl} instr & \rightarrow & \text{if} (\ expr) \ instr \ \text{else} \ instr \\ & | & \text{while} (\ expr) \ instr \\ & | & \{ \ lista_instr \ \} \end{array}$$

entonces las palabras clave **if**, **while** y el símbolo **{** nos indican qué alternativa es la única que quizá podría tener éxito, si vamos a buscar una instrucción.

El siguiente algoritmo recolecta la información de los conjuntos PRIMERO y SIGUIENTE en una tabla de análisis predictivo $M[A, a]$, un arreglo bidimensional, en donde A es un no terminal y a es un terminal o el símbolo $\$$, el marcador de fin de la entrada. El algoritmo se basa en la siguiente idea: se elige la producción $A \rightarrow \alpha$ si el siguiente símbolo de entrada a se encuentra en $\text{PRIMERO}(\alpha)$. La única complicación ocurre cuando $\alpha = \epsilon$, o en forma más general, $\alpha \xrightarrow{*} \epsilon$. En este caso, debemos elegir de nuevo $A \rightarrow \alpha$ si el símbolo de entrada actual se encuentra en $\text{SIGUIENTE}(A)$, o si hemos llegado al $\$$ en la entrada y $\$$ se encuentra en $\text{SIGUIENTE}(A)$.

Algoritmo 4.31: Construcción de una tabla de análisis sintáctico predictivo.

ENTRADA: La gramática G .

SALIDA: La tabla de análisis sintáctico M .

MÉTODO: Para cada producción $A \rightarrow \alpha$ de la gramática, hacer lo siguiente:

1. Para cada terminal a en $\text{PRIMERO}(A)$, agregar $A \rightarrow \alpha$ a $M[A, a]$.
2. Si ϵ está en $\text{PRIMERO}(\alpha)$, entonces para cada terminal b en $\text{SIGUIENTE}(A)$, se agrega $A \rightarrow \alpha$ a $M[A, b]$. Si ϵ está en $\text{PRIMERO}(\alpha)$ y $\$$ se encuentra en $\text{SIGUIENTE}(A)$, se agrega $A \rightarrow \alpha$ a $M[A, \$]$ también.

Si después de realizar lo anterior, no hay producción en $M[A, a]$, entonces se establece $M[A, a]$ a **error** (que, por lo general, representamos mediante una entrada vacía en la tabla). \square

Ejemplo 4.32: Para la gramática de expresiones (4.28), el Algoritmo 4.31 produce la tabla de análisis sintáctico en la figura 4.17. Los espacios en blanco son entradas de error; los espacios que no están en blanco indican una producción con la cual se expande un no terminal.

NO TERMINAL	SÍMBOLO DE ENTRADA					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T		$T \rightarrow FT'$		$T \rightarrow FT'$		
T'			$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$	$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

Figura 4.17: Tabla de análisis sintáctico M para el ejemplo 4.32

Considere la producción $E \rightarrow TE'$. Como

$$\text{PRIMERO}(TE') = \text{PRIMERO}(T) = \{(), \text{id}\}$$

esta producción se agrega a $M[E, ()]$ y $M[E, \text{id}]$. La producción $E' \rightarrow +TE'$ se agrega a $M[E', +]$, ya que $\text{PRIMERO}(+TE') = \{+\}$. Como $\text{SIGUIENTE}(E') = \{(), \$\}$, la producción $E' \rightarrow \epsilon$ se agrega a $M[E', ()]$ y a $M[E', \$]$. \square

El algoritmo 4.31 puede aplicarse a cualquier gramática G para producir una tabla de análisis sintáctico M . Para cada gramática LL(1), cada entrada en la tabla de análisis sintáctico identifica en forma única a una producción, o indica un error. Sin embargo, para algunas gramáticas, M puede tener algunas entradas que tengan múltiples definiciones. Por ejemplo, si G es recursiva por la izquierda o ambigua, entonces M tendrá por lo menos una entrada con múltiples definiciones. Aunque la eliminación de la recursividad por la izquierda y la factorización por la izquierda son fáciles de realizar, hay algunas gramáticas para las cuales ningún tipo de alteración producirá una gramática LL(1).

El lenguaje en el siguiente ejemplo no tiene una gramática LL(1).

Ejemplo 4.33: La siguiente gramática, que abstrae el problema del else colgante, se repite aquí del ejemplo 4.22:

$$\begin{aligned} S &\rightarrow iEtSS' \mid a \\ S' &\rightarrow eS \mid \epsilon \\ E &\rightarrow b \end{aligned}$$

La tabla de análisis sintáctico para esta gramática aparece en la figura 4.18. La entrada para $M[S', e]$ contiene tanto a $S' \rightarrow eS$ como a $S' \rightarrow \epsilon$.

La gramática es ambigua y la ambigüedad se manifiesta mediante una elección de qué producción usar cuando se ve una e (**else**). Podemos resolver esta ambigüedad eligiendo $S' \rightarrow eS$.

No TERMINAL	SÍMBOLO DE ENTRADA					
	<i>a</i>	<i>b</i>	<i>e</i>	<i>i</i>	<i>t</i>	$\$$
<i>S</i>	$S \rightarrow a$			$S \rightarrow iEtSS'$		
<i>S'</i>			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
<i>E</i>		$E \rightarrow b$				

Figura 4.18: Tabla de análisis sintáctico M para el ejemplo 4.33

Esta elección corresponde a la asociación de un **else** con el **then** anterior más cercano. Observe que la elección $S' \rightarrow \epsilon$ evitaría que *e* se metiera en la pila o se eliminara de la entrada, y eso definitivamente está mal. \square

4.4.4 Análisis sintáctico predictivo no recursivo

Podemos construir un analizador sintáctico predictivo no recursivo mediante el mantenimiento explícito de una pila, en vez de hacerlo mediante llamadas recursivas implícitas. El analizador sintáctico imita una derivación por la izquierda. Si *w* es la entrada que se ha relacionado hasta ahora, entonces la pila contiene una secuencia de símbolos gramaticales α de tal forma que:

$$S \xrightarrow[lm]{*} w\alpha$$

El analizador sintáctico controlado por una tabla, que se muestra en la figura 4.19, tiene un búfer de entrada, una pila que contiene una secuencia de símbolos gramaticales, una tabla de análisis sintáctico construida por el Algoritmo 4.31, y un flujo de salida. El búfer de entrada contiene la cadena que se va a analizar, seguida por el marcador final $\$$. Reutilizamos el símbolo $\$$ para marcar la parte inferior de la pila, que al principio contiene el símbolo inicial de la gramática encima de $\$$.

El analizador sintáctico se controla mediante un programa que considera a X , el símbolo en la parte superior de la pila, y a a , el símbolo de entrada actual. Si X es un no terminal, el analizador sintáctico elige una producción X mediante una consulta a la entrada $M[X, a]$ de la tabla de análisis sintáctico M (aquí podría ejecutarse código adicional; por ejemplo, el código para construir un nodo en un árbol de análisis sintáctico). En cualquier otro caso, verifica si hay una coincidencia entre el terminal X y el símbolo de entrada actual a .

El comportamiento del analizador sintáctico puede describirse en términos de sus *configuraciones*, que proporcionan el contenido de la pila y el resto de la entrada. El siguiente algoritmo describe la forma en que se manipulan las configuraciones.

Algoritmo 4.34: Análisis sintáctico predictivo, controlado por una tabla.

ENTRADA: Una cadena *w* y una tabla de análisis sintáctico M para la gramática G .

SALIDA: Si *w* está en $L(G)$, una derivación por la izquierda de *w*; en caso contrario, una indicación de error.

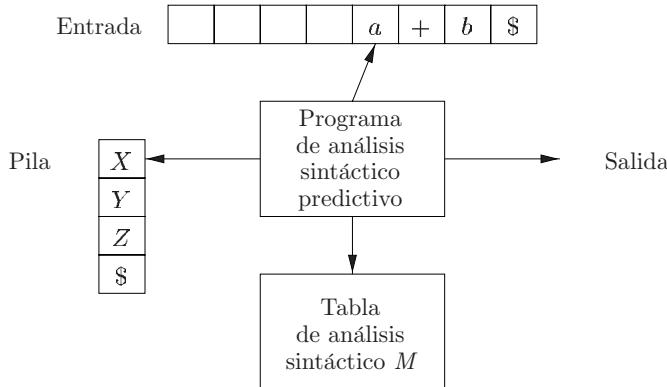


Figura 4.19: Modelo de un analizador sintáctico predictivo, controlado por una tabla

MÉTODO: Al principio, el analizador sintáctico se encuentra en una configuración con $w\$$ en el búfer de entrada, y el símbolo inicial S de G en la parte superior de la pila, por encima de $\$$. El programa en la figura 4.20 utiliza la tabla de análisis sintáctico predictivo M para producir un análisis sintáctico predictivo para la entrada. \square

```

establecer  $ip$  para que apunte al primer símbolo de  $w$ ;
establecer  $X$  con el símbolo de la parte superior de la pila;
while (  $X \neq \$$  ) { /* la pila no está vacía */
    if (  $X$  es  $a$  ) sacar de la pila y avanzar  $ip$ ;
    else if (  $X$  es un terminal ) error();
    else if (  $M[X, a]$  es una entrada de error ) error();
    else if (  $M[X, a] = X \rightarrow Y_1 Y_2 \dots Y_k$  ) {
        enviar de salida la producción  $X \rightarrow Y_1 Y_2 \dots Y_k$ ;
        sacar de la pila;
        meter  $Y_k, Y_{k-1}, \dots, Y_1$  en la pila, con  $Y_1$  en la parte superior;
    }
    establecer  $X$  con el símbolo de la parte superior de la pila;
}

```

Figura 4.20: Algoritmo de análisis sintáctico predictivo

Ejemplo 4.35: Considere la gramática (4.28); se muestra la tabla de análisis sintáctico en la figura 4.17. Con la entrada **id** + **id** * **id**, el analizador predictivo sin recursividad del Algoritmo 4.34 realiza la secuencia de movimientos en la figura 4.21. Estos movimientos corresponden a una derivación por la izquierda (vea la figura 4.12 para la derivación completa):

$$E \xrightarrow{lm} TE' \xrightarrow{lm} FT'E' \xrightarrow{lm} \mathbf{id} \ T'E' \xrightarrow{lm} \mathbf{id} \ E' \xrightarrow{lm} \mathbf{id} + \ TE' \xrightarrow{lm} \dots$$

COINCIDENCIA	PILA	ENTRADA	ACCIÓN
	$E\$$	$id + id * id\$$	
	$TE\$$	$id + id * id\$$	emitir $E \rightarrow TE'$
	$FT'E\$$	$id + id * id\$$	emitir $T \rightarrow FT'$
id	$T'E\$$	$id + id * id\$$	emitir $F \rightarrow id$
id	$E\$$	$+ id * id\$$	relacionar id
id	$+ TE\$$	$+ id * id\$$	emitir $T' \rightarrow \epsilon$
$id +$	$TE\$$	$id * id\$$	emitir $E' \rightarrow + TE'$
$id +$	$FT'E\$$	$id * id\$$	relacionar $+$
$id +$	$id T'E\$$	$id * id\$$	emitir $T \rightarrow FT'$
$id + id$	$T'E\$$	$* id\$$	emitir $F \rightarrow id$
$id + id$	$* FT'E\$$	$* id\$$	relacionar $*$
$id + id *$	$FT'E\$$	$id\$$	emitir $F \rightarrow id$
$id + id *$	$id T'E\$$	$id\$$	relacionar id
$id + id * id$	$T'E\$$	$\$$	emitir $T' \rightarrow \epsilon$
$id + id * id$	$E\$$	$\$$	emitir $E' \rightarrow \epsilon$
$id + id * id$	$\$$	$\$$	

Figura 4.21: Movimientos que realiza un analizador sintáctico predictivo con la entrada $id + id * id$

Observe que las formas de frases en esta derivación corresponden a la entrada que ya se ha relacionado (en la columna COINCIDENCIA), seguida del contenido de la pila. La entrada relacionada se muestra sólo para resaltar la correspondencia. Por la misma razón, la parte superior de la pila está a la izquierda; cuando consideremos el análisis sintáctico ascendente, será más natural mostrar la parte superior de la pila a la derecha. El apuntador de la entrada apunta al símbolo por la izquierda de la cadena en la columna ENTRADA. \square

4.4.5 Recuperación de errores en el análisis sintáctico predictivo

Esta discusión sobre la recuperación de errores se refiere a la pila de un analizador sintáctico predictivo controlado por una tabla, ya que hace explícitas los terminales y no terminales que el analizador sintáctico espera relacionar con el resto de la entrada; las técnicas también pueden usarse con el análisis sintáctico de descenso recursivo.

Durante el análisis sintáctico predictivo, un error se detecta cuando el terminal en la parte superior de la pila no coincide con el siguiente símbolo de entrada, o cuando el no terminal A se encuentra en la parte superior de la pila, a es el siguiente símbolo de entrada y $M[A, a]$ es **error** (es decir, la entrada en la tabla de análisis sintáctico está vacía).

Modo de pánico

La recuperación de errores en modo de pánico se basa en la idea de omitir símbolos en la entrada hasta que aparezca un token en un conjunto seleccionado de tokens de sincronización. Su

efectividad depende de la elección del conjunto de sincronización. Los conjuntos deben elegirse de forma que el analizador sintáctico se recupere con rapidez de los errores que tengan una buena probabilidad de ocurrir en la práctica. Algunas heurísticas son:

1. Como punto inicial, colocar todos los símbolos que están en $SIGUIENTE(A)$ en el conjunto de sincronización para el no terminal A . Si omitimos tokens hasta que se vea un elemento de $SEGUIMENTO(A)$ y sacamos a A de la pila, es probable que el análisis sintáctico pueda continuar.
2. No basta con usar $SIGUIENTE(A)$ como el conjunto de sincronización para A . Por ejemplo, si los signos de punto y coma terminan las instrucciones, como en C, entonces las palabras reservadas que empiezan las instrucciones no pueden aparecer en el conjunto $SIGUIENTE$ del no terminal que representa a las expresiones. Un punto y coma faltante después de una asignación puede, por lo tanto, ocasionar que se omita la palabra reservada que empieza la siguiente instrucción. A menudo hay una estructura jerárquica en las construcciones en un lenguaje; por ejemplo, las expresiones aparecen dentro de las instrucciones, las cuales aparecen dentro de bloques, y así sucesivamente. Al conjunto de sincronización de una construcción de bajo nivel podemos agregar los símbolos que empiezan las construcciones de un nivel más alto. Por ejemplo, podríamos agregar palabras clave que empiezan las instrucciones a los conjuntos de sincronización para los no terminales que generan las expresiones.
3. Si agregamos los símbolos en $PRIMERO(A)$ al conjunto de sincronización para el no terminal A , entonces puede ser posible continuar con el análisis sintáctico de acuerdo con A , si en la entrada aparece un símbolo que se encuentre en $PRIMERO(A)$.
4. Si un no terminal puede generar la cadena vacía, entonces la producción que deriva a ϵ puede usarse como predeterminada. Al hacer esto se puede posponer cierta detección de errores, pero no se puede provocar la omisión de un error. Este método reduce el número de terminales que hay que considerar durante la recuperación de errores.
5. Si un terminal en la parte superior de la pila no se puede relacionar, una idea simple es sacar el terminal, emitir un mensaje que diga que se insertó el terminal, y continuar con el análisis sintáctico. En efecto, este método requiere que el conjunto de sincronización de un token consista de todos los demás tokens.

Ejemplo 4.36: El uso de los símbolos en $PRIMERO$ y $SIGUIENTE$ como tokens de sincronización funciona razonablemente bien cuando las expresiones se analizan de acuerdo con la gramática usual (4.28). La tabla de análisis sintáctico para esta gramática de la figura 4.17 se repite en la figura 4.22, en donde “sinc” indica los tokens de sincronización obtenidos del conjunto $SIGUIENTE$ de la no terminal en cuestión. Los conjuntos $SIGUIENTE$ para las no terminales se obtienen del ejemplo 4.30.

La tabla en la figura 4.22 debe usarse de la siguiente forma. Si el analizador sintáctico busca la entrada $M[A, a]$ y descubre que está en blanco, entonces se omite el símbolo de entrada a . Si la entrada es “sinc”, entonces se saca el no terminal que está en la parte superior de la pila, en un intento por continuar con el análisis sintáctico. Si un token en la parte superior de la pila no coincide con el símbolo de entrada, entonces sacamos el token de la pila, como dijimos antes.

No TERMINAL	SÍMBOLO DE ENTRADA					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	sinc	sinc
E'		$E \rightarrow +TE'$			$E \rightarrow \epsilon$	$E \rightarrow \epsilon$
T	$T \rightarrow FT'$	sinc		$T \rightarrow FT'$	sinc	sinc
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$	sinc	sinc	$F \rightarrow (E)$	sinc	sinc

Figura 4.22: Tokens de sincronización agregados a la tabla de análisis sintáctico de la figura 4.17

Con la entrada errónea $\mathbf{id} * + \mathbf{id}$, el analizador sintáctico y el mecanismo de recuperación de errores de la figura 4.22 se comporta como en la figura 4.23. \square

PILA	ENTRADA	COMENTARIO
$E \$$	$) \mathbf{id} * + \mathbf{id} \$$	error, omitir $)$
$E \$$	$\mathbf{id} * + \mathbf{id} \$$	\mathbf{id} está en PRIMERO(E)
$TE' \$$	$\mathbf{id} * + \mathbf{id} \$$	
$FT'E' \$$	$\mathbf{id} * + \mathbf{id} \$$	
$\mathbf{id} T'E' \$$	$\mathbf{id} * + \mathbf{id} \$$	
$T'E' \$$	$* + \mathbf{id} \$$	
$* FT'E' \$$	$* + \mathbf{id} \$$	
$FT'E' \$$	$+ \mathbf{id} \$$	error, $M[F, +] = \text{sinc}$
$T'E' \$$	$+ \mathbf{id} \$$	Se sacó F
$E' \$$	$+ \mathbf{id} \$$	
$+ TE' \$$	$+ \mathbf{id} \$$	
$TE' \$$	$\mathbf{id} \$$	
$FT'E' \$$	$\mathbf{id} \$$	
$\mathbf{id} T'E' \$$	$\mathbf{id} \$$	
$T'E' \$$	$\$$	
$E' \$$	$\$$	
$\$$	$\$$	

Figura 4.23: Movimientos de análisis sintáctico y recuperación de errores realizados por un analizador sintáctico predictivo

La discusión anterior sobre la recuperación en modo de pánico no señala el punto importante relacionado con los mensajes de error. El diseñador del compilador debe proporcionar mensajes de error informativos que no sólo describan el error, sino que también llamen la atención hacia el lugar en donde se descubrió el error.

Recuperación a nivel de frase

La recuperación de errores a nivel de frase se implementa llenando las entradas en blanco en la tabla de análisis sintáctico predictivo con apuntadores a rutinas de error. Estas rutinas pueden modificar, insertar o eliminar símbolos en la entrada y emitir mensajes de error apropiados. También pueden sacar de la pila. La alteración de los símbolos de la pila o el proceso de meter nuevos símbolos a la pila es cuestionable por dos razones. En primer lugar, los pasos que realiza el analizador sintáctico podrían entonces no corresponder a la derivación de ninguna palabra en el lenguaje. En segundo lugar, debemos asegurarnos de que no haya posibilidad de un ciclo infinito. Verificar que cualquier acción de recuperación ocasione en un momento dado que se consuma un símbolo de entrada (o que se reduzca la pila si se ha llegado al fin de la entrada) es una buena forma de protegerse contra tales ciclos.

4.4.6 Ejercicios para la sección 4.4

Ejercicio 4.4.1: Para cada una de las siguientes gramáticas, idee analizadores sintácticos predictivos y muestre las tablas de análisis sintáctico. Puede factorizar por la izquierda o eliminar la recursividad por la izquierda de sus gramáticas primero.

- a) La gramática del ejercicio 4.2.2(a).
- b) La gramática del ejercicio 4.2.2(b).
- c) La gramática del ejercicio 4.2.2(c).
- d) La gramática del ejercicio 4.2.2(d).
- e) La gramática del ejercicio 4.2.2(e).
- f) La gramática del ejercicio 4.2.2(g).

!! **Ejercicio 4.4.2:** ¿Es posible, mediante la modificación de la gramática en cualquier forma, construir un analizador sintáctico predictivo para el lenguaje del ejercicio 4.2.1 (expresiones postfijo con el operando a)?

Ejercicio 4.4.3: Calcule PRIMERO y SIGUIENTE para la gramática del ejercicio 4.2.1.

Ejercicio 4.4.4: Calcule PRIMERO y SIGUIENTE para cada una de las gramáticas del ejercicio 4.2.2.

Ejercicio 4.4.5: La gramática $S \rightarrow a \ S \ a \mid a \ a$ genera todas las cadenas de longitud uniforme de as . Podemos idear un analizador sintáctico de descenso recursivo con rastreo hacia atrás para esta gramática. Si elegimos expandir mediante la producción $S \rightarrow a \ a$ primero, entonces sólo debemos reconocer la cadena aa . Por ende, cualquier analizador sintáctico de descenso recursivo razonable probará $S \rightarrow a \ S \ a$ primero.

- a) Muestre que este analizador sintáctico de descenso recursivo reconoce las entradas aa , $aaaa$ y $aaaaaaaa$, pero no $aaaaaa$.
- !! b) ¿Qué lenguaje reconoce este analizador sintáctico de descenso recursivo?

Los siguientes ejercicios son pasos útiles en la construcción de una gramática en la “Forma Normal de Chomsky” a partir de gramáticas arbitrarias, como se define en el ejercicio 4.4.8.

! Ejercicio 4.4.6: Una gramática es *libre de ϵ* si ningún cuerpo de las producciones es ϵ (a lo cual se le llama *producción ϵ*).

- Proporcione un algoritmo para convertir cualquier gramática en una gramática libre de ϵ que genere el mismo lenguaje (con la posible excepción de la cadena vacía; ninguna gramática libre de ϵ puede generar a ϵ).
- Aplique su algoritmo a la gramática $S \rightarrow aSbS \mid bSaS \mid \epsilon$. *Sugerencia:* Primero busque todas las no terminales que sean *anulables*, lo cual significa que generan a ϵ , tal vez mediante una derivación extensa.

! Ejercicio 4.4.7: Una *producción simple* es una producción cuyo cuerpo es una sola no terminal; por ejemplo, una producción de la forma $A \rightarrow A$.

- Proporcione un algoritmo para convertir cualquier gramática en una gramática libre de ϵ , sin producciones simples, que genere el mismo lenguaje (con la posible excepción de la cadena vacía) *Sugerencia:* Primero elimine las producciones ϵ y después averigüe para qué pares de no terminales A y B se cumple que $A \xrightarrow{*} B$ mediante una secuencia de producciones simples.
- Aplice su algoritmo a la gramática (4.1) en la sección 4.1.2.
- Muestre que, como consecuencia de la parte (a), podemos convertir una gramática en una gramática equivalente que no tenga *ciclos* (derivaciones de uno o más pasos, en los que $A \xrightarrow{*} A$ para cierta no terminal A).

!! Ejercicio 4.4.8: Se dice que una gramática está en *Forma Normal de Chomsky* (FNC) si toda producción es de la forma $A \rightarrow BC$ o de la forma $A \rightarrow a$, en donde A , B y C son no terminales, y a es un terminal. Muestre cómo convertir cualquier gramática en una gramática FNC para el mismo lenguaje (con la posible excepción de la cadena vacía; ninguna gramática FNC puede generar a ϵ).

! Ejercicio 4.4.9: Todo lenguaje que tiene una gramática libre de contexto puede reconocerse en un tiempo máximo de $O(n^3)$ para las cadenas de longitud n . Una manera simple de hacerlo, conocida como el algoritmo de *Cocke-Younger-Kasami* (o CYK), se basa en la programación dinámica. Es decir, dada una cadena $a_1a_2 \dots a_n$, construimos una tabla T de n por n de tal forma que T_{ij} sea el conjunto de no terminales que generen la subcadena $a_ia_{i+1} \dots a_j$. Si la gramática subyacente está en FNC (vea el ejercicio 4.4.8), entonces una entrada en la tabla puede llenarse en un tiempo $O(n)$, siempre y cuando llenemos las entradas en el orden apropiado: el menor valor de $j - i$ primero. Escriba un algoritmo que llene en forma correcta las entradas de la tabla, y muestre que su algoritmo requiere un tiempo $O(n^3)$. Después de llenar la tabla, ¿cómo podemos determinar si $a_1a_2 \dots a_n$ está en el lenguaje?

! Ejercicio 4.4.10: Muestre cómo, después de llenar la tabla como en el ejercicio 4.4.9, podemos recuperar en un tiempo $O(n)$ un árbol de análisis sintáctico para $a_1 a_2 \cdots a_n$. *Sugerencia:* Modifique la tabla de manera que registre, para cada no terminal A en cada entrada de la tabla T_{ij} , cierto par de no terminales en otras entradas en la tabla que justifiquen la acción de colocar a A en T_{ij} .

! Ejercicio 4.4.11: Modifique su algoritmo del ejercicio 4.4.9 de manera que busque, para cualquier cadena, el menor número de errores de inserción, eliminación y mutación (cada error de un solo carácter) necesarios para convertir la cadena en una cadena del lenguaje de la gramática subyacente.

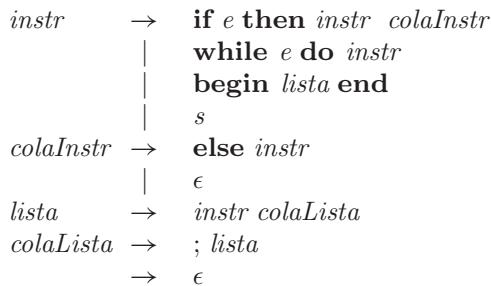


Figura 4.24: Una gramática para ciertos tipos de instrucciones

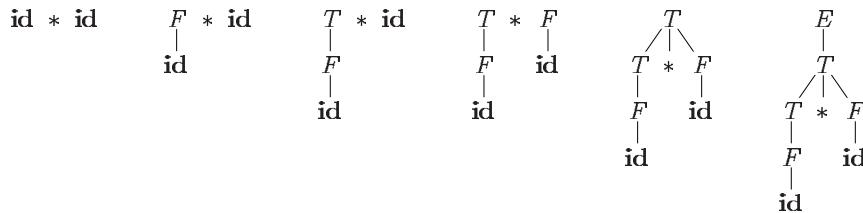
! Ejercicio 4.4.12: En la figura 4.24 hay una gramática para ciertas instrucciones. Podemos considerar que e y s son terminales que representan expresiones condicionales y “otras instrucciones”, respectivamente. Si resolvemos el conflicto en relación con la expansión de la instrucción “else” opcional (la no terminal $colaInstr$) al preferir consumir un **else** de la entrada cada vez que veamos uno, podemos construir un analizador sintáctico predictivo para esta gramática. Usando la idea de los símbolos de sincronización descritos en la sección 4.4.5:

- Construya una tabla de análisis sintáctico predictivo con corrección de errores para la gramática.
- Muestre el comportamiento de su analizador sintáctico con las siguientes entradas:

(i) **if** e **then** s ; **if** e **then** s **end**
(ii) **while** e **do** **begin** s ; **if** e **then** s ; **end**

4.5 Análisis sintáctico ascendente

Un análisis sintáctico ascendente corresponde a la construcción de un árbol de análisis sintáctico para una cadena de entrada que empieza en las hojas (la parte inferior) y avanza hacia la raíz (la parte superior). Es conveniente describir el análisis sintáctico como el proceso de construcción de árboles de análisis sintáctico, aunque de hecho un front-end de usuario podría realizar una traducción directamente, sin necesidad de construir un árbol explícito. La secuencia

Figura 4.25: Un análisis sintáctico ascendente para **id * id**

de imágenes de árboles en la figura 4.25 ilustra un análisis sintáctico ascendente del flujo de tokens **id * id**, con respecto a la gramática de expresiones (4.1).

Esta sección presenta un estilo general de análisis sintáctico ascendente, conocido como análisis sintáctico de desplazamiento-reducción. En las secciones 4.6 y 4.7 hablaremos sobre las gramáticas LR, la clase más extensa de gramáticas para las cuales pueden construirse los analizadores sintácticos de desplazamiento-reducción. Aunque es demasiado trabajo construir un analizador sintáctico LR en forma manual, las herramientas conocidas como generadores automáticos de analizadores sintácticos facilitan la construcción de analizadores sintácticos LR eficientes a partir de gramáticas adecuadas. Los conceptos en esta sección son útiles para escribir gramáticas adecuadas que nos permitan hacer un uso efectivo de un generador de analizadores sintácticos LR. En la sección 4.7 aparecen los algoritmos para implementar los generadores de analizadores sintácticos.

4.5.1 Reducciones

Podemos considerar el análisis sintáctico ascendente como el proceso de “reducir” una cadena w al símbolo inicial de la gramática. En cada paso de *reducción*, se sustituye una subcadena específica que coincide con el cuerpo de una producción por el no terminal que se encuentra en el encabezado de esa producción.

Las decisiones clave durante el análisis sintáctico ascendente son acerca de cuándo reducir y qué producción aplicar, a medida que procede el análisis sintáctico.

Ejemplo 4.37: Las imágenes en la figura 4.25 ilustran una secuencia de reducciones; la gramática es la gramática de expresiones (4.1). Hablaremos sobre las reducciones en términos de la siguiente secuencia de cadenas:

id * id, F * id, T * id, T * F, T, E

Las cadenas en esta secuencia se forman a partir de las raíces de todos los subárboles de las imágenes. La secuencia empieza con la cadena de entrada **id * id**. La primera reducción produce **F * id** al reducir el **id** por la izquierda a **F**, usando la producción $F \rightarrow \text{id}$. La segunda reducción produce **T * id** al reducir **F** a **T**.

Ahora tenemos una elección entre reducir la cadena **T**, que es el cuerpo de $E \rightarrow T$, y la cadena que consiste en el segundo **id**, que es el cuerpo de $F \rightarrow \text{id}$. En vez de reducir **T** a **E**, el segundo **id** se reduce a **T**, con lo cual se produce la cadena **T * F**. Despues, esta cadena se reduce a **T**. El análisis sintáctico termina con la reducción de **T** al símbolo inicial **E**. \square

Por definición, una reducción es el inverso de un paso en una derivación (recuerde que en una derivación, un no terminal en una forma de frase se sustituye por el cuerpo de una de sus producciones). Por lo tanto, el objetivo del análisis sintáctico ascendente es construir una derivación en forma inversa. La siguiente derivación corresponde al análisis sintáctico en la figura 4.25:

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

Esta derivación es de hecho una derivación por la derecha.

4.5.2 Poda de mangos

Durante una exploración de izquierda a derecha de la entrada, el análisis sintáctico ascendente construye una derivación por la derecha en forma inversa. De manera informal, un “mango” es una subcadena que coincide con el cuerpo de una producción, y cuya reducción representa un paso a lo largo del inverso de una derivación por la derecha.

Por ejemplo, si agregamos subíndices a los tokens **id** para mejorar la legibilidad, los mangos durante el análisis sintáctico de $\mathbf{id}_1 * \mathbf{id}_2$, de acuerdo con la gramática de expresiones (4.1), son como en la figura 4.26. Aunque T es el cuerpo de la producción $E \rightarrow T$, el símbolo T no es un mango en la forma de frase $T * \mathbf{id}_2$. Si T se sustituyera por E , obtendríamos la cadena $E * \mathbf{id}_2$, lo cual no puede derivarse del símbolo inicial E . Por ende, la subcadena por la izquierda que coincide con el cuerpo de alguna producción no necesita ser un mango.

FORMA DE FRASE DERECHA	MANGO	REDUCCIÓN DE LA PRODUCCIÓN
$\mathbf{id}_1 * \mathbf{id}_2$	\mathbf{id}_1	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \rightarrow F$
$T * \mathbf{id}_2$	\mathbf{id}_2	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$E \rightarrow T * F$

Figura 4.26: Mangos durante un análisis sintáctico de $\mathbf{id}_1 * \mathbf{id}_2$

De manera formal, si $S \xrightarrow{r_m} \alpha Aw \xrightarrow{r_m} \alpha\beta w$, como en la figura 4.27, entonces la producción $A \rightarrow \beta$ en la posición que sigue después de α es un *mango* de $\alpha\beta w$. De manera alternativa, un mango de la forma de frase derecha γ es una producción $A \rightarrow \beta$ y una posición de γ en donde puede encontrarse la cadena β , de tal forma que al sustituir β en esa posición por A se produzca la forma de frase derecha anterior en una derivación por la derecha de γ .

Observe que la cadena w a la derecha del mango debe contener sólo símbolos terminales. Por conveniencia, nos referimos al cuerpo β en vez de $A \rightarrow \beta$ como un mango. Observe que decimos “un mango” en vez de “el mango”, ya que la gramática podría ser ambigua, con más de una derivación por la derecha de $\alpha\beta w$. Si una gramática no tiene ambigüedad, entonces cada forma de frase derecha de la gramática tiene sólo un mango.

Puede obtenerse una derivación por la derecha en forma inversa mediante la “poda de mangos”. Es decir, empezamos con una cadena de terminales w a las que se les va a realizar

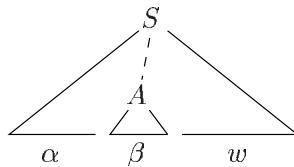


Figura 4.27: Un mango $A \rightarrow \beta$ en el árbol de análisis sintáctico para $\alpha\beta w$

el análisis sintáctico. Si w es un enunciado de la gramática a la mano, entonces dejamos que $w = \gamma_n$, en donde γ_n es la n -ésima forma de frase derecha de alguna derivación por la derecha, que todavía se desconoce:

$$S = \gamma_0 \xrightarrow{rm} \gamma_1 \xrightarrow{rm} \gamma_2 \xrightarrow{rm} \cdots \xrightarrow{rm} \gamma_{n-1} \xrightarrow{rm} \gamma_n = w$$

Para reconstruir esta derivación en orden inverso, localizamos el mango β_n en γ_n y sustituimos β_n por el encabezado de la producción $A_n \rightarrow \beta_n$ para obtener la forma de frase derecha γ_{n-1} anterior. Tenga en cuenta que todavía no sabemos cómo se van a encontrar los mangos, pero en breve veremos métodos para hacerlo.

Después repetimos este proceso. Es decir, localizamos el mango β_{n-1} en γ_{n-1} y reducimos este mango para obtener la forma de frase derecha γ_{n-2} . Si al continuar este proceso producimos una forma de frase derecha que consista sólo en el símbolo inicial S , entonces nos detenemos y anunciamos que el análisis sintáctico se completó con éxito. El inverso de la secuencia de producciones utilizadas en las reducciones es una derivación por la derecha para la cadena de entrada.

4.5.3 Análisis sintáctico de desplazamiento-reducción

El análisis sintáctico de desplazamiento-reducción es una forma de análisis sintáctico ascendente, en la cual una pila contiene símbolos gramaticales y un búfer de entrada contiene el resto de la cadena que se va a analizar. Como veremos, el mango siempre aparece en la parte superior de la pila, justo antes de identificarla como el mango.

Utilizamos el $\$$ para marcar la parte inferior de la pila y también el extremo derecho de la entrada. Por convención, al hablar sobre el análisis sintáctico ascendente, mostramos la parte superior de la pila a la derecha, en vez de a la izquierda como hicimos para el análisis sintáctico descendente. Al principio la pila está vacía, y la cadena w está en la entrada, como se muestra a continuación:

PILA	ENTRADA
$\$$	$w \$$

Durante una exploración de izquierda a derecha de la cadena de entrada, el analizador sintáctico desplaza cero o más símbolos de entrada y los mete en la pila, hasta que esté listo para reducir una cadena β de símbolos gramaticales en la parte superior de la pila. Despues reduce β al encabezado de la producción apropiada. El analizador sintáctico repite este ciclo hasta que haya detectado un error, o hasta que la pila contenga el símbolo inicial y la entrada esté vacía:

PILA	ENTRADA
$\$ S$	$\$$

Al entrar a esta configuración, el analizador sintáctico se detiene y anuncia que el análisis sintáctico se completó con éxito. La figura 4.28 avanza por pasos a través de las acciones que podría realizar un analizador sintáctico de desplazamiento-reducción al analizar la cadena de entrada $\mathbf{id}_1 * \mathbf{id}_2$, de acuerdo con la gramática de expresiones (4.1).

PILA	ENTRADA	ACCIÓN
\$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	desplazar
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reducir $F \rightarrow \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reducir $T \rightarrow F$
$\$ T$	$* \mathbf{id}_2 \$$	desplazar
$\$ T *$	$\mathbf{id}_2 \$$	desplazar
$\$ T * \mathbf{id}_2$	$\$$	reducir $F \rightarrow \mathbf{id}$
$\$ T * F$	$\$$	reducir $T \rightarrow T * F$
$\$ T$	$\$$	reducir $E \rightarrow T$
$\$ E$	$\$$	aceptar

Figura 4.28: Configuraciones de un analizador sintáctico de desplazamiento-reducción, con una entrada $\mathbf{id}_1 * \mathbf{id}_2$

Aunque las operaciones primarias son desplazar y reducir, en realidad hay cuatro acciones posibles que puede realizar un analizador sintáctico de desplazamiento-reducción: (1) desplazar, (2) reducir, (3) aceptar y (4) error.

1. *Desplazar*. Desplazar el siguiente símbolo de entrada y lo coloca en la parte superior de la pila.
2. *Reducir*. El extremo derecho de la cadena que se va a reducir debe estar en la parte superior de la pila. Localizar el extremo izquierdo de la cadena dentro de la pila y decidir con qué terminal se va a sustituir la cadena.
3. *Aceptar*. Anunciar que el análisis sintáctico se completó con éxito.
4. *Error*. Descubrir un error de sintaxis y llamar a una rutina de recuperación de errores.

El uso de una pila en el análisis sintáctico de desplazamiento-reducción se justifica debido a un hecho importante: el mango siempre aparecerá en algún momento dado en la parte superior de la pila, nunca en el interior. Este hecho puede demostrarse si consideramos las posibles formas de dos pasos sucesivos en cualquier derivación por la izquierda. La figura 4.29 ilustra los dos posibles casos. En el caso (1), A se sustituye por $\beta B y$, y después el no terminal B por la derecha en el cuerpo $\beta B y$ se sustituye por γ . En el caso (2), A se expande primero otra vez, pero ahora el cuerpo es una cadena y que consiste sólo en terminales. El siguiente no terminal B por la derecha se encontrará en alguna parte a la derecha de y .

En otras palabras:

$$\begin{aligned}
 (1) \quad & S \xrightarrow[\substack{rm \\ rm}]{} \alpha A z \Rightarrow \alpha \beta B y z \xrightarrow[\substack{rm \\ rm}]{} \alpha \beta \gamma y z \\
 (2) \quad & S \xrightarrow[\substack{rm \\ rm}]{} \alpha B x A z \Rightarrow \alpha B x y z \xrightarrow[\substack{rm \\ rm}]{} \alpha \gamma x y z
 \end{aligned}$$

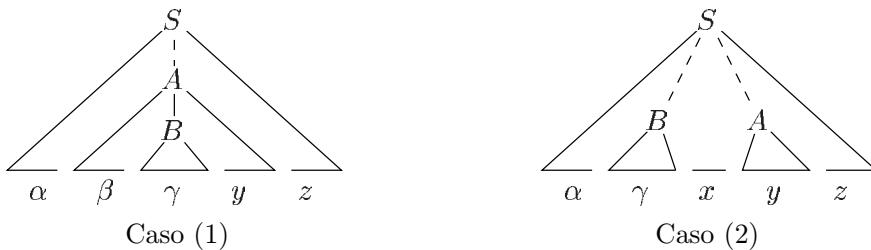


Figura 4.29: Casos para dos pasos sucesivos de una derivación por la derecha

Considere el caso (1) a la inversa, en donde un analizador sintáctico de desplazamiento-reducción acaba de llegar a la siguiente configuración:

PILA ENTRADA
 $\$ \alpha \beta \gamma$ $yz\$$

El analizador sintáctico reduce el mango γ a B para llegar a la siguiente configuración:

$\$ \alpha \beta B \qquad \qquad \qquad y z \$$

Ahora el analizador puede desplazar la cadena y y colocarla en la pila, mediante una secuencia de cero o más movimientos de desplazamiento para llegar a la configuración

$\$ \alpha \beta B y \quad z \$$

con el mango βBy en la parte superior de la pila, y se reduce a A .

Ahora considere el caso (2). En la configuración

$\$ \alpha \gamma \quad x y z \$$

el mango γ está en la parte superior de la pila. Después de reducir el mango γ a B , el analizador sintáctico puede reducir la cadena xy para meter el siguiente mango y en la parte superior de la pila, listo para reducirse a A :

$\$ \alpha B x y z \$$

En ambos casos, después de realizar una reducción, el analizador sintáctico tuvo que desplazar cero o más símbolos para meter el siguiente mango en la pila. Nunca tuvo que buscar el mango dentro de la pila.

4.5.4 Conflictos durante el análisis sintáctico de desplazamiento-reducción

Existen gramáticas libres de contexto para las cuales no se puede utilizar el análisis sintáctico de desplazamiento-reducción. Cada analizador sintáctico de desplazamiento-reducción para una gramática de este tipo puede llegar a una configuración en la cual el analizador sintáctico, conociendo el contenido completo de la pila y el siguiente símbolo de entrada, no puede decidir

si va a desplazar o a reducir (un *conflicto de desplazamiento/reducción*), o no puede decidir qué reducciones realizar (un *conflicto de reducción/reducción*). Ahora veremos algunos ejemplos de construcciones sintácticas que ocasionan tales gramáticas. Técnicamente, estas gramáticas no están en la clase $LR(k)$ de gramáticas definidas en la sección 4.7; nos referimos a ellas como gramáticas no LR . La k en $LR(k)$ se refiere al número de símbolos de preanálisis en la entrada. Las gramáticas que se utilizan en la compilación, por lo general, entran en la clase $LR(1)$, con un símbolo de anticipación a lo más.

Ejemplo 4.38: Una gramática ambigua nunca podrá ser LR . Por ejemplo, considere la gramática del *else* colgante (4.14) de la sección 4.3:

$$\begin{array}{lcl} instr & \rightarrow & \mathbf{if} \ expr \ \mathbf{then} \ instr \\ & | & \mathbf{if} \ expr \ \mathbf{then} \ instr \ \mathbf{else} \ instr \\ & | & \mathbf{otra} \end{array}$$

Si tenemos un analizador sintáctico de desplazamiento-reducción con la siguiente configuración:

PILA	ENTRADA
... if <i>expr then instr</i>	else ... \$

no podemos saber si **if** *expr then instr* es el mango, sin importar lo que aparezca debajo de él en la pila. Aquí tenemos un conflicto de desplazamiento/reducción. Dependiendo de lo que siga después del **else** en la entrada, podría ser correcto reducir **if** *expr then instr* a *instr*, o podría ser correcto desplazar el **else** y después buscar otro *instr* para completar la expresión alternativa **if** *expr then instr else instr*.

Observe que el análisis sintáctico de desplazamiento-reducción puede adaptarse para analizar ciertas gramáticas ambiguas, como la gramática *if-then-else* anterior. Si resolvemos el conflicto de desplazamiento/reducción en el **else** a favor del desplazamiento, el analizador sintáctico se comportará como esperamos, asociando cada **else** con el **then** anterior sin coincidencia. En la sección 4.8 hablaremos sobre los analizadores sintácticos para dichas gramáticas ambiguas. □

Otra configuración común para los conflictos ocurre cuando sabemos que tenemos un mango, pero el contenido de la pila y el siguiente símbolo de entrada no son suficientes para determinar qué producción debe usarse en una reducción. El siguiente ejemplo ilustra esta situación.

Ejemplo 4.39: Suponga que tenemos un analizador léxico que devuelve el nombre de token **id** para todos los nombres, sin importar su tipo. Suponga también que nuestro lenguaje invoca a los procedimientos proporcionando sus nombres, con los parámetros rodeados entre paréntesis, y que los arreglos se referencian mediante la misma sintaxis. Como la traducción de los índices en las referencias a arreglos y los parámetros en las llamadas a procedimientos son distintos, queremos usar distintas producciones para generar listas de parámetros e índices actuales. Por lo tanto, nuestra gramática podría tener producciones como las de la figura 4.30 (entre otras).

Una instrucción que empieza con **p(i,j)** aparecería como el flujo de tokens **id(id, id)** para el analizador sintáctico. Después de desplazar los primeros tres tokens en la pila, un analizador sintáctico de desplazamiento-reducción tendría la siguiente configuración:

(1)	<i>instr</i> →	id (<i>lista_parametros</i>)
(2)	<i>instr</i> →	<i>expr</i> := <i>expr</i>
(3)	<i>lista_parametros</i> →	<i>lista_parametros</i> , <i>parámetro</i>
(4)	<i>lista_parametros</i> →	<i>parámetro</i>
(5)	<i>parámetro</i> →	id
(6)	<i>expr</i> →	id (<i>lista_expr</i>)
(7)	<i>expr</i> →	id
(8)	<i>lista_expr</i> →	<i>lista_expr</i> , <i>expr</i>
(9)	<i>lista_expr</i> →	<i>expr</i>

Figura 4.30: Producciones que implican llamadas a procedimientos y referencias a arreglos



Es evidente que el **id** en la parte superior de la pila debe reducirse, pero ¿mediante qué producción? La elección correcta es la producción (5) si *p* es un procedimiento, pero si *p* es un arreglo, entonces es la producción (7). La pila no indica qué información debemos usar en la en la tabla de símbolos que se obtiene a partir de la declaración de *p*.

Una solución es cambiar el token **id** en la producción (1) a **procid** y usar un analizador léxico más sofisticado, que devuelva el nombre de token **procid** cuando reconozca un lexema que sea el nombre de un procedimiento. Para ello se requeriría que el analizador léxico consultara la tabla de símbolos, antes de devolver un token.

Si realizamos esta modificación, entonces al procesar *p(i,j)* el analizador se encontraría en la siguiente configuración:



o en la configuración anterior. En el caso anterior, elegimos la reducción mediante la producción (5); en el último caso mediante la producción (7). Observe cómo el tercer símbolo de la parte superior de la pila determina la reducción que se va a realizar, aun cuando no está involucrado en la reducción. El análisis sintáctico de desplazamiento-reducción puede utilizar información de más adentro en la pila, para guiar el análisis sintáctico. \square

4.5.5 Ejercicios para la sección 4.5

Ejercicio 4.5.1: Para la gramática $S \rightarrow 0 S 1 \mid 0 1$ del ejercicio 4.2.2(a), indique el mango en cada una de las siguientes formas de frases derechas:

- a) 000111.
- b) 00*S*11.

Ejercicio 4.5.2: Repita el ejercicio 4.5.1 para la gramática $S \rightarrow S S + \mid S S * \mid a$ del ejercicio 4.2.1 y las siguientes formas de frases derechas:

- a) $SSS + a * +.$
- b) $SS + a * a +.$
- c) $aaa * a + +.$

Ejercicio 4.5.3: Proporcione los análisis sintácticos ascendentes para las siguientes cadenas de entrada y gramáticas:

- a) La entrada 000111, de acuerdo a la gramática del ejercicio 4.5.1.
- b) La entrada $aaa * a++$, de acuerdo a la gramática del ejercicio 4.5.2.

4.6 Introducción al análisis sintáctico LR: SLR (LR simple)

El tipo más frecuente de analizador sintáctico ascendentes en la actualidad se basa en un concepto conocido como análisis sintáctico $LR(k)$; la “L” indica la exploración de izquierda a derecha de la entrada, la “R” indica la construcción de una derivación por la derecha a la inversa, y la k para el número de símbolos de entrada de preanálisis que se utilizan al hacer decisiones del análisis sintáctico. Los casos $k = 0$ o $k = 1$ son de interés práctico, por lo que aquí sólo consideraremos los analizadores sintácticos LR con $k \leq 1$. Cuando se omite (k) , se asume que k es 1.

Esta sección presenta los conceptos básicos del análisis sintáctico LR y el método más sencillo para construir analizadores sintácticos de desplazamiento-reducción, llamados “LR Simple” (o SLR). Es útil tener cierta familiaridad con los conceptos básicos, incluso si el analizador sintáctico LR se construye mediante un generador de analizadores sintácticos automático. Empezaremos con “elementos” y “estados del analizador sintáctico”; la salida de diagnóstico de un generador de analizadores sintácticos LR, por lo general, incluye estados del analizador sintáctico, los cuales pueden usarse para aislar las fuentes de conflictos en el análisis sintáctico.

La sección 4.7 introduce dos métodos más complejos (LR canónico y LALR) que se utilizan en la mayoría de los analizadores sintácticos LR.

4.6.1 ¿Por qué analizadores sintácticos LR?

Los analizadores sintácticos LR son controlados por tablas, en forma muy parecida a los analizadores sintácticos LL no recursivos de la sección 4.4.4. Se dice que una gramática para la cual podemos construir una tabla de análisis sintáctico, usando uno de los métodos en esta sección y en la siguiente, es una *gramática LR*. De manera intuitiva, para que una gramática sea LR, basta con que un analizador sintáctico de desplazamiento-reducción de izquierda a derecha pueda reconocer mangos de las formas de frases derechas, cuando éstas aparecen en la parte superior de la pila.

El análisis sintáctico LR es atractivo por una variedad de razones:

- Pueden construirse analizadores sintácticos LR para reconocer prácticamente todas las construcciones de lenguajes de programación para las cuales puedan escribirse gramáticas libres de contexto. Existen gramáticas libres de contexto que no son LR, pero por lo general se pueden evitar para las construcciones comunes de los lenguajes de programación.

- El método de análisis sintáctico LR es el método de análisis sintáctico de desplazamiento-reducción sin rastreo hacia atrás más general que se conoce a la fecha, y aún así puede implementarse con la misma eficiencia que otros métodos más primitivos de desplazamiento-reducción (vea las notas bibliográficas).
- Un analizador sintáctico LR puede detectar un error sintáctico tan pronto como sea posible en una exploración de izquierda a derecha de la entrada.
- La clase de gramáticas que pueden analizarse mediante los métodos LR es un superconjunto propio de la clase de gramáticas que pueden analizarse con métodos predictivos o LL. Para que una gramática sea $LR(k)$, debemos ser capaces de reconocer la ocurrencia del lado derecho de una producción en una forma de frase derecha, con k símbolos de entrada de preanálisis. Este requerimiento es mucho menos estricto que para las gramáticas $LL(k)$, en donde debemos ser capaces de reconocer el uso de una producción, viendo sólo los primeros símbolos k de lo que deriva su lado derecho. Por ende, no debe sorprender que las gramáticas LR puedan describir más lenguajes que las gramáticas LL.

La principal desventaja del método LR es que es demasiado trabajo construir un analizador sintáctico LR en forma manual para una gramática común de un lenguaje de programación. Se necesita una herramienta especializada: un generador de analizadores sintácticos LR. Por fortuna, hay varios generadores disponibles, y en la sección 4.9 hablaremos sobre uno de los que se utilizan con más frecuencia: **Yacc**. Dicho generador recibe una gramática libre de contexto y produce de manera automática un analizador para esa gramática. Si la gramática contiene ambigüedades u otras construcciones que sean difíciles de analizar en una exploración de izquierda a derecha de la entrada, entonces el generador de analizadores sintácticos localiza estas construcciones y proporciona mensajes de diagnóstico detallados.

4.6.2 Los elementos y el autómata $LR(0)$

¿Cómo sabe un analizador sintáctico de desplazamiento-reducción cuándo desplazar y cuándo reducir? Por ejemplo, con el contenido $\$T$ de la pila y el siguiente símbolo de entrada $*$ en la figura 4.28, ¿cómo sabe el analizador sintáctico que la T en la parte superior de la pila no es un mango, por lo cual la acción apropiada es desplazar y no reducir T a E ?

Un analizador sintáctico LR realiza las decisiones de desplazamiento-reducción mediante el mantenimiento de estados, para llevar el registro de la ubicación que tenemos en un análisis sintáctico. Los estados representan conjuntos de “elementos”. Un *elemento $LR(0)$* (*elemento*, para abreviar) de una gramática G es una producción de G con un punto en cierta posición del cuerpo. Por ende, la producción $A \rightarrow XYZ$ produce los siguientes cuatro elementos:

$$\begin{aligned} A &\rightarrow \cdot XYZ \\ A &\rightarrow X \cdot YZ \\ A &\rightarrow XY \cdot Z \\ A &\rightarrow XYZ \cdot \end{aligned}$$

La producción $A \rightarrow \epsilon$ genera sólo un elemento, $A \rightarrow \cdot$.

De manera intuitiva, un elemento indica qué parte de una producción hemos visto en un punto dado del proceso de análisis sintáctico. Por ejemplo, el elemento $A \rightarrow \cdot XYZ$ indica

Representación de conjuntos de elementos

Un generador de análisis sintáctico que produce un analizador descendente tal vez requiere representar elementos y conjuntos de elementos en una forma conveniente. Un elemento puede representarse mediante un par de enteros, el primero de los cuales es el número de una de las producciones de la gramática subyacente, y el segundo de los cuales es la posición del punto. Los conjuntos de elementos pueden representarse mediante una lista de estos pares. No obstante, como veremos pronto, los conjuntos necesarios de elementos a menudo incluyen elementos de “cierre”, en donde el punto se encuentra al principio del cuerpo. Estos siempre pueden reconstruirse a partir de los otros elementos en el conjunto, por lo que no tenemos que incluirlos en la lista.

que esperamos ver una cadena que pueda derivarse de XYZ a continuación en la entrada. El elemento $A \rightarrow X \cdot YZ$ indica que acabamos de ver en la entrada una cadena que puede derivarse de X , y que esperamos ver a continuación una cadena que pueda derivarse de YZ . El elemento $A \rightarrow XY \cdot Z$ indica que hemos visto el cuerpo XYZ y que puede ser hora de reducir XYZ a A .

Una colección de conjuntos de elementos $LR(0)$, conocida como la colección $LR(0)$ *canónica*, proporciona la base para construir un autómata finito determinista, el cual se utiliza para realizar decisiones en el análisis sintáctico. A dicho autómata se le conoce como *autómata $LR(0)$* .³ En especial, cada estado del autómata $LR(0)$ representa un conjunto de elementos en la colección $LR(0)$ canónica. El autómata para la gramática de expresiones (4.1), que se muestra en la figura 4.31, servirá como ejemplo abierto para hablar sobre la colección $LR(0)$ canónica para una gramática.

Para construir la colección $LR(0)$ canónica de una gramática, definimos una gramática aumentada y dos funciones, *CERRADURA* e *ir_A*. Si G es una gramática con el símbolo inicial S , entonces G' , la *gramática aumentada* para G , es G con un nuevo símbolo inicial S' y la producción $S' \rightarrow S$. El propósito de esta nueva producción inicial es indicar al analizador sintáctico cuándo debe dejar de analizar para anunciar la aceptación de la entrada. Es decir, la aceptación ocurre sólo cuando el analizador sintáctico está a punto de reducir mediante $S' \rightarrow S$.

Cerradura de conjuntos de elementos

Si I es un conjunto de elementos para una gramática G , entonces *CERRADURA*(I) es el conjunto de elementos que se construyen a partir de I mediante las siguientes dos reglas:

1. Al principio, agregar cada elemento en I a *CERRADURA*(I).
2. Si $A \rightarrow \alpha \cdot B\beta$ está en *CERRADURA*(I) y $B \rightarrow \gamma$ es una producción, entonces agregar el elemento $B \rightarrow \gamma$ a *CERRADURA*(I), si no se encuentra ya ahí. Aplicar esta regla hasta que no puedan agregarse más elementos nuevos a *CERRADURA*(I).

³Técnicamente, el autómata deja de ser determinista de acuerdo a la definición de la sección 3.6.4, ya que no tenemos un estado muerto, que corresponde al conjunto vacío de elementos. Como resultado, hay ciertos pares estado-entrada para los cuales no existe un siguiente estado.

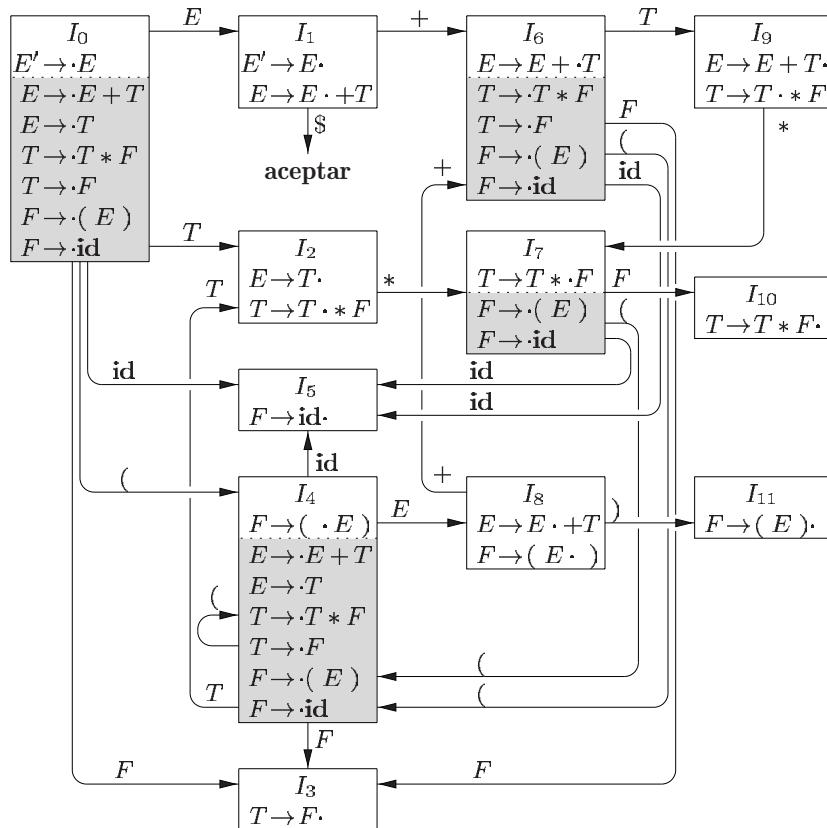


Figura 4.31: Autómata LR(0) para la gramática de expresiones (4.1)

De manera intuitiva, $A \rightarrow \alpha \cdot B \beta$ en $\text{CERRADURA}(I)$ indica que, en algún punto en el proceso de análisis sintáctico, creemos que podríamos ver a continuación una subcadena que pueda derivarse de $B\beta$ como entrada. La subcadena que pueda derivarse de $B\beta$ tendrá un prefijo que pueda derivarse de B , mediante la aplicación de una de las producciones B . Por lo tanto, agregamos elementos para todas las producciones B ; es decir, si $B \rightarrow \gamma$ es una producción, también incluimos a $B \rightarrow \cdot \gamma$ en $\text{CERRADURA}(I)$.

Ejemplo 4.40: Considere la siguiente gramática de expresiones aumentada:

$$\begin{aligned}
 E' &\rightarrow E \\
 E &\rightarrow E + T \mid T \\
 T &\rightarrow T * F \mid F \\
 E &\rightarrow (E) \mid \text{id}
 \end{aligned}$$

Si I es el conjunto de un elemento $\{[E' \rightarrow \cdot E]\}$, entonces $\text{CERRADURA}(I)$ contiene el conjunto de elementos I_0 en la figura 4.31.

Para ver cómo se calcula la CERRADURA, $E' \rightarrow \cdot E$ se coloca en $\text{CERRADURA}(I)$ mediante la regla (1). Como hay una E justo a la derecha de un punto, agregamos las producciones E con puntos en los extremos izquierdos: $E \rightarrow \cdot E + T$ y $E \rightarrow \cdot T$. Ahora hay una T justo a la derecha de un punto en este último elemento, por lo que agregamos $T \rightarrow \cdot T * F$ y $T \rightarrow \cdot F$. A continuación, la F a la derecha de un punto nos obliga a agregar $F \rightarrow \cdot(E)$ y $F \rightarrow \cdot \text{id}$, pero no necesita agregarse ningún otro elemento. \square

La cerradura puede calcularse como en la figura 4.32. Una manera conveniente de implementar la función *cerradura* es mantener un arreglo booleano llamado *agregado*, indexado mediante los no terminales de G , de tal forma que $\text{agregado}[B]$ se establezca a **true** si, y sólo si agregamos el elemento $B \rightarrow \cdot \gamma$ para cada producción B de la forma $B \rightarrow \gamma$.

```

ConjuntoDeElementos CERRADURA( $I$ ) {
     $J = I$ ;
    repeat
        for ( cada elemento  $A \rightarrow \alpha \cdot B \beta$  en  $J$  )
            for ( cada producción  $B \rightarrow \gamma$  de  $G$  )
                if (  $B \rightarrow \cdot \gamma$  no está en  $J$  )
                    agregar  $B \rightarrow \cdot \gamma$  a  $J$ ;
        until no se agreguen más elementos a  $J$  en una ronda;
    return  $J$ ;
}

```

Figura 4.32: Cálculo de CERRADURA

Observe que si se agrega una producción B al cierre de I con el punto en el extremo izquierdo, entonces se agregarán todas las producciones B de manera similar al cierre. Por ende, no es necesario en algunas circunstancias listar los elementos $B \rightarrow \gamma$ que se agregan a I mediante CERRADURA. Basta con una lista de los no terminales B cuyas producciones se agregaron. Dividimos todos los conjuntos de elementos de interés en dos clases:

1. *Elementos del corazón*: el elemento inicial, $S' \rightarrow \cdot S$, y todos los elementos cuyos puntos no estén en el extremo izquierdo.
2. *Elementos que no son del corazón*: todos los elementos con sus puntos en el extremo izquierdo, excepto $S' \rightarrow \cdot S$.

Además, cada conjunto de elementos de interés se forma tomando la cerradura de un conjunto de elementos del corazón; desde luego que los elementos que se agregan en la cerradura nunca podrán ser elementos del corazón. Por lo tanto, podemos representar los conjuntos de elementos en los que realmente estamos interesados con muy poco almacenamiento si descartamos todos los elementos que no sean del corazón, sabiendo que podrían regenerarse mediante el proceso de cerradura. En la figura 4.31, los elementos que no son del corazón se encuentran en la parte sombreada del cuadro para un estado.

La función ir_A

La segunda función útil es $\text{ir_A}(I, X)$, en donde I es un conjunto de elementos y X es un símbolo gramatical. $\text{ir_A}(I, X)$ se define como la cerradura del conjunto de todos los elementos $[A \rightarrow \alpha X \cdot \beta]$, de tal forma que $[A \rightarrow \alpha \cdot X \beta]$ se encuentre en I . De manera intuitiva, la función ir_A se utiliza para definir las transiciones en el autómata LR(0) para una gramática. Los estados del autómata corresponden a los conjuntos de elementos, y $\text{ir_A}(I, X)$ especifica la transición que proviene del estado para I , con la entrada X .

Ejemplo 4.41: Si I es el conjunto de dos elementos $\{[E' \rightarrow E \cdot], [E \rightarrow E \cdot + T]\}$, entonces $\text{ir_A}(I, +)$ contiene los siguientes elementos:

$$\begin{array}{lcl} E & \rightarrow & E + \cdot T \\ T & \rightarrow & \cdot T * F \\ T & \rightarrow & \cdot F \\ F & \rightarrow & \cdot(E) \\ F & \rightarrow & \cdot \mathbf{id} \end{array}$$

Para calcular $\text{ir_A}(I, +)$, examinamos I en busca de elementos con $+$ justo a la derecha del punto. $E' \rightarrow E \cdot$ no es uno de estos elementos, pero $E \rightarrow E \cdot + T$ sí lo es. Desplazamos el punto sobre el $+$ para obtener $E \rightarrow E + \cdot T$ y después tomamos la cerradura de este conjunto singleton. \square

Ahora estamos listos para que el algoritmo construya a C , la colección canónica de conjuntos de elementos LR(0) para una gramática aumentada G' ; el algoritmo se muestra en la figura 4.33.

```
void elementos(G') {
    C = CERRADURA({[S' → ·S]});
    repeat
        for ( cada conjunto de elementos I en C )
            for ( cada símbolo gramatical X )
                if ( ir_A(I, X) no está vacío y no está en C )
                    agregar ir_A(I, X) a C;
    until no se agreguen nuevos conjuntos de elementos a C en una iteración;
}
```

Figura 4.33: Cálculo de la colección canónica de conjuntos de elementos LR(0)

Ejemplo 4.42: La colección canónica de conjuntos de elementos LR(0) para la gramática (4.1) y la función ir_A se muestran en la figura 4.31. ir_A se codifica mediante las transiciones en la figura. \square

Uso del autómata LR(0)

La idea central del análisis sintáctico “LR simple”, o SLR, es la construcción del autómata LR(0) a partir de la gramática. Los estados de este autómata son los conjuntos de elementos de la colección LR(0) canónica, y las traducciones las proporciona la función ir_A . El autómata LR(0) para la gramática de expresiones (4.1) apareció antes en la figura 4.31.

El estado inicial del autómata LR(0) es $CERRADURA(\{[S' \rightarrow \cdot S]\})$, en donde S' es el símbolo inicial de la gramática aumentada. Todos los estados son de aceptaciones. Decimos que el “estado j ” se refiere al estado que corresponde al conjunto de elementos I_j .

¿Cómo puede ayudar el autómata LR(0) con las decisiones de desplazar-reducir? Estas decisiones pueden realizarse de la siguiente manera. Suponga que la cadena γ de símbolos gramaticales lleva el autómata LR(0) del estado inicial 0 a cierto estado j . Después, se realiza un desplazamiento sobre el siguiente símbolo de entrada a si el estado j tiene una transición en a . En cualquier otro caso, elegimos reducir; los elementos en el estado j nos indicarán qué producción usar.

El algoritmo de análisis sintáctico LR que presentaremos en la sección 4.6.3 utiliza su pila para llevar el registro de los estados, así como de los símbolos gramaticales; de hecho, el símbolo gramatical puede recuperarse del estado, por lo que la pila contiene los estados. El siguiente ejemplo proporciona una vista previa acerca de cómo pueden utilizarse un autómata LR(0) y una pila de estados para realizar decisiones de desplazamiento-reducción en el análisis sintáctico.

Ejemplo 4.43: La figura 4.34 ilustra las acciones de un analizador sintáctico de desplazamiento-reducción con la entrada **id * id**, usando el autómata LR(0) de la figura 4.31. Utilizamos una pila para guardar los estados; por claridad, los símbolos gramaticales que corresponden a los estados en la pila aparecen en la columna SÍMBOLOS. En la línea (1), la pila contiene el estado inicial 0 del autómata; el símbolo correspondiente es el marcador $\$$ de la parte inferior de la pila.

LÍNEA	PILA	SÍMBOLOS	ENTRADA	ACCIÓN
(1)	0	$\$$	id * id \$	desplazar 5
(2)	0 5	$\$ id$	$* id \$$	reducir $F \rightarrow id$
(3)	0 3	$\$ F$	$* id \$$	reducir $T \rightarrow F$
(4)	0 2	$\$ T$	$* id \$$	desplazar 7
(5)	0 2 7	$\$ T * id$	$id \$$	desplazar 5
(6)	0 2 7 5	$\$ T * id$	$\$$	reducir $F \rightarrow id$
(7)	0 2 7 10	$\$ T * F$	$\$$	reducir $T \rightarrow T * F$
(8)	0 2	$\$ T$	$\$$	reducir $E \rightarrow T$
(9)	0 1	$\$ E$	$\$$	aceptar

Figura 4.34: El análisis sintáctico de **id * id**

El siguiente símbolo de entrada es **id** y el estado 0 tiene una transición en **id** al estado 5. Por lo tanto, realizamos un desplazamiento. En la línea (2), el estado 5 (símbolo **id**) se ha metido en la pila. No hay transición desde el estado 5 con la entrada *****, por lo que realizamos una reducción. Del elemento $[F \rightarrow id \cdot]$ en el estado 5, la reducción es mediante la producción $F \rightarrow id$.

Con los símbolos, una reducción se implementa sacando el cuerpo de la producción de la pila (en la línea (2), el cuerpo es **id**) y metiendo el encabezado de la producción (en este caso, *F*). Con los estados, sacamos el estado 5 para el símbolo **id**, lo cual lleva el estado 0 a la parte superior y buscamos una transición en *F*, el encabezado de la producción. En la figura 4.31, el estado 0 tiene una transición en *F* al estado 3, por lo que metemos el estado 3, con el símbolo *F* correspondiente; vea la línea (3).

Como otro ejemplo, considere la línea (5), con el estado 7 (símbolo *) en la parte superior de la pila. Este estado tiene una transición al estado 5 con la entrada **id**, por lo que metemos el estado 5 (símbolo **id**). El estado 5 no tiene transiciones, así que lo reducimos mediante $F \rightarrow \text{id}$. Al sacar el estado 5 para el cuerpo **id**, el estado 7 pasa a la parte superior de la pila. Como el estado 7 tiene una transición en *F* al estado 10, metemos el estado 10 (símbolo *F*). \square

4.6.3 El algoritmo de análisis sintáctico LR

En la figura 4.35 se muestra un diagrama de un analizador sintáctico LR. Este diagrama consiste en una entrada, una salida, una pila, un programa controlador y una tabla de análisis sintáctico que tiene dos partes (ACCION y *ir_A*). El programa controlador es igual para todos los analizadores sintácticos LR; sólo la tabla de análisis sintáctico cambia de un analizador sintáctico a otro. El programa de análisis sintáctico lee caracteres de un búfer de entrada, uno a la vez. En donde un analizador sintáctico de desplazamiento-reducción desplazaría a un símbolo, un analizador sintáctico LR desplaza a un *estado*. Cada estado sintetiza la información contenida en la pila, debajo de éste.

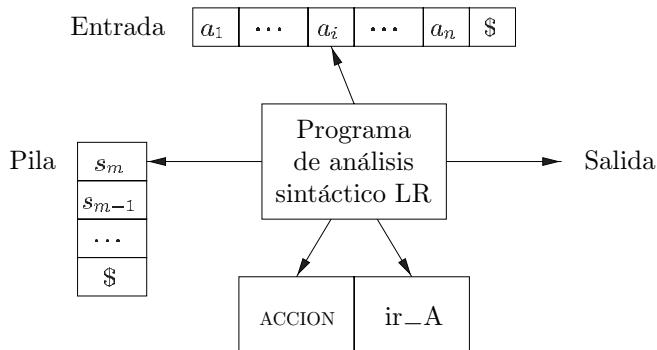


Figura 4.35: Modelo de un analizador sintáctico LR

La pila contiene una secuencia de estados, $s_0 s_1 \dots s_m$, en donde s_m , se encuentra en la parte superior. En el método SLR, la pila contiene estados del autómata $LR(0)$; los métodos LR canónico y LALR son similares. Por construcción, cada estado tiene un símbolo gramatical correspondiente. Recuerde que los estados corresponden a los conjuntos de elementos, y que hay una transición del estado i al estado j si $ir_A(I_i, X) = I_j$. Todas las transiciones al estado j deben ser para el mismo símbolo gramatical X . Por ende, cada estado, excepto el estado inicial 0, tiene un símbolo gramatical único asociado con él.⁴

⁴Lo opuesto no necesariamente es válido; es decir, más de un estado puede tener el mismo símbolo gramatical. Por

Estructura de la tabla de análisis sintáctico LR

La tabla de análisis sintáctico consiste en dos partes: una función de acción de análisis sintáctico llamada `ACCION` y una función `ir_A`.

1. La función `ACCION` recibe como argumentos un estado i y un terminal a (o $\$$, el marcador de fin de entrada). El valor de $\text{ACCION}[i, a]$ puede tener una de cuatro formas:
 - (a) Desplazar j , en donde j es un estado. La acción realizada por el analizador sintáctico desplaza en forma efectiva la entrada a hacia la pila, pero usa el estado j para representar la a .
 - (b) Reducir $A \rightarrow \beta$. La acción del analizador reduce en forma efectiva a β en la parte superior de la pila, al encabezado A .
 - (c) Aceptar. El analizador sintáctico acepta la entrada y termina el análisis sintáctico.
 - (d) Error. El analizador sintáctico descubre un error en su entrada y realiza cierta acción correctiva. En las secciones 4.8.3 y 4.8.4 hablaremos más acerca de cómo funcionan dichas rutinas de recuperación de errores.
2. Extendemos la función `ir_A`, definida en los conjuntos de elementos, a los estados: si $\text{ir}_A[I_i, A] = I_j$, entonces `ir_A` también asigna un estado i y un no terminal A al estado j .

Configuraciones del analizador sintáctico LR

Para describir el comportamiento de un analizador sintáctico LR, es útil tener una notación que represente el estado completo del analizador sintáctico: su pila y el resto de la entrada. Una *configuración* de un analizador sintáctico LR es un par:

$$(s_0s_1 \dots s_m, a_ia_{i+1} \dots a_n\$)$$

en donde el primer componente es el contenido de la pila (parte superior a la derecha) y el segundo componente es el resto de la entrada. Esta configuración representa la forma de frase derecha:

$$X_1X_2 \dots X_ma_ia_{i+1} \dots a_n$$

básicamente en la misma forma en que lo haría un analizador sintáctico de desplazamiento-reducción; la única diferencia es que en vez de símbolos gramaticales, la pila contiene estados a partir de los cuales pueden recuperarse los símbolos gramaticales. Es decir, X_i es el símbolo gramatical representado mediante el estado s_i . Observe que s_0 , el estado inicial del analizador sintáctico, no representa a un símbolo gramatical y sirve como marcador de la parte inferior de la pila, así como también juega un papel importante en el análisis sintáctico.

ejemplo, vea los estados 1 y 8 en el autómata LR(0) de la figura 4.31, a los cuales se entra mediante las transiciones en E , o los estados 2 y 9, a los cuales se entra mediante las transiciones en T .

Comportamiento del analizador sintáctico LR

El siguiente movimiento del analizador sintáctico a partir de la configuración anterior, se determina mediante la lectura de a_i , el símbolo de entrada actual, y s_m , el estado en la parte superior de la pila, y después se consulta la entrada $\text{ACCION}[s_m, a_i]$ en la tabla de acción del análisis sintáctico. Las configuraciones resultantes después de cada uno de los cuatro tipos de movimiento son:

1. Si $\text{ACCION}[s_m, a_i] = \text{desplazar } s$, el analizador ejecuta un movimiento de desplazamiento; desplaza el siguiente estado s y lo mete en la pila, introduciendo la siguiente configuración:

$$(s_0s_1 \dots s_ms, a_{i+1} \dots a_n\$)$$

El símbolo a_i no necesita guardarse en la pila, ya que puede recuperarse a partir de s , si es necesario (en la práctica, nunca lo es). Ahora, el símbolo de entrada actual es a_{i+1} .

2. Si $\text{ACCION}[s_m, a_i] = \text{reducir } A \rightarrow \beta$, entonces el analizador sintáctico ejecuta un movimiento de reducción, entrando a la siguiente configuración:

$$(s_0s_1 \dots s_{m-r}s, a_ia_{i+1} \dots a_n\$)$$

en donde r es la longitud de β , y $s = \text{ir_A}[s_{m-r}, A]$. Aquí, el analizador sintáctico primero sacó los símbolos del estado r de la pila, exponiendo al estado s_{m-r} . Después el analizador sintáctico metió a s , la entrada para $\text{ir_A}[s_{m-r}, A]$ en la pila. El símbolo de entrada actual no se cambia en un movimiento de reducción. Para los analizadores sintácticos LR que vamos a construir, $X_{m-r+1} \dots X_m$, la secuencia de los símbolos gramaticales correspondientes a los estados que se sacan de la pila, siempre coincidirá con β , el lado derecho de la producción reductora.

La salida de un analizador sintáctico LR se genera después de un movimiento de reducción, mediante la ejecución de la acción semántica asociada con la producción reductora. Por el momento, vamos a suponer que la salida consiste sólo en imprimir la producción reductora.

3. Si $\text{ACCION}[s_m, a_i] = \text{aceptar}$, se completa el análisis sintáctico.
4. Si $\text{ACCION}[s_m, a_i] = \text{error}$, el analizador ha descubierto un error y llama a una rutina de recuperación de errores.

A continuación se sintetiza el algoritmo de análisis sintáctico LR. Todos los analizadores sintácticos LR se comportan de esta manera; la única diferencia entre un analizador sintáctico LR y otro es la información en los campos ACCION e ir_A de la tabla de análisis sintáctico.

Algoritmo 4.44: Algoritmo de análisis sintáctico LR.

ENTRADA: Una cadena de entrada w y una tabla de análisis sintáctico LR con las funciones ACCION e ir_A , para una gramática G .

SALIDA: Si w está en $L(G)$, los pasos de reducción de un análisis sintáctico ascendentes para w ; en cualquier otro caso, una indicación de error.

MÉTODO: Al principio, el analizador sintáctico tiene s_0 en su pila, en donde s_0 es el estado inicial y $w\$$ está en el búfer de entrada. Entonces, el analizador ejecuta el programa en la figura 4.36. \square

```

hacer que  $a$  sea el primer símbolo de  $w\$$ ;
while(1) { /* repetir indefinidamente */
    hacer que  $s$  sea el estado en la parte superior de la pila;
    if ( ACCION[ $s, a$ ] = desplazar  $t$  ) {
        meter  $t$  en la pila;
        hacer que  $a$  sea el siguiente símbolo de entrada;
    } else if ( ACCION[ $s, a$ ] = reducir  $A \rightarrow \beta$  ) {
        sacar  $|\beta|$  símbolos de la pila;
        hacer que el estado  $t$  ahora esté en la parte superior de la pila;
        meter ir_A[ $t, A$ ] en la pila;
        enviar de salida la producción  $A \rightarrow \beta$ ;
    } else if ( ACCION[ $s, a$ ] = aceptar ) break; /* terminó el análisis sintáctico */
    else llamar a la rutina de recuperación de errores;
}

```

Figura 4.36: Programa de análisis sintáctico LR

Ejemplo 4.45: La figura 4.37 muestra las funciones ACCION e ir_A de una tabla de análisis sintáctico LR para la gramática de expresiones (4.1), que repetimos a continuación con las producciones enumeradas:

$$\begin{array}{ll}
 (1) & E \rightarrow E + T \\
 (2) & E \rightarrow T \\
 (3) & T \rightarrow T * F \\
 (4) & T \rightarrow F \\
 (5) & T \rightarrow (E) \\
 (6) & F \rightarrow \mathbf{id}
 \end{array}$$

Los códigos para las acciones son:

1. si significa desplazar y meter el estado i en la pila,
2. rq significa reducir mediante la producción enumerada como j ,
3. acc significa aceptar,
4. espacio en blanco significa error.

Observe que el valor de $ir_A[s, a]$ para el terminal a se encuentra en el campo ACCION conectado con la acción de desplazamiento en la entrada a , para el estado s . El campo ir_A proporciona $ir_A[s, A]$ para los no terminales A . Aunque no hemos explicado aún cómo se seleccionaron las entradas para la figura 4.37, en breve trataremos con esta cuestión.

ESTADO	ACCIÓN						ir_A		
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Figura 4.37: Tabla de análisis sintáctico para la gramática de expresiones

En la entrada **id * id + id**, la secuencia del contenido de la pila y de la entrada se muestra en la figura 4.38. Para fines de claridad, también se muestran las secuencias de los símbolos gramaticales que corresponden a los estados contenidos en la pila. Por ejemplo, en la línea (1) el analizador LR se encuentra en el estado 0, el estado inicial sin símbolo gramatical, y con **id** el primer símbolo de entrada. La acción en la fila 0 y la columna **id** del campo acción de la figura 4.37 es s5, lo cual significa desplazar metiendo el estado 5. Esto es lo que ha ocurrido en la línea (2) se ha metido en la pila el símbolo de estado 5, mientras que **id** se ha eliminado de la entrada.

Después, ***** se convierte en el símbolo de entrada actual, y la acción del estado 5 sobre la entrada ***** es reducir mediante $F \rightarrow \mathbf{id}$. Se saca un símbolo de estado de la pila. Después se expone el estado 0. Como el ir_A del estado 0 en F es 3, el estado 3 se mete a la pila. Ahora tenemos la configuración de la línea (3). Cada uno de los movimientos restantes se determina en forma similar. \square

4.6.4 Construcción de tablas de análisis sintáctico SLR

El método SLR para construir tablas de análisis sintáctico es un buen punto inicial para estudiar el análisis sintáctico LR. Nos referiremos a la tabla de análisis sintáctico construida por este método como una tabla SLR, y a un analizador sintáctico LR que utiliza una tabla de análisis sintáctico SLR como un analizador sintáctico SLR. Los otros dos métodos aumentan el método SLR con información de anticipación.

El método SLR empieza con elementos LR(0) y un autómata LR(0), que presentamos en la sección 4.5. Es decir, dada una gramática G , la aumentamos para producir G' , con un nuevo símbolo inicial S' . A partir de G' construimos a C , la colección canónica de conjuntos de elementos para G' , junto con la función ir_A.

	PILA	SÍMBOLOS	ENTRADA	ACCIÓN
(1)	0		$\mathbf{id} * \mathbf{id} + \mathbf{id} \$$	desplazar
(2)	0 5	\mathbf{id}	$* \mathbf{id} + \mathbf{id} \$$	reducir mediante $F \rightarrow \mathbf{id}$
(3)	0 3	F	$* \mathbf{id} + \mathbf{id} \$$	reducir mediante $T \rightarrow F$
(4)	0 2	T	$* \mathbf{id} + \mathbf{id} \$$	desplazar
(5)	0 2 7	$T *$	$\mathbf{id} + \mathbf{id} \$$	desplazar
(6)	0 2 7 5	$T * \mathbf{id}$	$+ \mathbf{id} \$$	reducir mediante $F \rightarrow \mathbf{id}$
(7)	0 2 7 10	$T * F$	$+ \mathbf{id} \$$	reducir mediante $T \rightarrow T * F$
(8)	0 2	T	$+ \mathbf{id} \$$	reducir mediante $E \rightarrow T$
(9)	0 1	E	$+ \mathbf{id} \$$	desplazar
(10)	0 1 6	$E +$	$\mathbf{id} \$$	desplazar
(11)	0 1 6 5	$E + \mathbf{id}$	$\$$	reducir mediante $F \rightarrow \mathbf{id}$
(12)	0 1 6 3	$E + F$	$\$$	reducir mediante $T \rightarrow F$
(13)	0 1 6 9	$E + T$	$\$$	reducir mediante $E \rightarrow E + T$
(14)	0 1	E	$\$$	aceptar

Figura 4.38: Movimientos de un analizador sintáctico LR con $\mathbf{id} * \mathbf{id} + \mathbf{id}$

Después, las entradas ACCION e ir_A en la tabla de análisis sintáctico se construyen utilizando el siguiente algoritmo. Para ello, requerimos conocer SIGUIENTE(A) para cada no terminal A de una gramática (vea la sección 4.4).

Algoritmo 4.46: Construcción de una tabla de análisis sintáctico SLR.

ENTRADA: Una gramática aumentada G' .

SALIDA: Las funciones ACCION e ir_A para G' de la tabla de análisis sintáctico SLR.

MÉTODO:

1. Construir $C = \{ I_0, I_1, \dots, I_n \}$, la colección de conjuntos de elementos LR(0) para G' .
2. El estado i se construye a partir de I_i . Las acciones de análisis sintáctico para el estado i se determinan de la siguiente forma:
 - (a) Si $[A \rightarrow \alpha \cdot a\beta]$ está en I_i e $\text{ir}_A(I_i, a) = I_j$, entonces establecer $\text{ACCION}[i, a]$ a “desplazar j ”. Aquí, a debe ser una terminal.
 - (b) Si $[A \rightarrow \alpha \cdot]$ está en I_i , entonces establecer $\text{ACCION}[i, a]$ a “reducir $A \rightarrow \alpha$ ” para toda a en $\text{SIGUIENTE}(A)$; aquí, A tal vez no sea S' .
 - (c) Si $[S' \rightarrow S \cdot]$ está en I_i , entonces establecer $\text{ACCION}[i, \$]$ a “aceptar”.

Si resulta cualquier acción conflictiva debido a las reglas anteriores, decimos que la gramática no es SLR(1). El algoritmo no produce un analizador sintáctico en este caso.

3. Las transiciones de ir_A para el estado i se construyen para todos los no terminales A , usando la regla: Si $\text{ir_A}(I_i, A) = I_j$, entonces $\text{ir_A}[i, A] = j$.
4. Todas las entradas que no estén definidas por las reglas (2) y (3) se dejan como “error”.
5. El estado inicial del analizador sintáctico es el que se construyó a partir del conjunto de elementos que contienen $[S' \rightarrow \cdot S]$.

□

A la tabla de análisis sintáctico que consiste en las funciones ACCION e ir_A determinadas por el algoritmo 4.46 se le conoce como la *tabla SLR(1) para G* . A un analizador sintáctico LR que utiliza la tabla SLR(1) para G se le conoce como analizador sintáctico SLR(1) para G , y a una gramática que tiene una tabla de análisis sintáctico SLR(1) se le conoce como *SLR(1)*. Por lo general, omitimos el “(1)” después de “SLR”, ya que no trataremos aquí con los analizadores sintácticos que tienen más de un símbolo de preanálisis.

Ejemplo 4.47: Vamos a construir la tabla SLR para la gramática de expresiones aumentada. La colección canónica de conjuntos de elementos $\text{LR}(0)$ para la gramática se mostró en la figura 4.31. Primero consideremos el conjunto de elementos I_0 :

$$\begin{aligned} E' &\rightarrow \cdot E \\ E &\rightarrow \cdot E + T \\ E &\rightarrow \cdot T \\ T &\rightarrow \cdot T * F \\ T &\rightarrow \cdot F \\ F &\rightarrow \cdot (E) \\ F &\rightarrow \cdot \text{id} \end{aligned}$$

El elemento $F \rightarrow \cdot (E)$ produce la entrada $\text{ACCION}[0, ()] = \text{desplazar } 4$, y el elemento $F \rightarrow \cdot \text{id}$ produce la entrada $\text{ACCION}[0, \text{id}] = \text{desplazar } 5$. Los demás elementos en I_0 no producen ninguna acción. Ahora consideremos I_1 :

$$\begin{aligned} E' &\rightarrow E \cdot \\ E &\rightarrow E \cdot + T \end{aligned}$$

El primer elemento produce $\text{ACCION}[1, \$] = \text{aceptar}$, y el segundo produce $\text{ACCION}[1, +] = \text{desplazar } 6$. Ahora consideremos I_2 :

$$\begin{aligned} E &\rightarrow T \cdot \\ T &\rightarrow T \cdot * F \end{aligned}$$

Como $\text{SIGUIENTE}(E) = \{\$\, , \, +\, , \,)\}$, el primer elemento produce lo siguiente:

$$\text{ACCION}[2, \$] = \text{ACCION}[2, +] = \text{ACCION}[2, ()] = \text{reducir } E \rightarrow T$$

El segundo elemento produce $\text{ACCION}[2, *] = \text{desplazar } 7$. Si continuamos de esta forma obtendremos las tablas ACCION y ir_A que se muestran en la figura 4.31. En esa figura, los números de las producciones en las acciones de reducción son los mismos que el orden en el que aparecen en la gramática original (4.1). Es decir, $E \rightarrow E + T$ es el número 1, $E \rightarrow T$ es 2, y así sucesivamente. □

Ejemplo 4.48: Cada una de las gramáticas SLR(1) no tiene ambigüedad, pero hay muchas gramáticas sin ambigüedad que no son SLR(1). Considere la gramática con las siguientes producciones:

$$\begin{array}{lcl} S & \rightarrow & L = R \mid R \\ L & \rightarrow & *R \mid \mathbf{id} \\ R & \rightarrow & L \end{array} \quad (4.49)$$

Consideremos que L y R representan *l-value* y *r-value*, respectivamente, y que $*$ es un operador que indica “el contenido de”.⁵ La colección canónica de conjuntos de elementos LR(0) para la gramática (4.49) se muestra en la figura 4.39.

$$\begin{array}{llll} I_0: & S' \rightarrow \cdot S & I_5: & L \rightarrow \mathbf{id} \cdot \\ & S \rightarrow \cdot L = R & & \\ & S \rightarrow \cdot R & I_6: & S \rightarrow L = \cdot R \\ & L \rightarrow \cdot * R & & R \rightarrow \cdot L \\ & L \rightarrow \cdot \mathbf{id} & & L \rightarrow \cdot * R \\ & R \rightarrow \cdot L & & L \rightarrow \cdot \mathbf{id} \\ \\ I_1: & S' \rightarrow S \cdot & I_7: & L \rightarrow * R \cdot \\ \\ I_2: & S \rightarrow L \cdot = R & I_8: & R \rightarrow L \cdot \\ & R \rightarrow L \cdot & & \\ \\ I_3: & S \rightarrow R \cdot & I_9: & S \rightarrow L = R \cdot \\ \\ I_4: & L \rightarrow \cdot * R & & \end{array}$$

Figura 4.39: Colección LR(0) canónica para la gramática (4.49)

Considere el conjunto de elementos I_2 . El primer elemento en este conjunto hace que ACCION[2, =] sea “desplazar 6”. Como SIGUIENTE(R) contiene $=$ (para ver por qué, considere la derivación $S \Rightarrow L = R \Rightarrow *R = R$), el segundo elemento establece ACCION[2, =] a “reducir $R \rightarrow L$ ”. Como hay tanto una entrada de desplazamiento como una de reducción en ACCION[2, =], el estado 2 tiene un conflicto de desplazamiento/reducción con el símbolo de entrada $=$.

La gramática (4.49) no es ambigua. Este conflicto de desplazamiento/reducción surge del hecho de que el método de construcción del analizador sintáctico SLR no es lo bastante poderoso como para recordar el suficiente contexto a la izquierda para decidir qué acción debe realizar el analizador sintáctico sobre la entrada $=$, habiendo visto una cadena que puede reducirse a L . Los métodos canónico y LALR, que veremos a continuación, tendrán éxito en una colección más extensa de gramáticas, incluyendo la gramática (4.49). Sin embargo, observe que

⁵Al igual que en la sección 2.8.3, un *l-value* designa una ubicación y un *r-value* es un valor que puede almacenarse en una ubicación.

hay gramáticas sin ambigüedad para las cuales todos los métodos de construcción de analizadores sintácticos LR producirán una tabla de acciones de análisis sintáctico con conflictos. Por fortuna, dichas gramáticas pueden, por lo general, evitarse en las aplicaciones de los lenguajes de programación. \square

4.6.5 Prefijos viables

¿Por qué pueden usarse los autómatas LR(0) para realizar decisiones de desplazamiento-reducción? El autómata LR(0) para una gramática caracteriza las cadenas de símbolos gramaticales que pueden aparecer en la pila de un analizador sintáctico de desplazamiento-reducción para la gramática. El contenido de la pila debe ser un prefijo de una forma de frase derecha. Si la pila contiene a α y el resto de la entrada es x , entonces una secuencia de reducciones llevará a αx a S . En términos de derivaciones, $S \xrightarrow[rm]^* \alpha x$.

Sin embargo, no todos los prefijos de las formas de frases derechas pueden aparecer en la pila, ya que el analizador sintáctico no debe desplazar más allá del mango. Por ejemplo, suponga que:

$$E \xrightarrow[rm]^* F * \mathbf{id} \xrightarrow[rm]^* (E) * \mathbf{id}$$

Entonces, en diversos momentos durante el análisis sintáctico, la pila contendrá $($, (E) y (E) , pero no debe contener $(E)^*$, ya que (E) es un mango, que el analizador sintáctico debe reducir a F antes de desplazar a $*$.

Los prefijos de las formas de frases derechas que pueden aparecer en la pila de un analizador sintáctico de desplazamiento-reducción se llaman *prefijos viables*. Se definen de la siguiente manera: un prefijo viable es un prefijo de una forma de frase derecha que no continúa más allá del extremo derecho del mango por la derecha de esa forma de frase. Mediante esta definición, siempre es posible agregar símbolos terminales al final de un prefijo viable para obtener una forma de frase derecha.

El análisis sintáctico SLR se basa en el hecho de que los autómatas LR(0) reconocen los prefijos viables. Decimos que el elemento $A \rightarrow \beta_1 \cdot \beta_2$ es *válido* para un prefijo viable $\alpha \beta_1$ si hay una derivación $S' \xrightarrow[rm]^* \alpha Aw \xrightarrow[rm]^* \alpha \beta_1 \beta_2 w$. En general, un elemento será válido para muchos prefijos viables.

El hecho de que $A \rightarrow \beta_1 \cdot \beta_2$ sea válido para $\alpha \beta_1$ nos dice mucho acerca de si debemos desplazar o reducir cuando encontramos a $\alpha \beta_1$ en la pila de análisis sintáctico. En especial, si $\beta_2 \neq \epsilon$, entonces sugiere que no hemos desplazado aún el mango hacia la pila, por lo que el desplazamiento es nuestro siguiente movimiento. Si $\beta_2 = \epsilon$, entonces parece que $A \rightarrow \beta_1$ es el mango, y debemos reducir mediante esta producción. Desde luego que dos elementos válidos pueden indicarnos que debemos hacer distintas cosas para el mismo prefijo viable. Podemos resolver algunos de estos conflictos analizando el siguiente símbolo de entrada, y otros podemos resolverlos mediante los métodos de la sección 4.8, pero no debemos suponer que todos los conflictos de acciones de análisis sintáctico pueden resolverse si se aplica el método LR a una gramática arbitraria.

Podemos calcular con facilidad el conjunto de elementos válidos para cada prefijo viable que puede aparecer en la pila de un analizador sintáctico LR. De hecho, un teorema central de la teoría de análisis sintáctico LR nos dice que el conjunto de elementos válidos para un prefijo viable γ es exactamente el conjunto de elementos a los que se llega desde el estado inicial,

Los elementos como estados de un AFN

Podemos construir un autómata finito no determinista N para reconocer prefijos viables si tratamos a los mismos elementos como estados. Hay una transición desde $A \rightarrow \alpha \cdot X\beta$ hacia $A \rightarrow \alpha X \cdot \beta$ etiquetada como X , y hay una transición desde $A \rightarrow \alpha \cdot B\beta$ hacia $B \rightarrow \cdot \gamma$ etiquetada como ϵ . Entonces, $\text{CERRADURA}(I)$ para el conjunto de elementos (estados de N) I es exactamente el cierre ϵ de un conjunto de estados de un AFN definidos en la sección 3.7.1. Por ende, $\text{ir_A}(I, X)$ proporciona la transición proveniente de I en el símbolo X del AFD construido a partir de N mediante la construcción del subconjunto. Si lo vemos de esta forma, el procedimiento $\text{elementos}(G')$ en la figura 4.33 es sólo la construcción del mismo subconjunto que se aplica al AFN N , con los elementos como estados.

a lo largo de la ruta etiquetada como γ en el autómata LR(0) para la gramática. En esencia, el conjunto de elementos válidos abarca toda la información útil que puede deducirse de la pila. Aunque aquí no demostraremos este teorema, vamos a ver un ejemplo.

Ejemplo 4.50: Vamos a considerar de nuevo la gramática de expresiones aumentada, cuyos conjuntos de elementos y la función ir_A se exhiben en la figura 4.31. Sin duda, la cadena $E + T^*$ es un prefijo viable de la gramática. El autómata de la figura 4.31 se encontrará en el estado 7, después de haber leído $E + T^*$. El estado 7 contiene los siguientes elementos:

$$\begin{aligned} T &\rightarrow T^* \cdot F \\ F &\rightarrow \cdot(E) \\ F &\rightarrow \cdot\text{id} \end{aligned}$$

que son precisamente los elementos válidos para $E + T^*$. Para ver por qué, considere las siguientes tres derivaciones por la derecha:

$$\begin{array}{lll} E' \xrightarrow{rm} E & E' \xrightarrow{rm} E & E' \xrightarrow{rm} E \\ \xrightarrow{rm} E + T & \xrightarrow{rm} E + T & \xrightarrow{rm} E + T \\ \xrightarrow{rm} E + T^* F & \xrightarrow{rm} E + T^* F & \xrightarrow{rm} E + T^* F \\ \xrightarrow{rm} E + T^* (E) & \xrightarrow{rm} E + T^* (E) & \xrightarrow{rm} E + T^* \text{id} \end{array}$$

La primera derivación muestra la validez de $T \rightarrow T^* \cdot F$, la segunda muestra la validez de $F \rightarrow \cdot(E)$, y la tercera muestra la validez de $F \rightarrow \cdot\text{id}$. Se puede mostrar que no hay otros elementos válidos para $E + T^*$, aunque aquí no demostraremos ese hecho. \square

4.6.6 Ejercicios para la sección 4.6

Ejercicio 4.6.1: Describa todos los prefijos viables para las siguientes gramáticas:

- La gramática $S \rightarrow 0 S 1 \mid 0 1$ del ejercicio 4.2.2(a).

- ! b) La gramática $S \rightarrow S S + | S S * | a$ del ejercicio 4.2.1.
- ! c) La gramática $S \rightarrow S (S) | \epsilon$ del ejercicio 4.2.2(c).

Ejercicio 4.6.2: Construya los conjuntos SLR de elementos para la gramática (aumentada) del ejercicio 4.2.1. Calcule la función ir_A para estos conjuntos de elementos. Muestre la tabla de análisis sintáctico para esta gramática. ¿Es una gramática SLR?

Ejercicio 4.6.3: Muestre las acciones de su tabla de análisis sintáctico del ejercicio 4.6.2 sobre la entrada $aa * a +$.

Ejercicio 4.6.4: Para cada una de las gramáticas (aumentadas) del ejercicio 4.2.2(a)-(g):

- Construya los conjuntos SLR de elementos y su función ir_A .
- Indique cualquier conflicto de acciones en sus conjuntos de elementos.
- Construya la tabla de análisis sintáctico SLR, si es que existe.

Ejercicio 4.6.5: Muestre que la siguiente gramática:

$$\begin{array}{lcl} S & \rightarrow & A \ a \ A \ b \ | \ B \ b \ B \ a \\ A & \rightarrow & \epsilon \\ B & \rightarrow & \epsilon \end{array}$$

es LL(1), pero no SLR(1).

Ejercicio 4.6.6: Muestre que la siguiente gramática:

$$\begin{array}{lcl} S & \rightarrow & S \ A \ | \ A \\ A & \rightarrow & a \end{array}$$

es SLR(1), pero no LL(1).

!! **Ejercicio 4.6.7:** Considere la familia de gramáticas G_n definidas por:

$$\begin{array}{lcl} S & \rightarrow & A_i \ b_i & \text{para } 1 \leq i \leq n \\ A_i & \rightarrow & a_j \ A_i \ | \ a_j & \text{para } 1 \leq i, j \leq n \text{ e } i \neq j \end{array}$$

Muestre que:

- G_n tiene $2n^2 - 2$ producciones.
- G_n tiene $2n^2 + n^2 + n$ conjuntos de elementos LR(0).
- G_n es SLR(1).

¿Qué dice este análisis acerca de la extensión que pueden llegar a tener los analizadores sintácticos LR?

! Ejercicio 4.6.8: Sugerimos que los elementos individuales pudieran considerarse como estados de un autómata finito no determinista, mientras que los conjuntos de elementos válidos son los estados de un autómata finito determinista (vea el recuadro titulado “Los elementos como estados de un AFN” en la sección 4.6.5). Para la gramática $S \rightarrow S S + | S S * | a$ del ejercicio 4.2.1:

- Dibuje el diagrama de transición (AFN) para los elementos válidos de esta gramática, de acuerdo a la regla que se proporciona en el recuadro antes citado.
- Aplique la construcción de subconjuntos (Algoritmo 3.20) a su AFN, a partir de (a). ¿Cómo se compara el AFD resultante con el conjunto de elementos LR(0) para la gramática?
- Muestre que en todos los casos, la construcción de subconjuntos que se aplica al AFN que proviene de los elementos válidos para una gramática produce los conjuntos LR(0) de elementos.

! Ejercicio 4.6.9: La siguiente es una gramática ambigua:

$$\begin{array}{l} S \rightarrow A S | b \\ A \rightarrow S A | a \end{array}$$

Construya para esta gramática su colección de conjuntos de elementos LR(0). Si tratamos de construir una tabla de análisis sintáctico LR para la gramática, hay ciertas acciones en conflicto. ¿Qué son? Suponga que tratamos de usar la tabla de análisis sintáctico eligiendo en forma no determinista una posible acción, cada vez que haya un conflicto. Muestre todas las posibles secuencias de acciones con la entrada *bab*.

4.7 Analizadores sintácticos LR más poderosos

En esta sección vamos a extender las técnicas anteriores de análisis sintáctico LR, para usar un símbolo de preanálisis en la entrada. Hay dos métodos distintos:

- El método “LR canónico”, o simplemente “LR”, que utiliza al máximo el (los) símbolo(s) de preanálisis. Este método utiliza un extenso conjunto de elementos, conocidos como elementos LR(1).
- El método “LR con símbolo de preanálisis” o “LALR(lookahead LR)”, que se basa en los conjuntos de elementos LR(0), y tiene mucho menos estados que los analizadores sintácticos comunes, basados en los elementos LR(1). Si introducimos con cuidado lecturas anticipadas en los elementos LR(0), podemos manejar muchas gramáticas más con el método LALR que con el SLR, y construir tablas de análisis sintáctico que no sean más grandes que las tablas SLR. LALR es el método de elección en la mayoría de las situaciones.

Después de presentar ambos métodos, concluiremos con una explicación acerca de cómo compactar las tablas de análisis sintáctico LR para los entornos con memoria limitada.

4.7.1 Elementos LR(1) canónicos

Ahora presentaremos la técnica más general para construir una tabla de análisis sintáctico LR a partir de una gramática. Recuerde que en el método SLR, el estado i llama a la reducción mediante $A \rightarrow \alpha$ si el conjunto de elementos I_i contiene el elemento $[A \rightarrow \alpha \cdot]$ y α se encuentra en $\text{SIGUIENTE}(A)$. No obstante, en algunas situaciones cuando el estado i aparece en la parte superior de la pila, el prefijo viable $\beta\alpha$ en la pila es tal que βA no puede ir seguida de a en ninguna forma de frase derecha. Por ende, la reducción mediante $A \rightarrow \alpha$ debe ser inválida con la entrada a .

Ejemplo 4.51: Vamos a reconsiderar el ejemplo 4.48, en donde en el estado 2 teníamos el elemento $R \rightarrow L$, el cual podía corresponder a la $A \rightarrow \alpha$ anterior, y a podía ser el signo $=$, que se encuentra en $\text{SIGUIENTE}(R)$. Por ende, el analizador sintáctico SLR llama a la reducción mediante $R \rightarrow L$ en el estado 2, con $=$ como el siguiente símbolo de entrada (también se llama a la acción de desplazamiento, debido al elemento $S \rightarrow L = R$ en el estado 2). Sin embargo, no hay forma de frase derecha de la gramática en el ejemplo 4.48 que empiece como $R = \dots$. Por lo tanto, el estado 2, que es el estado correspondiente al prefijo viable L solamente, en realidad no debería llamar a la reducción de esa L a R . \square

Es posible transportar más información en el estado, que nos permita descartar algunas de estas reducciones inválidas mediante $A \rightarrow \alpha$. Al dividir estados según sea necesario, podemos hacer que cada estado de un analizador sintáctico LR indique con exactitud qué símbolos de entrada pueden ir después de un mango α para el cual haya una posible reducción a A .

La información adicional se incorpora al estado mediante la redefinición de elementos, para que incluyan un símbolo terminal como un segundo componente. La forma general de un elemento se convierte en $[A \rightarrow \alpha \cdot \beta, a]$, en donde $A \rightarrow \alpha\beta$ es una producción y a es un terminal o el delimitador $\$$ derecho. A un objeto de este tipo le llamamos *elemento LR(1)*. El 1 se refiere a la longitud del segundo componente, conocido como la *lectura anticipada* del elemento.⁶ La lectura anticipada no tiene efecto sobre un elemento de la forma $[A \rightarrow \alpha \cdot \beta, a]$, en donde β no es ϵ , pero un elemento de la forma $[A \rightarrow \alpha \cdot, a]$ llama a una reducción mediante $A \rightarrow \alpha$ sólo si el siguiente símbolo de entrada es a . Por ende, nos vemos obligados a reducir mediante $A \rightarrow \alpha$ sólo con esos símbolos de entrada a para los cuales $[A \rightarrow \alpha \cdot, a]$ es un elemento LR(1) en el estado en la parte superior de la pila. El conjunto de tales a s siempre será un subconjunto de $\text{SIGUIENTE}(A)$, pero podría ser un subconjunto propio, como en el ejemplo 4.51.

De manera formal, decimos que el elemento LR(1) $[A \rightarrow \alpha \cdot \beta, a]$ es *válido* para un prefijo viable γ si hay una derivación $S \xrightarrow[\text{rm}]{*} \delta A w \xrightarrow[\text{rm}]{*} \delta \alpha \beta w$, en donde

1. $\gamma = \delta\alpha$, y
2. a es el primer símbolo de w , o w es ϵ y, a es $\$$.

Ejemplo 4.52: Consideremos la siguiente gramática:

⁶ Desde luego que son posibles las lecturas anticipadas que sean cadenas de una longitud mayor a uno, pero no las consideraremos en este libro.

$$\begin{aligned} S &\rightarrow B \ B \\ B &\rightarrow a \ B \mid b \end{aligned}$$

Hay una derivación por la derecha $S \xrightarrow{*} aaBab \xrightarrow{rm} aaaBab$. Podemos ver que el elemento $[B \rightarrow a \cdot B, a]$ es válido para un prefijo viable $\gamma = aaa$, si dejamos que $\delta = aa$, $A = B$, $w = ab$ $\alpha = a$ y $\beta = B$ en la definición anterior. También hay una derivación por la derecha $S \xrightarrow{*} BaB \xrightarrow{rm} BaaB$. De esta derivación podemos ver que el elemento $[B \rightarrow \alpha \cdot B, \$]$ es válido para el prefijo viable Baa . \square

4.7.2 Construcción de conjuntos de elementos LR(1)

El método para construir la colección de conjuntos de elementos LR(1) válidos es en esencia el mismo que para construir la colección canónica de conjuntos de elementos LR(0). Sólo necesitamos modificar los dos procedimientos CERRADURA e ir_A.

```

ConjuntoDeElementos CERRADURA( $I$ ) {
    repeat
        for ( cada elemento  $[A \rightarrow \alpha \cdot B\beta, a]$  en  $I$  )
            for ( cada producción  $B \rightarrow \gamma$  en  $G'$  )
                for ( cada terminal  $b$  en PRIMERO( $\beta a$ ) )
                    agregar  $[B \rightarrow \cdot \gamma, b]$  al conjunto  $I$ ;
    until no se agreguen más elementos a  $I$ ;
    return  $I$ ;
}

ConjuntoDeElementos ir_A( $I, X$ ) {
    inicializar  $J$  para que sea el conjunto vacío;
    for ( cada elemento  $[A \rightarrow \alpha \cdot X\beta, a]$  en  $I$  )
        agregar el elemento  $[A \rightarrow \alpha X \cdot \beta, a]$  al conjunto  $J$ ;
    return CERRADURA( $J$ );
}

void elementos( $G'$ ) {
    inicializar  $C$  a CERRADURA( $\{[S' \rightarrow \cdot S, \$]\}$ );
    repeat
        for ( cada conjunto de elementos  $I$  en  $C$  )
            for ( cada símbolo gramatical  $X$  )
                if ( ir_A( $I, X$ ) no está vacío y no está en  $C$  )
                    agregar ir_A( $I, X$ ) a  $C$ ;
    until no se agreguen nuevos conjuntos de elementos a  $C$ ;
}

```

Figura 4.40: Construcción de conjuntos de elementos LR(1) para la gramática G'

Para apreciar la nueva definición de la operación CERRADURA, en especial, por qué b debe estar en $\text{PRIMERO}(\beta a)$, considere un elemento de la forma $[A \rightarrow \alpha B \beta, a]$ en el conjunto de elementos válido para cierto prefijo viable γ . Entonces hay una derivación por la derecha $S \xrightarrow{rm} \delta A a x \xrightarrow{rm} \delta \alpha B \beta a x$, en donde $\gamma = \delta \alpha$. Suponga que $\beta a x$ deriva a la cadena de terminales by . Entonces, para cada producción de la forma $B \rightarrow \eta$ para cierta η , tenemos la derivación $S \xrightarrow{rm} \gamma B b y \xrightarrow{rm} \gamma \eta b y$. Por ende, $[B \rightarrow \cdot \eta, b]$ es válida para γ . Observe que b puede ser el primer terminal derivado a partir de β , o que es posible que β derive a ϵ en la derivación $\beta a x \xrightarrow{rm} b y$ y, por lo tanto, b puede ser a . Para resumir ambas posibilidades, decimos que b puede ser cualquier terminal en $\text{PRIMERO}(\beta a x)$, en donde PRIMERO es la función de la sección 4.4. Observe que x no puede contener la primera terminal de by , por lo que $\text{PRIMERO}(\beta a x) = \text{PRIMERO}(\beta a)$. Ahora proporcionaremos la construcción de los conjuntos de elementos LR(1).

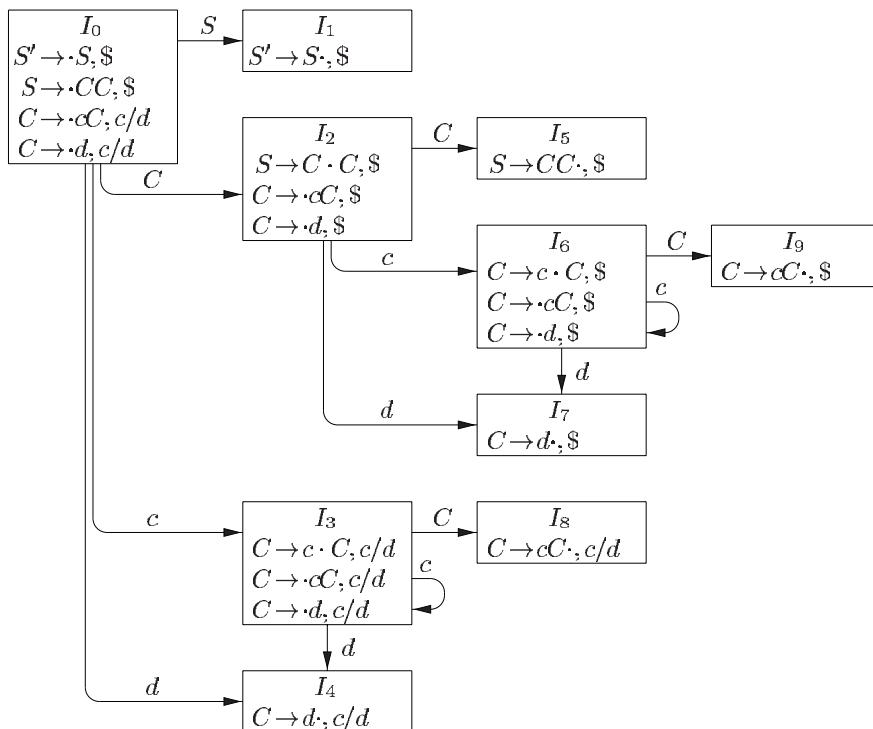


Figura 4.41: El gráfico de ir_A para la gramática (4.55)

Algoritmo 4.53: Construcción de los conjuntos de elementos LR(1).

ENTRADA: Una gramática aumentada G' .

SALIDA: Los conjuntos de elementos LR(1) que son el conjunto de elementos válido para uno o más prefijos viables de G' .

MÉTODO: Los procedimientos CERRADURA e ir_A, y la rutina principal *elementos* para construir los conjuntos de elementos se mostraron en la figura 4.40. \square

Ejemplo 4.54: Considere la siguiente gramática aumentada:

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & C \ C \\ C & \rightarrow & c \ C \mid d \end{array} \quad (4.55)$$

Empezamos por calcular la cerradura de $\{[S' \rightarrow \cdot S, \$]\}$. Relacionamos el elemento $[S' \rightarrow \cdot S, \$]$ con el elemento $[A \rightarrow \alpha \cdot B \beta, a]$ en el procedimiento CERRADURA. Es decir, $A = S'$, $\alpha = \epsilon$, $B = S$, $\beta = \epsilon$ y $a = \$$. La función CERRADURA nos indica que debemos agregar $[B \rightarrow \cdot \gamma, b]$ para cada producción $B \rightarrow y$ y la terminal b en PRIMERO(βa). En términos de la gramática actual, $B \rightarrow \gamma$ debe ser $S \rightarrow CC$, y como β es ϵ y a es $\$$, b sólo puede ser $\$$. Por ende, agregamos $[S \rightarrow \cdot CC, \$]$.

Para seguir calculando la cerradura, agregamos todos los elementos $[C \rightarrow \cdot \gamma, b]$ para b en PRIMERO($C\$$). Es decir, si relacionamos $[S \rightarrow \cdot CC, \$]$ con $[A \rightarrow \alpha \cdot B \beta, a]$, tenemos que $A = S$, $\alpha = \epsilon$, $B = C$, $\beta = C$ y $a = \$$. Como C no deriva a la cadena vacía, PRIMERO($C\$$) = PRIMERO(C). Como PRIMERO(C) contiene los terminales c y d , agregamos los elementos $[C \rightarrow \cdot cC, c]$, $[C \rightarrow \cdot cC, d]$, $[C \rightarrow \cdot d, c]$ y $[C \rightarrow \cdot d, d]$. Ninguno de los elementos nuevos tiene un no terminal justo a la derecha del punto, por lo que hemos completado nuestro primer conjunto de elementos LR(1). El conjunto inicial de elementos es:

$$\begin{array}{l} I_0 : \quad S \rightarrow \cdot S, \$ \\ \quad S \rightarrow \cdot CC, \$ \\ \quad C \rightarrow \cdot cC, c/d \\ \quad C \rightarrow \cdot d, c/d \end{array}$$

Hemos omitido los corchetes por conveniencia de notación, y utilizamos la notación $[C \rightarrow \cdot cC, c/d]$ como abreviación para los dos elementos $[C \rightarrow \cdot cC, c]$ y $[C \rightarrow \cdot cC, d]$.

Ahora calculamos $\text{ir}_A(I_0, X)$ para los diversos valores de X . Para $X = S$ debemos la cerradura del elemento $[S' \rightarrow S, \$]$. No es posible una cerradura adicional, ya que el punto está en el extremo derecho. Por ende, tenemos el siguiente conjunto de elementos:

$$I_1 : \quad S' \rightarrow S \cdot, \$$$

Para $X = C$ calculamos la cerradura $[S \rightarrow \cdot CC, \$]$. Agregamos las producciones C con el segundo componente $\$$ y después no podemos agregar más, produciendo lo siguiente:

$$\begin{array}{l} I_2 : \quad S \rightarrow \cdot CC, \$ \\ \quad C \rightarrow \cdot cC, \$ \\ \quad C \rightarrow \cdot d, \$ \end{array}$$

Ahora, dejamos que $X = c$. Debemos calcular la cerradura $\{[C \rightarrow \cdot cC, c/d]\}$. Agregamos las producciones C con el segundo componente c/d , produciendo lo siguiente:

$$I_3 : \begin{aligned} C &\rightarrow c \cdot C, c/d \\ C &\rightarrow \cdot cC, c/d \\ C &\rightarrow \cdot d, c/d \end{aligned}$$

Por último, dejamos que $X = d$, y terminamos con el siguiente conjunto de elementos:

$$I_4 : C \rightarrow d \cdot, c/d$$

Hemos terminado de considerar a ir_A con I_0 . No obtenemos nuevos conjuntos de I_1 , pero I_2 , tiene ir_A en C , c y d . Para $\text{ir_A}(I_2, C)$, obtenemos lo siguiente:

$$I_5 : S \rightarrow CC \cdot, \$$$

sin que se requiera una cerradura. Para calcular $\text{ir_A}(I_2, c)$ tomamos la cerradura de $\{[C \rightarrow c \cdot C, \$]\}$, para obtener lo siguiente:

$$I_6 : \begin{aligned} C &\rightarrow c \cdot C, \$ \\ C &\rightarrow \cdot cC, \$ \\ C &\rightarrow \cdot d, \$ \end{aligned}$$

Observe que I_6 difiere de I_3 sólo en los segundos componentes. Más adelante veremos que es común para ciertos conjuntos de elementos LR(1) que una gramática tenga los mismos primeros componentes y que difieran en sus segundos componentes. Cuando construyamos la colección de conjuntos de elementos LR(0) para la misma gramática, cada conjunto de LR(0) coincidirá con el conjunto de los primeros componentes de uno o más conjuntos de elementos LR(1). Cuando hablemos sobre el análisis sintáctico LALR, veremos más sobre este fenómeno.

Continuando con la función ir_A para I_2 , $\text{ir_A}(I_2, d)$ se ve de la siguiente manera:

$$I_7 : C \rightarrow d \cdot, \$$$

Si pasamos ahora a I_3 , los ir_A de I_3 en c y d son I_3 e I_4 , respectivamente, y $\text{ir_A}(I_3, C)$ es:

$$I_8 : C \rightarrow cC \cdot, c/d$$

I_4 e I_5 no tienen ir_As , ya que todos los elementos tienen sus puntos en el extremo derecho. Los ir_As de I_6 en c y d son I_6 e I_7 , respectivamente, y $\text{ir_A}(I_6, C)$ es:

$$I_9 : C \rightarrow cC \cdot, \$$$

Los conjuntos restantes de elementos no producen mas ir_A , por lo que hemos terminado. La figura 4.41 muestra los diez conjuntos de elementos con sus ir_A . \square

4.7.3 Tablas de análisis sintáctico LR(1) canónico

Ahora proporcionaremos las reglas para construir las funciones ACCION e ir_A de LR(1), a partir de los conjuntos de elementos LR(1). Estas funciones se representan mediante una tabla, como antes. La única diferencia está en los valores de las entradas.

Algoritmo 4.56: Construcción de tablas de análisis sintáctico LR canónico.

ENTRADA: Una gramática aumentada G' .

SALIDA: Las funciones ACCION e ir_A de la tabla de análisis sintáctico LR canónico para G' .

MÉTODO:

1. Construir $C' = \{ I_0, I_1, \dots, I_n \}$, la colección de conjuntos de elementos LR(1) para G' .
2. El estado i del analizador sintáctico se construye a partir de I_i . La acción de análisis sintáctico para el estado i se determina de la siguiente manera.
 - (a) Si $[A \rightarrow \alpha \cdot a\beta, b]$ está en I_i , e $\text{ir_A}(I_i, a) = I_j$, entonces hay que establecer $\text{ACCION}[i, a]$ a “desplazar j ”. Aquí, a debe ser una terminal.
 - (b) Si $[A \rightarrow \alpha \cdot, a]$ está en I_i , $A \neq S'$, entonces hay que establecer $\text{ACCION}[i, a]$ a “reducir $A \rightarrow \alpha$ ”.
 - (c) Si $[S' \rightarrow S \cdot, \$]$ está en I_i , entonces hay que establecer $\text{ACCION}[i, \$]$ a “aceptar”.

Si resulta cualquier acción conflictiva debido a las reglas anteriores, decimos que la gramática no es LR(1). El algoritmo no produce un analizador sintáctico en este caso.

3. Las transiciones ir_A para el estado i se construyen para todos los no terminales A usando la regla: Si $\text{ir_A}(I_i, A) = I_j$, entonces $\text{ir_A}[i, A] = j$.
4. Todas las entradas no definidas por las reglas (2) y (3) se vuelven “error”.
5. El estado inicial del analizador sintáctico es el que se construye a partir del conjunto de elementos que contienen $[S' \rightarrow \cdot S, \$]$.

□

A la tabla que se forma a partir de la acción de análisis sintáctico y las funciones producidas por el Algoritmo 4.44 se le conoce como la tabla de análisis LR(1) *canónica*. Si la función de acción de análisis sintáctico no tiene entradas definidas en forma múltiple, entonces a la gramática dada se le conoce como *gramática LR(1)*. Como antes, omitimos el “(1)” si queda comprendida su función.

Ejemplo 4.57: La tabla de análisis sintáctico canónica para la gramática (4.55) se muestra en la figura 4.42. Las producciones 1, 2 y 3 son $S \rightarrow CC$, $C \rightarrow cC$ y $C \rightarrow d$, respectivamente. □

Cada gramática SLR(1) es una gramática LR(1), pero para una gramática SLR(1) el analizador sintáctico LR canónico puede tener más estados que el analizador sintáctico SLR para la misma gramática. La gramática de los ejemplos anteriores es SLR, y tiene un analizador sintáctico SLR con siete estados, en comparación con los diez de la figura 4.42.

ESTADO	ACCIÓN			ir_A	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Figura 4.42: Tabla de análisis sintáctico canónica para la gramática (4.55)

4.7.4 Construcción de tablas de análisis sintáctico LALR

Ahora presentaremos nuestro último método de construcción de analizadores sintácticos, la técnica LALR (LR con *lectura anticipada*). Este método se utiliza con frecuencia en la práctica, ya que las tablas que se obtienen son considerablemente menores que las tablas LR canónicas, y a pesar de ello la mayoría de las construcciones sintácticas comunes de los lenguajes de programación pueden expresarse en forma conveniente mediante una gramática LALR. Lo mismo es casi válido para las gramáticas SLR, pero hay algunas construcciones que las técnicas SLR no pueden manejar de manera conveniente (vea el ejemplo 4.48).

Para una comparación del tamaño de los analizadores sintácticos, las tablas SLR y LALR para una gramática siempre tienen el mismo número de estados, y este número consiste, por lo general, en cientos de estados, para un lenguaje como C. La tabla LR canónica tendría, por lo general, varios miles de estados para el lenguaje con el mismo tamaño. Por ende, es mucho más fácil y económico construir tablas SLR y LALR que las tablas LR canónicas.

Con el fin de una introducción, consideremos de nuevo la gramática (4.55), cuyos conjuntos de elementos LR(1) se mostraron en la figura 4.41. Tomemos un par de estados con apariencia similar, como I_4 e I_7 . Cada uno de estos estados sólo tiene elementos con el primer componente $C \rightarrow d$. En I_4 , los símbolos de anticipación son c o d ; en I_7 , $\$$ es el único símbolo de anticipación.

Para ver las diferencias entre las funciones de I_4 e I_7 en el analizador sintáctico, observe que la gramática genera el lenguaje regular c^*dc^*d . Al leer una entrada $cc\cdots cdcc\cdots cd$, el analizador sintáctico desplaza el primer grupo de cs y la d subsiguiente, y las mete en la pila, entrando al estado 4 después de leer la d . Después, el analizador llama a una reducción mediante $C \rightarrow d$, siempre y cuando el siguiente símbolo de entrada sea c o d . El requerimiento de que sigue c o d tiene sentido, ya que éstos son los símbolos que podrían empezar cadenas en c^*d . Si $\$$ sigue después de la primera d , tenemos una entrada como ccd , que no está en el lenguaje, y el estado 4 declara en forma correcta un error si $\$$ es la siguiente entrada.

El analizador sintáctico entra al estado 7 después de leer la segunda d . Después, el analizador debe ver a $\$$ en la entrada, o de lo contrario empezó con una cadena que no es de la forma

c*dc*d. Por ende, tiene sentido que el estado 7 deba reducir mediante $C \rightarrow d$ en la entrada \$, y declarar un error en entradas como c o d .

Ahora vamos a sustituir I_4 e I_7 por I_{47} , la unión de I_4 e I_7 , que consiste en el conjunto de tres elementos representados por $[C \rightarrow d, c/d/\$]$. Las transacciones ir_A en d que pasan a I_4 o a I_7 desde I_0, I_2, I_3 e I_6 ahora entran a I_{47} . La acción del estado 47 es reducir en cualquier entrada. El analizador sintáctico revisado se comporta en esencia igual que el original, aunque podría reducir d a C en circunstancias en las que el original declararía un error, por ejemplo, en una entrada como ccd o $cdcdc$. En un momento dado, el error se atrapará; de hecho, se atrapará antes de que se desplacen más símbolos de entrada.

En forma más general, podemos buscar conjuntos de elementos LR(1) que tengan el mismo *corazón*; es decir, el mismo conjunto de primeros componentes, y podemos combinar estos conjuntos con corazones comunes en un solo conjunto de elementos. Por ejemplo, en la figura 4.41, I_4 e I_7 forman dicho par, con el corazón $\{C \rightarrow d\}$. De manera similar, I_3 e I_6 forman otro par, con el corazón $\{C \rightarrow c \cdot C, C \rightarrow \cdot cC, C \rightarrow \cdot d\}$. Hay un par más, I_8 e I_9 , con el corazón común $\{C \rightarrow c \cdot C\}$. Observe que, en general, un corazón es un conjunto de elementos LR(0) para la gramática en cuestión, y que una gramática LR(1) puede producir más de dos conjuntos de elementos con el mismo corazón.

Como el corazón de $\text{ir_A}(I, X)$ depende sólo del corazón de I , las transacciones ir_A de los conjuntos combinados pueden combinarse entre sí mismos. Por ende, no hay problema al revisar la función ir_A a medida que combinamos conjuntos de elementos. Las funciones activas se modifican para reflejar las acciones sin error de todos los conjuntos de elementos en la combinación.

Suponga que tenemos una gramática LR(1), es decir, una cuyos conjuntos de elementos LR(1) no produzcan conflictos de acciones en el análisis sintáctico. Si sustituimos todos los estados que tengan el mismo corazón con su unión, es posible que la unión resultante tenga un conflicto, pero es poco probable debido a lo siguiente: Suponga que en la unión hay un conflicto en el símbolo anticipado, debido a que hay un elemento $[A \rightarrow \alpha \cdot, a]$ que llama a una reducción mediante $A \rightarrow \alpha$, y que hay otro elemento $[B \rightarrow \beta \cdot \gamma, b]$ que llama a un desplazamiento. Entonces, cierto conjunto de elementos a partir del cual se formó la unión tiene el elemento $[A \rightarrow \alpha \cdot, a]$, y como los corazones de todos estos estados son iguales, debe tener un elemento $[B \rightarrow \beta \cdot \gamma, c]$ para alguna c . Pero entonces, este estado tiene el mismo conflicto de desplazamiento/reducción en a , y la gramática no era LR(1) como supusimos. Por ende, la combinación de estados con corazones comunes nunca podrá producir un conflicto de desplazamiento/reducción que no haya estado presente en uno de los estados originales, ya que las acciones de desplazamiento dependen sólo del corazón, y no del símbolo anticipado.

Sin embargo, es posible que una combinación produzca un conflicto de reducción/reducción, como se muestra en el siguiente ejemplo.

Ejemplo 4.58: Considere la siguiente gramática:

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & a A d \mid b B d \mid a B e \mid b A e \\ A & \rightarrow & c \\ B & \rightarrow & c \end{array}$$

la cual genera las cuatro cadenas acd , ace , bcd y bce . El lector puede comprobar que la gramática es LR(1) mediante la construcción de los conjuntos de elementos. Al hacer esto, encontramos

el conjunto de elementos $\{[A \rightarrow c, d], [B \rightarrow c, e]\}$ válido para el prefijo viable ac y $\{[A \rightarrow c, e], [B \rightarrow c, d]\}$ válido para bc . Ninguno de estos conjuntos tiene un conflicto, y sus corazones son iguales. Sin embargo, su unión, que es:

$$\begin{array}{l} A \rightarrow c, d/e \\ B \rightarrow c, d/e \end{array}$$

genera un conflicto de reducción/reducción, ya que las reducciones mediante $A \rightarrow c$ y $B \rightarrow c$ se llaman para las entradas d y e . \square

Ahora estamos preparados para proporcionar el primero de dos algoritmos de construcción de tablas LALR. La idea general es construir los conjuntos de elementos LR(1), y si no surgen conflictos, combinar los conjuntos con corazones comunes. Después, construiremos la tabla de análisis sintáctico a partir de la colección de conjuntos de elementos combinados. El método que vamos a describir sirve principalmente como una definición de las gramáticas LALR(1). El proceso de construir la colección completa de conjuntos de elementos LR(1) requiere demasiado espacio y tiempo como para que sea útil en la práctica.

Algoritmo 4.59: Una construcción de tablas LALR sencilla, pero que consume espacio.

ENTRADA: Una gramática aumentada G' .

SALIDA: Las funciones de la tabla de análisis sintáctico LALR ACCION e ir_A para G' .

MÉTODO:

1. Construir $C = \{I_0, I_1, \dots, I_n\}$, la colección de conjuntos de elementos LR(1).
2. Para cada corazón presente entre el conjunto de elementos LR(1), buscar todos los conjuntos que tengan ese corazón y sustituir estos conjuntos por su unión.
3. Dejar que $C' = \{J_0, J_1, \dots, J_m\}$ sean los conjuntos resultantes de elementos LR(1). Las acciones de análisis sintáctico para el estado i se construyen a partir de J_i , de la misma forma que en el Algoritmo 4.56. Si hay un conflicto de acciones en el análisis sintáctico, el algoritmo no produce un analizador sintáctico y decimos que la gramática no es LALR(1).
4. La tabla ir_A se construye de la siguiente manera. Si J es la unión de uno o más conjuntos de elementos LR(1), es decir, $J = I_1 \cap I_2 \cap \dots \cap I_k$, entonces los corazones de $\text{ir_A}(I_1, X)$, $\text{ir_A}(I_2, X)$, ..., $\text{ir_A}(I_k, X)$ son iguales, ya que I_1, I_2, \dots, I_k tienen el mismo corazón. Dejar que K sea la unión de todos los conjuntos de elementos que tienen el mismo corazón que $\text{ir_A}(I_1, X)$. Entonces, $\text{ir_A}(J, X) = K$.

\square

A la tabla producida por el algoritmo 4.59 se le conoce como la *tabla de análisis sintáctico LALR* para G . Si no hay conflictos de acciones en el análisis sintáctico, entonces se dice que la gramática dada es una *gramática LALR(1)*. A la colección de conjuntos de elementos que se construye en el paso (3) se le conoce como *colección LALR(1)*.

Ejemplo 4.60: Consideré de nuevo la gramática (4.55), cuyo gráfico de ir_A se mostró en la figura 4.41. Como dijimos antes, hay tres pares de conjuntos de elementos que pueden combinarse. I_3 e I_6 se sustituyen por su unión:

$$\begin{aligned} I_{36}: \quad & C \rightarrow c \cdot C, c/d/\$ \\ & C \rightarrow \cdot cC, c/d/\$ \\ & C \rightarrow \cdot d, c/d/\$ \end{aligned}$$

I_4 e I_7 se sustituyen por su unión:

$$I_{47}: \quad C \rightarrow d \cdot, c/d/\$$$

después, I_8 e I_9 se sustituyen por su unión:

$$I_{89}: \quad C \rightarrow cC, c/d/\$$$

Las funciones de acción e ir_A LALR para los conjuntos combinados de elementos se muestran en la figura 4.43.

ESTADO	ACCIÓN			ir_A	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Figura 4.43: Tabla de análisis sintáctico LALR para la gramática del ejemplo 4.54

Para ver cómo se calculan los ir_A , considere el $\text{ir_A}(I_{36}, C)$. En el conjunto original de elementos LR(1), $\text{ir_A}(I_3, C) = I_8$, y ahora I_8 es parte de I_{89} , por lo que hacemos que $\text{ir_A}(I_{36}, C)$ sea I_{89} . Podríamos haber llegado a la misma conclusión si consideráramos a I_6 , la otra parte de I_{36} . Es decir, $\text{ir_A}(I_6, C) = I_9$, y ahora I_9 forma parte de I_{89} . Para otro ejemplo, considere a $\text{ir_A}(I_2, c)$, una entrada que se ejerce después de la acción de desplazamiento de I_2 en la entrada c . En los conjuntos originales de elementos LR(1), $\text{ir_A}(I_2, c) = I_6$. Como I_6 forma ahora parte de I_{36} , $\text{ir_A}(I_2, c)$ se convierte en I_{36} . Por lo tanto, la entrada en la figura 4.43 para el estado 2 y la entrada c se convierte en s36, lo cual significa desplazar y meter el estado 36 en la pila. \square

Cuando se les presenta una cadena del lenguaje $\mathbf{c}^* \mathbf{d} \mathbf{c}^* \mathbf{d}$, tanto el analizador sintáctico LR de la figura 4.42 como el analizador sintáctico LALR de la figura 4.43 realizan exactamente la misma secuencia de desplazamientos y reducciones, aunque los nombres de los estados en la pila pueden diferir. Por ejemplo, si el analizador sintáctico LR mete a I_3 o I_6 en la pila, el analizador sintáctico LALR meterá a I_{36} en la pila. Esta relación se aplica en general para

una gramática LALR. Los analizadores sintácticos LR y LALR se imitarán uno al otro en las entradas correctas.

Si se le presenta una entrada errónea, el analizador sintáctico LALR tal vez proceda a realizar ciertas reducciones, una vez que el analizador sintáctico LR haya declarado un error. Sin embargo, el analizador sintáctico LALR nunca desplazará otro símbolo después de que el analizador sintáctico LR declare un error. Por ejemplo, en la entrada *ccd* seguida de $\$$, el analizador sintáctico LR de la figura 4.42 meterá lo siguiente en la pila:

0 3 3 4

y en el estado 4 descubrirá un error, ya que $\$$ es el siguiente símbolo de entrada y el estado 4 tiene una acción de error en $\$$. En contraste, el analizador sintáctico LALR de la figura 4.43 realizará los movimientos correspondientes, metiendo lo siguiente en la pila:

0 36 36 47

Pero el estado 47 en la entrada $\$$ tiene la acción de reducir $C \rightarrow d$. El analizador sintáctico LALR cambiará, por lo tanto, su pila a:

0 36 36 89

Ahora, la acción del estado 89 en la entrada $\$$ es reducir $C \rightarrow cC$. La pila se convierte en lo siguiente:

0 36 89

en donde se llama a una reducción similar, con lo cual se obtiene la pila:

0 2

Por último, el estado 2 tiene una acción de error en la entrada $\$$, por lo que ahora se descubre el error.

4.7.5 Construcción eficiente de tablas de análisis sintáctico LALR

Hay varias modificaciones que podemos realizar al Algoritmo 4.59 para evitar construir la colección completa de conjuntos de elementos LR(1), en el proceso de crear una tabla de análisis sintáctico LALR(1).

- En primer lugar, podemos representar cualquier conjunto de elementos I LR(0) o LR(1) mediante su corazón (kernel); es decir, mediante aquellos elementos que sean el elemento inicial, $[S' \rightarrow \cdot S]$ o $[S' \rightarrow \cdot S, \$]$, o que tengan el punto en algún lugar que no sea al principio del cuerpo de la producción.
- Podemos construir los corazones de los elementos LALR(1) a partir de los corazones de los elementos LR(0), mediante un proceso de propagación y generación espontánea de lecturas adelantadas, lo cual describiremos en breve.
- Si tenemos los corazones LALR(1), podemos generar la tabla de análisis sintáctico LALR(1) cerrando cada corazón, usando la función CERRADURA de la figura 4.40, y después calculando las entradas en la tabla mediante el Algoritmo 4.56, como si los conjuntos de elementos LALR(1) fueran conjuntos de elementos LR(1) canónicos.

Ejemplo 4.61: Vamos a usar, como un ejemplo del eficiente método de construcción de una tabla LALR(1), la gramática que no es SLR del ejemplo 4.48, la cual reproducimos a continuación en su forma aumentada:

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & L = R \mid R \\ L & \rightarrow & *R \mid \mathbf{id} \\ R & \rightarrow & L \end{array}$$

En la figura 4.39 se mostraron los conjuntos completos de elementos LR(0) para esta gramática. Los corazones de estos elementos se muestran en la figura 4.44. \square

$$\begin{array}{ll} I_0: & S' \rightarrow \cdot S \\ I_1: & S' \rightarrow S \cdot \\ I_2: & S \rightarrow L \cdot = R \\ & R \rightarrow L \cdot \\ I_3: & S \rightarrow R \cdot \\ I_4: & L \rightarrow * \cdot R \\ & I_5: & L \rightarrow \mathbf{id} \cdot \\ & I_6: & S \rightarrow L = \cdot R \\ & I_7: & L \rightarrow * R \cdot \\ & I_8: & R \rightarrow L \cdot \\ & I_9: & S \rightarrow L = R \cdot \end{array}$$

Figura 4.44: Corazones de los conjuntos de elementos LR(0) para la gramática (4.49)

Ahora debemos adjuntar los símbolos de anticipación apropiados para los elementos LR(0) en los corazones, para crear los corazones de los conjuntos de elementos LALR(1). Hay dos formas en que se puede adjuntar un símbolo de anticipación b a un elemento LR(0) $B \rightarrow \gamma \cdot \delta$, en cierto conjunto de elementos LALR(1) J :

1. Hay un conjunto de elementos I , con un elemento de corazón $A \rightarrow \alpha \cdot \beta$, a , y $J = \text{ir_A}(I, X)$, y la construcción de

$$\text{ir_A}(\text{CERRADURA}(\{[A \rightarrow \alpha \cdot \beta, a]\}), X)$$

como se proporciona en la figura 4.40, contiene $[B \rightarrow \gamma \cdot \delta, b]$, sin importar a . Se considera que dicho símbolo de anticipación b se genera en forma *espontánea* para $B \rightarrow \gamma \cdot \delta$.

2. Como caso especial, el símbolo de anticipación $\$$ se genera en forma espontánea para el elemento $S' \rightarrow \cdot S$ en el conjunto inicial de elementos.
3. Todo es como en (1), pero $a = b$ e $\text{ir_A}(\text{CERRADURA}(\{[A \rightarrow \alpha \cdot \beta, b]\}), X)$, como se proporciona en la figura 4.40, contiene $[B \rightarrow \gamma \cdot \delta, b]$ sólo porque $A \rightarrow \alpha \cdot \beta$ tiene a b como uno de sus símbolos de anticipación asociados. En tal caso, decimos que los símbolos de anticipación *se propagan* desde $A \rightarrow \alpha \cdot \beta$ en el corazón de I , hasta $B \rightarrow \gamma \cdot \delta$ en el corazón de J . Observe que la propagación no depende del símbolo de anticipación específico; o todos los símbolos de anticipación se propagan desde un elemento hasta otro, o ninguno lo hace.

Debemos determinar los símbolos de anticipación generados en forma espontánea para cada conjunto de elementos $LR(0)$, y también determinar cuáles elementos propagan los símbolos de anticipación desde cuáles otros. En realidad, la prueba es bastante simple. Hagamos que $\#$ sea un símbolo que no esté en la gramática en cuestión. Hagamos que $A \rightarrow \alpha \cdot \beta$ sea un elemento de corazón $LR(0)$ en el conjunto I . Calcule, para cada X , $J = ir_A(CERRADURA(\{[A \rightarrow \alpha \cdot \beta, \#]\}), X)$. Para cada elemento de corazón en J , examinamos su conjunto de símbolos de anticipación. Si $\#$ es un símbolo de anticipación, entonces los símbolos de anticipación se propagan hacia ese elemento desde $A \rightarrow \alpha \cdot \beta$. Cualquier otro símbolo de anticipación se genera de manera espontánea. Estas ideas se hacen precisas en el siguiente algoritmo, el cual también hace uso del hecho de que los únicos elementos de corazón en J deben tener a X justo a la izquierda del punto; es decir, deben ser de la forma $B \rightarrow \gamma X \cdot \delta$.

Algoritmo 4.62: Determinación de los símbolos de anticipación.

ENTRADA: El corazón K de un conjunto de elementos $LR(0)$ I y un símbolo gramatical X .

SALIDA: Los símbolos de anticipación generados en forma espontánea por los elementos en I , para los elementos de corazón en $ir_A(I, X)$ y los elementos en I , a partir de los cuales los símbolos de anticipación se propagan hacia los elementos de corazón en $ir_A(I, X)$.

MÉTODO: El algoritmo se proporciona en la figura 4.45. \square

```

for ( cada elemento  $A \rightarrow \alpha \cdot \beta$  en  $K$  ) {
     $J := CERRADURA(\{[A \rightarrow \alpha \cdot \beta, \#]\})$ ;
    if (  $[B \rightarrow \gamma \cdot X \delta, a]$  está en  $J$ , y  $a$  no es  $\#$  )
        concluir que el símbolo de anticipación  $a$  se genera en forma espontánea
        para el elemento  $B \rightarrow \gamma X \cdot \delta$  en  $ir\_A(I, X)$ ;
    if (  $[B \rightarrow \gamma \cdot X \delta, \#]$  está en  $J$  )
        concluir que los símbolos de anticipación se propagan desde  $A \rightarrow \alpha \cdot \beta$  en  $I$ 
        hacia  $B \rightarrow \gamma X \cdot \delta$  en  $ir\_A(I, X)$ ;
}

```

Figura 4.45: Descubrimiento de símbolos de anticipación propagados y espontáneos

Ahora estamos listos para adjuntar los símbolos de anticipación a los corazones de los conjuntos de elementos $LR(0)$ para formar los conjuntos de elementos $LALR(1)$. En primer lugar, sabemos que $\$$ es un símbolo de anticipación para $S' \rightarrow \cdot S$ en el conjunto inicial de elementos $LR(0)$. El Algoritmo 4.62 nos proporciona todos los símbolos de anticipación que se generan en forma espontánea. Después de listar todos esos símbolos de anticipación, debemos permitir que se propaguen hasta que ya no puedan hacerlo más. Hay muchos métodos distintos, todos los cuales en cierto sentido llevan la cuenta de los “nuevos” símbolos de anticipación que se han propagado en un elemento, pero que todavía no se propagan hacia fuera. El siguiente algoritmo describe una técnica para propagar los símbolos de anticipación hacia todos los elementos.

Algoritmo 4.63: Cálculo eficiente de los corazones de la colección de conjuntos de elementos $LALR(1)$.

ENTRADA: Una gramática aumentada G' .

SALIDA: Los corazones de la colección de conjuntos de elementos LALR(1) para G' .

MÉTODO:

1. Construir los corazones de los conjuntos de elementos LR(0) para G . Si el espacio no es de extrema importancia, la manera más simple es construir los conjuntos de elementos LR(0), como en la sección 4.6.2, y después eliminar los elementos que no sean del corazón. Si el espacio está restringido en extremo, tal vez sea conveniente almacenar sólo los elementos del corazón de cada conjunto, y calcular ir_A para un conjunto de elementos I , para lo cual primero debemos calcular la cerradura de I .
2. Aplicar el Algoritmo 4.62 al corazón de cada conjunto de elementos LR(0) y el símbolo gramatical X para determinar qué símbolos de anticipación se generan en forma espontánea para los elementos del corazón en $\text{ir_A}(I, X)$, y a partir los cuales se propagan los elementos en los símbolos de anticipación I a los elementos del corazón en $\text{ir_A}(I, X)$.
3. Inicializar una tabla que proporcione, para cada elemento del corazón en cada conjunto de elementos, los símbolos de anticipación asociados. Al principio, cada elemento tiene asociados sólo los símbolos de anticipación que determinamos en el paso (2) que se generaron en forma espontánea.
4. Hacer pasadas repetidas sobre los elementos del corazón en todos los conjuntos. Al visitar un elemento i , buscamos los elementos del corazón para los cuales i propaga sus símbolos de anticipación, usando la información que se tabuló en el paso (2). El conjunto actual de símbolos de anticipación para i se agrega a los que ya están asociados con cada uno de los elementos para los cuales i propaga sus símbolos de anticipación. Continuamos realizando pasadas sobre los elementos del corazón hasta que no se propaguen más símbolos nuevos de anticipación.

□

Ejemplo 4.64: Vamos a construir los corazones de los elementos LALR(1) para la gramática del ejemplo 4.61. Los corazones de los elementos LR(0) se mostraron en la figura 4.44. Al aplicar el Algoritmo 4.62 al corazón del conjunto de elementos I_0 , primero calculamos $\text{CERRADURA}(\{[S' \rightarrow \cdot S, \#]\})$, que viene siendo:

$$\begin{array}{ll} S' \rightarrow \cdot S, \# & L \rightarrow \cdot * R, \# / = \\ S \rightarrow \cdot L = R, \# & L \rightarrow \cdot \text{id}, \# / = \\ S \rightarrow \cdot R, \# & R \rightarrow \cdot L, \# \end{array}$$

Entre los elementos en la cerradura, vemos dos en donde se ha generado el símbolo de anticipación $=$ en forma espontánea. El primero de éstos es $L \rightarrow \cdot * R$. Este elemento, con $*$ a la derecha del punto, produce $[L \rightarrow \cdot * R, =]$. Es decir, $=$ es un símbolo de anticipación generado en forma espontánea para $L \rightarrow \cdot * R$, que se encuentra en el conjunto de elementos I_4 . De manera similar, $[L \rightarrow \cdot \text{id}, =]$ nos indica que $=$ es un símbolo de anticipación generado en forma espontánea para $L \rightarrow \text{id}\cdot$ en I_5 .

Como $\#$ es un símbolo de anticipación para los seis elementos en la cerradura, determinamos que el elemento $S' \rightarrow S$ en I_0 propaga los símbolos de anticipación hacia los siguientes seis elementos:

$$\begin{array}{ll}
 S' \rightarrow S \cdot \text{ in } I_1 & L \rightarrow * \cdot R \text{ in } I_4 \\
 S \rightarrow L \cdot = R \text{ in } I_2 & L \rightarrow \mathbf{id} \cdot \text{ in } I_5 \\
 S \rightarrow R \cdot \text{ in } I_3 & R \rightarrow L \cdot \text{ in } I_2
 \end{array}$$

DESDE	HACIA
$I_0: S' \rightarrow \cdot S$	$I_1: S' \rightarrow S \cdot$ $I_2: S \rightarrow L \cdot = R$ $I_2: R \rightarrow L \cdot$ $I_3: S \rightarrow R \cdot$ $I_4: L \rightarrow * \cdot R$ $I_5: L \rightarrow \mathbf{id} \cdot$
$I_2: S \rightarrow L \cdot = R$	$I_6: S \rightarrow L = \cdot R$
$I_4: L \rightarrow * \cdot R$	$I_4: L \rightarrow * \cdot R$ $I_5: L \rightarrow \mathbf{id} \cdot$ $I_7: L \rightarrow * R \cdot$ $I_8: R \rightarrow L \cdot$
$I_6: S \rightarrow L = \cdot R$	$I_4: L \rightarrow * \cdot R$ $I_5: L \rightarrow \mathbf{id} \cdot$ $I_8: R \rightarrow L \cdot$ $I_9: S \rightarrow L = R \cdot$

Figura 4.46: Propagación de los símbolos de anticipación

En la figura 4.47, mostramos los pasos (3) y (4) del Algoritmo 4.63. La columna etiquetada como INIC muestra los símbolos de anticipación generados en forma espontánea para cada elemento del corazón. Éstas son sólo las dos ocurrencias de $=$ que vimos antes, y el símbolo de anticipación $\$$ espontáneo para el elemento inicial $S' \rightarrow \cdot S$.

En la primera pasada, el símbolo $\$$ de anticipación se propaga de $S' \rightarrow S$ en I_0 hacia los seis elementos listados en la figura 4.46. El símbolo $=$ de anticipación se propaga desde $L \rightarrow * \cdot R$ en I_4 hacia los elementos $L \rightarrow * R \cdot$ en I_7 y $R \rightarrow L \cdot$ en I_8 . También se propaga hacia sí mismo y hacia $L \rightarrow \mathbf{id} \cdot$ en I_5 , pero estos símbolos de anticipación ya están presentes. En la segunda y tercera pasada, el único símbolo nuevo de anticipación que se propaga es $\$$, descubierto para los sucesores de I_2 e I_4 en la pasada 2 y para el sucesor de I_6 en la pasada 3. En la pasada 4 no se propagan nuevos símbolos de anticipación, por lo que el conjunto final de símbolos de anticipación se muestra en la columna por la derecha de la figura 4.47.

Observe que el conflicto de desplazamiento/reducción en el ejemplo 4.48 que utiliza el método SLR ha desaparecido con la técnica LALR. La razón es que sólo el símbolo $\$$ de anticipación está asociado con $R \rightarrow L \cdot$ en I_2 , por lo que no hay conflicto con la acción de análisis sintáctico de desplazamiento en $=$, generada por el elemento $S \rightarrow L \cdot = R$ en I_2 . \square

CONJUNTO	ELEMENTO	SÍMBOLOS DE ANTICIPACIÓN			
		INIC	PASADA 1	PASADA 2	PASADA 3
	$I_0: S' \rightarrow \cdot S$	\$	\$	\$	\$
	$I_1: S' \rightarrow S \cdot$		\$	\$	\$
	$I_2: S \rightarrow L \cdot = R$		\$	\$	\$
	$R \rightarrow L \cdot$		\$	\$	\$
	$I_3: S \rightarrow R \cdot$		\$	\$	\$
		=	=/\$	=/\$	=/\$
	$I_5: L \rightarrow \mathbf{id} \cdot$	=	=/\$	=/\$	=/\$
	$I_6: S \rightarrow L = \cdot R$			\$	\$
	$I_7: L \rightarrow *R \cdot$		=	=/\$	=/\$
	$I_8: R \rightarrow L \cdot$		=	=/\$	=/\$
	$I_9: S \rightarrow L = R \cdot$				\$

Figura 4.47: Cálculo de los símbolos de anticipación

4.7.6 Compactación de las tablas de análisis sintáctico LR

Una gramática de lenguaje de programación común, con una cantidad de 50 a 100 terminales y 100 producciones, puede tener una tabla de análisis sintáctico LALR con varios cientos de estados. La función de acción podría fácilmente tener 20 000 entradas, cada una requiriendo por lo menos 8 bits para codificarla. En los dispositivos pequeños, puede ser importante tener una codificación más eficiente que un arreglo bidimensional. En breve mencionaremos algunas técnicas que se han utilizado para comprimir los campos ACCION e ir_A de una tabla de análisis sintáctico LR.

Una técnica útil para compactar el campo de acción es reconocer que, por lo general, muchas filas de la tabla de acciones son idénticas. Por ejemplo, en la figura 4.42 los estados 0 y 3 tienen entradas de acción idénticas, al igual que los estados 2 y 6. Por lo tanto, podemos ahorrar una cantidad de espacio considerable, con muy poco costo en relación con el tiempo, si creamos un apuntador para cada estado en un arreglo unidimensional. Los apuntadores para los estados con las mismas acciones apuntan a la misma ubicación. Para acceder a la información desde este arreglo, asignamos a cada terminal un número desde cero hasta uno menos que el número de terminales, y utilizamos este entero como un desplazamiento a partir del valor del apuntador para cada estado. En un estado dado, la acción de análisis sintáctico para la i -ésima terminal se encontrará a i ubicaciones más allá del valor del apuntador para ese estado.

Puede lograrse una mejor eficiencia en cuanto al espacio, a expensas de un analizador sintáctico un poco más lento, mediante la creación de una lista para las acciones de cada estado. Esta lista consiste en pares (terminal-símbolo, acción). La acción más frecuente para un estado puede

colocarse al final de una lista, y en lugar de un terminal podemos usar la notación “**cualquiera**”, indicando que si no se ha encontrado el símbolo de entrada actual hasta ese punto en la lista, debemos realizar esa acción sin importar lo que sea la entrada. Además, las entradas de error pueden sustituirse sin problemas por acciones de reducción, para una mayor uniformidad a lo largo de una fila. Los errores se detectarán más adelante, antes de un movimiento de desplazamiento.

Ejemplo 4.65: Considere la tabla de análisis sintáctico de la figura 4.37. En primer lugar, observe que las acciones para los estados 0, 4, 6 y 7 coinciden. Podemos representarlas todas mediante la siguiente lista:

SÍMBOLO	ACCIÓN
id	s5
(s4
cualquiera	error

El estado 1 tiene una lista similar:

+	s6
\$	acc
cualquiera	error

En el estado 2, podemos sustituir las entradas de error por r2, para que se realice la reducción mediante la producción 2 en cualquier entrada excepto *. Por ende, la lista para el estado 2 es:

*	s7
cualquiera	r2

El estado 3 tiene sólo entradas de error y r4. Podemos sustituir la primera por la segunda, de manera que la lista para el estado 3 consista sólo en el par (**cualquiera**, r4). Los estados 5, 10 y 11 pueden tratarse en forma similar. La lista para el estado 8 es:

+	s6
)	s11
cualquiera	error

y para el estado 9 es:

*	s7
)	s11
cualquiera	r1

□

También podemos codificar la tabla ir_A mediante una lista, pero aquí es más eficiente crear una lista de pares para cada no terminal A. Cada par en la lista para A es de la forma (*estadoActual*, *siguienteEstado*), lo cual indica que:

$$\text{ir_A}[\text{estadoActual}, A] = \text{siguienteEstado}$$

Esta técnica es útil, ya que tiende a haber menos estados en cualquier columna de la tabla `ir_A`. La razón es que el `ir_A` en el no terminal A sólo puede ser un estado que pueda derivarse a partir de un conjunto de elementos en los que algunos elementos tengan a A justo a la izquierda de un punto. Ningún conjunto tiene elementos con X y Y justo a la izquierda de un punto si $X \neq Y$. Por ende, cada estado aparece como máximo en una columna `ir_A`.

Para una mayor reducción del espacio, hay que observar que las entradas de error en la tabla de `ir_A` nunca se consultan. Por lo tanto, podemos sustituir cada entrada de error por la entrada más común sin error en su columna. Esta entrada se convierte en la opción predeterminada; se representa en la lista para cada columna mediante un par con **cualquiera** en vez de *estadoActual*.

Ejemplo 4.66: Considere de nuevo la figura 4.37. La columna para F tiene la entrada 10 para el estado 7, y todas las demás entradas son 3 o error. Podemos sustituir error por 3 y crear, para la columna F , la siguiente lista:

ESTADOACTUAL	SIGUIENTEESTADO
7	10
cualquiera	3

De manera similar, una lista adecuada para la columna T es:

6	9
cualquiera	2

Para la columna E podemos elegir 1 o 8 como la opción predeterminada; son necesarias dos entradas en cualquier caso. Por ejemplo, podríamos crear para la columna E la siguiente lista:

4	8
cualquiera	1

□

El ahorro de espacio en estos pequeños ejemplos puede ser engañoso, ya que el número total de entradas en las listas creadas en este ejemplo y el anterior, junto con los apuntadores desde los estados hacia las listas de acción, y desde las no terminales hacia las listas de los siguientes estados, producen un ahorro de espacio mínimo, en comparación con la implementación de una matriz de la figura 4.37. En las gramáticas prácticas, el espacio necesario para la representación de la lista es, por lo general, menos del diez por ciento de lo necesario para la representación de la matriz. Los métodos de compresión de tablas para los autómatas finitos que vimos en la sección 3.9.8 pueden usarse también para representar las tablas de análisis sintáctico LR.

4.7.7 Ejercicios para la sección 4.7

Ejercicio 4.7.1: Construya los conjuntos de elementos

- LR canónicos, y
- LALR.

para la gramática $S \rightarrow S\ S + \mid S\ S\ * \mid a$ del ejercicio 4.2.1.

Ejercicio 4.7.2: Repita el ejercicio 4.7.1 para cada una de las gramáticas (aumentadas) del ejercicio 4.2.2(a)-(g).

! Ejercicio 4.7.3: Para la gramática del ejercicio 4.7.1, use el Algoritmo 4.63 para calcular la colección de conjuntos de elementos LALR, a partir de los corazones de los conjuntos de elementos LR(0).

! Ejercicio 4.7.4: Muestre que la siguiente gramática:

$$\begin{array}{l} S \rightarrow A a \mid b A c \mid d c \mid b d a \\ A \rightarrow d \end{array}$$

es LALR(1), pero no SLR(1).

! Ejercicio 4.7.5: Muestre que la siguiente gramática:

$$\begin{array}{l} S \rightarrow A a \mid b A c \mid B c \mid d B a \\ A \rightarrow d \\ B \rightarrow d \end{array}$$

es LR(1), pero no LALR(1).

4.8 Uso de gramáticas ambiguas

Es un hecho que ninguna gramática ambigua es LR y, por ende, no se encuentra en ninguna de las clases de gramáticas que hemos visto en las dos secciones anteriores. No obstante, ciertos tipos de gramáticas ambiguas son bastante útiles en la especificación e implementación de lenguajes. Para las construcciones de lenguajes como las expresiones, una gramática ambigua proporciona una especificación más corta y natural que cualquier gramática no ambigua equivalente. Otro uso de las gramáticas ambiguas es el de aislar las construcciones sintácticas que ocurren con frecuencia para la optimización de casos especiales. Con una gramática ambigua, podemos especificar las construcciones de casos especiales, agregando con cuidado nuevas producciones a la gramática.

Aunque las gramáticas que usamos no son ambiguas, en todos los casos especificamos reglas para eliminar la ambigüedad, las cuales sólo permiten un árbol de análisis sintáctico para cada enunciado. De esta forma, se eliminan las ambigüedades de la especificación general del lenguaje, y algunas veces es posible diseñar un analizador sintáctico LR que siga las mismas opciones para resolver las ambigüedades. Debemos enfatizar que las construcciones ambiguas deben utilizarse con medida y en una forma estrictamente controlada; de no ser así, no puede haber garantía en el lenguaje que reconozca un analizador sintáctico.

4.8.1 Precedencia y asociatividad para resolver conflictos

Considere la gramática ambigua (4.3) para las expresiones con los operadores + y *, que repetimos a continuación por conveniencia:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

Esta gramática es ambigua, ya que no especifica la asociatividad ni la precedencia de los operadores + y *. La gramática sin ambigüedad (4.1), que incluye las producciones $E \rightarrow E + T$ y $T \rightarrow T * F$, genera el mismo lenguaje, pero otorga a + una precedencia menor que la de *, y hace que ambos operadores sean asociativos por la izquierda. Hay dos razones por las cuales podría ser más conveniente preferir el uso de la gramática ambigua. En primer lugar, como veremos más adelante, podemos cambiar con facilidad la asociatividad y la precedencia de los operadores + y * sin perturbar las producciones de (4.3) o el número de estados en el analizador sintáctico resultante. En segundo lugar, el analizador sintáctico para la gramática sin ambigüedad invertirá una fracción considerable de su tiempo realizando reducciones mediante las producciones $E \rightarrow T$ y $T \rightarrow F$, cuya única función es hacer valer la asociatividad y la precedencia. El analizador sintáctico para la gramática sin ambigüedad (4.3) no desperdiciará tiempo realizando reducciones mediante estas producciones *simples* (producciones cuyo cuerpo consiste en un solo no terminal).

Los conjuntos de elementos LR(0) para la gramática de expresiones sin ambigüedad (4.3) aumentada por $E' \rightarrow E$ se muestran en la figura 4.48. Como la gramática (4.3) es ambigua, habrá conflictos de acciones de análisis sintáctico cuando tratemos de producir una tabla de análisis sintáctico LR a partir de los conjuntos de elementos. Los estados que corresponden a los conjuntos de elementos I_7 e I_8 generan estos conflictos. Suponga que utilizamos el método SLR para construir la tabla de acciones de análisis sintáctico. El conflicto generado por I_7 entre la reducción mediante $E \rightarrow E + E$ y el desplazamiento en + o * no puede resolverse, ya que + y * se encuentran en SIGUIENTE(E). Por lo tanto, se llamaría a ambas acciones en las entradas + y *. I_8 genera un conflicto similar, entre la reducción mediante $E \rightarrow E * E$ y el desplazamiento en las entradas + y *. De hecho, cada uno de nuestros métodos de construcción de tablas de análisis sintáctico LR generarán estos conflictos.

No obstante, estos problemas pueden resolverse mediante el uso de la información sobre la precedencia y la asociatividad para + y *. Considere la entrada **id** + **id** * **id**, la cual hace que un analizador sintáctico basado en la figura 4.48 entre al estado 7 después de procesar **id** + **id**; de manera específica, el analizador sintáctico llega a la siguiente configuración:

PREFIJO	PILA	ENTRADA
$E + E$	0 1 4 7	* id \$

Por conveniencia, los símbolos que corresponden a los estados 1, 4 y 7 también se muestran bajo PREFIJO.

Si * tiene precedencia sobre +, sabemos que el analizador sintáctico debería desplazar a * hacia la pila, preparándose para reducir el * y sus símbolos **id** circundantes a una expresión. El analizador sintáctico SLR de la figura 4.37 realizó esta elección, con base en una gramática sin ambigüedad para el mismo lenguaje. Por otra parte, si + tiene precedencia sobre *, sabemos que el analizador sintáctico debería reducir $E + E$ a E . Por lo tanto, la precedencia relativa

$I_0:$	$E' \rightarrow \cdot E$ $E \rightarrow \cdot E + E$ $E \rightarrow \cdot E * E$ $E \rightarrow \cdot (E)$ $E \rightarrow \cdot \mathbf{id}$	$I_5:$	$E \rightarrow E * \cdot E$ $E \rightarrow \cdot E + E$ $E \rightarrow \cdot E * E$ $E \rightarrow \cdot (E)$ $E \rightarrow \cdot \mathbf{id}$
$I_1:$	$E' \rightarrow E \cdot$ $E \rightarrow E \cdot + E$ $E \rightarrow E \cdot * E$	$I_6:$	$E \rightarrow (E) \cdot$ $E \rightarrow E \cdot + E$ $E \rightarrow E \cdot * E$
$I_2:$	$E \rightarrow (\cdot E)$ $E \rightarrow \cdot E + E$ $E \rightarrow \cdot E * E$ $E \rightarrow \cdot (E)$ $E \rightarrow \cdot \mathbf{id}$	$I_7:$	$E \rightarrow E + E \cdot$ $E \rightarrow E \cdot + E$ $E \rightarrow E \cdot * E$
$I_3:$	$E \rightarrow \mathbf{id} \cdot$	$I_8:$	$E \rightarrow E * E \cdot$ $E \rightarrow E \cdot + E$ $E \rightarrow E \cdot * E$
$I_4:$	$E \rightarrow E + \cdot E$ $E \rightarrow \cdot E + E$ $E \rightarrow \cdot E * E$ $E \rightarrow \cdot (E)$ $E \rightarrow \cdot \mathbf{id}$	$I_9:$	$E \rightarrow (E) \cdot$

Figura 4.48: Conjuntos de elementos LR(0) para una gramática de expresiones aumentada

de $+$ seguido de $*$ determina en forma única la manera en que debería resolverse el conflicto de acciones de análisis sintáctico entre la reducción $E \rightarrow E + E$ y el desplazamiento sobre $*$ en el estado 7.

Si la entrada hubiera sido $\mathbf{id} + \mathbf{id} + \mathbf{id}$, el analizador sintáctico llegaría de todas formas a una configuración en la cual tendría la pila 0 1 4 7 después de procesar la entrada $\mathbf{id} + \mathbf{id}$. En la entrada $+$ hay de nuevo un conflicto de desplazamiento/reducción en el estado 7. Sin embargo, ahora la asociatividad del operador $+$ determina cómo debe resolverse este conflicto. Si $+$ es asociativo a la izquierda, la acción correcta es reducir mediante $E \rightarrow E + E$. Es decir, los símbolos \mathbf{id} que rodean el primer $+$ deben agruparse primero. De nuevo, esta elección coincide con lo que haría el analizador sintáctico SLR para la gramática sin ambigüedad.

En resumen, si asumimos que $+$ es asociativo por la izquierda, la acción del estado 7 en la entrada $+$ debería ser reducir mediante $E \rightarrow E + E$, y suponiendo que $*$ tiene precedencia sobre $+$, la acción del estado 7 en la entrada $*$ sería desplazar. De manera similar, suponiendo que $*$ sea asociativo por la izquierda y tenga precedencia sobre $+$, podemos argumentar que el estado 8, que puede aparecer en la parte superior de la pila sólo cuando $E * E$ son los tres símbolos gramaticales de la parte superior, debería tener la acción de reducir $E \rightarrow E * E$ en las entradas $+$ y $*$. En el caso de la entrada $+$, la razón es que $*$ tiene precedencia sobre $+$, mientras que en el caso de la entrada $*$, el fundamento es que $*$ es asociativo por la izquierda.

Si procedemos de esta forma, obtendremos la tabla de análisis sintáctico LR que se muestra en la figura 4.49. Las producciones de la 1 a la 4 son $E \rightarrow E + E$, $E \rightarrow E * E$, $\rightarrow (E)$ y $E \rightarrow \mathbf{id}$, respectivamente. Es interesante que una tabla de acciones de análisis sintáctico similar se produzca eliminando las reducciones mediante las producciones simples $E \rightarrow T$ y $T \rightarrow F$ a partir de la tabla SLR para la gramática de expresiones sin ambigüedad (4.1) que se muestra en la figura 4.37. Las gramáticas ambiguas como la que se usa para las expresiones pueden manejarse en una forma similar, en el contexto de los análisis sintácticos LALR y LR canónico.

ESTADO	ACCIÓN					ir_A
	id	+	*	()	\$	
	<i>E</i>					
0	s3		s2			1
1		s4	s5		acc	
2	s3			s2		6
3		r4	r4		r4 r4	
4	s3			s2		7
5	s3			s2		8
6		s4	s5		s9	
7		r1	s5		r1 r1	
8		r2	r2		r2 r2	
9		r3	r3		r3 r3	

Figura 4.49: Tabla de análisis sintáctico para la gramática (4.3)

4.8.2 La ambigüedad del “else colgante”

Considere de nuevo la siguiente gramática para las instrucciones condicionales:

$$\begin{aligned} instr &\rightarrow \mathbf{if} \ expr \ \mathbf{then} \ instr \ \mathbf{else} \ instr \\ &\quad | \quad \mathbf{if} \ expr \ \mathbf{then} \ instr \\ &\quad | \quad \mathbf{otras} \end{aligned}$$

Como vimos en la sección 4.3.2, esta gramática no tiene ambigüedades, ya que no resuelve la ambigüedad del else colgante. Para simplificar la discusión, vamos a considerar una abstracción de esta gramática, en donde *i* representa a **if** *expr then*, *e* representa a **else**, y *a* representa a “todas las demás producciones”. De esta forma podemos escribir la gramática, con la producción aumentada $S' \rightarrow S$, como

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow i \ S \ e \ S \ | \ i \ S \ | \ a \end{aligned} \tag{4.67}$$

Los conjuntos de elementos LR(0) para la gramática (4.67) se muestran en la figura 4.50. La ambigüedad en (4.67) produce un conflicto de desplazamiento/reducción en I_4 . Ahí, $S \rightarrow iS \cdot eS$ llama a un desplazamiento de *e* y, como $\text{SIGUIENTE}(S) = \{e, \$\}$, el elemento $S \rightarrow iS \cdot$ llama a la reducción mediante $S \rightarrow iS$ en la entrada *e*.

$I_0:$	$S' \rightarrow \cdot S$ $S \rightarrow \cdot iSeS$ $S \rightarrow \cdot iS$ $S \rightarrow \cdot a$	$I_3:$	$S \rightarrow a \cdot$
$I_1:$	$S' \rightarrow S \cdot$	$I_4:$	$S \rightarrow iS \cdot eS$
$I_2:$	$S \rightarrow i \cdot SeS$ $S \rightarrow i \cdot S$ $S \rightarrow \cdot iSeS$ $S \rightarrow \cdot iS$ $S \rightarrow \cdot a$	$I_5:$	$S \rightarrow iSe \cdot S$ $S \rightarrow \cdot iSeS$ $S \rightarrow \cdot iS$ $S \rightarrow \cdot a$
		$I_6:$	$S \rightarrow iSeS \cdot$

Figura 4.50: Estados LR(0) para la gramática aumentada (4.67)

Traduciendo esto de vuelta a la terminología **if-then-else**, si tenemos a:

if *expr then instr*

en la pila y a **else** como el primer símbolo de entrada, ¿debemos desplazar el **else** hacia la pila (es decir, desplazar a *e*) o reducir **if** *expr then instr* (es decir, reducir mediante $S \rightarrow iS$)? La respuesta es que debemos desplazar el **else**, ya que está “asociado” con el **then** anterior. En la terminología de la gramática (4.67), la *e* en la entrada, que representa a **else**, sólo puede formar parte del cuerpo que empieza con la *iS* que está ahora en la parte superior de la pila. Si lo que sigue después de *e* en la entrada no puede analizarse como una *S*, para completar el cuerpo *iSeS*, entonces podemos demostrar que no hay otro análisis sintáctico posible.

Concluimos que el conflicto de desplazamiento/reducción en I_4 debe resolverse a favor del desplazamiento en la entrada *e*. La tabla de análisis sintáctico SLR que se construyó a partir de los conjuntos de elementos de la figura 4.48, que utiliza esta resolución del conflicto de acciones de análisis sintáctico en I_4 con la entrada *e*, se muestra en la figura 4.51. Las producciones de la 1 a la 3 son $S \rightarrow iSeS$, $S \rightarrow iS$ y $S \rightarrow a$, respectivamente.

ESTADO	ACCIÓN				ir_A
	<i>i</i>	<i>e</i>	<i>a</i>	\$	
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	

Figura 4.51: Tabla de análisis sintáctico LR para la gramática del “else colgante”

Por ejemplo, en la entrada *iiae*, el analizador sintáctico realiza los movimientos que se muestran en la figura 4.52, correspondientes a la resolución correcta del “else colgante”. En la línea (5), el estado 4 selecciona la acción de desplazamiento en la entrada *e*, mientras que en la línea (9), el estado 4 llama a la reducción mediante $S \rightarrow iS$ en la entrada $\$$.

PILA	SÍMBOLOS	ENTRADA	ACCIÓN
(1) 0		<i>iiae</i> $\$$	desplazar
(2) 0 2	<i>i</i>	<i>iiae</i> $\$$	desplazar
(3) 0 2 2	<i>ii</i>	<i>a</i> e a $\$$	desplazar
(4) 0 2 2 3	<i>ii</i> <i>a</i>	e a $\$$	desplazar
(5) 0 2 2 4	<i>ii</i> S	e a $\$$	reducir $S \rightarrow a$
(6) 0 2 2 4 5	<i>ii</i> S <i>e</i>	a $\$$	desplazar
(7) 0 2 2 4 5 3	<i>ii</i> S <i>e</i> <i>a</i>	$\$$	reducir $S \rightarrow a$
(8) 0 2 2 4 5 6	<i>ii</i> S <i>e</i> S	$\$$	reducir $S \rightarrow iSeS$
(9) 0 2 4	<i>i</i> S	$\$$	reducir $S \rightarrow iS$
(10) 0 1	S	$\$$	aceptar

Figura 4.52: Acciones de análisis sintáctico con la entrada *iiae*

Con el fin de comparar, si no podemos usar una gramática ambigua para especificar instrucciones condicionales, entonces tendríamos que usar una gramática más robusta a lo largo de las líneas del ejemplo 4.16.

4.8.3 Recuperación de errores en el análisis sintáctico LR

Un analizador sintáctico LR detectará un error al consultar la tabla de acciones de análisis sintáctico y encontrar una entrada de error. Los errores nunca se detectan al consultar la tabla de *ir_A*. Un analizador sintáctico LR anunciará un error tan pronto como no haya una continuación válida para la porción de la entrada que se ha explorado hasta ese momento. Un analizador sintáctico LR canónico no realizará ni siquiera una sola reducción antes de anunciar un error. Los analizadores sintácticos SLR y LALR pueden realizar varias reducciones antes de anunciar un error, pero nunca desplazarán un símbolo de entrada erróneo hacia la pila.

En el análisis sintáctico LR, podemos implementar la recuperación de errores en modo de pánico de la siguiente manera. Exploramos la pila en forma descendente hasta encontrar un estado *s* con un *ir_A* en un no terminal *A* específico. Después, se descartan cero o más símbolos de entrada hasta encontrar un símbolo *a* que pueda seguir a *A* de manera legítima. A continuación, el analizador sintáctico mete el estado *ir_A(s, A)* en la pila y continúa con el análisis sintáctico normal. Podría haber más de una opción para el no terminal *A*. Por lo general, éstos serían no terminales que representen las piezas principales del programa, como una expresión, una instrucción o un bloque. Por ejemplo, si *A* es el no terminal *instr*, *a* podría ser un punto y coma o *,*, lo cual marca el final de una secuencia de instrucciones.

Este método de recuperación de errores trata de eliminar la frase que contiene el error sintáctico. El analizador sintáctico determina que una cadena que puede derivarse de *A* contiene un error. Parte de esa cadena ya se ha procesado, y el resultado de este procesamiento es una

secuencia de estados en la parte superior de la pila. El resto de la cadena sigue en la entrada, y el analizador sintáctico trata de omitir el resto de esta cadena buscando un símbolo en la entrada que pueda seguir de manera legítima a A . Al eliminar estados de la pila, el analizador sintáctico simula que ha encontrado una instancia de A y continúa con el análisis sintáctico normal.

Para implementar la recuperación a nivel de frase, examinamos cada entrada de error en la tabla de análisis sintáctico LR y decidimos, en base al uso del lenguaje, el error más probable del programador que pudiera ocasionar ese error. Después podemos construir un procedimiento de recuperación de errores apropiado; se supone que la parte superior de la pila y los primeros símbolos de entrada se modificarían de una forma que se considera como apropiada para cada entrada de error.

Al diseñar rutinas de manejo de errores específicas para un analizador sintáctico LR, podemos llenar cada entrada en blanco en el campo de acción con un apuntador a una rutina de error que tome la acción apropiada, seleccionada por el diseñador del compilador. Las acciones pueden incluir la inserción o eliminación de símbolos de la pila o de la entrada (o de ambas), o la alteración y transposición de los símbolos de entrada. Debemos realizar nuestras elecciones de tal forma que el analizador sintáctico LR no entre en un ciclo infinito. Una estrategia segura asegurará que por lo menos se elimine o se desplace un símbolo de entrada en un momento dado, o que la pila se reduzca si hemos llegado al final de la entrada. Debemos evitar sacar un estado de la pila que cubra un no terminal, ya que esta modificación elimina de la pila una construcción que ya se haya analizado con éxito.

Ejemplo 4.68: Considere de nuevo la siguiente gramática de expresiones:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

La figura 4.53 muestra la tabla de análisis sintáctico LR de la figura 4.49 para esta gramática, modificada para la detección y recuperación de errores. Hemos modificado cada estado que llama a una reducción específica en ciertos símbolos de entrada, mediante la sustitución de las entradas de error en ese estado por la reducción. Este cambio tiene el efecto de posponer la detección de errores hasta que se realicen una o más reducciones, pero el error seguirá atrapándose antes de que se realice cualquier desplazamiento. Las entradas restantes en blanco de la figura 4.49 se han sustituido por llamadas a las rutinas de error.

Las rutinas de error son las siguientes:

e1: Esta rutina se llama desde los estados 0, 2, 4 y 5, y todos ellos esperan el principio de un operando, ya sea un **id** o un paréntesis izquierdo. En vez de ello, se encontró +, * o el final de la entrada.

meter el estado 3 (el ir_A de los estados 0, 2, 4 y 5 en **id**);
emitir el diagnóstico “falta operando”.

e2: Se llama desde los estados 0, 1, 2, 4 y 5 al encontrar un paréntesis derecho.

eliminar el paréntesis derecho de la entrada;
emitir el diagnóstico “paréntesis derecho desbalanceado”.

ESTADO	ACCIÓN						ir_A
	id	+	*	()	\$	
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	acc	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

Figura 4.53: Tabla de análisis sintáctico LR con rutinas de error

e3: Se llama desde los estados 1 o 6 cuando se espera un operador y se encuentra un **id** o paréntesis derecho.

meter el estado 4 (correspondiente al símbolo +) en la pila;
emitter el diagnóstico “falta un operador”.

e4: Se llama desde el estado 6 cuando se encuentra el final de la entrada.

meter el estado 9 (para un paréntesis derecho) a la pila;
emitter el diagnóstico “falta paréntesis derecho”.

En la entrada errónea **id** +), la secuencia de configuraciones que introduce el analizador sintáctico se muestra en la figura 4.54. \square

4.8.4 Ejercicios para la sección 4.8

! Ejercicio 4.8.1: La siguiente es una gramática ambigua para las expresiones con n operadores binarios infijo, con n niveles distintos de precedencia:

$$E \rightarrow E \theta_1 E \mid E \theta_2 E \mid \dots \mid E \theta_n E \mid (E) \mid \mathbf{id}$$

- Como una función de n , ¿cuáles son los conjuntos de elementos SLR?
- ¿Cómo resolvería los conflictos en los elementos SLR, de manera que todos los operadores sean asociativos a la izquierda, y que θ_1 tenga precedencia sobre θ_2 , que tiene precedencia sobre θ_3 , y así sucesivamente?
- Muestre la tabla de análisis sintáctico SLR que resulta de sus decisiones en la parte (b).

PILA	SÍMBOLOS	ENTRADA	ACCIÓN
0		id +) \$	
0 3	id	+) \$	
0 1	<i>E</i>	+) \$	
0 1 4	<i>E</i> +) \$	“paréntesis derecho desbalanceado”
0 1 4	<i>E</i> +	\$	e2 elimina el paréntesis derecho “falta un operando”
0 1 4 3	<i>E</i> + id	\$	e1 mete el estado 3
0 1 4 7	<i>E</i> +	\$	en la pila
0 1	<i>E</i> +	\$	

Figura 4.54: Movimientos de análisis sintáctico y recuperación de errores realizados por un analizador sintáctico LR

- d) Repita las partes (a) y (c) para la gramática sin ambigüedad, la cual define el mismo conjunto de expresiones, como se muestra en la figura 4.55.
- e) ¿Cómo se comparan los conteos del número de conjuntos de elementos y los tamaños de las tablas para las dos gramáticas (ambigua y sin ambigüedad)? ¿Qué nos dice esa comparación acerca del uso de las gramáticas de expresiones ambiguas?

$$\begin{array}{rcl}
 E_1 & \rightarrow & E_1 \theta E_2 \mid E_2 \\
 E_2 & \rightarrow & E_2 \theta E_3 \mid E_3 \\
 & \dots & \\
 E_n & \rightarrow & E_n \theta E_{n+1} \mid E_{n+1} \\
 E_{n+1} & \rightarrow & (E_1) \mid \mathbf{id}
 \end{array}$$

Figura 4.55: Gramática sin ambigüedad para n operadores

! **Ejercicio 4.8.2:** En la figura 4.56 hay una gramática para ciertas instrucciones, similar a la que vimos en el ejercicio 4.4.12. De nuevo, e y s son terminales que representan expresiones condicionales y “otras instrucciones”, respectivamente.

- a) Construya una tabla de análisis sintáctico LR para esta gramática, resolviendo los conflictos de la manera usual para el problema del else colgante.
- b) Implemente la corrección de errores, llenando las entradas en blanco en la tabla de análisis sintáctico con acciones de reducción adicionales, o rutinas de recuperación de errores adecuadas.
- c) Muestre el comportamiento de su analizador sintáctico con las siguientes entradas:

- (i) **if** e **then** s ; **if** e **then** s **end**
(ii) **while** e **do** **begin** s ; **if** e **then** s ; **end**

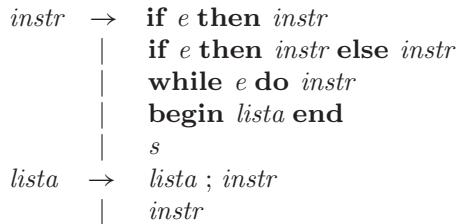


Figura 4.56: Una gramática para ciertos tipos de instrucciones

4.9 Generadores de analizadores sintácticos

En esta sección veremos cómo puede usarse un generador de analizadores sintácticos para facilitar la construcción del front-end de usuario de un compilador. Utilizaremos el generador de analizadores sintácticos LALR de nombre **Yacc** como la base de nuestra explicación, ya que implementa muchos de los conceptos que vimos en las dos secciones anteriores, y se emplea mucho. **Yacc** significa “yet another compiler-compiler” (otro compilador-de compiladores más), lo cual refleja la popularidad de los generadores de analizadores sintácticos a principios de la década de 1970, cuando S. C. Johnson creó la primera versión de **Yacc**. Este generador está disponible en forma de comando en el sistema en UNIX, y se ha utilizado para ayudar a implementar muchos compiladores de producción.

4.9.1 El generador de analizadores sintácticos Yacc

Puede construirse un traductor mediante el uso de **Yacc** de la forma que se ilustra en la figura 4.57. En primer lugar se prepara un archivo, por decir **traducir.y**, el cual contiene una especificación de **Yacc** del traductor. El siguiente comando del sistema UNIX:

```
yacc traducir.y
```

transforma el archivo **traducir.y** en un programa en C llamado **y.tab.c**, usando el método LALR descrito en el algoritmo 4.63. El programa **y.tab.c** es una representación de un analizador sintáctico LALR escrito en C, junto con otras rutinas en C que el usuario puede haber preparado. La tabla de análisis sintáctico LR se compacta según lo descrito en la sección 4.7. Al compilar **y.tab.c** junto con la biblioteca **1y** que contiene el programa de análisis sintáctico LR mediante el uso del comando:

```
cc y.tab.c -ly
```

obtenemos el programa objeto **a.out** deseado, el cual realiza la traducción especificada por el programa original en **Yacc**.⁷ Si se necesitan otros procedimientos, pueden compilarse o cargarse con **y.tab.c**, de igual forma que con cualquier programa en C.

Un programa fuente en **Yacc** tiene tres partes:

⁷El nombre **1y** es dependiente del sistema.

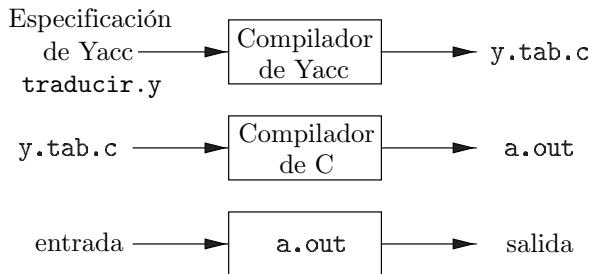


Figura 4.57: Creación de un traductor de entrada/salida con Yacc

declaraciones
 %%
 reglas de traducción
 %%
 soporte de las rutinas en C

Ejemplo 4.69: Para ilustrar cómo preparar un programa fuente en Yacc, vamos a construir una calculadora de escritorio simple que lee una expresión aritmética, la evalúa e imprime su valor numérico. Vamos a construir la calculadora de escritorio empezando con la siguiente gramática para las expresiones aritméticas:

$$\begin{aligned}
 E &\rightarrow E + T \mid T \\
 T &\rightarrow T * F \mid F \\
 R &\rightarrow (E) \mid \text{digito}
 \end{aligned}$$

El token **digito** es un solo dígito entre 0 y 9. En la figura 4.58 se muestra un programa de calculadora de escritorio en Yacc, derivado a partir de esta gramática. \square

La parte de las declaraciones

Hay dos secciones en la parte de las declaraciones de un programa en Yacc; ambas son opcionales. En la primera sección, colocamos las declaraciones ordinarias en C, delimitadas mediante `%{` y `%}`. Aquí colocamos las declaraciones de cualquier valor temporal usado por las reglas de traducción o los procedimientos de las secciones segunda y tercera. En la figura 4.58, esta sección contiene sólo la siguiente instrucción de inclusión:

```
#include <ctype.h>
```

la cual ocasiona que el preprocesador de C incluya el archivo de encabezado estándar `<ctype.h>`, el cual contiene el predicado `isdigit`.

Además, en la parte de las declaraciones se encuentran las declaraciones de los tokens de gramática. En la figura 4.58, la instrucción

```
%token DIGITO
```

```

%{
#include <ctype.h>
%}

%token DIGITO

%%
linea  :  expr '\n'          { printf("%d\n", $1); }
        ;
expr    :  expr '+' term   { $$ = $1 + $3; }
        |  term
        ;
term    :  term '*' factor { $$ = $1 * $3; }
        |  factor
        ;
factor  :  '(' expr ')'
        |  DIGITO
        ;
%%

yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGITO;
    }
    return c;
}

```

Figura 4.58: Especificación de Yacc de una calculadora de escritorio simple

declara a **DIGITO** como un token. Los tokens declarados en esta sección pueden usarse en las partes segunda y tercera de la especificación de Yacc. Si se utiliza **Lex** para crear el analizador léxico que pasa el token al analizador sintáctico **Yacc**, entonces estas declaraciones de tokens también se vuelven disponibles para el analizador generado por **Lex**, como vimos en la sección 3.5.2.

La parte de las reglas de traducción

En la parte de la especificación de Yacc después del primer par de **%%**, colocamos las reglas de traducción. Cada regla consiste en una producción gramatical y la acción semántica asociada. Un conjunto de producciones que hemos estado escribiendo como:

$$\langle \text{encabezado} \rangle \rightarrow \langle \text{cuerpo} \rangle_1 \mid \langle \text{cuerpo} \rangle_2 \mid \dots \mid \langle \text{cuerpo} \rangle_n$$

podría escribirse en Yacc de la siguiente manera:

```

⟨encabezado⟩ : ⟨cuerpo⟩1 { ⟨acción semántica⟩1 }
| ⟨cuerpo⟩2 { ⟨acción semántica⟩2 }
...
| ⟨cuerpo⟩n { ⟨acción semántica⟩n }
;

```

En una producción de **Yacc**, las cadenas sin comillas de letras y dígitos que no se declaren como tokens se consideran como no terminales. Un solo carácter entre comillas, por ejemplo 'c', se considera como el símbolo terminal c, así como el código entero para el token representado por ese carácter (es decir, **Lex** devolvería el código de carácter para 'c' al analizador sintáctico, como un entero). Los cuerpos alternativos pueden separarse mediante una barra vertical; además se coloca un punto y coma después de cada encabezado con sus alternativas y sus acciones semánticas. El primer encabezado se considera como el símbolo inicial.

Una acción semántica de **Yacc** es una secuencia de instrucciones en C. En una acción semántica, el símbolo \$\$ se refiere al valor del atributo asociado con el no terminal del encabezado, mientras que \$i se refiere al valor asociado con el *i*-ésimo símbolo gramatical (terminal o no terminal) del cuerpo. La acción semántica se realiza cada vez que reducimos mediante la producción asociada, por lo que normalmente la acción semántica calcula un valor para \$\$ en términos de los \$i's. En la especificación de **Yacc**, hemos escrito las dos producciones *E* siguientes:

$$E \rightarrow E + T \mid T$$

y sus acciones semánticas asociadas como:

```

expr : expr '+' term    { $$ = $1 + $3; }
| term
;

```

Observe que el no terminal **term** en la primera producción es el tercer símbolo gramatical del cuerpo, mientras que + es el segundo. La acción semántica asociada con la primera producción agrega el valor de la **expr** y la **term** del cuerpo, y asigna el resultado como el valor para el no terminal **expr** del encabezado. Hemos omitido del todo la acción semántica para la segunda producción, ya que copiar el valor es la acción predeterminada para las producciones con un solo símbolo gramatical en el cuerpo. En general, { \$\$ = \$1; } es la acción semántica predeterminada.

Observe que hemos agregado una nueva producción inicial:

```
linea : expr '\n' { printf("%d\n", $1); }
```

a la especificación de **Yacc**. Esta producción indica que una entrada para la calculadora de escritorio debe ser una expresión seguida de un carácter de nueva línea. La acción semántica asociada con esta producción imprime el valor decimal de la expresión que va seguida de un carácter de nueva línea.

La parte de las rutinas de soporte en C

La tercera parte de una especificación de **Yacc** consiste en las rutinas de soporte en C. Debe proporcionarse un analizador léxico mediante el nombre **yylex()**. La elección común es usar **Lex** para producir **yylex()**; vea la sección 4.9.3. Pueden agregarse otros procedimientos como las rutinas de recuperación de errores, según sea necesario.

El analizador léxico **yylex()** produce tokens que consisten en un nombre de token y su valor de atributo asociado. Si se devuelve el nombre de un token como **DIGITO**, el nombre del token debe declararse en la primera sección de la especificación de **Yacc**. El valor del atributo asociado con un token se comunica al analizador sintáctico, a través de una variable **yyval** definida por **Yacc**.

El analizador léxico en la figura 4.58 es bastante burdo. Lee un carácter de entrada a la vez, usando la función de C **getchar()**. Si el carácter es un dígito, el valor del dígito se almacena en la variable **yyval** y se devuelve el nombre de token **DIGITO**. En cualquier otro caso, se devuelve el mismo carácter como el nombre de token.

4.9.2 Uso de Yacc con gramáticas ambiguas

Ahora vamos a modificar la especificación de **Yacc**, de tal forma que la calculadora de escritorio resultante sea más útil. En primer lugar, vamos a permitir que la calculadora de escritorio evalúe una secuencia de expresiones, de una a una línea. También vamos a permitir líneas en blanco entre las expresiones. Para ello, cambiaremos la primer regla a:

```
lineas : lineas expr '\n'      { printf("%g\n", $2); }
      | lineas '\n'
      | /* vacia */
      ;
```

En **Yacc**, una alternativa vacía, como lo es la tercera línea, denota a ϵ .

En segundo lugar, debemos agrandar la clase de expresiones para incluir números en vez de dígitos individuales, y para incluir los operadores aritméticos **+**, **-**, (tanto binarios como unarios), ***** y **/**. La manera más sencilla de especificar esta clase de expresiones es utilizar la siguiente gramática ambigua:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid - E \mid \text{numero}$$

La especificación resultante de **Yacc** se muestra en la figura 4.59.

Como la gramática en la especificación de **Yacc** en la figura 4.59 es ambigua, el algoritmo **LALR** generará conflictos de acciones de análisis sintáctico. **Yacc** reporta el número de conflictos de acciones de análisis sintáctico que se generan. Podemos obtener una descripción de los conjuntos de elementos y los conflictos de acciones de análisis sintáctico si invocamos a **Yacc** con una opción **-v**. Esta opción genera un archivo adicional **y.output**, el cual contiene los corazones de los conjuntos de elementos encontrados para la gramática, una descripción de los conflictos de acciones de análisis sintáctico generados por el algoritmo **LALR**, y una representación legible de la tabla de análisis sintáctico **LR** que muestra cómo se resolvieron los conflictos de acciones de análisis sintáctico. Cada vez que **Yacc** reporta que ha encontrado

```

%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* tipo double para la pila de Yacc */
%}
%token NUMERO

%left '+', '-'
%left '*', '/'
%right UMENOS
%%

lines  : lines expr '\n'   { printf("%g\n", $2); }
        | lines '\n'
        | /* vacia */

expr   : expr '+' expr    { $$ = $1 + $3; }
        | expr '-' expr    { $$ = $1 - $3; }
        | expr '*' expr   { $$ = $1 * $3; }
        | expr '/' expr   { $$ = $1 / $3; }
        | '(' expr ')'
        | '-' expr %prec UMENOS { $$ = - $2; }
        | NUMERO
        ;
%%

yylex() {
    int c;
    while ( ( c = getchar() ) == ' ' );
    if ( ( c == '.' ) || (isdigit(c)) ) {
        unget(c, stdin);
        scanf( "%lf", &yyval);
        return NUMERO;
    }
    return c;
}

```

Figura 4.59: Especificación de Yacc para una calculadora de escritorio más avanzada

conflictos de acciones de análisis sintáctico, es conveniente crear y consultar el archivo `y.output` para ver por qué se generaron los conflictos de acciones de análisis sintáctico y si se resolvieron en forma correcta.

A menos que se indique lo contrario, `Yacc` resolverá todos los conflictos de las acciones de análisis sintáctico mediante las siguientes dos reglas:

1. Un conflicto de reducción/reducción se resuelve eligiendo la producción en conflicto que se presente primero en la especificación de `Yacc`.
2. Un conflicto de desplazamiento/reducción se resuelve a favor del desplazamiento. Esta regla resuelve en forma correcta el conflicto de desplazamiento/reducción ocasionado por la ambigüedad del `else` colgante.

Como estas reglas predeterminadas no siempre pueden ser lo que desea el escritor de compiladores, `Yacc` proporciona un mecanismo general para resolver los conflictos de desplazamiento/reducción. En la porción de las declaraciones, podemos asignar precedencias y asociatividades a las terminales. La siguiente declaración:

```
%left '+' '-'
```

hace que `+` y `-` sean de la misma precedencia y asociativos a la izquierda. Podemos declarar un operador como asociativo a la derecha si escribimos lo siguiente:

```
%right '^'
```

y podemos forzar a un operador para que sea un operador binario sin asociatividad (es decir, no pueden combinarse dos ocurrencias del operador de ninguna manera) escribiendo lo siguiente:

```
%nonassoc '<'
```

Los tokens reciben las precedencias en el orden en el que aparecen en la parte de las declaraciones, en donde la menor precedencia va primero. Los tokens en la misma declaración tienen la misma precedencia. Así, la declaración

```
%right UMENOS
```

en la figura 4.59 proporciona al token `UMENOS` un nivel de precedencia mayor que el de las cinco terminales anteriores.

`Yacc` resuelve los conflictos de desplazamiento/reducción adjuntando una precedencia y una asociatividad a cada una de las producciones involucradas en un conflicto, así como también a cada terminal involucrada en un conflicto. Si debe elegir entre desplazar el símbolo de entrada `a` y reducir mediante la producción $A \rightarrow \alpha$, `Yacc` reduce si la precedencia de la producción es mayor que la de `a`, o si las precedencias son iguales y la asociatividad de la producción es `left`. En cualquier otro caso, el desplazamiento es la acción elegida.

Por lo general, la precedencia de una producción se considera igual a la de su terminal por la derecha. Ésta es la decisión sensata en la mayoría de los casos. Por ejemplo, dadas las siguientes producciones:

$$E \rightarrow E + E \mid E \cdot E$$

sería preferible reducir mediante $E \rightarrow E+E$ con el símbolo de anticipación $+$, ya que el $+$ en el cuerpo tiene la misma precedencia que el símbolo de anticipación, pero es asociativo a la izquierda. Con el símbolo de anticipación $*$, sería más preferible desplazar, ya que éste tiene una precedencia mayor que la del $+$ en la producción.

En esas situaciones en las que el terminal por la derecha no proporciona la precedencia apropiada a una producción, podemos forzar el uso de una precedencia si adjuntamos a una producción la siguiente etiqueta:

```
%prec {terminal}
```

La precedencia y la asociatividad de la producción serán entonces iguales que la del terminal, que se supone está definida en la sección de declaraciones. Yacc no reporta los conflictos de desplazamiento/reducción que se resuelven usando este mecanismo de precedencia y asociatividad.

Este “terminal” podría ser un receptáculo como **UMENOS** en la figura 4.59: el analizador léxico no devuelve este terminal, sino que está declarado con el único fin de definir una precedencia para una producción. En la figura 4.59, la declaración

```
%right UMENOS
```

asigna al token **UMENOS** una precedencia mayor que la de $*$ y $/$. En la parte de las reglas de traducción, la etiqueta:

```
%prec UMENOS
```

al final de la producción

```
expr : '-' expr
```

hace que el operador de resta unario en esta producción tenga una menor precedencia que cualquier otro operador.

4.9.3 Creación de analizadores léxicos de Yacc con Lex

Lex se diseñó para producir analizadores léxicos que pudieran utilizarse con Yacc. La biblioteca 11 de Lex proporciona un programa controlador llamado **yylex()**, el nombre que Yacc requiere para su analizador léxico. Si se utiliza Lex para producir el analizador léxico, sustituimos la rutina **yylex()** en la tercera parte de la especificación de Yacc con la siguiente instrucción:

```
#include "lex.yy.c"
```

y hacemos que cada acción de Lex devuelva un terminal conocido a Yacc. Al usar la instrucción **#include "lex.yy.c"**, el programa **yylex** tiene acceso a los nombres de Yacc para los tokens, ya que el archivo de salida de Lex se compila como parte del archivo de salida **y.tab.c** de Yacc.

En el sistema UNIX, si la especificación de Lex está en el archivo **primero.1** y la especificación de Yacc en **segundo.y**, podemos escribir lo siguiente:

```
lex primero.l
yacc segundo.y
cc y.tab.c -ly -ll
```

para obtener el traductor deseado.

La especificación de Lex en la figura 4.60 puede usarse en vez del analizador léxico de la figura 4.59. El último patrón, que significa “cualquier carácter”, debe escribirse como `\nl.` ya que el punto en Lex coincide con cualquier carácter, excepto el de nueva línea.

```
numero      [0-9]+\e.?|[0-9]*\e.[0-9]+
%
[ ]         { /* omitir espacios en blanco */ }
{numero}    { sscanf(yytext, "%lf", &yyval);
              return NUMERO; }
\nl.        { return yytext[0]; }
```

Figura 4.60: Especificación de Lex para yylex() en la figura 4.59

4.9.4 Recuperación de errores en Yacc

En Yacc, la recuperación de errores utiliza una forma de producciones de error. En primer lugar, el usuario decide qué no terminales “importantes” tendrán la recuperación de errores asociado con ellas. Las elecciones típicas son cierto subconjunto de los no terminales que generan expresiones, instrucciones, bloques y funciones. Después el usuario agrega a las producciones de error gramaticales de la forma $A \rightarrow \text{error } \alpha$, en donde A es un no terminal importante y α es una cadena de símbolos gramaticales, tal vez la cadena vacía; **error** es una palabra reservada de Yacc. Yacc generará un analizador sintáctico a partir de dicha especificación, tratando a las producciones de error como producciones ordinarias.

No obstante, cuando el analizador sintáctico generado por Yacc encuentra un error, trata a los estados cuyos conjuntos de elementos contienen producciones de error de una manera especial. Al encontrar un error, Yacc saca símbolos de su pila hasta que encuentra el estado en la parte superior de su pila cuyo conjunto subyacente de elementos incluya a un elemento de la forma $A \rightarrow \cdot \text{error } \alpha$. Después, el analizador sintáctico “desplaza” un token ficticio **error** hacia la pila, como si hubiera visto el token **error** en su entrada.

Cuando α es ϵ , se realiza una reducción a A de inmediato y se invoca la acción semántica asociada con la producción $A \rightarrow \cdot \text{error}$ (que podría ser una rutina de recuperación de errores especificada por el usuario). Después, el analizador sintáctico descarta los símbolos de entrada hasta que encuentra uno con el cual pueda continuar el análisis sintáctico normal.

Si α no está vacía, Yacc sigue recorriendo la entrada, ignorando los símbolos hasta que encuentra una subcadena que pueda reducirse a α . Si α consiste sólo en terminales, entonces busca esta cadena de terminales en la entrada y los “reduce” al desplazarlas hacia la pila. En este punto, el analizador sintáctico tendrá a **error** α en la parte superior de su pila. Después, el analizador sintáctico reducirá **error** α a A y continuará con el análisis sintáctico normal.

Por ejemplo, una producción de error de la siguiente forma:

```

%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* tipo double para la pila de Yacc */
%}
%token NUMERO

%left '+' '-'
%left '*' '/'
%right UMENOS
%%

lineas : lineas expr '\n'  { printf("%g\n", $2); }
| lineas '\n'
| /* vacia */
| error '\n' { yyerror("reintroduzca linea anterior:");
               yyerrok; }

;
expr  : expr '+' expr      { $$ = $1 + $3; }
| expr '-' expr      { $$ = $1 - $3; }
| expr '*' expr      { $$ = $1 * $3; }
| expr '/' expr      { $$ = $1 / $3; }
| '(' expr ')'
| '-' expr %prec UMENOS { $$ = - $2; }
| NUMERO
;
%%

#include "lex.yy.c"

```

Figura 4.61: Calculadora de escritorio con recuperación de errores

instr → **error** ;

especificaría al analizador sintáctico que debe omitir lo que esté más allá después del siguiente punto y coma al ver un error, y debe suponer que se ha encontrado una instrucción. La rutina semántica para esta producción de error no tendría que manipular la entrada, pero podría generar un mensaje de diagnóstico y establecer una bandera para inhibir la generación de código objeto, por ejemplo.

Ejemplo 4.70: La figura 4.61 muestra la calculadora de escritorio Yacc de la figura 4.59, con la siguiente producción de error:

lineas : error '\n'

Esta producción de error hace que la calculadora de escritorio suspenda el análisis sintáctico normal al encontrar un error sintáctico en una línea de entrada. Al encontrar el error, el ana-

lizador sintáctico en la calculadora de escritorio empieza a sacar símbolos de su pila hasta que encuentra un estado con una acción de desplazamiento en el token **error**. El estado 0 es un estado de este tipo (en este ejemplo, es el único estado así), ya que sus elementos incluyen:

$$\text{lineas} \rightarrow \cdot \text{error} \text{ '}\backslash\text{n}\text{'}$$

Además, el estado 0 siempre se encuentra en la parte inferior de la pila. El analizador sintáctico desplaza el token **error** hacia la pila y después continúa ignorando símbolos en la entrada hasta encontrar un carácter de nueva línea. En este punto, el analizador sintáctico desplaza el carácter de nueva línea hacia la pila, reduce **error** 'n' a *lineas*, y emite el mensaje de diagnóstico "reintroduzca linea anterior:". La rutina especial de Yacc llamada `yyerrok` restablece el analizador sintáctico a su modo normal de operación. \square

4.9.5 Ejercicios para la sección 4.9

! Ejercicio 4.9.1: Escriba un programa en Yacc que reciba expresiones booleanas como entrada [según lo indicado por la gramática del ejercicio 4.2.2(g)] y produzca el valor verdadero de las expresiones.

! Ejercicio 4.9.2: Escriba un programa en Yacc que reciba listas (según lo definido por la gramática del ejercicio 4.2.2(e), pero con cualquier carácter individual como elemento, no sólo *a*) y produzca como salida una representación lineal de la misma lista; por ejemplo, una lista individual de los elementos, en el mismo orden en el que aparecen en la entrada.

! Ejercicio 4.9.3: Escriba un programa en Yacc que indique si su entrada es un *palíndromo* (secuencia de caracteres que se leen igual al derecho y al revés).

!! Ejercicio 4.9.4: Escriba un programa en Yacc que reciba expresiones regulares (según lo definido por la gramática del ejercicio 4.2.2(d), pero con cualquier carácter individual como argumento, no sólo *a*) y produzca como salida una tabla de transición para un autómata finito no determinista que reconozca el mismo lenguaje.

4.10 Resumen del capítulo 4

- ◆ *Analizadores sintácticos.* Un analizador sintáctico recibe como entrada tokens del analizador léxico, y trata los nombres de los tokens como símbolos terminales de una gramática libre de contexto. Después, el analizador construye un árbol de análisis sintáctico para su secuencia de tokens de entrada; el árbol de análisis sintáctico puede construirse en sentido figurado (pasando por los pasos de derivación correspondientes) o en forma literal.
- ◆ *Gramáticas libres de contexto.* Una gramática especifica un conjunto de símbolos terminales (entradas), otro conjunto de no terminales (símbolos que representan construcciones sintácticas) y un conjunto de producciones, cada una de las cuales proporciona una forma en la que se pueden construir las cadenas representadas por un no terminal, a partir de símbolos terminales y cadenas representados por otros no terminales. Una producción consiste en un encabezado (el no terminal a sustituir) y un cuerpo (la cadena de símbolos gramaticales de sustitución).

- ◆ *Derivaciones.* Al proceso de empezar con el no terminal inicial de una gramática y sustituirlo en forma repetida por el cuerpo de una de sus producciones se le conoce como derivación. Si siempre se sustituye el no terminal por la izquierda (o por la derecha), entonces a la derivación se le llama por la izquierda (o respectivamente, por la derecha).
- ◆ *Árboles de análisis sintáctico.* Un árbol de análisis sintáctico es una imagen de una derivación, en la cual hay un nodo para cada no terminal que aparece en la derivación. Los hijos de un nodo son los símbolos mediante los cuales se sustituye este no terminal en la derivación. Hay una correspondencia de uno a uno entre los árboles de análisis sintáctico, las derivaciones por la izquierda y las derivaciones por la derecha de la misma cadena de terminales.
- ◆ *Ambigüedad.* Una gramática para la cual cierta cadena de terminales tiene dos o más árboles de análisis sintáctico distintos, o en forma equivalente, dos o más derivaciones por la izquierda, o dos o más derivaciones por la derecha, se considera ambigua. En la mayoría de los casos de interés práctico, es posible rediseñar una gramática ambigua de tal forma que se convierta en una gramática sin ambigüedad para el mismo lenguaje. No obstante, las gramáticas ambiguas con ciertos trucos aplicados nos llevan algunas veces a la producción de analizadores sintácticos más eficientes.
- ◆ *Análisis sintáctico descendente y ascendente.* Por lo general, los analizadores sintácticos se diferencian en base a si trabajan de arriba hacia abajo (si empiezan con el símbolo inicial de la gramática y construyen el árbol de análisis sintáctico partiendo de la parte superior) o de abajo hacia arriba (si empiezan con los símbolos terminales que forman las hojas del árbol de análisis sintáctico y construyen el árbol partiendo de la parte inferior). Los analizadores sintácticos descendentes incluyen los analizadores sintácticos con descenso recursivo y LL, mientras que las formas más comunes de analizadores sintácticos ascendentes son analizadores sintácticos LR.
- ◆ *Diseño de gramáticas.* A menudo, las gramáticas adecuadas para el análisis sintáctico descendente son más difíciles de diseñar que las utilizadas por los analizadores sintácticos ascendentes. Es necesario eliminar la recursividad por la izquierda, una situación en la que un no terminal deriva a una cadena que empieza con el mismo no terminal. También debemos factorizar por la izquierda; las producciones de grupo para el mismo no terminal que tengan un prefijo común en el cuerpo.
- ◆ *Analizadores sintácticos de descenso recursivo.* Estos analizadores sintácticos usan un procedimiento para cada no terminal. El procedimiento analiza su entrada y decide qué producción aplicar para su no terminal. Los terminales en el cuerpo de la producción se relacionan con la entrada en el momento apropiado, mientras que las no terminales en el cuerpo producen llamadas a su procedimiento. El rastreo hacia atrás, en el caso de cuando se elige la producción incorrecta, es una posibilidad.
- ◆ *Analizadores sintácticos LL(1).* Una gramática en la que es posible elegir la producción correcta con la cual se pueda expandir un no terminal dado, con solo analizar el siguiente símbolo de entrada, se conoce como LL(1). Estas gramáticas nos permiten construir una tabla de análisis sintáctico predictivo que proporcione, para cada no terminal y cada símbolo de preanálisis, la elección de la producción correcta. La corrección de errores se puede facilitar al colocar las rutinas de error en algunas, o en todas las entradas en la tabla que no tengan una producción legítima.

- ◆ *Análisis sintáctico de desplazamiento-reducción.* Por lo general, los analizadores sintácticos ascendentes operan mediante la elección, en base al siguiente símbolo de entrada (símbolo de anticipación) y el contenido de la pila, de si deben desplazar la siguiente entrada hacia la pila, o reducir algunos símbolos en la parte superior de la misma. Un paso de reducción toma un cuerpo de producción de la parte superior de la pila y lo sustituye por el encabezado de la producción.
- ◆ *Prefijos viables.* En el análisis sintáctico de desplazamiento-reducción, el contenido de la pila siempre es un prefijo viable; es decir, un prefijo de cierta forma de frase derecha que termina a la derecha, no más allá del final del mango de ésta. El mango es la subcadena que se introdujo en el último paso de la derivación por la derecha de esa forma de frase.
- ◆ *Elementos válidos.* Un elemento es una producción con un punto en alguna parte del cuerpo. Un elemento es válido para un prefijo viable si la producción de ese elemento se utiliza para generar el mango, y el prefijo viable incluye todos esos símbolos a la izquierda del punto, pero no los que están abajo.
- ◆ *Analizadores sintácticos LR.* Cada uno de los diversos tipos de analizadores sintácticos LR opera construyendo primero los conjuntos de elementos válidos (llamados estados LR) para todos los prefijos viables posibles, y llevando el registro del estado para cada prefijo en la pila. El conjunto de elementos válidos guía la decisión de análisis sintáctico de desplazamiento-reducción. Preferimos reducir si hay un elemento válido con el punto en el extremo derecho del cuerpo, y desplazamos el símbolo de anticipación hacia la pila si ese símbolo aparece justo a la derecha del punto, en algún elemento válido.
- ◆ *Analizadores sintácticos LR simples.* En un analizador sintáctico SLR, realizamos una reducción implicada por un elemento válido con un punto en el extremo derecho, siempre y cuando el símbolo de anticipación pueda seguir el encabezado de esa producción en alguna forma de frase. La gramática es SLR, y este método puede aplicarse si no hay conflictos de acciones de análisis sintáctico; es decir, que para ningún conjunto de elementos y para ningún símbolo de anticipación haya dos producciones mediante las cuales se pueda realizar una reducción, ni exista la opción de reducir o desplazar.
- ◆ *Analizadores sintácticos LR canónicos.* Esta forma más compleja de analizador sintáctico LR utiliza elementos que se aumentan mediante el conjunto de símbolos de anticipación que pueden seguir el uso de la producción subyacente. Las reducciones sólo se eligen cuando hay un elemento válido con el punto en el extremo derecho, y el símbolo actual de anticipación es uno de los permitidos para este elemento. Un analizador sintáctico LR canónico puede evitar algunos de los conflictos de acciones de análisis sintáctico que están presentes en los analizadores sintácticos SLR; pero a menudo tiene más estados que el analizador sintáctico SLR para la misma gramática.
- ◆ *Analizadores sintácticos LR con lectura anticipada.* Los analizadores sintácticos LALR ofrecen muchas de las ventajas de los analizadores sintácticos SLR y LR canónicos, mediante la combinación de estados que tienen los mismos corazones (conjuntos de elementos, ignorando los conjuntos asociados de símbolos de anticipación). Por ende, el número de estados es el mismo que el del analizador sintáctico SLR, pero algunos conflictos de acciones de análisis sintáctico presentes en el analizador sintáctico SLR pueden eliminarse en el analizador sintáctico LALR. Los analizadores sintácticos LALR se han convertido en el método más usado.

- ◆ *Análisis sintáctico ascendente de gramáticas ambiguas.* En muchas situaciones importantes, como en el análisis sintáctico de expresiones aritméticas, podemos usar una gramática ambigua y explotar la información adicional, como la precedencia de operadores, para resolver conflictos entre desplazar y reducir, o entre la reducción mediante dos reducciones distintas. Por ende, las técnicas de análisis sintáctico LR se extienden a muchas gramáticas ambiguas.
- ◆ **Yacc.** El generador de analizadores sintácticos **Yacc** recibe una gramática (posiblemente) ambigua junto con la información de resolución de conflictos, y construye los estados del LALR. Después produce una función que utiliza estos estados para realizar un análisis sintáctico ascendente y llama a una función asociada cada vez que se realiza una reducción.

4.11 Referencias para el capítulo 4

El formalismo de las gramáticas libres de contexto se originó con Chomsky [5], como parte de un estudio acerca del lenguaje natural. La idea también se utilizó en la descripción sintáctica de dos de los primeros lenguajes: Fortran por Backus [2] y Algol 60 por Naur [26]. El erudito Panini ideó una notación sintáctica equivalente para especificar las reglas de la gramática Sanskrit entre los años 400 a.C. y 200 a.C. [19].

Cantor [4] y Floyd [13] fueron los primeros que observaron el fenómeno de la ambigüedad. La Forma Normal de Chomsky (ejercicio 4.4.8) proviene de [6]. La teoría de las gramáticas libres de contexto se resume en [17].

El análisis sintáctico de descenso recursivo fue el método preferido para los primeros compiladores, como [16], y los sistemas para escribir compiladores, como META [28] y TMG [25]. Lewis y Stearns [24] introdujeron las gramáticas LL. El ejercicio 4.4.5, la simulación en tiempo lineal del descenso recursivo, proviene de [3].

Una de las primeras técnicas de análisis sintáctico, que se debe a Floyd [14], implicaba la precedencia de los operadores. Wirth y Weber [29] generalizaron la idea para las partes del lenguaje que no involucran operadores. Estas técnicas se utilizan raras veces hoy en día, pero podemos verlas como líderes en una cadena de mejoras para el análisis sintáctico LR.

Knuth [22] introdujo los analizadores sintácticos LR, y las tablas de análisis sintáctico LR canónicas se originaron ahí. Este método no se consideró práctico, debido a que las tablas de análisis sintáctico eran más grandes que las memorias principales de las computadoras típicas de esa época, hasta que Korenjak [23] proporcionó un método para producir tablas de análisis sintáctico de un tamaño razonable para los lenguajes de programación comunes. DeRemer desarrolló los métodos LALR [8] y SLR [9] que se usan en la actualidad. La construcción de las tablas de análisis sintáctico LR para las gramáticas ambiguas provienen de [1] y [12].

El generador **Yacc** de Johnson demostró con mucha rapidez la habilidad práctica de generar analizadores sintácticos con un generador de analizadores sintácticos LALR para los compiladores de producción. El manual para el generador de analizadores sintácticos **Yacc** se encuentra en [20]. La versión de código-abierto, **Bison**, se describe en [10]. Hay un generador de analizadores sintácticos similar llamado **CUP** [18], el cual se basa en LALR y soporta acciones escritas en Java. Los generadores de analizadores sintácticos descendentes incluyen a **Antlr** [27], un generador de analizadores sintácticos de descenso recursivo que acepta acciones en C++, Java o C#, y **LLGen** [15], que es un generador basado en LL(1).

Dain [7] proporciona una bibliografía acerca del manejo de errores sintácticos.

El algoritmo de análisis sintáctico de programación dinámica de propósito general descrito en el ejercicio 4.4.9 lo inventaron en forma independiente J. Cocke (sin publicar), Younger [30] y Kasami [21]; de aquí que se le denomine “algoritmo CYK”. Hay un algoritmo más complejo de propósito general que creó Earley [11], que tabula los elementos LR para cada subcadena de la entrada dada; este algoritmo, que también requiere un tiempo $O(n^3)$ en general, sólo requiere un tiempo $O(n^2)$ en las gramáticas sin ambigüedad.

1. Aho, A. V., S. C. Johnson y J. D. Ullman, “Deterministic parsing of ambiguous grammars”, *Comm. ACM* **18**:8 (Agosto, 1975), pp. 441-452.
2. Backus, J. W, “The syntax and semantics of the proposed international algebraic language of the Zurich-ACM-GAMM Conference”, *Proc. Intl. Conf. Information Processing*, UNESCO, París (1959), pp. 125-132.
3. Birman, A. y J. D. Ullman, “Parsing algorithms with backtrack”, *Information and Control* **23**:1 (1973), pp. 1-34.
4. Cantor, D. C., “On the ambiguity problem of Backus systems”, *J. ACM* **9**:4 (1962), pp. 477-479.
5. Chomsky, N., “Three models for the description of language”, *IRE Trans. on Information Theory* **IT-2**:3 (1956), pp. 113-124.
6. Chomsky, N., “On certain formal properties of grammars”, *Information and Control* **2**:2 (1959), pp. 137-167.
7. Dain, J., “Bibliography on Syntax Error Handling in Language Translation Systems”, 1991. Disponible en el grupo de noticias `comp.compilers`; vea <http://compilers.iecc.com/comparch/article/91-04-050>.
8. DeRemer, F., “Practical Translators for LR(k) Languages”, Tesis Ph.D., MIT, Cambridge, MA, 1969.
9. DeRemer, F., “Simple LR(k) grammars”, *Comm. ACM* **14**:7 (Julio, 1971), pp. 453-460.
10. Donnelly, C. y R. Stallman, “Bison: The YACC-compatible Parser Generator”, <http://www.gnu.org/software/bison/manual/>.
11. Earley, J., “An efficient context-free parsing algorithm”, *Comm. ACM* **13**:2 (Febrero, 1970), pp. 94-102.
12. Earley, J., “Ambiguity and precedence in syntax description”, *Acta Informatica* **4**:2 (1975), pp. 183-192.
13. Floyd, R. W., “On ambiguity in phrase-structure languages”, *Comm. ACM* **5**:10 (Octubre, 1962), pp. 526-534.
14. Floyd, R. W., “Syntactic analysis and operator precedence”, *J. ACM* **10**:3 (1963), pp. 316-333.

15. Grune, D. y C. J. H. Jacobs, “A programmer-friendly LL(1) parser generator”, *Software Practice and Experience* **18**:1 (Enero, 1988), pp. 29-38. Vea también <http://www.cs.vu.nl/~ceriel/LLgen.html>.
16. Hoare, C. A. R., “Report on the Elliott Algol translator”, *Computer J.* **5**:2 (1962), pp. 127-129.
17. Hopcroft, J. E., R. Motwani y J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Boston, MA, 2001.
18. Hudson, S. E. *et al.*, “CUP LALR Parser Generator in Java”, Disponible en <http://www2.cs.tum.edu/projects/cup/>.
19. Ingberman, P. Z., “Panini-Backus form suggested”, *Comm. ACM* **10**:3 (Marzo, 1967), p. 137.
20. Johnson, S. C., “Yacc — Yet Another Compiler Compiler”, Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, 1975. Disponible en <http://dinosaur.compilertools.net/yacc/>.
21. Kasami, T., “An efficient recognition and syntax analysis algorithm for context-free languages”, AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.
22. Knuth, D. E., “On the translation of languages from left to right”, *Information and Control* **8**:6 (1965), pp. 607-639.
23. Korenjak, A. J., “A practical method for constructing LR(k) processors”, *Comm. ACM* **12**:11 (Noviembre, 1969), pp. 613-623.
24. Lewis, P. M. II y R. E. Stearns, “Syntax-directed transduction”, *J. ACM* **15**:3 (1968), pp. 465-488.
25. McClure, R. M., “TMG — a syntax-directed compiler”, *proc. 20th ACM Natl. Conf.* (1965), pp. 262-274.
26. Naur, P. *et al.*, “Report on the algorithmic language ALGOL 60”, *Comm. ACM* **3**:5 (Mayo, 1960), pp. 299-314. Vea también *Comm. ACM* **6**:1 (Enero, 1963), pp. 1-17.
27. Parr, T., “ANTLR”, <http://www.antlr.org/>.
28. Schorre, D. V., “Meta-II: a syntax-oriented compiler writing language”, *Proc. 19th ACM Natl. Conf.* (1964), pp. D1.3-1-D1.3-11.
29. Wirth, N. y H. Weber, “Euler: a generalization of Algol and its formal definition: Part I”, *Comm. ACM* **9**:1 (Enero, 1966), pp. 13-23.
30. Younger, D. H., “Recognition and parsing of context-free languages in time n^3 ”, *Information and Control* **10**:2 (1967), pp. 189-208.

Capítulo 5

Traducción orientada por la sintaxis

Este capítulo desarrolla el tema de la sección 2.3: la traducción de los lenguajes guiados por las gramáticas libres de contexto. Las técnicas de traducción de este capítulo se aplicarán en el capítulo 6 a la comprobación de tipos y la generación de código intermedio. Estas técnicas son también útiles en la implementación de pequeños lenguajes para tareas especializadas; este capítulo incluye un ejemplo de composición tipográfica.

Para asociar la información con una construcción del lenguaje, adjuntamos *atributos* al (los) símbolo(s) gramatical(es) que representa(n) la construcción, como vimos en la sección 2.3.2. Una definición orientada por la sintaxis especifica los valores de los atributos mediante la asociación de las reglas semánticas con las producciones gramaticales. Por ejemplo, un traductor de infijo a postfijo podría tener la siguiente producción con la siguiente regla:

PRODUCCIÓN	REGLA SEMÁNTICA	
$E \rightarrow E_1 + T$	$E.codigo = E_1.codigo \parallel T.codigo \parallel '+'$	(5.1)

Esta producción tiene dos no terminales, E y T ; el subíndice en E_1 diferencia la ocurrencia de E en el cuerpo de la producción de la ocurrencia de E como el encabezado. Tanto E como T tienen un atributo *codigo* con valor de cadena. La regla semántica especifica que la cadena $E.codigo$ debe formarse mediante la concatenación de $E_1.codigo$, $T.codigo$ y el carácter $'+'$. Aunque la regla deja explícito que la traducción de E se conforma a partir de las traducciones de E_1 , T y $'+'$, puede ser ineficiente implementar la traducción en forma directa, mediante la manipulación de cadenas.

Como vimos en la sección 2.3.5, un esquema de traducción orientado por la sintaxis incrusta fragmentos de programa, llamados acciones semánticas, con cuerpos de producciones, como en

$$E \rightarrow E_1 + T \{ \text{print } '+' \} \quad (5.2)$$

Por convención, las acciones semánticas se encierran entre llaves (si las llaves ocurren como símbolos gramaticales, las encerramos entre comillas sencillas, como en $'\{'$ y $\}'$). La posición

de una acción semántica en el cuerpo de una producción determina el orden en el que se ejecuta la acción. En la producción (5.2), la acción ocurre al final, después de todos los símbolos gramaticales; en general, las acciones semánticas pueden ocurrir en cualquier posición dentro del cuerpo de una producción.

Entre las dos notaciones, las definiciones dirigidas por la sintaxis pueden ser más legibles, y por ende más útiles para las especificaciones. No obstante, los esquemas de traducción pueden ser más eficientes y, por lo tanto, más útiles para las implementaciones.

El método más general para la traducción orientada por la sintaxis es construir un árbol de análisis sintáctico o un árbol sintáctico, y después calcular los valores de los atributos en los nodos del árbol, visitándolas. En muchos casos, la traducción puede realizarse durante el análisis sintáctico sin construir un árbol explícito. Por lo tanto, vamos a estudiar una clase de traducciones orientadas a la sintaxis, conocidas como “traducciones con atributos heredados por la izquierda” (L indica de izquierda a derecha), los cuales abarcan prácticamente todas las traducciones que pueden realizarse durante el análisis sintáctico. También estudiaremos una clase más pequeña, llamada “traducciones con atributos sintetizados” (S de sintetizados), las cuales pueden realizarse con facilidad en conexión con un análisis sintáctico ascendente.

5.1 Definiciones dirigidas por la sintaxis

Una *definición dirigida por la sintaxis* es una gramática libre de contexto, junto con atributos y reglas. Los atributos sintetizados se asocian con los símbolos gramaticales y las reglas se asocian con las producciones. Si X es un símbolo y a es uno de sus atributos, entonces escribimos $X.a$ para denotar el valor de a en el nodo específico de un árbol de análisis sintáctico, etiquetado como X . Si implementamos los nodos del árbol de análisis sintáctico mediante registros u objetos, entonces los atributos de X pueden implementarse mediante campos de datos en los registros, que representen los nodos para X . Los atributos pueden ser de cualquier tipo: por ejemplo, números, tipos, referencias de tablas o cadenas. Las cadenas pueden incluso ser secuencias largas de código, por decir código tenemos el lenguaje intermedio utilizado por un compilador.

5.1.1 Atributos heredados y sintetizados

Vamos a manejar dos tipos de atributos para los no terminales:

1. Un *atributo sintetizado* para un no terminal A en un nodo N de un árbol sintáctico se define mediante una regla semántica asociada con la producción en N . Observe que la producción debe tener a A como su encabezado. Un atributo sintetizado en el nodo N se define sólo en términos de los valores de los atributos en el hijo de N , y en el mismo N .
2. Un *atributo heredado* para un no terminal B en el nodo N de un árbol de análisis sintáctico se define mediante una regla semántica asociada con la producción en el padre de N . Observe que la producción debe tener a B como un símbolo en su cuerpo. Un atributo heredado en el nodo N se define sólo en términos de los valores de los atributos en el padre de N , en el mismo N y en sus hermanos.

Una definición alternativa de los atributos heredados

No se habilitan traducciones adicionales si permitimos que un atributo heredado $B.c$ en un nodo N se defina en términos de los valores de los atributos en los hijos de N , así como en el mismo N , en su padre y en sus hermanos. Dichas reglas pueden “simularse” mediante la creación de atributos adicionales de B , por ejemplo, $B.c_1, B.c_2, \dots$. Éstos son atributos sintetizados que copian los atributos necesarios de los hijos del nodo etiquetado como B . Despues calculamos a $B.c$ como un atributo heredado, usando los atributos $B.c_1, B.c_2, \dots$ en vez de los atributos en el hijo. Dichos atributos sintetizados se necesitan raras veces en la práctica.

Aunque no permitimos que un atributo heredado en el nodo N se defina en términos de los valores de los atributos en el hijo del nodo N , sí permitimos que un atributo sintetizado en el nodo N se defina en términos de los valores de los atributos heredados en el mismo nodo N .

Los terminales pueden tener atributos sintetizados, pero no atributos heredados. Los atributos para los terminales tienen valores léxicos que suministra el analizador léxico; no hay reglas semánticas en la misma definición dirigida por la sintaxis para calcular el valor de un atributo para un terminal.

Ejemplo 5.1: La definición dirigida por la sintaxis en la figura 5.1 se basa en nuestra conocida gramática para las expresiones aritméticas con los operadores $+$ y $*$. Evalúa las expresiones que terminan con un marcador final **n**. En la definición dirigida por la sintaxis, cada una de los no terminales tiene un solo atributo sintetizado, llamado *val*. También suponemos que el terminal **dígito** tiene un atributo sintetizado *valex*, el cual es un valor entero que devuelve el analizador léxico.

PRODUCCIÓN	REGLAS SEMÁNTICAS
1) $L \rightarrow E \text{ n}$	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F.val = E.val$
7) $F \rightarrow \text{dígito}$	$F.val = \text{dígito}.lexval$

Figura 5.1: Definición orientada por la sintaxis de una calculadora de escritorio simple

La regla para la producción 1, $L \rightarrow E \text{ n}$, establece $L.val$ a $E.val$, que veremos que es el valor numérico de toda la expresión.

La producción 2, $E \rightarrow E_1 + T$, también tiene una regla, la cual calcula el atributo *val* para el encabezado E como la suma de los valores en E_1 y T . En cualquier nodo N de un árbol sintáctico, etiquetado como E , el valor de *val* para E es la suma de los valores de *val* en los hijos del nodo N , etiquetados como E y T .

La producción 3, $E \rightarrow T$, tiene una sola regla que define el valor de *val* para *E* como el mismo que el valor de *val* en el hijo para *T*. La producción 4 es similar a la segunda producción; su regla multiplica los valores en los hijos, en vez de sumarlos. Las reglas para las producciones 5 y 6 copian los valores en un hijo, como el de la tercera producción. La producción 7 proporciona a *F.val* el valor de un dígito; es decir, el valor numérico del token **dígito** que devolvió el analizador léxico. \square

A una definición dirigida por la sintaxis que sólo involucra atributos sintetizados se le conoce como definición dirigida por la sintaxis con *atributos sintetizados*; la definición dirigida por la sintaxis en la figura 5.1 tiene esta propiedad. En una definición dirigida por la sintaxis con atributos sintetizados, cada regla calcula un atributo para el no terminal en el encabezado de una producción, a partir de los atributos que se toman del cuerpo de la producción.

Por simpleza, los ejemplos en esta sección tienen reglas semánticas sin efectos adicionales. En la práctica, es conveniente permitir que las definiciones dirigidas por la sintaxis tengan efectos adicionales limitados, como imprimir el resultado calculado por una calculadora de escritorio o interactuar con una tabla de símbolos. Una vez que veamos el orden de evaluación de los atributos en la sección 5.2, permitiremos que las reglas semánticas calculen funciones arbitrarias, lo cual es probable que involucre efectos adicionales.

Una definición dirigida por la sintaxis con atributos sintetizados puede implementarse de manera natural, en conjunción con un analizador sintáctico LR. De hecho, la definición dirigida por la sintaxis en la figura 5.1 es un reflejo del programa de Yacc de la figura 4.58, el cual ilustra la traducción durante el análisis sintáctico LR. La diferencia es que, en la regla para la producción 1, el programa de Yacc imprime el valor *E.val* como un efecto adicional, en vez de definir el atributo *L.val*.

A una definición dirigida por la sintaxis sin efectos adicionales se le llama algunas veces *gramática atribuida*. Las reglas en una gramática atribuida definen el valor de un atributo, sólo en términos de los valores de otros atributos y constantes.

5.1.2 Evaluación de una definición dirigida por la sintaxis en los nodos de un árbol de análisis sintáctico

Para visualizar la traducción especificada por una definición dirigida por la sintaxis, es útil trabajar con los árboles de análisis sintáctico, aun cuando un traductor en realidad no necesita construir un árbol de análisis sintáctico. Imagine, por lo tanto, que las reglas de una definición dirigida por la sintaxis se aplican construyendo primero un árbol de análisis sintáctico, y después usando las reglas para evaluar todos los atributos en cada uno de los nodos del árbol. A un árbol sintáctico, que muestra el (los) valor(es) de su(s) atributo(s) se le conoce como *árbol de análisis sintáctico anotado*.

¿Cómo construimos un árbol de análisis sintáctico anotado? ¿En qué orden evaluamos los atributos? Antes de poder evaluar un atributo en un nodo del árbol de análisis sintáctico, debemos evaluar todos los atributos de los cuales depende su valor. Por ejemplo, si todos los atributos sintetizados son sintetizados, como en el ejemplo 5.1, entonces debemos evaluar los atributos *val* en todos los hijos de un nodo, para poder evaluar el atributo *val* en el mismo nodo.

Con los atributos sintetizados, podemos evaluar los atributos en cualquier orden de abajo hacia arriba, como el de un recorrido postorden del árbol de análisis sintáctico; en la sección 5.2.3 hablaremos sobre la evaluación de las definiciones con atributos sintetizados.

Para las definiciones dirigidas por la sintaxis con atributos heredados y sintetizados, no hay garantía de que haya siquiera un orden en el que se puedan evaluar los atributos en los nodos. Por ejemplo, considere los no terminales A y B , con los atributos sintetizados y heredados $A.s$ y $B.i$, respectivamente, junto con la siguiente producción y las siguientes reglas:

PRODUCCIÓN	REGLAS SEMÁNTICAS
$A \rightarrow B$	$A.s = B.i;$
	$B.i = A.s + 1$

Estas reglas son circulares; es imposible evaluar $A.s$ en un nodo N o $B.i$ como el hijo de N sin primero evaluar el otro. La dependencia circular de $A.s$ y $B.i$ en cierto par de nodos en un árbol sintáctico se sugiere mediante la figura 5.2.

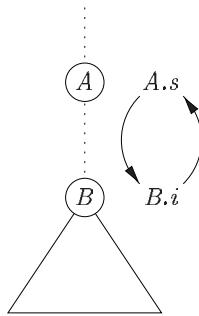


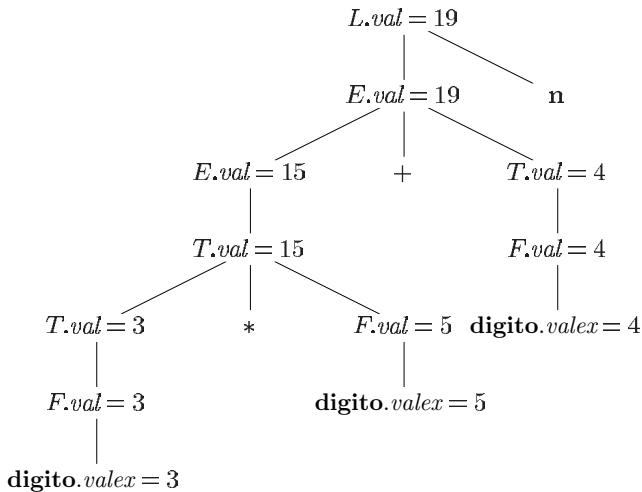
Figura 5.2: La dependencia circular de $A.s$ y $B.i$, uno del otro

En términos computacionales, es difícil determinar si existen o no circularidades en cualquiera de los árboles de análisis sintáctico que tenga que traducir una definición dirigida por la sintaxis dada.¹ Por fortuna, hay subclases útiles de definiciones dirigidas por la sintaxis que bastan para garantizar que exista un orden de evaluación, como veremos en la sección 5.2.

Ejemplo 5.2: La figura 5.3 muestra un árbol de análisis sintáctico anotado para la cadena de entrada $3 * 5 + 4 \mathbf{n}$, construida mediante el uso de la gramática y las reglas de la figura 5.1. Se supone que el analizador léxico proporciona los valores de *valex*. Cada uno de los nodos para los no terminales tiene el atributo *val* calculado en orden de abajo hacia arriba, y podemos ver los valores resultantes asociados con cada nodo. Por ejemplo, en el nodo con un hijo etiquetado como $*$, después de calcular $T.val = 3$ y $F.val = 5$ en sus hijos primero y tercero, aplicamos la regla que dice que $T.val$ es el producto de estos dos valores, o 15. \square

Los atributos heredados son útiles cuando la estructura de un árbol de análisis sintáctico no “coincide” con la sintaxis abstracta del código fuente. El siguiente ejemplo muestra cómo pueden usarse los atributos heredados para solucionar dicho conflicto, debido a una gramática diseñada para el análisis sintáctico, en vez de la traducción.

¹Sin entrar en detalles, aunque el problema puede decidirse, no puede resolverse mediante un algoritmo en tiempo polinomial, incluso si $\mathcal{P} = \mathcal{NP}$, ya que tiene una complejidad de tiempo exponencial.

Figura 5.3: Árbol de análisis sintáctico anotado para $3 * 5 + 4 \mathbf{n}$

Ejemplo 5.3: La definición dirigida por la sintaxis en la figura 5.4 calcula términos como $3 * 5$ y $3 * 5 * 7$. El análisis sintáctico descendente de la entrada $3 * 5$ empieza con la producción $T \rightarrow FT'$. Aquí, F genera el dígito 3, pero el operador $*$ se genera mediante T' . Por ende, el operando izquierdo 3 aparece en un subárbol distinto del árbol de análisis sintáctico de $*$. Por lo tanto, se utilizará un atributo heredado para pasar el operando al operador.

La gramática en este ejemplo es un extracto de una versión no recursiva por la izquierda de la conocida gramática de expresiones; utilizamos dicha gramática como un ejemplo para ilustrar el análisis sintáctico descendente en la sección 4.4.

PRODUCCIÓN	REGLAS SEMÁNTICAS
1) $T \rightarrow FT'$	$T'.her = F.val$ $T.val = T'.sin$
2) $T' \rightarrow *FT'_1$	$T'_1.her = T'.her \times F.val$ $T'.sin = T'_1.sin$
3) $T' \rightarrow \epsilon$	$T'.sin = T'.her$
4) $F \rightarrow \mathbf{digito}$	$F.val = \mathbf{digito}.valex$

Figura 5.4: Una definición dirigida por la sintaxis basada en una gramática adecuada para el análisis sintáctico de descendente

Cada uno de los no terminales T y F tiene un atributo sintetizado val ; el terminal **digito** tiene un atributo sintetizado $valex$. El no terminal T' tiene dos atributos: un atributo heredado her y un atributo sintetizado sin .

Las reglas semánticas se basan en la idea de que el operando izquierdo del operador $*$ es heredado. Dicho en forma más precisa, el encabezado T' de la producción $T' \rightarrow *FT'_1$ hereda el operando izquierdo de $*$ en el cuerpo de la producción. Dado un término $x * y * z$, la raíz del subárbol para $*y * z$ hereda a x . Entonces, la raíz del subárbol para $*z$ hereda el valor de $x * y$, y así en lo sucesivo, si hay más factores en el término. Una vez que se han acumulado todos los factores, el resultado se pasa de vuelta al árbol, mediante el uso de atributos sintetizados.

Para ver cómo se utilizan las reglas semánticas, considere el árbol de análisis sintáctico anotado para $3 * 5$ en la figura 5.5. La hoja de más a la izquierda en el árbol de análisis sintáctico, etiquetada como **digito**, tiene el valor de atributo $valex = 3$, en donde el 3 lo suministra el analizador léxico. Su padre es para la producción 4, $F \rightarrow \text{digito}$. La única regla semántica asociada con esta producción define a $F.val = \text{digito}.valex$, que es igual a 3.

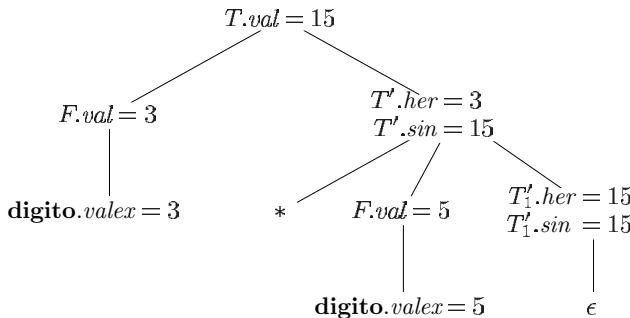


Figura 5.5: Árbol de análisis sintáctico anotado para $3 * 5$

En el segundo hijo de la raíz, el atributo heredado $T'.her$ se define mediante la regla semántica $T'.her = F.val$, asociada con la producción 1. Por ende, el operando izquierdo 3 para el operador $*$ se pasa de izquierda a derecha, a lo largo de los hijos de la raíz.

La producción en el nodo para T' es $T' \rightarrow *FT'_1$ (retenemos el subíndice 1 en el árbol de análisis sintáctico anotado para diferenciar entre los dos nodos para T'). El atributo heredado $T'_1.her$ se define mediante la regla semántica $T'_1.her = T'.her \times F.val$ asociada con la producción 2.

Con $T'.her = 3$ y $F.val = 5$, obtenemos $T'_1.her = 15$. En el nodo inferior para T'_1 , la producción es $T' \rightarrow \epsilon$. La regla semántica $T'.sin = T'.her$ define $T'_1.sin = 15$. Los atributos *sintetizados* en los nodos para T' pasan el valor 15 hacia arriba del árbol, hasta el nodo para T , en donde $T.val = 15$. \square

5.1.3 Ejercicios para la sección 5.1

Ejercicio 5.1.1: En la definición dirigida por la sintaxis de la figura 5.1, proporcione árboles de análisis sintáctico anotados para las siguientes expresiones:

- a) $(3 + 4) * (5 + 6)$ **n.**

- b) $1 * 2 * 3 * (4 + 5) \mathbf{n}$.
 c) $(9 + 8 * (7 + 6) + 5) * 4 \mathbf{n}$.

Ejercicio 5.1.2: Extienda la definición dirigida por la sintaxis de la figura 5.4 para manejar las expresiones como en la figura 5.1.

Ejercicio 5.1.3: Repita el ejercicio 5.1.1, usando su definición dirigida por la sintaxis del ejercicio 5.1.2.

5.2 Órdenes de evaluación para las definiciones dirigidas por la sintaxis

Los “grafos de dependencias” son una herramienta útil en la determinación de un orden de evaluación para las instancias de los atributos en un árbol de análisis sintáctico dado. Mientras que un árbol de análisis sintáctico anotado muestra los valores de los atributos, un grafo de dependencias nos ayuda a determinar cómo pueden calcularse esos valores.

En esta sección, además de los grafos de dependencias definiremos dos clases importantes de definiciones dirigidas por la sintaxis: la definición dirigida por la sintaxis “con atributos sintetizados” y la definición dirigida por la sintaxis más general “con atributos heredados por la izquierda”. Las traducciones especificadas por estas dos clases se adaptan bien con los métodos de análisis sintáctico que hemos estudiado, y la mayoría de las transiciones que se encuentran en la práctica pueden escribirse para conformarse a los requerimientos de por lo menos una de estas clases.

5.2.1 Gráficos de dependencias

Un *grafo de dependencias* describe el flujo de información entre las instancias de atributos en un árbol de análisis sintáctico específico; una flecha de una instancia de atributo a otra significa que el valor de la primera se necesita para calcular la segunda. Las flechas expresan las restricciones que imponen las reglas semánticas. Dicho en forma más detallada:

- Para cada nodo del árbol de análisis sintáctico, por decir un nodo etiquetado mediante el símbolo gramatical X , el grafo de dependencias tiene un nodo para cada atributo asociado con X .
- Suponga que una regla semántica asociada con una producción p define el valor del atributo sintetizado $A.b$ en términos del valor de $X.c$ (la regla puede definir a $A.b$ en términos de los demás atributos, aparte de $X.c$). Entonces, el grafo de dependencias tiene una flecha desde $X.c$ hasta $A.b$. Dicho en forma más precisa, en cada nodo N etiquetado como A en el que se aplica la producción p , se crea una flecha que va al atributo b en N , desde el atributo c en el hijo de N que corresponde a esta instancia del símbolo X en el cuerpo de la producción.²

²Como un nodo N puede tener varios hijos etiquetados como X , de nuevo suponemos que los subíndices diferencian los usos del mismo símbolo en distintos lugares en la producción.

- Suponga que una regla semántica asociada con una producción p define el valor del atributo heredado $B.c$ en términos del valor de $X.a$. Entonces, el grafo de dependencias tiene una flecha que va desde $X.a$ hasta $B.c$. Para cada nodo N etiquetado como B , que corresponda a una ocurrencia de esta B en el cuerpo de la producción p , se crea una flecha que va al atributo c en N , desde el atributo a en el nodo M que corresponde a esta ocurrencia de X . Observe que M podría ser el padre o un hermano de N .

Ejemplo 5.4: Considere la siguiente producción y regla:

PRODUCCIÓN	REGLA SEMÁNTICA
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

En cada nodo N etiquetado como E , con hijos que corresponden al cuerpo de esta producción, el atributo sintetizado val en N se calcula usando los valores de val en los dos hijos, etiquetados como E_1 y T . Por ende, una parte del grafo de dependencias para cada árbol de análisis sintáctico en el cual se utilice esta producción, se verá como el de la figura 5.6. Como convención, vamos a mostrar las flechas del árbol de análisis sintáctico como líneas punteadas, mientras que las flechas del grafo de dependencias son sólidas.

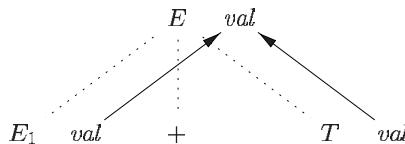


Figura 5.6: $E.val$ se sintetiza a partir de $E_1.val$ y $E_2.val$

Ejemplo 5.5: En la figura 5.7 aparece un ejemplo de un grafo de dependencias completo. Los nodos del grafo de dependencias, que se representan mediante los números del 1 al 9, corresponden a los atributos en el árbol de análisis sintáctico anotado en la figura 5.5.

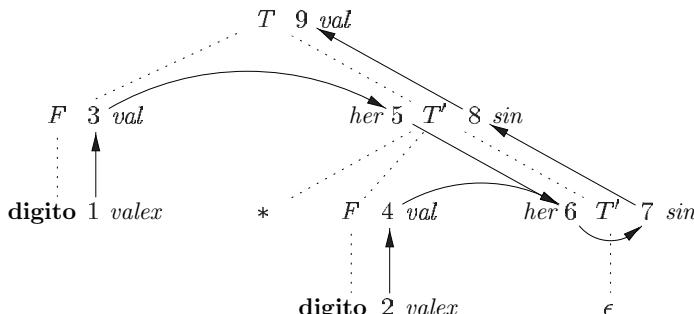


Figura 5.7: Grafo de dependencias para el árbol de análisis sintáctico anotado de la figura 5.5

Los nodos 1 y 2 representan el atributo $valex$ asociado con las dos hojas etiquetadas como **digito**. Los nodos 3 y 4 representan el atributo val asociado con los dos nodos etiquetados como F .

Las flechas que van al nodo 3 desde 1 y al nodo 4 desde 2 resultan de la regla semántica que define a $F.val$ en términos de **digito**. $valex$. De hecho, $F.val$ es igual a **digito**. $valex$, sólo que la flecha representa la dependencia, no la igualdad.

Los nodos 5 y 6 representan el atributo heredado $T'.her$ asociado con cada una de las ocurrencias del no terminal T' . La flecha de 5 a 3 se debe a la regla $T'.her = F.val$, la cual define a $T'.her$ en el hijo derecho de la raíz que parte de $F.val$ en el hijo izquierdo. Vemos flechas que van a 6 desde el nodo 5 para $T'.her$, y desde el nodo 4 para $F.val$, ya que estos valores se multiplican para evaluar el atributo her en el nodo 6.

Los nodos 7 y 8 representan el atributo sintetizado sin , asociado con las ocurrencias de T' . La flecha que va al nodo 7 desde 6 se debe a la regla semántica $T'.sin = T'.her$ asociada con la producción 3 en la figura 5.4. La flecha que va al nodo 8 desde 7 se debe a una regla semántica asociada con la producción 2.

Por último, el nodo 9 representa el atributo $T.val$. La flecha que va a 9 desde 8 se debe a la regla semántica $T.val = T'.sin$, asociada con la producción 1. \square

5.2.2 Orden de evaluación

El grafo de dependencias caracteriza los órdenes posibles en los cuales podemos evaluar los atributos en los diversos nodos de un árbol de análisis sintáctico. Si el grafo de dependencias tiene una flecha que va del nodo M al nodo N , entonces el atributo correspondiente a M debe evaluarse antes del atributo de N . Por ende, los únicos órdenes de evaluación permisibles son aquellas secuencias de nodos N_1, N_2, \dots, N_k , de tal forma que si hay una flecha del grafo de dependencias que va desde N_i hasta N_j , entonces $i < j$. Dicho ordenamiento incrusta un grafo dirigido en un orden lineal, a lo cual se le conoce como *orden topológico* del grafo.

Si hay un ciclo en el grafo, entonces no hay órdenes topológicos; es decir, no hay forma de evaluar la definición dirigida por la sintaxis en este árbol de análisis sintáctico. No obstante, si no hay ciclos, entonces hay por lo menos un orden topológico. Para ver por qué, debido a que no hay ciclos, de seguro podemos encontrar un nodo en el que no entren flechas. En caso de que no hubiera dicho nodo, podríamos proceder de predecesor en predecesor hasta regresar a algún nodo que ya hayamos visto, produciendo un ciclo. Debemos hacer que este nodo sea el primero en el orden topológico, eliminarlo del grafo de dependencias y repetir el proceso en los nodos restantes.

Ejemplo 5.6: El grafo de dependencias de la figura 5.7 no tiene ciclos. Un orden topológico es el orden en el que ya se han numerado los nodos: 1, 2, ..., 9. Observe que cada flecha del grafo pasa de un nodo a otro de mayor numeración, por lo que sin duda este orden es topológico. Hay otros órdenes topológicos también, como 1, 3, 5, 2, 4, 6, 7, 8, 9. \square

5.2.3 Definiciones con atributos sintetizados

Como dijimos antes, dada una definición dirigida por la sintaxis, es muy difícil saber si existen árboles de análisis sintáctico cuyos grafo de dependencias tengan ciclos. En la práctica, las traducciones pueden implementarse mediante el uso de clases de definiciones dirigidas por la sintaxis que garanticen un orden de evaluación, ya que no permiten grafos de dependencias con

ciclos. Además, las dos clases presentadas en esta sección pueden implementarse de manera eficiente en conexión con el análisis sintáctico descendente o ascendente.

La primera clase se define de la siguiente manera:

- Una definición dirigida por la sintaxis tiene *atributos sintetizados* si todos los atributos sintetizados son sintetizados.

Ejemplo 5.7: La definición dirigida por la sintaxis de la figura 5.1 es un ejemplo de una definición con atributos sintetizados. Cada atributo, $L.val$, $E.val$, $T.val$ y $F.val$ es sintetizado. \square

Cuando una definición dirigida por la sintaxis tiene atributos sintetizados, podemos evaluar sus atributos en cualquier orden de abajo hacia arriba de los nodos del árbol de análisis sintáctico. A menudo, es bastante sencillo evaluar los atributos mediante la realización de un recorrido postorden del árbol y evaluar los atributos en un nodo N cuando el recorrido sale de N por última vez. Es decir, aplicamos la función *postorden*, que se define a continuación, a la raíz del árbol de análisis sintáctico (vea también el recuadro “Recorridos preorden y postorden” en la sección 2.3.4):

```
postorden( $N$ ) {
    for ( cada hijo  $C$  de  $N$ , empezando desde la izquierda ) postorden( $C$ );
    evaluar los atributos asociados con el nodo  $N$ ;
}
```

Las definiciones de los atributos sintetizados pueden implementarse durante el análisis sintáctico de ascendente, ya que un análisis sintáctico de este tipo corresponde a un recorrido postorden. En forma específica, el postorden corresponde exactamente al orden en el que un analizador sintáctico LR reduce el cuerpo de una producción a su encabezado. Este hecho se utilizará en la sección 5.4.2 para evaluar los atributos sintetizados y almacenarlos en la pila durante el análisis sintáctico LR, sin crear los nodos del árbol en forma explícita.

5.2.4 Definiciones con atributos heredados

A la segunda clase de definiciones dirigidas por la sintaxis se le conoce como *definiciones con atributos heredados*. La idea de esta clase es que, entre los atributos asociados con el cuerpo de una producción, las flechas del grafo de dependencias pueden ir de izquierda a derecha, pero no al revés (de aquí que se les denomine “atributos heredados por la izquierda”). Dicho en forma más precisa, cada atributo debe ser:

1. Sintetizado.
2. Heredado, pero con las reglas limitadas de la siguiente manera. Suponga que hay una producción $A \rightarrow X_1X_2 \dots X_n$, y que hay un atributo heredado $X_i.a$, el cual se calcula mediante una regla asociada con esta producción. Entonces, la regla puede usar sólo:
 - (a) Los atributos heredados asociados con el encabezado A .
 - (b) Los atributos heredados o sintetizados asociados con las ocurrencias de los símbolos X_1, X_2, \dots, X_{i-1} ubicados a la izquierda de X_i .

- (c) Los atributos heredados o sintetizados asociados con esta misma ocurrencia de X_i , pero sólo en una forma en la que no haya ciclos en un grafo de dependencias formado por los atributos de esta X_i .

Ejemplo 5.8: La definición dirigida por la sintaxis en la figura 5.4 tiene atributos heredados por la izquierda. Para ver por qué, considere las reglas semánticas para los atributos heredados, que repetimos a continuación por conveniencia:

PRODUCCIÓN	REGLAS SEMÁNTICA
$T \rightarrow F T'$	$T'.her = F.val$
$T' \rightarrow * F T'_1$	$T'_1.her = T'.her \times F.val$

La primera de estas reglas define el atributo heredado $T'.her$, usando sólo $F.val$ y F aparece a la izquierda de T' en el cuerpo de la producción, según se requiera. La segunda regla define a $T'_1.her$ usando el atributo heredado $T'.her$, asociado con el encabezado, y $F.val$, en donde F aparece a la izquierda de T'_1 en el cuerpo de la producción.

En cada uno de estos casos, las reglas utilizan la información “de arriba o de la izquierda”, según lo requiera la clase. El resto de los atributos sintetizados son sintetizados. Por ende, la definición dirigida por la sintaxis tiene atributos heredados por la izquierda. \square

Ejemplo 5.9: Cualquier definición dirigida por la sintaxis que contenga la siguiente producción y reglas no puede tener atributos heredados por la izquierda:

PRODUCCIÓN	REGLAS SEMÁNTICAS
$A \rightarrow B C$	$A.s = B.b;$ $B.i = f(C.c, A.s)$

La primera regla, $A.s = B.b$, es una regla legítima en una definición dirigida por la sintaxis con atributos sintetizados o atributos heredados por la izquierda. Define a un atributo sintetizado $A.s$ en términos de un atributo en un hijo (es decir, un símbolo dentro del cuerpo de la producción).

La segunda regla define a un atributo heredado $B.i$, de forma que la definición dirigida por la sintaxis completa no puede tener atributos sintetizados. Además, aunque la regla es legal, la definición dirigida por la sintaxis no puede tener atributos heredados por la izquierda, debido a que el atributo $C.c$ se utiliza para ayudar a definir $B.i$, y C está a la derecha de B en el cuerpo de la producción. Aunque los atributos en los hermanos de un árbol de análisis sintáctico pueden usarse en las definiciones dirigidas por la sintaxis con atributos heredados por la izquierda, deben estar a la izquierda del símbolo cuyo atributo se está definiendo. \square

5.2.5 Reglas semánticas con efectos adicionales controlados

En la práctica, las traducciones involucran efectos adicionales: una calculadora de escritorio podría imprimir un resultado; un generador de código podría introducir el tipo de un identificador en una tabla de símbolos. Con las definiciones dirigidas por la sintaxis, encontramos un equilibrio entre las gramáticas de atributos y los esquemas de traducción. Las gramáticas atribuidas no tienen efectos adicionales y permiten cualquier orden de evaluación consistente con el grafo de dependencias. Los esquemas de traducción imponen la evaluación de izquierda

a derecha y permiten que las acciones semánticas contengan cualquier fragmento del programa; en la sección 5.4 hablaremos sobre los esquemas de traducción.

Controlaremos los efectos adicionales en las definiciones dirigidas por la sintaxis, en una de las dos formas siguientes:

- Permitir los efectos adicionales incidentales que no restrinjan la evaluación de los atributos. En otras palabras, hay que permitir los efectos adicionales cuando la evaluación de los atributos basada en cualquier orden topológico del grafo de dependencias produzca una traducción “correcta”, en donde “correcta” depende de la aplicación.
- Restringir los órdenes de evaluación permitidos, de manera que se produzca la misma traducción para cualquier orden permitido. Las restricciones pueden considerarse como flechas implícitas agregadas al grafo de dependencias.

Como ejemplo de un efecto adicional incidental, vamos a modificar la calculadora de escritorio del ejemplo 5.1 para imprimir un resultado. En vez de la regla $L.val = E.val$, que almacena el resultado en el atributo sintetizado $L.val$, considere lo siguiente:

PRODUCCIÓN	REGLA SEMÁNTICA
1) $L \rightarrow E \mathbf{n}$	$print(E.val)$

Las reglas semánticas que se ejecuten por sus efectos adicionales, como $print(E.val)$, se tratarán como las definiciones de los atributos falsos sintetizados, asociados con el encabezado de la producción. La definición dirigida por la sintaxis modificada produce la misma traducción bajo cualquier orden topológico, ya que la instrucción `print` se ejecuta al final, después de calcular el resultado y colocarlo en $E.val$.

Ejemplo 5.10: La definición dirigida por la sintaxis en la figura 5.8 recibe una declaración D simple, que consiste en un tipo T básico seguido de una lista L de identificadores. T puede ser **int** o **float**. Para cada identificador en la lista, se introduce el tipo en la entrada en la tabla de símbolos para ese identificador. Asumimos que al introducir el tipo para un identificador, no se ve afectada la entrada en la tabla de símbolos de ningún otro identificador. Por ende, las entradas pueden actualizarse en cualquier orden. Esta definición dirigida por la sintaxis no verifica si un identificador se declara más de una vez; puede modificarse para ello.

PRODUCCIÓN	REGLAS SEMÁNTICAS
1) $D \rightarrow T \ L$	$L.her = T.tipo$
2) $T \rightarrow \mathbf{int}$	$T.tipo = \text{integer}$
3) $T \rightarrow \mathbf{float}$	$T.tipo = \text{float}$
4) $L \rightarrow L_1, \mathbf{id}$	$L_1.her = L.her$ $\text{agregarTipo}(\mathbf{id.entrada}, L.her)$
5) $L \rightarrow \mathbf{id}$	$\text{agregarTipo}(\mathbf{id.entrada}, L.her)$

Figura 5.8: Definición orientada por la sintaxis para las declaraciones de tipos simples

La no terminal D representa una declaración que, a partir de la producción 1, consiste en un tipo T seguido de una lista L de identificadores. T tiene un atributo, $T.tipo$, el cual es el tipo en la declaración D . La no terminal L tiene también un atributo, al cual llamaremos her para enfatizar que es un atributo heredado. El propósito de $L.her$ es pasar el tipo declarado hacia abajo en la lista de identificadores, de manera que pueda agregarse a las entradas apropiadas en la tabla de símbolos.

Cada una de las producciones 2 y 3 evalúan el atributo sintetizado $T.tipo$, proporcionándole el valor apropiado, integer o float. Este tipo se pasa al atributo $L.her$ en la regla para la producción 1. La producción 4 pasa $L.her$ hacia abajo en el árbol de análisis sintáctico. Es decir, el valor $L_1.her$ se calcula en un nodo del árbol de análisis sintáctico mediante la copia del valor de $L.her$ del padre de ese nodo; el padre corresponde al encabezado de la producción.

Las producciones 4 y 5 también tienen una regla en la que se hace una llamada a la función *agregarTipo* con dos argumentos:

1. **id.entrada**, un valor léxico que apunta a un objeto en la tabla de símbolos.
 2. **L.her**, el tipo que se va a asignar a todos los identificadores en la lista.

Suponemos que la función *agregarTipo* instala en forma apropiada el tipo *L.her* como el tipo del identificador representado.

En la figura 5.9 aparece un grafo de dependencias para la cadena de entrada **float id₁, id₂, id₃**. Los números del 1 al 10 representan los nodos del grafo de dependencias. Los nodos 1, 2 y 3 representan el atributo *entrada* asociado con cada una de las hojas etiquetadas como **id**. Los nodos 6, 8 y 10 son los atributos falsos que representan la aplicación de la función *agregarTipo* a un tipo y uno de estos valores *entrada*.

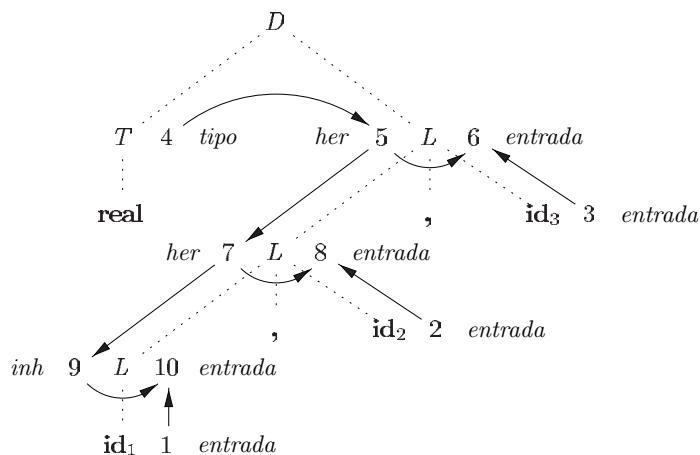


Figura 5.9: Grafo de dependencias para una declaración **float id₁, id₂, id₃**

El nodo 4 representa el atributo $T.tipo$, y en realidad es en donde empieza la evaluación de atributos. Después, este tipo se pasa a los nodos 5, 7 y 9 que representan a $L.her$ asociado con cada una de las ocurrencias del no terminal L . \square

5.2.6 Ejercicios para la sección 5.2

Ejercicio 5.2.1: ¿Cuáles son todos los órdenes topológicos para el grafo de dependencias de la figura 5.7?

Ejercicio 5.2.2: Para la definición dirigida por la sintaxis de la figura 5.8, proporcione los árboles de análisis sintáctico anotados para las siguientes expresiones:

- a) `int a, b, c.`
- b) `float w, x, y, z.`

Ejercicio 5.2.3: Suponga que tenemos una producción $A \rightarrow BCD$. Cada una de los cuatro no terminales A , B , C y D tienen dos atributos: s es un atributo sintetizado, y i es un atributo heredado. Para cada uno de los siguientes conjuntos de reglas, indique si (i) las reglas son consistentes con una definición con atributos sintetizados, (ii) las reglas son consistentes con una definición con atributos heredados por la izquierda, y (iii) si las reglas son consistentes con algún orden de evaluación.

- a) $A.s = B.i + C.s.$
- b) $A.s = B.i + C.s$ y $D.i = A.i + B.s.$
- c) $A.s = B.s + D.s.$
- d) $A.s = D.i$, $B.i = A.s + C.s$, $C.i = B.s$ y $D.i = B.i + C.i.$

! **Ejercicio 5.2.4:** Esta gramática genera números binarios con un punto “decimal”:

$$\begin{aligned} S &\rightarrow L \cdot L \mid L \\ L &\rightarrow L B \mid B \\ B &\rightarrow 0 \mid 1 \end{aligned}$$

Diseñe una definición dirigida por la sintaxis con atributos heredados por la izquierda para calcular $S.val$, el valor de número decimal de una cadena de entrada. Por ejemplo, la traducción de la cadena `101.101` debe ser el número decimal `5.625`. *Sugerencia:* use un atributo heredado $L.lado$ que nos indique en qué lado del punto decimal se encuentra un bit.

!! **Ejercicio 5.2.5:** Diseñe una definición dirigida por la sintaxis con atributos sintetizados para la gramática y la traducción descritas en el ejercicio 5.2.4.

!! **Ejercicio 5.2.6:** Implemente el Algoritmo 3.23, que convierte una expresión regular en un autómata finito no determinista, mediante una definición dirigida por la sintaxis con atributos heredados por la izquierda en una gramática analizable mediante el análisis sintáctico descendente. Suponga que hay un token `car` que representa a cualquier carácter, y que `car.valex` es el carácter que representa. También puede suponer la existencia de una función `nuevo()` que devuelva un nuevo estado, es decir, un estado que esta función nunca haya devuelto. Use cualquier notación conveniente para especificar las transiciones del AFN.

5.3 Aplicaciones de la traducción orientada por la sintaxis

Las técnicas de traducción orientada por la sintaxis de este capítulo se aplicarán en el capítulo 6 a la comprobación de tipos y la generación de código intermedio. Aquí consideraremos ejemplos seleccionados para ilustrar algunas definiciones dirigidas por la sintaxis representativas.

La principal aplicación en esta sección es la construcción de árboles de análisis sintáctico. Como algunos compiladores utilizan árboles sintácticos como una representación intermedia, una forma común de definición dirigida por la sintaxis convierte su cadena de entrada en un árbol. Para completar la traducción en código intermedio, el compilador puede recorrer el árbol sintáctico, usando otro conjunto de reglas que son en realidad una definición dirigida por la sintaxis en el árbol sintáctico, en vez del árbol de análisis sintáctico. El capítulo 6 también habla sobre los métodos para la generación de código intermedio que aplican una definición dirigida por la sintaxis sin siquiera construir un árbol en forma explícita.

Vamos a considerar dos definiciones dirigidas por la sintaxis en la construcción de árboles sintácticos para las expresiones. La primera, una definición con atributos sintetizados, es adecuada para usarse durante el análisis sintáctico ascendente. La segunda, con atributos heredados por la izquierda, es adecuada para usarse durante el análisis sintáctico descendente.

El ejemplo final de esta sección es una definición con atributos heredados a la izquierda que maneja los tipos básicos y de arreglos.

5.3.1 Construcción de árboles de análisis sintáctico

Como vimos en la sección 2.8.2, cada nodo en un árbol sintáctico representa a una construcción; los hijos del nodo representan los componentes significativos de la construcción. Un nodo de árbol sintáctico que representa a una expresión $E_1 + E_2$ tiene la etiqueta `+` y dos hijos que representan las subexpresiones E_1 y E_2 .

Vamos a implementar los nodos de un árbol sintáctico mediante objetos con un número adecuado de campos. Cada objeto tendrá un campo `op` que es la etiqueta del nodo. Los objetos tendrán los siguientes campos adicionales:

- Si el nodo es una hoja, un campo adicional contiene el valor léxico para esa hoja. Un constructor `Hoja(op, val)` crea un objeto hoja. De manera alternativa, si los nodos se ven como registros, entonces `Hoja` devuelve un apuntador a un nuevo registro para una hoja.
- Si el nodo es interior, hay tantos campos adicionales como hijos que tiene el nodo en el árbol sintáctico. Un constructor `Nodo` recibe dos o más argumentos: `Nodo(op, c1, c2, ..., ck)` crea un objeto con el primer campo `op` y k campos adicionales para los k hijos c_1, \dots, c_k .

Ejemplo 5.11: La definición con atributos sintetizados en la figura 5.10 construye árboles sintácticos para una gramática de expresiones simple, que involucra sólo a los operadores binarios `+` y `-`. Como siempre, estos operadores tienen el mismo nivel de precedencia y ambos son asociativos por la izquierda. Todos los no terminales tienen un atributo sintetizado llamado `nodo`, el cual representa a un nodo del árbol sintáctico.

Cada vez que se utiliza la primera producción $E \rightarrow E_1 + T$, su regla crea un nodo con `'+'` para `op` y dos hijos, `E1.nodo` y `T.nodo`, para las subexpresiones. La segunda producción tiene una regla similar.

PRODUCCIÓN	REGLAS SEMÁNTICAS
1) $E \rightarrow E_1 + T$	$E.\text{nodo} = \text{new } \text{Nodo}('+', E_1.\text{nodo}, T.\text{nodo})$
2) $E \rightarrow E_1 - T$	$E.\text{nodo} = \text{new } \text{Nodo}(' - ', E_1.\text{nodo}, T.\text{nodo})$
3) $E \rightarrow T$	$E.\text{nodo} = T.\text{nodo}$
4) $T \rightarrow (E)$	$T.\text{nodo} = E.\text{nodo}$
5) $T \rightarrow \text{id}$	$T.\text{nodo} = \text{new } \text{Hoja}(\text{id}, \text{id}.\text{entrada})$
6) $T \rightarrow \text{num}$	$T.\text{nodo} = \text{new } \text{Hoja}(\text{num}, \text{num}.\text{val})$

Figura 5.10: Construcción de árboles sintácticos para expresiones simples

Para la producción 3, $E \rightarrow T$, no se crea ningún nodo, ya que $E.\text{nodo}$ es igual que $T.\text{nodo}$. De manera similar, no se crea ningún nodo para la producción 4, $T \rightarrow (E)$. El valor de $T.\text{nodo}$ es igual que $E.\text{nodo}$, ya que los paréntesis sólo se utilizan para agrupar; tienen influencia sobre la estructura del árbol de análisis sintáctico y del árbol en sí, pero una vez que termina su trabajo, no hay necesidad de retenerlos en el árbol sintáctico.

Las últimas dos producciones T tienen un solo terminal a la derecha. Utilizamos el constructor *Hoja* para crear un nodo adecuado, el cual se convierte en el valor de $T.\text{nodo}$.

La figura 5.11 muestra la construcción de un árbol sintáctico para la entrada $a - 4 + c$. Los nodos del árbol sintáctico se muestran como registros, con el campo *op* primero. Las flechas del árbol sintáctico ahora se muestran como líneas sólidas. El árbol de análisis sintáctico subyacente, que en realidad no necesita construirse, se muestra con flechas punteadas. El tercer tipo de línea, que se muestra discontinua, representa los valores de $E.\text{nodo}$ y $T.\text{nodo}$; cada línea apunta al nodo apropiado del árbol sintáctico.

En la parte inferior podemos ver hojas para a , 4 y c , construidas por *Hoja*. Suponemos que el valor léxico **id.entrada** apunta a la tabla de símbolos, y que el valor léxico **num.val** es el valor numérico de una constante. Estas hojas, o los apuntadores a ellas, se convierten en el valor de $T.\text{nodo}$ en los tres nodos del árbol sintáctico etiquetados como T , de acuerdo con las reglas 5 y 6. Observe que en base a la regla 3, el apuntador a la hoja para a es también el valor de $E.\text{nodo}$ para la E de más a la izquierda en el árbol de análisis sintáctico.

La regla 2 ocasiona la creación de un nodo con *op* igual al signo negativo y los apuntadores a las primeras dos hojas. Después, la regla 1 produce el nodo raíz del árbol sintáctico, mediante la combinación del nodo para $-$ con la tercera hoja.

Si las reglas se evalúan durante un recorrido postorden del árbol de análisis sintáctico, o con reducciones durante un análisis sintáctico ascendente, entonces la secuencia de pasos que se muestra en la figura 5.12 termina con p_5 apuntando a la raíz del árbol sintáctico construido. \square

Con una gramática diseñada para el análisis sintáctico descendente se construyen los mismos árboles sintácticos, mediante la misma secuencia de pasos, aun cuando la estructura de los árboles de análisis sintáctico difiere de manera considerable de la de los árboles sintácticos.

Ejemplo 5.12: La definición con atributos heredados por la izquierda en la figura 5.13 realiza la misma traducción que la definición con atributos sintetizados en la figura 5.10. Los atributos para los símbolos gramaticales E , T , **id** y **num** son como vimos en el ejemplo 5.11.

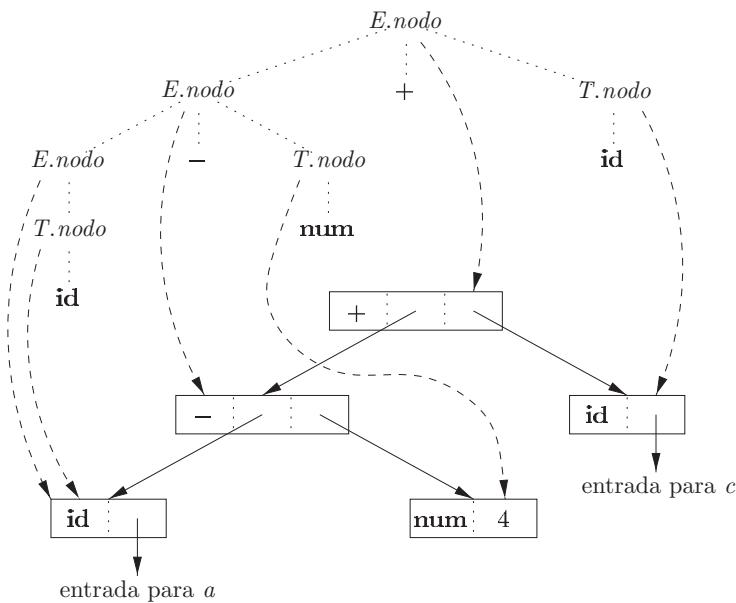


Figura 5.11: Árbol sintáctico para $a - 4 + c$

- 1) $p_1 = \text{new Hoja}(\text{id}, \text{entrada-}a);$
 - 2) $p_2 = \text{new Hoja}(\text{num}, 4);$
 - 3) $p_3 = \text{new Nodo}('-, p_1, p_2);$
 - 4) $p_4 = \text{new Hoja}(\text{id}, \text{entrada-}c);$
 - 5) $p_5 = \text{new Nodo}('+, p_3, p_4);$

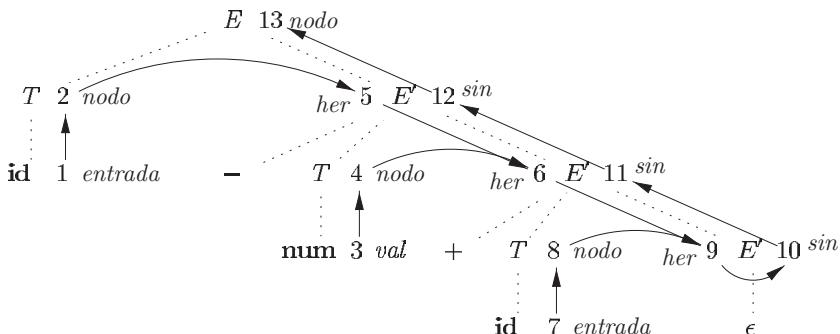
Figura 5.12: Pasos en la construcción del árbol sintáctico para $a - 4 + c$

Las reglas para construir árboles sintácticos en este ejemplo son similares a las reglas para la calculadora de escritorio en el ejemplo 5.3. En este ejemplo se evalúo un término $x * y$, pasando x como un atributo heredado, ya que x y $* y$ aparecían en distintas porciones del árbol de análisis sintáctico. Aquí, la idea es construir un árbol sintáctico para $x + y$, pasando x como un atributo heredado, ya que x y $+ y$ aparecen en distintos subárboles. El no terminal E' es la contraparte del no terminal T' en el ejemplo 5.3. Compare el grafo de dependencias para $a - 4 + c$ en la figura 5.14, con el grafo para $3 * 5$ en la figura 5.7.

El no terminal E' tiene un atributo heredado *her* y un atributo sintetizado *sin*. El atributo $E'.her$ representa el árbol sintáctico parcial que se ha construido hasta ahora. En forma específica, representa la raíz del árbol para el prefijo de la cadena de entrada que se encuentra a la izquierda del subárbol para E' . En el nodo 5 en el grafo de dependencias de la figura 5.14, $E'.her$ denota la raíz del árbol sintáctico parcial para el identificador a ; es decir, la hoja para a . En el nodo 6, $E'.her$ denota la raíz para el árbol sintáctico parcial para la entrada $a - 4$. En el nodo 9, $E'.her$ denota el árbol sintáctico para $a - 4 + c$.

PRODUCCIÓN	REGLAS SEMÁNTICAS
1) $E \rightarrow TE'$	$E.\text{nodo} = E'.\text{sin}$ $E'.\text{her} = T.\text{nodo}$
2) $E' \rightarrow + TE'_1$	$E'_1.\text{her} = \text{new } \text{Nodo}(' + ', E'.\text{her}, T.\text{nodo})$ $E'.\text{sin} = E'_1.\text{sin}$
3) $E' \rightarrow - TE'_1$	$E'_1.\text{her} = \text{new } \text{nodo}(' - ', E'.\text{her}, T.\text{nodo})$ $E'.\text{sin} = E'_1.\text{sin}$
4) $E' \rightarrow \epsilon$	$E'.\text{sin} = E'.\text{her}$
5) $T \rightarrow (E)$	$T.\text{nodo} = E.\text{nodo}$
6) $T \rightarrow \text{id}$	$T.\text{nodo} = \text{new } \text{Hoja}(\text{id}, \text{id}.\text{entrada})$
7) $T \rightarrow \text{num}$	$T.\text{nodo} = \text{new } \text{Hoja}(\text{num}, \text{num}.\text{entrada})$

Figura 5.13: Construcción de árboles sintácticos durante el análisis sintáctico descendente

Figura 5.14: Gráfico de dependencias para $a - 4 + c$, con la definición dirigida por la sintaxis de la figura 5.13

Como no hay más entrada, en el nodo 9, $E'.\text{her}$ apunta a la raíz de todo el árbol sintáctico. Los atributos sintetizados pasan este valor de vuelta hacia arriba del árbol de análisis sintáctico, hasta que se convierte en el valor de $E.\text{nodo}$. De manera específica, el valor del atributo en el nodo 10 se define mediante la regla $E'.\text{sin} = E'.\text{her}$ asociada con la producción $E' \rightarrow \epsilon$. El valor del atributo en el nodo 11 se define mediante la regla $E'.\text{sin} = E'_1.\text{sin}$ asociada con la producción 2 en la figura 5.13. Hay reglas similares que definen los valores de los atributos en los nodos 12 y 13. \square

5.3.2 La estructura de tipos

Los atributos heredados son útiles cuando la estructura del árbol de análisis sintáctico difiere de la sintaxis abstracta de la entrada; entonces, los atributos pueden usarse para llevar información de una parte del árbol de análisis sintáctico a otra. El siguiente ejemplo muestra cómo

un conflicto en la estructura puede deberse al diseño del lenguaje, y no a las restricciones que impone el método de análisis sintáctico.

Ejemplo 5.13: En C, el tipo **int** [2][3] puede leerse como “arreglo de 2 arreglos de 3 enteros”. La expresión del tipo correspondiente $\text{arreglo}(2, \text{arreglo}(3, \text{integer}))$ se representa mediante el árbol en la figura 5.15. El operador *arreglo* recibe dos parámetros, un número y un tipo. Si los tipos se representan mediante árboles, entonces este operador devuelve un nodo de árbol etiquetado como *arreglo*, con dos hijos para el número y el tipo.

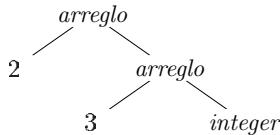


Figura 5.15: Expresión de los tipos para **int**[2][3]

Con la definición dirigida por la sintaxis en la figura 5.16, el no terminal *T* genera un tipo básico o un tipo arreglo. El no terminal *B* genera uno de los tipos básicos **int** y **float**. *T* genera un tipo básico cuando *T* deriva a *B C* y *C* deriva a ϵ . En cualquier otro caso, *C* genera componentes de un arreglo que consisten en una secuencia de enteros, en donde cada entero está rodeado por llaves.

PRODUCCIÓN	REGLAS SEMÁNTICAS
$T \rightarrow B C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow \text{int}$	$B.t = \text{integer}$
$B \rightarrow \text{float}$	$B.t = \text{float}$
$C \rightarrow [\text{num}] C_1$	$C.t = \text{arreglo}(\text{num}.val, C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

Figura 5.16: *T* genera un tipo básico o un tipo de arreglo

Los no terminales *B* y *T* tienen un atributo sintetizado *t* que representa un tipo. El no terminal *C* tiene dos atributos: un atributo heredado *b* y un atributo sintetizado *t*. Los atributos *b* heredados pasan un tipo básico hacia abajo del árbol, y los atributos *t* sintetizados acumulan el resultado.

En la figura 5.17 se muestra un árbol de análisis sintáctico anotado para la cadena de entrada **int**[2][3]. La expresión de tipo correspondiente en la figura 5.15 se construye pasando el tipo *integer* desde *B*, hacia abajo por la cadena de *Cs* y a través de los atributos heredados *b*. El tipo *arreglo* se sintetiza hacia arriba por la cadena de *Cs*, a través de los atributos *t*.

Con más detalle, en la raíz para $T \rightarrow B C$, el no terminal *C* hereda el tipo de *B*, usando el atributo heredado *C.b*. En el nodo de más a la derecha para *C*, la producción es $C \rightarrow \epsilon$, por lo que *C.t* es igual a *C.b*. Las reglas semánticas para la producción $C \rightarrow [\text{num}] C_1$ forman *C.t* mediante la aplicación del operador *arreglo* a los operandos *num.val* y *C_1.t*. \square

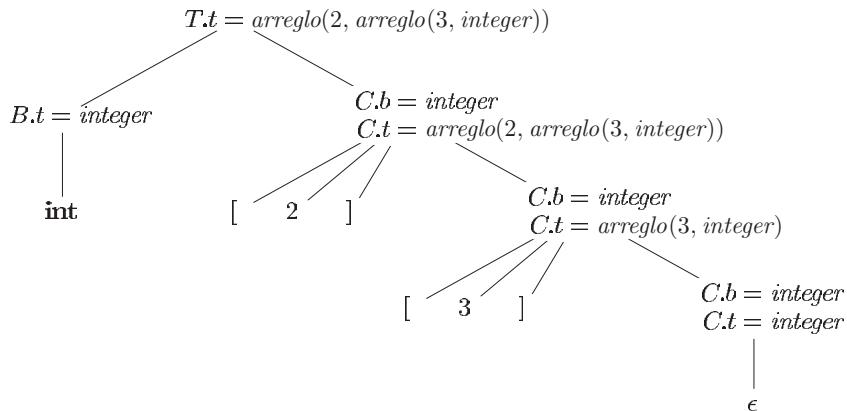


Figura 5.17: Traducción orientada por la sintaxis de los tipos de arreglos

5.3.3 Ejercicios para la sección 5.3

Ejercicio 5.3.1: A continuación se muestra una gramática para expresiones, en la que se invoca el operador $+$ y los operandos de entero o punto flotante. Los números de punto flotante se diferencian debido a que tienen un punto decimal.

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow \mathbf{num} \cdot \mathbf{num} \mid \mathbf{num} \end{aligned}$$

- Proporcione una definición dirigida por la sintaxis para determinar el tipo de cada término T y cada expresión E .
- Extienda su definición dirigida por la sintaxis de (a) para traducir expresiones a la notación postfijo. Use el operador unario **intToFloat** para convertir un entero en un número equivalente de punto flotante.

! Ejercicio 5.3.2: Proporcione una definición dirigida por la sintaxis para traducir las expresiones infijo con $+$ y $*$ en expresiones equivalentes sin paréntesis redundantes. Por ejemplo, como ambos operadores asocian por la izquierda, y $*$ tiene precedencia sobre $+$, $((a*(b+c))*(d))$ se traduce en $a * (b + c) * d$.

! Ejercicio 5.3.3: Proporcione una definición dirigida por la sintaxis para diferenciar expresiones como $x * (3 * x + x * x)$, en las que se involucren los operadores $+$ y $*$, la variable x y constantes. Suponga que no ocurre ninguna simplificación, de forma que, por ejemplo, $3 * x$ se traducirá en $3 * 1 + 0 * x$.

5.4 Esquemas de traducción orientados por la sintaxis

Los esquemas de traducción orientados por la sintaxis son una notación complementaria para las definiciones dirigidas a la sintaxis. Todas las aplicaciones de las definiciones dirigidas por la sintaxis en la sección 5.3 pueden implementarse mediante el uso de esquemas de traducción orientados por la sintaxis.

En la sección 2.3.5 vimos que un *esquema de traducción orientado por la sintaxis* (esquema de traducción orientado a la sintaxis) es una gramática libre de contexto, con fragmentos de programa incrustados dentro de los cuerpos de las producciones. A estos fragmentos se les conoce como *acciones semánticas* y pueden aparecer en cualquier posición dentro del cuerpo de una producción. Por convención, colocamos llaves alrededor de las acciones; si las llaves se necesitan como símbolos gramaticales, entonces las encerramos entre comillas.

Podemos implementar cualquier esquema de traducción orientado a la sintaxis, construyendo primero un árbol de análisis sintáctico y después realizando las acciones en un orden de izquierda a derecha, con búsqueda en profundidad; es decir, durante un recorrido preorden. En la sección 5.4.3 aparece un ejemplo.

Por lo general, los esquemas de traducción orientados a la sintaxis se implementan durante el análisis sintáctico, sin construir un árbol de análisis sintáctico. En esta sección nos enfocaremos en el uso de esquemas de traducción orientados a la sintaxis para implementar dos clases importantes de definiciones dirigidas por la sintaxis:

1. Puede utilizarse el análisis sintáctico LR en la gramática subyacente, y la definición dirigida por la sintaxis tiene atributos sintetizados.
2. Puede utilizarse el análisis sintáctico LL en la gramática subyacente, y la definición dirigida por la sintaxis tiene atributos heredados por la izquierda.

Más adelante veremos cómo en ambos casos, las reglas semánticas en una definición dirigida por la sintaxis pueden convertirse en un esquema de traducción orientado a la sintaxis con acciones que se ejecuten en el momento adecuado. Durante el análisis sintáctico, una acción en el cuerpo de una producción se ejecuta tan pronto como los símbolos gramaticales a la izquierda de la acción tengan coincidencias.

Los esquemas de traducción orientados a la sintaxis que pueden implementarse durante el análisis sintáctico se caracterizan mediante la introducción de *no terminales marcadores* en lugar de cada acción incrustada; cada marcador M sólo tiene una producción, $M \rightarrow \epsilon$. Si la gramática con no terminales marcadores puede analizarse mediante un método dado, entonces el esquema de traducción orientado a la sintaxis puede implementarse durante el análisis sintáctico.

5.4.1 Esquemas de traducción postfijos

Hasta ahora, la implementación de definiciones dirigidas por la sintaxis más simple ocurre cuando podemos analizar sintácticamente la gramática de abajo hacia arriba, y la definición dirigida por la sintaxis tiene atributos sintetizados. En ese caso, podemos construir un esquema de traducción orientado a la sintaxis en el cual cada acción se coloque al final de la producción y se ejecute junto con la reducción del cuerpo para el encabezado de esa producción. Los esquemas de traducción orientados a la sintaxis con todas las acciones en los extremos derechos de los cuerpos de las producciones se llaman *esquemas de traducción orientados a la sintaxis postfijo*.

Ejemplo 5.14: El esquema de traducción orientado a la sintaxis postfijo en la figura 5.18 implementa la definición dirigida por la sintaxis de la calculadora de escritorio de la figura 5.1, con un cambio: la acción para la primera producción imprime un valor. El resto de las acciones son contrapartes exactas de las reglas semánticas. Como la gramática subyacente es LR, y la definición dirigida por la sintaxis tiene atributos sintetizados, estas acciones pueden realizarse en forma correcta, junto con los pasos de reducción del analizador sintáctico. \square

$L \rightarrow E \mathbf{n}$	{ imprimir($E.val$); }
$E \rightarrow E_1 + T$	{ $E.val = E_1.val + T.val$; }
$E \rightarrow T$	{ $E.val = T.val$; }
$T \rightarrow T_1 * F$	{ $T.val = T_1.val \times F.val$; }
$T \rightarrow F$	{ $T.val = F.val$; }
$F \rightarrow (E)$	{ $F.val = E.val$; }
$F \rightarrow \mathbf{digito}$	{ $F.val = \mathbf{digito}.valex$; }

Figura 5.18: Implementación de un esquema de traducción orientado a la sintaxis postfijo de la calculadora de escritorio

5.4.2 Implementación de esquemas de traducción orientados a la sintaxis postfijo con la pila del analizador sintáctico

Los esquemas de traducción orientados a la sintaxis postfijo pueden implementarse durante el análisis sintáctico LR mediante la ejecución de las acciones cuando ocurren las reducciones. El (los) atributo(s) de cada símbolo gramatical puede(n) colocarse en la pila, en un lugar en el que puedan encontrarse durante la reducción. El mejor plan es colocar los atributos junto con los símbolos gramaticales (o con los estados LR que representan estos símbolos) en registros en la misma pila.

En la figura 5.19, la pila del analizador sintáctico contiene registros con un campo para un símbolo gramatical (o estado del analizador sintáctico) y, debajo de él, un campo para un atributo. Los tres símbolos gramaticales $X \ Y \ Z$ están en la parte superior de la pila; tal vez estén a punto de reducirse en base a una producción como $A \rightarrow X \ Y \ Z$. Aquí, mostramos $X.x$ como el único atributo de X , y así en lo sucesivo. En general, podemos permitir más atributos, ya sea haciendo los registros lo bastante extensos o colocando apuntadores a registros en la pila. Con atributos pequeños, puede ser más simple hacer los registros lo bastante grandes, aun cuando algunos campos no se utilizan la mayor parte del tiempo. No obstante, si uno o más atributos sintetizados son de un tamaño sin límite (por decir, si son cadenas de caracteres), entonces sería mejor colocar un apuntador al valor del atributo en el registro de la pila y almacenar el valor real en alguna área de almacenamiento compartido más extensa, que no forme parte de la pila.

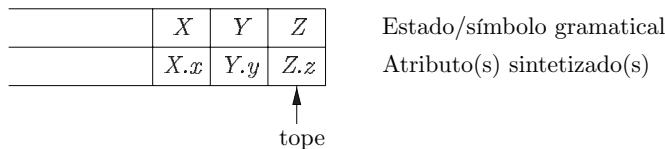


Figura 5.19: La pila del analizador sintáctico, con un campo para los atributos sintetizados

Si los atributos sintetizados son todos sintetizados, y las acciones ocurren en los extremos de las producciones, entonces podemos calcular los atributos para el encabezado cuando reduzcamos el cuerpo al encabezado. Si reducimos mediante una producción como $A \rightarrow X \ Y \ Z$, entonces tenemos todos los atributos de X , Y y Z disponibles, en posiciones conocidas de la pila, como en la figura 5.19. Después de la acción, A y sus atributos sintetizados se encuentran en la parte superior de la pila, en la posición del registro para X .

Ejemplo 5.15: Vamos a rescribir las acciones del esquema de traducción orientado a la sintaxis de la calculadora de escritorio del ejemplo 5.14, de manera que manipulen la pila del analizador sintáctico en forma explícita. Por lo general, el analizador sintáctico es el que realiza en forma automática la manipulación de la pila.

PRODUCCIÓN	ACCIONES
$L \rightarrow E \mathbf{n}$	{ imprimir($pila[tope - 1].val$); $tope = tope - 1;$ }
$E \rightarrow E_1 + T$	{ $pila[tope - 2].val = pila[tope - 2].val + pila[tope].val$; $tope = tope - 2;$ }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ $pila[tope - 2].val = pila[tope - 2].val \times pila[tope].val$; $tope = tope - 2;$ }
$T \rightarrow F$	
$F \rightarrow (E)$	{ $pila[tope - 2].val = pila[tope - 1].val$; $tope = tope - 2;$ }
$F \rightarrow \mathbf{digito}$	

Figura 5.20: Implementación de la calculadora de escritorio en una pila de análisis sintáctico ascendente

Suponga que la pila se mantiene en un arreglo de registros llamado *pila*, en donde *tope* es un cursor para la parte superior de la pila. Por ende, *pila[tope]* se refiere al registro superior en la pila, *pila[superior - 1]* al registro debajo de éste, y así en lo sucesivo. Además, suponemos que cada registro tiene un campo llamado *val*, el cual contiene el atributo de cualquier símbolo gramatical que se represente en ese registro. Por ende, podemos referirnos al atributo *E.val* que aparece en la tercera posición en la pila como *pila[tope - 2].val*. El esquema de traducción orientado por la sintaxis completo se muestra en la figura 5.20.

Por ejemplo, en la segunda producción, $E \rightarrow E_1 + T$, recorremos dos posiciones debajo de la parte superior para obtener el valor de E_1 , y encontramos el valor de T en superiores tope. La suma resultante se coloca en donde debe aparecer el encabezado *E* después de la reducción; es decir, dos posiciones debajo del tope actual. La razón es que, después de la reducción, los tres símbolos de la parte del tope de la pila se sustituyen por uno solo. Después de calcular *E.val*, sacamos dos símbolos del tope de la pila, por lo que el registro en donde colocamos *E.val* ahora estará en el tope de la pila.

En la tercera producción, $E \rightarrow T$, no es necesaria ninguna acción ya que la longitud de la pila no cambia, y el valor de *T.val* en el tope de la pila simplemente se convertirá en el valor de *E.val*. La misma observación se aplica a las producciones $T \rightarrow F$ y $F \rightarrow \mathbf{digito}$. La producción $F \rightarrow (E)$ es un poco distinta. Aunque el valor no cambia, dos posiciones se eliminan de la pila durante la reducción, por lo que el valor tiene que avanzar a la posición que está después de la reducción.

Observe que hemos omitido los pasos que manipulan el primer campo de los registros de la pila; el campo que proporciona el estado LR o, en cualquier otro caso, que representa el símbolo gramatical. Si vamos a realizar un análisis sintáctico LR, la tabla de análisis sintáctico nos in-

dica cuál es el nuevo estado cada vez que realizamos una reducción; vea el algoritmo 4.44. Por ende, simplemente podemos colocar el estado en el registro para el nuevo tope de la pila. \square

5.4.3 Esquemas de traducción orientados a la sintaxis con acciones dentro de las producciones

Puede colocarse una acción en cualquier posición dentro del cuerpo de una producción. Ésta se ejecuta de inmediato, después de procesar todos los símbolos a su izquierda. Por ende, si tenemos una producción $B \rightarrow X \{ a \} Y$, la acción a se realiza después de que reconozcamos a X (si X es un terminal) o todos los terminales derivados de X (si X es un no terminal). Dicho en forma más precisa,

- Si el análisis sintáctico es ascendente, entonces ejecutamos la acción a tan pronto como aparezca esta ocurrencia de X en el tope de la pila de análisis sintáctico.
- Si el análisis sintáctico es descendente, ejecutamos la acción a justo antes de tratar de expandir esta ocurrencia de Y (si Y es una no terminal) o comprobamos a Y en la entrada (si Y es una terminal).

Los esquemas de traducción orientados a la sintaxis que pueden implementarse durante el análisis sintáctico incluyen a los esquemas de traducción orientados a la sintaxis postfijos y a una clase de esquemas de traducción orientados a la sintaxis que veremos en la sección 5.5, los cuales implementan definiciones con atributos heredados por la izquierda. No todos los esquemas de traducción orientados a la sintaxis pueden implementarse durante el análisis sintáctico, como veremos en el siguiente ejemplo.

Ejemplo 5.16: Como un ejemplo extremo de un esquema de traducción orientado a la sintaxis problemático, suponga que convertimos nuestro ejemplo de una calculadora de escritorio en un esquema de traducción orientado a la sintaxis que imprime la forma prefijo de una expresión, en vez de evaluar esa expresión. En la figura 5.21 se muestran las producciones y las acciones.

- 1) $L \rightarrow E \mathbf{n}$
- 2) $E \rightarrow \{ \text{print}('+'); \} \ E_1 + T$
- 3) $E \rightarrow T$
- 4) $T \rightarrow \{ \text{print}('*'); \} \ T_1 * F$
- 5) $T \rightarrow F$
- 6) $F \rightarrow (E)$
- 7) $F \rightarrow \text{digito} \{ \text{print}(\text{digito}.valex); \}$

Figura 5.21: Esquema de traducción orientado a la sintaxis problemático para la traducción de infijo a postfijo durante el análisis sintáctico

Por desgracia, es imposible implementar este esquema de traducción orientado a la sintaxis durante el análisis sintáctico descendente o ascendente, debido a que el analizador sintáctico tendría que realizar acciones críticas, como imprimir instancias de $*$ o $+$, mucho antes de saber si estos símbolos aparecerán en su entrada.

Usando los no terminales como marcadores M_2 y M_4 para las acciones en las producciones 2 y 4, respectivamente, en la entrada 3 un analizador sintáctico de desplazamiento-reducción (vea la sección 4.5.3) tiene conflictos entre reducir mediante $M_2 \rightarrow \epsilon$, reducir mediante $M_4 \rightarrow \epsilon$, o desplazar el dígito. \square

Cualquier esquema de traducción orientado a la sintaxis puede implementarse de la siguiente manera:

1. Ignorando las acciones, se analiza sintácticamente la entrada y se produce un árbol de análisis sintáctico como resultado.
2. Despues se examina cada nodo interior N , por ejemplo, uno para la producción $A \rightarrow \alpha$. Se agregan hijos adicionales a N para las acciones en α , para que los hijos de N de izquierda a derecha tengan exactamente los símbolos y las acciones de α .
3. Realizar un recorrido en preorden (vea la sección 2.3.4) del árbol, y tan pronto como se visite un nodo etiquetado por una acción, realizar esa acción.

Por ejemplo, la figura 5.22 muestra el árbol de análisis sintáctico para la expresión $3 * 5 + 4$ con acciones insertadas. Si visitamos los nodos en preorden, obtenemos la forma prefijo de la expresión: $+ * 3 5 4$.

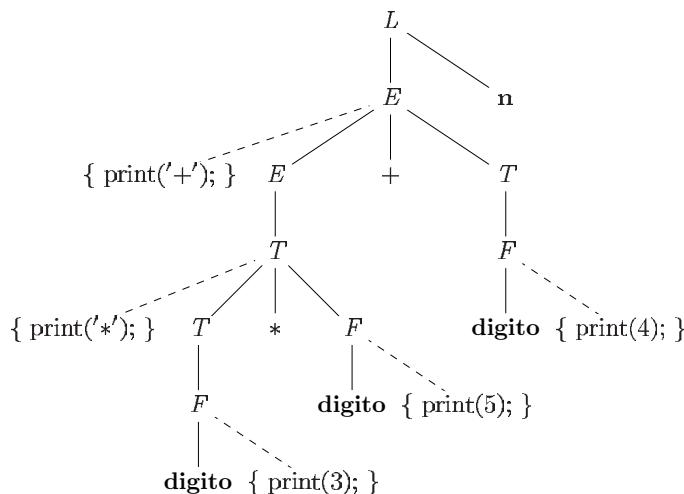


Figura 5.22: Árbol de análisis sintáctico con acciones incrustadas

5.4.4 Eliminación de la recursividad por la izquierda de los esquemas de traducción orientados a la sintaxis

Como ninguna gramática con recursividad por la izquierda puede analizarse sintácticamente en forma determinista, de forma descendente, en la sección 4.3.3 examinamos la eliminación de la recursividad por la izquierda. Cuando la gramática forma parte de un esquema de traducción orientado a la sintaxis, también debemos preocuparnos de cómo se manejan las acciones.

En primer lugar, considere el caso simple, en donde lo único que nos importa es el orden en el que se realizan las acciones en un esquema de traducción orientado a la sintaxis. Por ejemplo, si cada acción sólo imprime una cadena, sólo nos preocupa el orden en el que se imprimen las cadenas. En este caso, el siguiente principio puede servirnos de guía:

- Al transformar la gramática, trate a las acciones como si fueran símbolos terminales.

Este principio se basa en la idea de que la transformación gramatical preserva el orden de los terminales en la cadena generada. Por lo tanto, las acciones se ejecutan en el mismo orden en cualquier análisis sintáctico de izquierda a derecha, de arriba hacia abajo o de abajo hacia arriba.

El “truco” para eliminar la recursividad por la izquierda es tomar dos producciones:

$$A \rightarrow A\alpha \mid \beta$$

que generen cadenas que consisten en una β y cualquier número de α s, y sustituirlas por producciones que generen las mismas cadenas, usando un nuevo no terminal R (de “resto”) de la primera producción:

$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \mid \epsilon \end{aligned}$$

Si β no empieza con A , entonces A ya no tiene una producción recursiva por la izquierda. En términos de las definiciones regulares, con ambos conjuntos de producciones, A se define mediante $\beta(\alpha)^*$. En la sección 4.3.3 podrá consultar la sección sobre el manejo de situaciones en las que A tiene más producciones recursivas o no recursivas.

Ejemplo 5.17: Considera las siguientes producciones E de un esquema de traducción orientado a la sintaxis, para traducir expresiones infijo a la notación postfijo:

$$\begin{aligned} E &\rightarrow E_1 + T \quad \{ \text{print}('+'); \} \\ E &\rightarrow T \end{aligned}$$

Si aplicamos la transformación estándar a E , el resto de la producción recursiva por la izquierda es:

$$\alpha = + T \quad \{ \text{print}('+'); \}$$

y β , el cuerpo de la otra producción es T . Si introducimos R para el resto de E , obtenemos el siguiente conjunto de producciones:

$$\begin{aligned} E &\rightarrow T R \\ R &\rightarrow + T \quad \{ \text{print}('+'); \} R \\ R &\rightarrow \epsilon \end{aligned}$$

□

Cuando las acciones de una definición dirigida por la sintaxis calculan atributos en vez de sólo imprimir la salida, debemos tener más cuidado en la forma de eliminar la recursividad por la izquierda de una gramática. No obstante, si la definición dirigida por la sintaxis tiene atributos sintetizados, entonces siempre podremos construir un esquema de traducción orientado a la sintaxis colocando las acciones de cálculo de atributos en posiciones apropiadas en las producciones nuevas.

Vamos a ver un esquema general para el caso de una sola producción recursiva, una sola producción no recursiva, y un solo atributo del no terminal recursiva por la izquierda; la generalización a muchas producciones de cada tipo no es difícil, pero en cuanto a notación es compleja. Suponga que las dos producciones son:

$$\begin{array}{lcl} A & \rightarrow & A_1 Y \{ A.a = g(A_1.a, Y.y) \} \\ A & \rightarrow & X \{ A.a = f(X.x) \} \end{array}$$

Aquí, $A.a$ es el atributo sintetizado del no terminal A recursivo por la izquierda, y X y Y son símbolos gramaticales individuales con los atributos sintetizados $X.x$ y $Y.y$, respectivamente. Éstos podrían representar a una cadena de varios símbolos gramaticales, cada uno con su(s) propio(s) atributo(s), ya que el esquema tiene una función arbitraria g que calcula a $A.a$ en la producción recursiva, y una función arbitraria f que calcula a $A.a$ en la segunda producción. En cada caso, f y g reciben como argumentos los atributos a los que tengan permitido el acceso, si la definición dirigida por la sintaxis tiene atributos sintetizados.

Queremos convertir la gramática subyacente en:

$$\begin{array}{lcl} A & \rightarrow & X R \\ R & \rightarrow & Y R \mid \epsilon \end{array}$$

La figura 5.23 sugiere lo que debe hacer el esquema de traducción orientado a la sintaxis con la nueva gramática. En (a) podemos ver el efecto del esquema de traducción orientado a la sintaxis postfijo sobre la gramática original. Aplicamos f una vez, que corresponde al uso de la producción $A \rightarrow X$, y después aplicamos g todas las veces que utilicemos la producción $A \rightarrow A Y$. Como R genera un “residuo” de Y s, su traducción depende de la cadena a su izquierda, una cadena de la forma $XYY \dots Y$. Cada uso de la producción $R \rightarrow Y R$ resulta en una aplicación de g . Para R , utilizamos un atributo heredado $R.i$ para acumular el resultado de aplicar la función g en forma sucesiva, empezando con el valor de $A.a$.

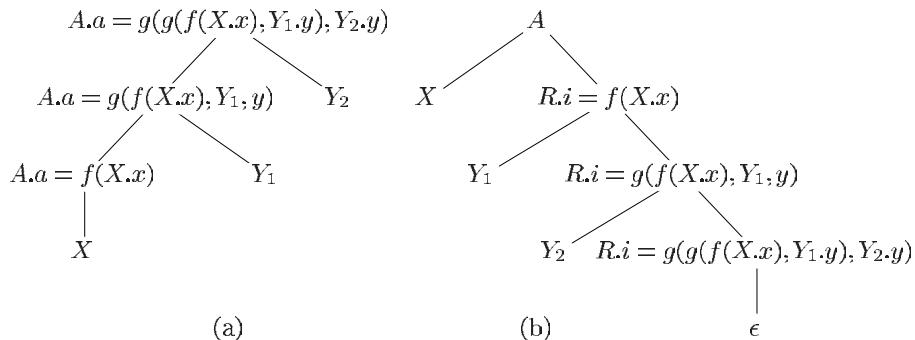


Figura 5.23: Eliminación de la recursividad por la izquierda de un esquema de traducción orientado a la sintaxis postfijo

Además, R tiene un atributo sintetizado $R.s$, el cual no se muestra en la figura 5.23. Este atributo se calcula primero cuando R termina su generación de símbolos Y , como se indica mediante el uso de la producción $R \rightarrow \epsilon$. Después, $R.s$ se copia hacia arriba en el árbol, para que pueda convertirse en el valor de $A.a$ para toda la expresión $XYY \dots Y$. El caso en el que A genera a XYY se muestra en la figura 5.23, y podemos ver que el valor de $A.a$ en la raíz de (a) tiene dos usos de g . Lo mismo pasa con $R.i$ en la parte inferior del árbol (b), y es el valor de $R.s$ el que se copia hacia arriba de ese árbol.

Para lograr esta traducción, utilizamos el siguiente *esquema de traducción orientado a la sintaxis*:

$$\begin{array}{lcl}
 A & \rightarrow & X \{R.i = f(X.x)\} \quad R \{A.a = R.s\} \\
 R & \rightarrow & Y \{R_1.i = g(R.i, Y.y)\} \quad R_1 \{R.s = R_1.s\} \\
 R & \rightarrow & \epsilon \{R.s = R.i\}
 \end{array}$$

Observe que el atributo heredado $R.i$ se evalúa de inmediato, antes de usar R en el cuerpo, mientras que los atributos sintetizados $A.a$ y $R.s$ se evalúan al final de las producciones. Por ende, cualquiera que sean los valores que se tengan que calcular, estos atributos estarán disponibles en lo que se haya calculado a la izquierda.

5.4.5 Esquemas de traducción orientados a la sintaxis para definiciones con atributos heredados por la izquierda

En la sección 5.4.1 vimos la conversión de definiciones dirigidas por la sintaxis con atributos sintetizados en esquemas de traducción orientados a la sintaxis postfijo, con acciones en los extremos derechos de las producciones. Siempre y cuando la gramática subyacente sea LR, los Esquemas de traducción orientados a la sintaxis postfijos podrán analizarse sintácticamente y traducirse de abajo hacia arriba.

Ahora, vamos a considerar el caso más general de un definición dirigida por la sintaxis con atributos heredados por la izquierda. Vamos a suponer que la gramática subyacente puede analizarse de arriba hacia abajo, porque de lo contrario sería con frecuencia imposible realizar la traducción en conexión con un analizador sintáctico LL o LR. Con cualquier gramática, la técnica antes mencionada puede implementarse adjuntando las acciones a un árbol de análisis sintáctico y ejecutándolas durante el recorrido del árbol.

A continuación se muestran las reglas para convertir una definición dirigida por la sintaxis con atributos heredados por la izquierda en un esquema de traducción orientado a la sintaxis:

1. Incrustar la acción que calcule los atributos heredados para un no terminal A , justo después de esa ocurrencia de A en el cuerpo de la producción. Si varios atributos heredados para A dependen uno del otro en forma acíclica, ordenar la evaluación de los atributos, de manera que los que se necesiten primero sean los que se calculen primero.
2. Colocar las acciones que calculan un atributo sintetizado para el encabezado de una producción, al final del cuerpo de esa producción.

Vamos a ilustrar estos principios con dos ejemplos completos. El primero involucra a la composición tipográfica. Muestra cómo pueden usarse las técnicas de compilación en el procesamiento de lenguajes, para aplicaciones distintas a las que consideramos, por lo general, como lenguajes de programación. El segundo ejemplo trata acerca de la generación de código intermedio para una construcción común en un lenguaje de programación: una forma de instrucción while.

Ejemplo 5.18: Este ejemplo está inspirado en los lenguajes para las fórmulas matemáticas de la composición tipográfica. **Eqn** es uno de los primeros ejemplos de dicho lenguaje; las ideas de **Eqn** se siguen utilizando en el sistema de composición tipográfica **TEX**, que se utilizó para producir este libro.

Vamos a concentrarnos sólo en la capacidad de definir subíndices, subíndices de subíndices, y así en lo sucesivo, ignorando los superíndices, las fracciones y subfracciones y todas las demás características matemáticas. En el lenguaje **Eqn**, escribimos **a sub i sub j** para establecer la expresión a_{ij} . Una gramática simple para los *cuadros* (elementos de texto delimitados por un rectángulo) es:

$$B \rightarrow B_1 B_2 \mid B_1 \mathbf{sub} B_2 \mid (B_1) \mid \mathbf{texto}$$

De manera correspondiente a estas cuatro producciones, un cuadro puede ser:

1. Dos cuadros yuxtapuestos, con la primera producción B_1 , a la izquierda de la otra producción B_2 .
2. Un cuadro y un cuadro de subíndice. El segundo cuadro aparece en un tamaño más pequeño, más abajo y a la derecha del primer cuadro.
3. Un cuadro entre paréntesis, para agrupar cuadros y subíndices. Eqn y TeX utilizan llaves para agrupar, pero nosotros usaremos paréntesis redondos ordinarios para evitar que se confundan con las llaves que rodean a las acciones en los esquemas de traducción orientados a la sintaxis.
4. Una cadena de texto; es decir, cualquier cadena de caracteres.

Esta gramática es ambigua, pero aún podemos usarla para el análisis sintáctico ascendente, si hacemos al subíndice y la yuxtaposición asociativos a la derecha, en donde **sub** tenga mayor precedencia que la yuxtaposición.

Se aplicará la composición tipográfica a las expresiones mediante la construcción de cuadros más grandes que delimiten a los más pequeños. En la figura 5.24, se va a crear una yuxtaposición entre los cuadros para E_1 y $.altura$, para formar el cuadro para $E_1.altura$. El cuadro izquierdo para E_1 se construye a partir del cuadro para E y el subíndice 1. Para manejar el subíndice 1, se reduce su cuadro aproximadamente un 30%, se baja y se coloca después del cuadro para E . Aunque debemos tratar a $.altura$ como una cadena de texto, los rectángulos dentro de su cuadro muestran cómo puede construirse a partir de los cuadros para las letras individuales.



Figura 5.24: Construcción de cuadros más grandes, a partir de cuadros más pequeños

En este ejemplo, nos concentraremos sólo en la geometría vertical de los cuadros. La geometría horizontal (la anchura de los cuadros) también es interesante, en especial cuando caracteres distintos tienen anchuras diferentes. Tal vez no sea aparente a la vista, pero cada uno de los caracteres distintos en la figura 5.24 tienen anchura diferente.

A continuación se describen los valores asociados con la geometría vertical de los cuadros:

- a) El *tamaño de punto* se utiliza para establecer el texto dentro de un cuadro. Vamos a suponer que los caracteres que no están en subíndices se establecen en un tipo de punto 10, el tamaño del tipo en este libro. Además, si un cuadro tiene el tamaño de punto p , entonces su cuadro de subíndice tiene el tamaño de punto más pequeño $0.7p$. El atributo heredado $B.tp$ representará el tamaño de punto del bloque B . Este atributo debe ser heredado, ya que el contexto determina cuánto necesita reducirse un cuadro dado, debido al número de niveles de subíndices.

- b) Cada cuadro tiene una *línea base*, la cual es una posición vertical que corresponde a la parte inferior de las líneas de texto, sin contar las letras, como “g” que se extiende debajo de la línea base normal. En la figura 5.24, la línea punteada representa la línea base para los cuadros E , $.altura$ y toda la expresión completa. La línea base para el cuadro que contiene el subíndice 1 se ajusta para bajar el subíndice.
- c) Un cuadro tiene una *altura*, que es la distancia a partir de la parte superior del cuadro hasta la línea base. El atributo sintetizado $B.al$ proporciona la altura del cuadro B .
- d) Un cuadro tiene una *profundidad*, que es la distancia a partir de la línea base, hasta la parte inferior del cuadro. El atributo sintetizado $B.pr$ proporciona la profundidad del cuadro B .

La definición dirigida por la sintaxis en la figura 5.25 nos proporciona las reglas para calcular los tamaños de punto, las alturas y las profundidades. La producción 1 se utiliza para asignar $B.tp$ como el valor inicial 10.

PRODUCCIÓN	REGLAS SEMÁNTICAS
1) $S \rightarrow B$	$B.tp = 10$
2) $B \rightarrow B_1 B_2$	$B_1.tp = B.tp$ $B_2.tp = B.tp$ $B.al = \max(B_1.al, B_2.al)$ $B.pr = \max(B_1.pr, B_2.pr)$
3) $B \rightarrow B_1 \text{ sub } B_2$	$B_1.tp = B.tp$ $B_2.tp = 0.7 \times B.tp$ $B.al = \max(B_1.al, B_2.al - 0.25 \times B.tp)$ $B.pr = \max(B_1.pr, B_2.pr + 0.25 \times B.tp)$
4) $B \rightarrow (B_1)$	$B_1.tp = B.tp$ $B.al = B_1.al$ $B.pr = B_1.pr$
5) $B \rightarrow \text{texto}$	$B.al = \text{obtenerAl}(B.tp, \text{texto.valex})$ $B.pr = \text{obtenerPr}(B.tp, \text{texto.valex})$

Figura 5.25: Definición dirigida por la sintaxis para la composición tipográfica de los cuadros

La producción 2 maneja la yuxtaposición. Los tamaños de los puntos se copian hacia abajo del árbol de análisis sintáctico; es decir, dos subcuadros de un cuadro heredan el mismo tamaño de punto del cuadro más grande. Las alturas y las profundidades se calculan hacia arriba del árbol, tomando el máximo. Esto es, la altura del cuadro más grande es el máximo de las alturas de sus dos componentes, y esto es igual para la profundidad.

La producción 3 maneja el uso de subíndices, y es la más sutil. En este ejemplo bastante simplificado, asumimos que el tamaño de punto de un cuadro con subíndice es 70% del tamaño de punto de su padre. La realidad es mucho más compleja, ya que los subíndices no se pueden

reducir en forma indefinida; en la práctica, después de unos cuantos niveles, los tamaños de los subíndices casi no se reducen. Además, suponemos que la línea base de un cuadro de subíndice se reduce un 25% del tamaño de punto del padre; de nuevo, la realidad es más compleja.

La producción 4 copia los atributos en forma apropiada, cuando se utilizan paréntesis. Por último, la producción 5 maneja las hojas que representan los cuadros de texto. También en esta cuestión, la situación verdadera es complicada, por lo que sólo mostramos dos funciones sin especificaciones llamadas *obtenerAl* y *obtenerPr*, las cuales examinan las tablas creadas con cada fuente para determinar la máxima altura y la máxima profundidad de cualquier carácter en la cadena de texto. Se supone que se proporciona la misma cadena como el atributo *valex* de el terminal **texto**.

Nuestra última tarea es convertir esta definición dirigida por la sintaxis en un esquema de traducción orientado a la sintaxis, siguiendo las reglas para una definición dirigida por la sintaxis con atributos heredados por la izquierda, y la figura 5.25 es de este tipo. El esquema de traducción orientado a la sintaxis apropiado se muestra en la figura 5.26. Por cuestión de legibilidad, como los cuerpos de las producciones se vuelven extensos, los dividimos entre varias líneas y alineamos las acciones. Por lo tanto, los cuerpos de las producciones consisten en el contenido de todas las líneas, hasta el encabezado de la siguiente producción. \square

PRODUCCIÓN	ACCIONES
1) $S \rightarrow B$	$\{ B.tp = 10; \}$
2) $B \rightarrow B_1$	$\{ B_1.tp = B.tp; \}$
$B \rightarrow B_2$	$\{ B_2.tp = B.tp; \}$ $\{ B.al = \max(B_1.al, B_2.al); \}$ $\{ B.pr = \max(B_1.pr, B_2.pr); \}$
3) $B \rightarrow B_1 \text{ sub } B_2$	$\{ B_1.tp = B.tp; \}$ $\{ B_2.tp = 0.7 \times B.tp; \}$ $\{ B.al = \max(B_1.al, B_2.al - 0.25 \times B.tp); \}$ $\{ B.pr = \max(B_1.pr, B_2.pr + 0.25 \times B.tp); \}$
4) $B \rightarrow (B_1)$	$\{ B_1.tp = B.tp; \}$ $\{ B.al = B_1.al; \}$ $\{ B.pr = B_1.pr; \}$
5) $B \rightarrow \text{texto}$	$\{ B.al = \text{obtenerAl}(B.tp, \text{texto.valex}); \}$ $\{ B.pr = \text{obtenerPr}(B.tp, \text{texto.valex}); \}$

Figura 5.26: Esquema de traducción orientado a la sintaxis para la composición tipográfica de los cuadros

Nuestro siguiente ejemplo se concentra en una instrucción `while` simple y la generación de código intermedio para este tipo de instrucción. El código inmediato se tratará como un atributo con valor de cadena. Más adelante, exploraremos técnicas que involucran la escritura de piezas de un atributo con valor de cadena a medida que realizamos el análisis sintáctico, con lo cual evitamos la copia de cadenas extensas, para construir cadenas aún más largas. La técnica

se presentó en el ejemplo 5.17, en donde generamos la forma postfija de una expresión infija “al instante”, en vez de calcularla como un atributo. No obstante, en nuestra primera formulación, creamos un atributo con valor de cadena mediante la concatenación.

Ejemplo 5.19: Para este ejemplo, sólo necesitamos una producción:

$$S \rightarrow \mathbf{while} (C) S_1$$

Aquí, S es el no terminal que genera todo tipo de instrucciones, que supuestamente incluye instrucciones if, instrucciones de asignación y otras. En este ejemplo, C representa a una expresión condicional: una expresión booleana que se evalúa como verdadera o falsa.

En este ejemplo de flujo de control, lo único que vamos a generar son etiquetas. Se asume que todas las demás instrucciones de código intermedio se van a generar mediante partes del esquema de traducción orientado a la sintaxis que no se muestran. En forma específica, generaremos instrucciones explícitas de la forma **etiqueta** L , en donde L es un identificador, para indicar que L es la etiqueta de la instrucción que le sigue. Asumimos que el código intermedio es como el que se presentó en la sección 2.8.4.

El propósito de nuestra instrucción while es que se evalúe la C condicional. Si es verdadera, el control pasa al principio del código para S_1 . Si es falsa, entonces el control pasa al código que sigue del código de la instrucción while. El código para S_1 debe diseñarse de manera que salte al principio del código para la instrucción while al terminar; el salto al principio del código que evalúa a C no se muestra en la figura 5.27.

Utilizamos los siguientes atributos para generar el código intermedio apropiado:

1. El atributo heredado $S.sigiente$ etiqueta el principio del código que debe ejecutarse una vez que S termine.
2. El atributo sintetizado $S.codigo$ es la secuencia de pasos de código intermedio que implementa a una instrucción S y termina con un salto a $S.sigiente$.
3. El atributo heredado $C.true$ etiqueta el principio del código que debe ejecutarse si C es verdadera.
4. El atributo heredado $C.false$ etiqueta el principio del código que debe ejecutarse si C es falsa.
5. El atributo sintetizado $C.codigo$ es la secuencia de pasos de código intermedio que implementa a la condición C , y salta a $C.true$ o a $C.false$, dependiendo de si C es verdadera o falsa.

La definición dirigida por la sintaxis que calcula estos atributos para la instrucción while se muestra en la figura 5.27. Hay varios puntos que ameritan una explicación:

- La función *new* genera nuevas etiquetas.
- Las variables $L1$ y $L2$ contienen etiquetas que necesitamos en el código. $L1$ es el principio del código para la instrucción while, y tenemos que arreglar que S_1 salte ahí una vez que termine. Ésta es la razón por la cual establecemos $S_1.sigiente$ a $L1$. $L2$ es el principio del código para S_1 , y se convierte en el valor de $C.true$, debido a que bifurcamos hacia allá cuando C es verdadera.

$$\begin{array}{ll}
 S \rightarrow \mathbf{while} (C) S_1 & L1 = \text{new}(); \\
 & L2 = \text{new}() \\
 & S_1.\text{siguiente} = L1; \\
 & C.\text{false} = S.\text{siguiente}; \\
 & C.\text{true} = L2; \\
 & S.\text{codigo} = \mathbf{etiqueta} \parallel L1 \parallel C.\text{codigo} \parallel \mathbf{etiqueta} \parallel L2 \parallel S_1.\text{codigo}
 \end{array}$$

Figura 5.27: Esquema de definición dirigida por la sintaxis para las instrucciones while

- Observe que $C.\text{false}$ se establece a $S.\text{siguiente}$, ya que cuando la condición es falsa, ejecutamos el código que vaya después del código para S .
- Utilizamos \parallel como el símbolo de concatenación de los fragmentos de código intermedio. Por lo tanto, el valor de $S.\text{codigo}$ empieza con la etiqueta $L1$, después el código para la condición C , otra etiqueta $L2$ y el código para S_1 .

Esta definición dirigida por la sintaxis tiene atributos heredados por la izquierda. Al convertirla en un esquema de traducción orientado a la sintaxis, lo único que queda pendiente es cómo manejar las etiquetas $L1$ y $L2$, que son variables y no atributos. Si tratamos a las acciones como no terminales falsos, entonces dichas variables pueden tratarse como los atributos sintetizados de los no terminales falsos. Como $L1$ y $L2$ no dependen de ningún otro atributo, pueden asignarse a la primera acción en la producción. En la figura 5.28 se muestra el esquema de traducción orientado a la sintaxis resultante con acciones incrustadas que implementa a esta definición con atributos heredados por la izquierda. \square

$$\begin{array}{ll}
 S \rightarrow \mathbf{while} (& \{ L1 = \text{new}(); L2 = \text{new}(); C.\text{false} = S.\text{siguiente}; C.\text{true} = L2; \} \\
 C) & \{ S_1.\text{siguiente} = L1; \} \\
 S_1 & \{ S.\text{codigo} = \mathbf{etiqueta} \parallel L1 \parallel C.\text{codigo} \parallel \mathbf{etiqueta} \parallel L2 \parallel S_1.\text{codigo}; \}
 \end{array}$$

Figura 5.28: Esquema de traducción orientado a la sintaxis para las instrucciones while

5.4.6 Ejercicios para la sección 5.4

Ejercicio 5.4.1: En la sección 5.4.2 mencionamos que es posible deducir, a partir del estado LR en la pila de análisis sintáctico, cuál es el símbolo gramatical que se representa mediante el estado. ¿Cómo descubriríamos esta información?

Ejercicio 5.4.2: Reescriba el siguiente esquema de traducción orientado a la sintaxis:

$$\begin{array}{l}
 A \rightarrow A \{a\} B \mid A B \{b\} \mid 0 \\
 B \rightarrow B \{c\} A \mid B A \{d\} \mid 1
 \end{array}$$

de manera que la gramática subyacente quede sin recursividad por la izquierda. Aquí, a , b , c y d son acciones, y 0 y 1 son no terminales.

! Ejercicio 5.4.3: El siguiente esquema de traducción orientado a la sintaxis calcula el valor de una cadena de 0s y 1s, interpretada como un entero binario positivo.

$$\begin{array}{lcl} B & \rightarrow & B_1 0 \{B.val = 2 \times B_1.val\} \\ & | & B_1 1 \{B.val = 2 \times B_1.val + 1\} \\ & | & 1 \{B.val = 1\} \end{array}$$

Reescriba este esquema de traducción orientado a la sintaxis, de manera que la gramática subyacente no sea recursiva por la izquierda, y de todas formas se calcule el mismo valor de $B.val$ para toda la cadena de entrada completa.

! Ejercicio 5.4.4: Escriba definiciones dirigidas por la sintaxis con atributos heredados por la izquierda, que sean análogas a las del ejemplo 5.19 para las siguientes producciones, cada una de las cuales representa a una construcción de flujo de control conocida, como en el lenguaje de programación C. Tal vez deba generar una instrucción de tres direcciones para saltar hacia una etiqueta L específica, en cuyo caso debe generar **goto** L .

- a) $S \rightarrow \text{if (} C \text{) } S_1 \text{ else } S_2$
- b) $S \rightarrow \text{do } S_1 \text{ while (} C \text{) }$
- c) $S \rightarrow \{' L ' \{'; L \rightarrow L S | \epsilon$

Observe que cualquier instrucción en la lista puede tener un salto desde su parte media hasta la siguiente instrucción, por lo que no basta con sólo generar código para cada instrucción en orden.

Ejercicio 5.4.5: Convierta cada una de sus definiciones dirigidas por la sintaxis del ejercicio 5.4.4 en un esquema de traducción orientado a la sintaxis, como en el ejemplo 5.19.

Ejercicio 5.4.6: Modifique la definición dirigida por la sintaxis de la figura 5.25 para incluir un atributo sintetizado $B.le$, la longitud de un cuadro. La longitud de la concatenación de dos cuadros es la suma de las longitudes de cada uno. Despues agregue sus nuevas reglas a las posiciones apropiadas en el esquema de traducción orientado a la sintaxis de la figura 5.26.

Ejercicio 5.4.7: Modifique la definición dirigida por la sintaxis de la figura 5.25 para incluir superíndices denotados por el operador **sup** entre los cuadros. Si el cuadro B_2 es un superíndice del cuadro B_1 , entonces posicione la línea base de B_2 a 0.6 veces el tamaño de punto de B_1 encima de la línea base de B_1 . Agregue la nueva producción y las reglas al esquema de traducción orientado a la sintaxis de la figura 5.26.

5.5 Implementación de definiciones dirigidas por la sintaxis con atributos heredados por la izquierda

Debido a que pueden tratarse muchas aplicaciones de traducción mediante el uso de definiciones con atributos heredados por la izquierda, vamos a considerar su implementación con más detalle en esta sección. Los siguientes métodos realizan la traducción mediante el recorrido de un árbol de análisis sintáctico:

1. *Construir el árbol de análisis sintáctico y anotar.* Este método funciona para cualquier definición dirigida por la sintaxis acíclica en cualquier caso. En la sección 5.1.2 vimos los árboles de análisis sintáctico anotados.
2. *Construir el árbol de análisis sintáctico, agregar las acciones y ejecutarlas en preorden.* Este método funciona para cualquier definición con atributos heredados por la izquierda. En la sección 5.4.5 vimos cómo convertir una definición dirigida por la sintaxis con atributos heredados por la izquierda en un esquema de traducción orientado a la sintaxis; en especial, hablamos sobre cómo incrustar las acciones en producciones, de acuerdo con las reglas semánticas de dicha definición dirigida por la sintaxis.

En esta sección, hablaremos sobre los siguientes métodos para traducir durante el análisis sintáctico:

3. *Usar un analizador de descenso recursivo* con una función para cada no terminal. La función para el no terminal A recibe los atributos heredados de A como argumentos, y devuelve los atributos sintetizados de A .
4. *Generar código al instante*, mediante el uso de un analizador sintáctico de descenso recursivo.
5. *Implementar un esquema de traducción orientado a la sintaxis en conjunto con un analizador sintáctico LL.* Los atributos sintetizados se mantienen en la pila de análisis sintáctico, y las reglas obtienen los atributos necesarios de ubicaciones conocidas en la pila.
6. *Implementar un esquema de traducción orientado a la sintaxis en conjunto con un analizador sintáctico LR.* Este método puede sorprenderle, ya que por lo general el esquema de traducción orientado a la sintaxis para una definición dirigida por la sintaxis con atributos heredados por la izquierda tiene acciones en medio de las producciones, y no podemos estar seguros que durante un análisis sintáctico LR nos encontremos siquiera en esa producción, sino hasta que se haya construido todo su cuerpo. Sin embargo, más adelante veremos que si la gramática subyacente es LL, siempre podremos manejar tanto el análisis como la traducción de abajo hacia arriba.

5.5.1 Traducción durante el análisis sintáctico de descenso recursivo

Un analizador sintáctico de descenso recursivo tiene una función A para cada no terminal A , como vimos en la sección 4.4.1. Podemos extender el analizador sintáctico en un traductor, de la siguiente manera:

- a) Los argumentos de la función A son los atributos heredados del no terminal A .
- b) El valor de retorno de la función A es la colección de atributos sintetizados del no terminal A .

En el cuerpo de la función A , debemos analizar y manejar los atributos:

1. Decidir la producción que utilizaremos para expandir A .
2. Comprobar que cada terminal aparezca en la entrada cuando se le requiera. Debemos asumir que no es necesario el rastreo hacia atrás, pero la extensión al análisis sintáctico de descenso recursivo con este rastreo puede llevarse a cabo mediante la restauración de la posición de entrada al momento de la falla, como vimos en la sección 4.4.1.

3. Preservar en las variables locales los valores de todos los atributos necesarios para calcular los atributos heredados para los no terminales en el cuerpo, o los atributos sintetizados para el no terminal del encabezado.
4. Llamar a las funciones correspondientes a los no terminales en el cuerpo de la producción seleccionada, proporcionándoles los argumentos apropiados. Como la definición dirigida por la sintaxis subyacente tiene atributos heredados por la izquierda, ya hemos calculado estos atributos y los almacenamos en variables locales.

Ejemplo 5.20: Vamos a considerar la definición dirigida por la sintaxis y el esquema de traducción orientado a la sintaxis del ejemplo 5.19 para las instrucciones while. En la figura 5.29 aparece una muestra en seudocódigo de las partes relevantes de la función S .

```

cadena  $S(\text{etiqueta siguiente})$  {
  cadena  $Scodigo, Ccodigo$ ; /* las variables locales contienen fragmentos de código */
  etiqueta  $L1, L2$ ; /* las etiquetas locales */
  if ( entrada actual == token while ) {
    avanzar entrada;
    comprobar que siga '(' en la entrada, y avanzar;
     $L1 = new()$ ;
     $L2 = new()$ ;
     $Ccodigo = C(\text{siguiente}, L2)$ ;
    comprobar que siga ')' en la entrada, y avanzar;
     $Scodigo = S(L1)$ ;
    return("etiqueta" ||  $L1$  ||  $Ccodigo$  || "etiqueta" ||  $L2$  ||  $Scodigo$ );
  }
  else /* otros tipos de instrucciones */
}

```

Figura 5.29: Implementación de instrucciones while con un analizador sintáctico de descenso recursivo

Mostramos a S para almacenar y devolver cadenas largas. En la práctica, sería mucho más eficiente para las funciones como S y C devolver apuntadores a registros que representen estas cadenas. Así, la instrucción de retorno en la función S no concatenaría físicamente los componentes mostrados, sino que construiría un registro, o tal vez un árbol de registros, expresando la concatenación de las cadenas representadas por $Scodigo$ y $Ccodigo$, las etiquetas $L1$ y $L2$, y las dos ocurrencias de la cadena literal "etiqueta". \square

Ejemplo 5.21: Ahora vamos a tomar el esquema de traducción orientado a la sintaxis de la figura 5.26 para cuadros de composición tipográfica. Primero señalaremos el análisis sintáctico, ya que la gramática subyacente en la figura 5.26 es ambigua. La siguiente gramática transformada hace a la yuxtaposición y al uso de subíndices asociativos a la derecha, en donde **sub** tiene precedencia sobre la yuxtaposición:

$$\begin{array}{lcl}
 S & \rightarrow & B \\
 B & \rightarrow & T B_1 \mid T \\
 T & \rightarrow & F \mathbf{sub} T_1 \mid F \\
 F & \rightarrow & (B) \mid \mathbf{texto}
 \end{array}$$

Los dos nuevos no terminales, T y F , se motivan mediante términos y factores en las expresiones. Aquí, un “factor” generado por F es un cuadro entre paréntesis o una cadena de texto. Un “término” generado por T es un “factor” con una secuencia de subíndices, y un cuadro generado por B es una secuencia de “términos” yuxtapuestos.

Los atributos de B se transportan hacia T y F , ya que los nuevas no terminales también denotan cuadros; se introdujeron sólo para ayudar en el análisis sintáctico. Por ende, tanto T como F tienen un atributo heredado tp , además de los atributos heredados al y pr , con acciones semánticas que pueden adaptarse del esquema de traducción orientado a la sintaxis en la figura 5.26.

La gramática aún no está lista para el análisis sintáctico descendente, ya que las producciones para B y T tienen prefijos comunes. Por ejemplo, considere a T . Un analizador sintáctico descendente no puede elegir una de las dos producciones para T con sólo ver un símbolo por adelantado en la entrada. Por fortuna, podemos usar una forma de factorización por la izquierda, descrita en la sección 4.3.4, para dejar lista la gramática. Con los esquemas de traducción orientados a la sintaxis, la noción de prefijo común se aplica a las acciones también. Ambas producciones para T empiezan con el no terminal F , que hereda el atributo tp de T .

El seudocódigo en la figura 5.30 para $T(tp)$ se mezcla con el código para $F(tp)$. Después de aplicar la factorización por la izquierda a $T \rightarrow F \mathbf{sub} T_1 \mid F$, sólo hay una llamada a F ; el seudocódigo muestra el resultado de sustituir el código para F en vez de esta llamada.

La llamada a la función T se hace como $T(10.0)$ por medio de la función para B , la cual no mostramos. Devuelve un par que consiste en la altura y la profundidad del cuadro generado por el no terminal T ; en la práctica, devolvería un registro que contiene la altura y la profundidad.

La función T empieza comprobando si hay un paréntesis izquierdo, en cuyo caso debe tener la producción $F \rightarrow (B)$ para trabajar con ella. Guarda lo que devuelva la B dentro de los paréntesis, pero si esa B no va seguida de un paréntesis derecho, entonces hay un error de sintaxis, el cual debe manejarse de una forma que no se muestra.

En cualquier otro caso, si la entrada actual es **texto**, entonces la función T utiliza *obtenerAl* y *obtenerPr* para determinar la anchura y la profundidad de este texto.

Después, T decide si el siguiente cuadro es un subíndice o no, y ajusta el tamaño del punto, en caso de que sí sea. Utilizamos las acciones asociadas con la producción $B \rightarrow B \mathbf{sub} B$ en la figura 5.26 para la altura y la profundidad del cuadro más grande. En cualquier otro caso, sólo devolvemos lo que F hubiera devuelto: $(a1, p1)$. \square

5.5.2 Generación de código al instante

La construcción de cadenas largas de código que son valores de atributos, como en el ejemplo 5.20, no es apropiado por varias razones, incluyendo el tiempo requerido para copiar o mover cadenas largas. En casos comunes como nuestro ejemplo de generación de código, podemos en su lugar generar piezas en forma incremental del código y colocarlas en un arreglo o archivo de salida, mediante la ejecución de las acciones en un esquema de traducción orientado a la sintaxis. Los elementos que necesitamos para hacer funcionar esta técnica son:

```

(float, float) T (float tp) {
    float a1, a2, p1, p2; /* variables locales que guardan alturas y profundidades */
    /* código inicial para F(tp) */
    if ( entrada actual == '(' ) {
        avanzar entrada;
        (a1, p1) = B(tp);
        if (entrada actual != ')' ) error de sintaxis: se esperaba ')';
        avanzar entrada;
    }
    else if ( entrada actual == texto ) {
        hacer que el valor léxico de texto.valex sea igual a t;
        avanzar entrada;
        a1 = obtenerAl(tp, t);
        p1 = obtenerPr(tp, t);
    }
    else error de sintaxis: se esperaba texto o '(';
    /* fin del código para F(tp) */
    if ( entrada actual == sub ) {
        avanzar entrada;
        (a2, p2) = T (0.7 * tp);
        return (max(a1, a2 - 0.25 * tp), max(p1, p2 + 0.25 * tp));
    }
    return (a1, p1);
}

```

Figura 5.30: Composición tipográfica de los cuadros con descenso recursivo

1. Para una o más no terminales existe un atributo *principal*. Por conveniencia, vamos a suponer que todos los atributos principales tienen valor de cadena. En el ejemplo 5.20, los atributos sintetizados *S.codigo* y *C.codigo* son atributos principales; los demás atributos no lo son.
2. Los atributos principales están sintetizados.
3. Las reglas que evalúan el (los) atributo(s) principal(es) aseguran que:
 - (a) El atributo principal sea la concatenación de los atributos principales de los no terminales que aparecen en el cuerpo de la producción involucrada, tal vez con otros elementos que no sean atributos principales, como la cadena **etiqueta** o los valores de las etiquetas *L1* y *L2*.
 - (b) Los atributos principales de los no terminales aparecen en la regla, en el mismo orden que los mismos no terminales aparecen en el cuerpo de la producción.

Como consecuencia de las condiciones antes mencionadas, el atributo principal puede construirse mediante la emisión de los elementos de la concatenación que no son atributos principales. Podemos confiar en las llamadas recursivas a las funciones para los no terminales en un cuerpo de producción, para emitir el valor de su atributo principal en forma incremental.

El tipo de los atributos principales

Nuestra suposición de simplificación de que los atributos principales son de tipo cadena es en realidad demasiado restrictiva. El verdadero requerimiento es que el tipo de todos los atributos principales debe tener valores que puedan construirse mediante la concatenación de elementos. Por ejemplo, una lista de objetos de cualquier tipo sería apropiada, siempre y cuando representemos estas listas de una forma que permita adjuntar en forma eficiente los elementos al final de la lista. Por ende, si el propósito del atributo principal es representar una secuencia de instrucciones de código intermedio, podríamos producir este código intermedio escribiendo instrucciones al final de un arreglo de objetos. Desde luego que los requerimientos declarados en la sección 5.5.2 se siguen aplicando a las listas; por ejemplo, los atributos principales deben ensamblarse a partir de otros atributos principales, mediante la concatenación en orden.

Ejemplo 5.22: Podemos modificar la función de la figura 5.29 para emitir los elementos de la traducción principal $S.codigo$, en vez de guardarlos para la concatenación en un valor de retorno de $S.codigo$. La función S modificada aparece en la figura 5.31.

```

void  $S(\text{etiqueta } siguiente)$  {
    etiqueta  $L1, L2$ ; /* las etiquetas locales */
    if ( entrada actual == token while ) {
        avanzar entrada;
        comprobar que siga '(' en la entrada, y avanzar;
         $L1 = new()$ ;
         $L2 = new()$ ;
         $print("etiqueta", L1)$ ;
         $C(siguiente, L2)$ ;
        comprobar que siga ')' en la entrada, y avanzar;
         $print("etiqueta", L2)$ ;
         $S(L1)$ ;
    }
    else /* otros tipos de instrucciones */
}

```

Figura 5.31: Generación de código de descenso recursivo al instante para instrucciones while

En la figura 5.32, S y C no tienen ahora valor de retorno, ya que sus únicos atributos sintetizados se producen mediante la impresión. Además, la posición de las primeras instrucciones es considerable. El orden en el que se imprime la salida es: primero **etiqueta** $L1$, después el código para C (que es igual que el valor de $C.codigo$ en la figura 5.29), después **etiqueta** $L2$, y por último el código de la llamada recursiva por S (que es igual que $S.codigo$ en la figura 5.29). Por ende, el código que imprime esta llamada a S es exactamente el mismo que el valor de $S.codigo$ que se devuelve en la figura 5.29). \square

De paso, podemos realizar el mismo cambio al esquema de traducción orientado a la sintaxis subyacente: convertir la construcción de un atributo principal en acciones que emitan los elementos de ese atributo. En la figura 5.32 podemos ver el esquema de traducción orientado a la sintaxis de la figura 5.28, modificado para generar código al instante.

```


$$\begin{array}{l}
S \rightarrow \mathbf{while} ( \quad \{ L1 = \mathit{new}(); L2 = \mathit{new}(); C.\mathit{false} = S.\mathit{siguiente}; \\
\quad \quad \quad C.\mathit{true} = L2; \mathit{print}("etiqueta", L1); \} \\
\quad \quad \quad S_1 \quad \quad \quad \{ S_1.\mathit{siguiente} = L1; \mathit{print}("etiqueta", L2); \}
\end{array}$$


```

Figura 5.32: Esquema de traducción orientado a la sintaxis para la generación de código al instante, para instrucciones while

5.5.3 Las definiciones dirigidas por la sintaxis con atributos heredados por la izquierda y el análisis sintáctico LL

Suponga que una definición dirigida por la sintaxis con atributos heredados por la izquierda está basada en una gramática LL, y que la hemos convertida en un esquema de traducción orientado a la sintaxis con acciones incrustadas en las producciones, como se describe en la sección 5.4.5. De esta forma, podemos realizar la traducción durante el análisis sintáctico LL, extendiendo la pila del analizador sintáctico para guardar las acciones y ciertos elementos de datos necesarios para la evaluación de los atributos. Por lo general, los elementos de datos son copias de los atributos.

Además de los registros que representan terminales y no terminales, la pila del analizador sintáctico contiene *registros de acción*, que representan las acciones que van a ejecutarse, y *registros de sintetizado* para guardar los atributos sintetizados para los no terminales. Utilizamos los siguientes dos principios para manejar los atributos en la pila:

- Los atributos heredados de una no terminal A se colocan en el registro de la pila que representa a ese no terminal. El código para evaluar esos atributos por lo general se representa mediante un registro de acción, justo encima del registro de pila para A ; de hecho, la conversión de las definiciones dirigidas por la sintaxis con atributos heredados por la izquierda a esquemas de traducción orientados a la sintaxis asegura que el registro de acción se encuentre de inmediato encima de A .
- Los atributos sintetizados para un no terminal A se colocan en un registro de sintetizado separado, el cual se encuentra justo debajo del registro para A en la pila.

Esta estrategia coloca registros de varios tipos en la pila de análisis sintáctico, confiando en que estos tipos de registros variantes pueden manejarse en forma apropiada como subclases de una clase “registro de pila”. En la práctica podríamos combinar varios registros en uno, pero tal vez las ideas se expliquen mejor al separar los datos utilizados para distintos fines en distintos registros.

Los registros de acción contienen apuntadores al código que se va a ejecutar. Las acciones también pueden aparecer en los registros sintetizados; estas acciones por lo general colocan copias del (los) atributo(s) sintetizado(s) en otros registros más abajo en la pila, en donde el

valor de ese atributo se requerirá una vez que se saquen de la pila el registro sintetizado y sus atributos.

Vamos a examinar brevemente el análisis sintáctico LL, para ver la necesidad de crear copias temporales de los atributos. En la sección 4.4.4 vimos que un analizador sintáctico LL controlado por una tabla imita a una derivación de más a la izquierda. Si w es la entrada que ha coincidido hasta ahora, entonces la pila contiene una secuencia de símbolos gramaticales α , de forma que $S \xrightarrow[ln]{*} w \alpha$, en donde S es el símbolo inicial. Cuando el analizador sintáctico se expande mediante una producción $A \rightarrow B C$, sustituye a A por $B C$ en el tope de la pila.

Suponga que el no terminal C tiene un atributo heredado $C.i$. Con $A \rightarrow B C$, el atributo heredado $C.i$ puede depender no sólo de los atributos heredados de A , sino de todos los atributos de B . Por ende, tal vez sea necesario procesar a B por completo, antes de poder evaluar a $C.i$. Entonces, guardamos copias temporales de todos los atributos necesarios para evaluar a $C.i$ en el registro de acción que evalúa a $C.i$. En cualquier otro caso, cuando el analizador sintáctico sustituye a A por $B C$ en la parte superior de la pila, los atributos heredados de A habrán desaparecido, junto con su registro de pila.

Como la definición dirigida por la sintaxis subyacente tiene atributos heredados por la izquierda, podemos estar seguros de que los valores de los atributos heredados de A están disponibles cuando A pasa a la parte superior de la pila. Por ende, los valores estarán disponibles a tiempo para copiarlos en el registro de acción que evalúa los atributos heredados de C . Además, el espacio para los atributos sintetizados de A no es un problema, ya que el espacio está en el registro de sintetizado para A , que permanece en la pila, debajo de B y C , cuando el analizador sintáctico expande mediante $A \rightarrow B C$.

A medida que se procesa B , podemos realizar acciones (a través de un registro justo encima de B en la pila) para copiar sus atributos heredados de manera que los utilice C según sea necesario, y después de procesar a B el registro de sintetizado para B puede copiar sus atributos sintetizados para que los use C , en caso de ser necesario. De igual forma, tal vez los atributos sintetizados de A necesiten valores temporales para ayudar a calcular su valor, y éstos pueden copiarse al registro de sintetizado para A , a medida que se procesa B y después C . El principio que hace que funcione todo este proceso de copiado de atributos es el siguiente:

- Todo el proceso de copiado se lleva a cabo entre los registros que se crean durante una expansión de un no terminal. Así, cada uno de estos registros sabe qué tan abajo en la pila se encuentra cada uno de los demás registros, y puede escribir valores en los registros de abajo con seguridad.

El siguiente ejemplo ilustra la implementación de los atributos heredados durante el análisis sintáctico LL, al copiar con diligencia los valores de los atributos. Es posible usar métodos abreviados u optimizaciones, en especial con las reglas de copiado, que sólo copian el valor de un atributo hacia otro. Veremos los métodos abreviados hasta el ejemplo 5.24, el cual ilustra también los registros de sintetizado.

Ejemplo 5.23: Este ejemplo implementa el esquema de traducción orientado a la sintaxis de la figura 5.32, que genera código al instante para la producción `while`. Este esquema de traducción orientado a la sintaxis no tiene atributos sintetizados, excepto los atributos falsos que representan las etiquetas.

La figura 5.33(a) muestra cuando estamos a punto de usar la producción `while` para expandir S , supuestamente debido a que el símbolo de lectura adelantada en la entrada es **while**.

El registro en la parte superior de la pila es para S , y contiene sólo el atributo heredado $S.sigüiente$, el cual suponemos tiene el valor x . Como ahora vamos a realizar el análisis sintáctico de arriba-abajo, mostramos el tope de la pila a la izquierda, de acuerdo con nuestra convención usual.

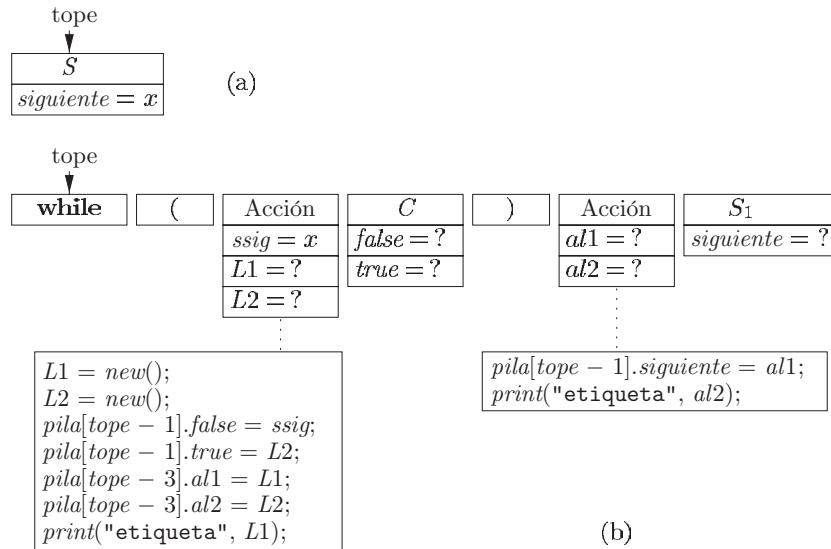
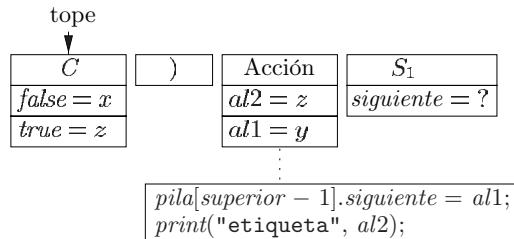


Figura 5.33: Expansión de S , de acuerdo a la producción de la instrucción while

La figura 5.33(b) muestra justo después de haber expandido S . Hay registros de acción en frente de los no terminales C y S_1 , que corresponden a las acciones en el esquema de traducción orientado a la sintaxis subyacente de la figura 5.32. El registro para C tiene espacio para los atributos heredados $true$ y $false$, mientras que el registro para S_1 tiene espacio para el atributo $sigüiente$, como lo deben tener todos los registros S . Mostramos los valores para estos campos como ?, ya que todavía no conocemos sus valores.

A continuación, el analizador sintáctico reconoce los símbolos **while** y (en la entrada, y saca sus registros de la pila. Ahora, la primera acción está en el topo, y debe ejecutarse. Este registro de acción tiene un campo llamado $ssig$, el cual contiene una copia del atributo heredado $S.sigüiente$. Cuando se saca S de la pila, el valor de $S.sigüiente$ se copia al campo $ssig$ para usarlo durante la evaluación de los atributos heredados para C . El código para la primera acción genera nuevos valores para $L1$ y $L2$, que debemos suponer son y y z , respectivamente. El siguiente paso es hacer que z sea el valor de $C.true$. La asignación $pila[tope - 1].true = L2$ se escribe sabiendo que sólo se ejecuta cuando este registro de acción se encuentra en el topo de la pila, por lo que $tope - 1$ se refiere al registro debajo de él (el registro para C).

Luego, el primer registro de acción copia a $L1$ en el campo $al1$ en la segunda acción, en donde se utilizará para evaluar a $S_1.sigüiente$. También copia a $L2$ en un campo llamado $al2$ de la segunda acción; este valor se requiere para que ese registro de acción imprima su salida en forma apropiada. Por último, el primer registro de acción imprime **etiqueta** y en la salida.

Figura 5.34: Después de realizar la acción encima de C

La situación después de completar la primera acción y sacar su registro de la pila se muestra en la figura 5.34. Los valores de los atributos heredados en el registro para C se han llenado en forma apropiada, al igual que los valores temporales $al1$ y $al2$ en el segundo registro de acción. En este punto se expande C y suponemos que se genera el código para implementar su prueba que contiene saltos a las etiquetas x y z , según lo apropiado. Cuando el registro C se saca de la pila, el registro para $)$ se convierte en el tope y hace que el analizador sintáctico compruebe si hay un $)$ en su entrada.

Con la acción encima de S_1 en el tope de la pila, su código establece $S_1.sigüiente$ y emite **etiqueta** z . Cuando termina con eso, el registro para S_1 se convierte en el tope de la pila, y a medida que se expande, suponemos que genera en forma correcta el código que implementa el tipo de instrucción que sea y después salta a la etiqueta y . \square

Ejemplo 5.24: Ahora vamos a considerar la misma instrucción `while`, pero con una traducción que produzca la salida $S.codigo$ como un atributo sintetizado, en vez de la generación instantánea. Para poder seguir la explicación, es útil tener en cuenta la siguiente hipótesis inductiva o invariante, que suponemos se sigue para cualquier no terminal:

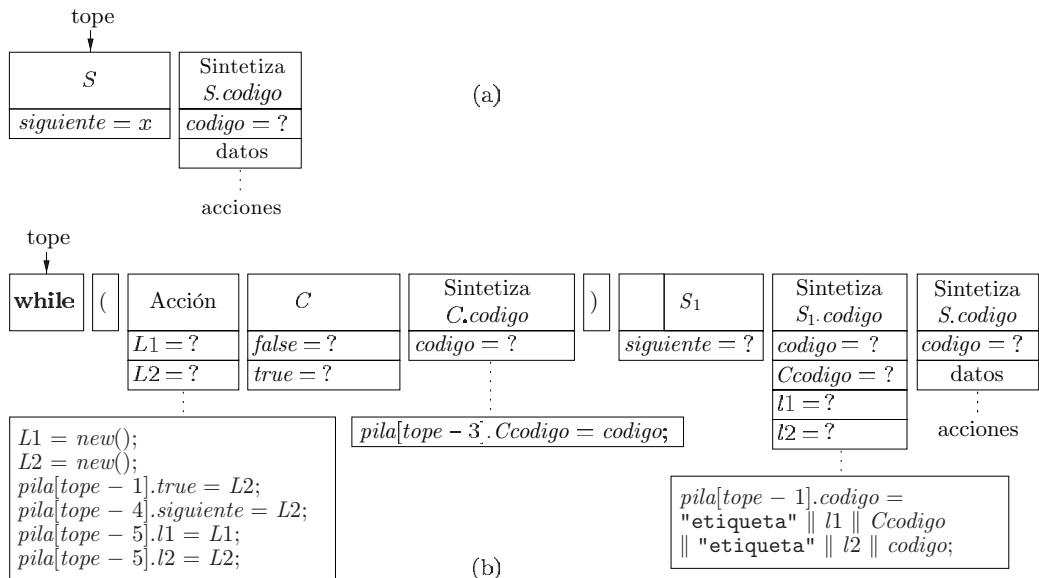
- Todo no terminal que tenga código asociado con el sale de ese código, en forma de cadena, en el registro de sintetizado que se encuentra justo debajo de ese no terminal en la pila.

Suponiendo que esta instrucción sea verdadera, vamos a manejar la producción `while` de manera que mantenga esta instrucción como una invariante.

La figura 5.35(a) muestra la situación justo antes de expandir S mediante el uso de la producción para las instrucciones `while`. En el tope de la pila podemos ver el registro para S ; tiene un campo para su atributo heredado $S.sigüiente$, como en el ejemplo 5.23. Justo debajo de ese registro se encuentra el registro de sintetizado para esta ocurrencia de S . Este registro tiene un campo para $S.codigo$, como todos los registros de sintetizado para S deben tener. También lo mostramos con otros campos para almacenamiento local y acciones, ya que el esquema de traducción orientado a la sintaxis para la producción `while` en la figura 5.28 sin duda forma parte de un esquema de traducción orientado a la sintaxis más grande.

Nuestra expansión de S se basa en el esquema de traducción orientado a la sintaxis de la figura 5.28, y se muestra en la figura 5.35(b). Como método abreviado, durante la expansión asumimos que el atributo heredado $S.sigüiente$ se asigna de manera directa a $C.false$, en vez de colocarse en la primera acción y después copiarse al registro para C .

Vamos a examinar lo que hace cada registro cuando se convierte en el tope de la pila. En primer lugar, el registro **while** hace que el token **while** se relacione con la entrada y se produzca

Figura 5.35: Expansión de *S* con el atributo sintetizado construido en la pila

una coincidencia, pues de lo contrario no habríamos expandido a *S* de esta manera. Después de sacar a **while** y (de la pila, se ejecuta el código para el registro de acción. Éste genera valores para *L1* y *L2*, y usamos el método abreviado de copiarlos directamente a los atributos heredados que los necesitan: *S₁.siguiente* y *C.true*. Los últimos dos pasos de la acción hacen que *L1* y *L2* se copien en el registro llamado “Sintetizar *S_{1.codigo}*”.

El registro de sintetizado para *S₁* realiza una doble función: no sólo contiene el atributo sintetizado *S_{1.codigo}*, sino que también sirve como registro de acción para completar la evaluación de los atributos para toda la producción $S \rightarrow \text{while } (C) S_1$. En especial, cuando llegue a la parte superior calculará el atributo sintetizado *S.codigo* y colocará su valor en el registro de sintetizado para el encabezado *S*.

Cuando *C* llega a la parte superior de la pila, se calculan sus dos atributos heredados. Mediante la hipótesis inductiva que mencionamos antes, suponemos que genera en forma correcta el código para ejecutar su condición y saltar a la etiqueta apropiada. También asumimos que las acciones realizadas durante la expansión de *C* colocan en forma correcta este código en el registro de abajo, como el valor del atributo sintetizado *C.codigo*.

Una vez que se saca *C*, el registro de sintetizado para *C.codigo* se convierte en la parte superior. Su código es necesario en el registro de sintetizado para *S_{1.codigo}*, debido a que ahí es en donde concatenamos todos los elementos de código para formar a *S.codigo*. Por lo tanto, el registro de sintetizado para *C.codigo* tiene una acción para copiar *C.codigo* en el registro de sintetizado para *S_{1.codigo}*. Después de hacerlo, el registro para el token) llega al topo de la pila y provoca que se compruebe si hay un) en la entrada. Si la prueba tiene éxito, el registro para *S₁* se convierte en el topo de la pila. Mediante nuestra hipótesis inductiva, este no terminal se expande y el efecto neto es que su código se construye en forma correcta, y se coloca en el campo para *codigo* en el registro de sintetizado para *S₁*.

¿Podemos manejar definiciones dirigidas por la sintaxis con atributos heredados por la izquierda en gramáticas LR?

En la sección 5.4.1 vimos que todos las definiciones dirigidas por la sintaxis con atributos sintetizados en una gramática LR se pueden implementar durante un análisis sintáctico ascendente. En la sección 5.5.3 vimos que todas las definiciones dirigidas por la sintaxis con atributos heredados por la izquierda en una gramática LL pueden analizarse de arriba hacia abajo. Como las gramáticas LL son un subconjunto propio de las gramáticas LR, y las definiciones dirigidas por la sintaxis con atributos sintetizados son un subconjunto propio de las definiciones dirigidas por la sintaxis con atributos heredados por la izquierda, ¿podemos manejar todas las gramáticas LR y las definiciones dirigidas por la sintaxis con atributos heredados por la izquierda de abajo hacia arriba?

No podemos, como lo demuestra el siguiente argumento intuitivo. Suponga que tenemos una producción $A \rightarrow B C$ en una gramática LR, y que hay un atributo $B.i$ heredado que depende de los atributos heredados de A . Al reducir a B , aún no hemos visto la entrada que genera C , por lo que no podemos estar seguros de tener un cuerpo de la producción $A \rightarrow B C$. Por ende, no podemos calcular $B.i$ todavía, ya que no sabemos si usar o no la regla asociada con esta producción.

Tal vez podríamos esperar hasta haber reducido a C , y sepamos que debemos reducir $B C$ a A . No obstante, incluso así no conocemos los atributos heredados de A , ya que después de cada reducción, tal vez no estemos seguros del cuerpo de la producción que contenga esta A . Podríamos razonar que esta decisión debería diferirse también y, por lo tanto, se diferiría aún más el cálculo de $B.i$. Si seguimos razonando de esta forma, pronto nos daremos cuenta de que no podemos tomar ninguna decisión sino hasta que se analice la entrada completa. En esencia, hemos llegado a la estrategia de “construir primero el árbol de análisis sintáctico y después realizar la traducción”.

Ahora, todos los campos de datos del registro sintetizados para S_1 se han llenado, por lo que cuando se convierta en el tope de la pila, podrá ejecutarse la acción en ese registro. La acción hace que las etiquetas y el código de $C.codigo$ y $S_1.codigo$ se concaténen en el orden apropiado. La cadena resultante se coloca en el registro de abajo; es decir, en el registro sintetizado para S . Ahora hemos calculado correctamente a $S.codigo$, y cuando el registro sintetizado para S se convierta en el tope, ese código estará disponible para colocarlo en otro registro más abajo en la pila, en donde en algún momento dado se ensamblará en una cadena de código más grande, para implementar el elemento de un programa del cual esta S forma parte. \square

5.5.4 Análisis sintáctico ascendente de las definiciones dirigidas por la sintaxis con atributos heredados por la izquierda

Podemos hacer cualquier traducción tanto de abajo hacia arriba, como de arriba hacia abajo. Dicho en forma más precisa, dada una definición dirigida por la sintaxis con atributos heredados por la izquierda en una gramática LL, podemos adaptar la gramática para calcular la misma definición dirigida por la sintaxis en la nueva gramática, durante un análisis sintáctico LR. El “truco” tiene tres partes:

1. Empezar con el esquema de traducción orientado a la sintaxis construido como en la sección 5.4.5, el cual coloca acciones incrustadas antes de cada no terminal, para calcular sus atributos heredados y una acción al final de la producción, para calcular los atributos sintetizados.
2. Introducir en la gramática un no terminal como marcador en vez de cada acción incrustada. Cada una de estas posiciones obtiene un marcador distinto, y hay una producción para cualquier marcador M , en específico, $M \rightarrow \epsilon$.
3. Modificar la acción a si el no terminal como marcador M la sustituye en alguna producción $A \rightarrow \alpha \{a\} \beta$, y asociar con $M \rightarrow \epsilon$ una acción a' que
 - (a) Copie, como atributo heredado de M , cualquier atributo de A o símbolo de α que la acción a requiera.
 - (b) Calcule los atributos de la misma forma que a , pero que los haga atributos sintetizados de M .

Este cambio parece ser ilegal, ya que por lo general la acción asociada con la producción $M \rightarrow \epsilon$ tendrá que acceder a los atributos que pertenezcan a los símbolos gramaticales que no aparezcan en esta producción. Sin embargo, vamos a implementar las acciones en la pila de análisis sintáctico LR, así que los atributos necesarios siempre estarán disponibles en un número conocido de posiciones hacia abajo en la pila.

Ejemplo 5.25: Suponga que hay una producción $A \rightarrow B C$ en una gramática LL, y que el atributo heredado $B.i$ se calcula a partir del atributo heredado $A.i$ mediante cierta fórmula $B.i = f(A.i)$. Es decir, el fragmento de un esquema de traducción orientado a la sintaxis que nos importa es:

$$A \rightarrow \{B.i = f(A.i);\} B C$$

Introducimos el marcador M con el atributo heredado $M.i$ y el atributo sintetizado $M.s$. El primero será una copia de $A.i$ y el segundo será $B.i$. El esquema de traducción orientado a la sintaxis se escribirá así:

$$\begin{aligned} A &\rightarrow M B C \\ A &\rightarrow \{M.i = A.i; M.s = f(M.i);\} \end{aligned}$$

Observe que la regla para M no tiene a $A.i$ disponible, pero de hecho vamos a arreglar que cada atributo heredado para un no terminal como A aparezca en la pila, justo debajo del lugar en el que se llevará a cabo la reducción a A . Así, cuando reduzcamos ϵ a M , encontraremos a $A.i$ justo debajo de ella, desde donde podrá leerse. Además, el valor de $M.s$, que se deja en la pila junto con M , es en realidad $B.i$ y se encuentra apropiadamente justo debajo del lugar en el que ocurrirá después la reducción a B . \square

Ejemplo 5.26: Vamos a convertir el esquema de traducción orientado a la sintaxis de la figura 5.28 en un esquema de traducción orientado a la sintaxis que pueda operar con un analizador sintáctico LR de la gramática modificada. Introduciremos un marcador M antes que C y un marcador N antes que S_1 , para que la gramática subyacente se convierta en lo siguiente:

Por qué funcionan los marcadores

Los marcadores son no terminales que derivan sólo a ϵ y que aparecen sólo una vez entre todos los cuerpos de todas las producciones. No proporcionaremos una prueba formal de que, cuando una gramática es LL, pueden agregarse no terminales como marcadores en cualquier posición en el cuerpo, y la gramática resultante seguirá siendo LR. Sin embargo, si una gramática es LL, entonces podemos determinar que una cadena w en la entrada se deriva del no terminal A , en una derivación que empieza con la producción $A \rightarrow \alpha$, con sólo ver el primer símbolo de w (o el siguiente símbolo, si $w = \epsilon$). Por ende, si analizamos a w de abajo hacia arriba, entonces el hecho de que un prefijo de w debe reducirse a α y después a S , se conoce tan pronto como aparece el principio de w en la entrada. En especial, si insertamos marcadores en cualquier parte de α , los estados LR incorporarán el hecho de que este marcador tiene que estar ahí, y reducirá ϵ al marcador en el punto apropiado en la entrada.

$$\begin{array}{l} S \rightarrow \mathbf{while} (M \, C) \, N \, S_1 \\ M \rightarrow \epsilon \\ N \rightarrow \epsilon \end{array}$$

Antes de hablar sobre las acciones asociadas con los marcadores M y N , vamos a describir la “hipótesis inductiva” de dónde se almacenan los atributos.

1. Debajo del cuerpo completo de la producción while (es decir, debajo de **while** en la pila) estará el atributo heredado $S.\text{siguiente}$. Tal vez no conozcamos el no terminal o el estado del analizador sintáctico asociado con este registro de pila, pero podemos estar seguros de que tendrá un campo, en una posición fija del registro, que contendrá a $S.\text{siguiente}$ antes de empezar a reconocer lo que se deriva de esta S .
2. Los atributos heredados $C.\text{true}$ y $C.\text{false}$ se encontrarán justo debajo del registro de la pila para C . Como se supone que la gramática es LL, la apariencia de **while** en la entrada nos asegura que la producción while sea la única que pueda reconocerse, por lo que podemos estar seguros de que M aparecerá justo debajo de C en la pila, y el registro de M contendrá los atributos heredados de C .
3. De manera similar, el atributo heredado $S_1.\text{siguiente}$ debe aparecer justo debajo de S_1 en la pila, para que podamos colocar ese atributo en el registro para N .
4. El atributo sintetizado $C.\text{codigo}$ aparecerá en el registro para C . Como siempre cuando tenemos una cadena extensa como valor de un atributo, esperamos que en la práctica aparezca un apuntador a (un objeto que representa) la cadena en el registro, mientras que la cadena en sí se encuentra fuera de la pila.
5. De manera similar, el atributo sintetizado $S_1.\text{codigo}$ aparecerá en el registro para S_1 .

Vamos a seguir el proceso de análisis sintáctico para una instrucción while. Suponga que un registro que contiene a $S.\text{siguiente}$ aparece en la parte superior de la pila, y que la siguiente

entrada es el terminal **while**. Desplazamos esta terminal hacia la pila. Después es evidente que la producción que se está reconociendo es while, por lo que el analizador sintáctico LR puede desplazar a "(" y determinar que su siguiente paso debe ser reducir ϵ a M . En la figura 5.36 se muestra la pila en ese momento. También mostramos en esa figura la acción que está asociada con la reducción a M . Creamos valores para $L1$ y $L2$, los cuales viven en campos del registro M , Además, en ese registro hay campos para $C.true$ y $C.false$. Estos atributos deben estar en los campos segundo y tercero del registro, para tener consistencia con otros registros de la pila que puedan aparecer debajo de C en otros contextos, y que también deban proporcionar estos atributos a C . La acción se completa asignando valores a $C.true$ y $C.false$, uno del valor $L2$ que se acaba de generar, y el otro recorriendo la pila hasta la posición en la que sabemos se encuentra $S.sigiente$.

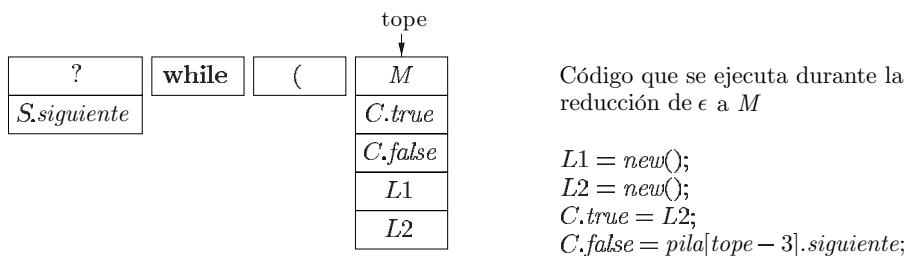


Figura 5.36: La pila de análisis sintáctico LR, después de reducir ϵ a M

Suponemos que las siguientes entradas se reducen a C en forma apropiada. Por lo tanto, el atributo sintetizado $C.codigo$ se coloca en el registro para C . Este cambio a la pila se muestra en la figura 5.37, que también incorpora los siguientes registros que se colocan posteriormente por encima de C en la pila.

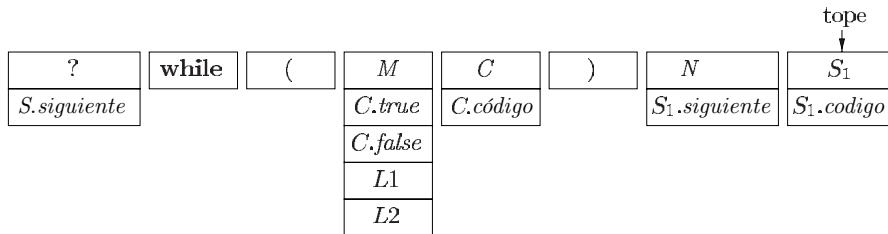


Figura 5.37: La pila, justo antes de la reducción del cuerpo de la producción while a S

Continuando con el reconocimiento de la instrucción while, el analizador sintáctico debe encontrar ahora un ")" en la entrada, al cual mete en la pila, en un registro propio. En ese punto, el analizador, que sabe está trabajando en una instrucción while, porque la gramática es LL, reducirá ϵ a N . La pieza individual de datos asociada con N es el atributo heredado

$S_1.\text{siguiente}$. Observe que este atributo necesita estar en el registro para N , ya que estará justo debajo del registro para S_1 . El código que se ejecuta para calcular el valor de $S_1.\text{siguiente}$ es:

$$S_1.\text{siguiente} = \text{pila}[\text{tope} - 3].L1;$$

Esta acción llega tres registros debajo de N , que se encuentra en el tope de la pila cuando se ejecuta el código, y obtiene el valor de $L1$.

A continuación, el analizador sintáctico reduce cierto prefijo de la entrada restante a S , a la cual nos hemos referido en forma consistente como S_1 , para diferenciarla de la S en el encabezado de la producción. El valor de $S_1.\text{codigo}$ se calcula y aparece en el registro de pila para S_1 . Este paso nos lleva a la condición que se ilustra en la figura 5.37.

En este punto, el analizador reconocerá todo, desde **while** hasta S_1 , y hasta S . El código que se ejecuta durante esta reducción es:

```

codigoTemp = etiqueta || pila[tope - 4].L1 || pila[tope - 3].codigo ||
            etiqueta || pila[tope - 4].L2 || pila[tope].codigo;
tope = tope - 5;
pila[tope].codigo = codigoTemp;

```

Esto es, construimos el valor de $S.\text{codigo}$ en una variable llamada codigoTemp . Ese código es el usual, que consiste en las dos etiquetas $L1$ y $L2$, el código para C y el código para S_1 . Se saca de la pila, para que aparezca S en donde se encontraba **while**. El valor del código para S se coloca en el campo codigo de ese registro, en donde puede interpretarse como el atributo sintetizado $S.\text{codigo}$. Observe que no mostramos, en ninguna parte de esta explicación, la manipulación de los estados LR, que también deben aparecer en la pila, en el campo que hemos llenado con símbolos gramaticales. \square

5.5.5 Ejercicios para la sección 5.5

Ejercicio 5.5.1: Implemente cada una de sus definiciones dirigidas por la sintaxis del ejercicio 5.4.4 como un analizador sintáctico de descenso recursivo, como en la sección 5.5.1.

Ejercicio 5.5.2: Implemente cada una de sus definiciones dirigidas por la sintaxis del ejercicio 5.4.4 como un analizador sintáctico de descenso recursivo, como en la sección 5.5.2.

Ejercicio 5.5.3: Implemente cada una de sus definiciones dirigidas por la sintaxis del ejercicio 5.4.4 como un analizador sintáctico LL, como en la sección 5.5.3, con el código generado “al instante”.

Ejercicio 5.5.4: Implemente cada una de sus definiciones dirigidas por la sintaxis del ejercicio 5.4.4 como un analizador sintáctico LL, como en la sección 5.5.3, pero con el código (o apun- tadores al código) almacenado en la pila.

Ejercicio 5.5.5: Implemente cada una de sus definiciones dirigidas por la sintaxis del ejercicio 5.4.4 con un analizador sintáctico LR, como en la sección 5.5.4.

Ejercicio 5.5.6: Implemente su definición dirigida por la sintaxis del ejercicio 5.2.4 como en la sección 5.5.1. ¿Sería distinta una implementación como en la sección 5.5.2?

5.6 Resumen del capítulo 5

- ◆ *Atributos heredados y sintetizados:* Las definiciones dirigidas por la sintaxis pueden utilizar dos tipos de atributos. Un atributo sintetizado en el nodo de un árbol sintáctico se calcula a partir de los atributos en sus hijos. Un atributo heredado en un nodo se calcula a partir de los atributos en su padre y/o hermanos.
- ◆ *Grafos de dependencias:* Dado un árbol de análisis sintáctico y una definición dirigida por la sintaxis, dibujamos flechas entre las instancias de los atributos asociados con cada nodo del árbol sintáctico, para denotar que el valor del atributo en el encabezado de la flecha se calcula en términos del valor del atributo en la parte final de la flecha.
- ◆ *Definiciones cíclicas:* En las definiciones dirigidas por la sintaxis problemáticas, encontramos que hay algunos árboles de análisis sintáctico para los cuales es imposible encontrar un orden en el que podamos calcular todos los atributos en todos los nodos. Estos árboles tienen ciclos en sus grafos de dependencias asociados. Es indecidible decidir si una definición dirigida por la sintaxis tiene dichos grafos de dependencias circular.
- ◆ *Definiciones con atributos sintetizados:* En una definición dirigida por la sintaxis con atributos sintetizados, todos los atributos sintetizados son sintetizados.
- ◆ *Definiciones con atributos heredados por la izquierda:* En una definición dirigida por la sintaxis con atributos heredados por la izquierda, los atributos pueden ser heredados o sintetizados. Sin embargo, los atributos heredados en el nodo de un árbol sintáctico pueden depender sólo de los atributos heredados de su padre y de (cualquiera de) los atributos de los hermanos a su izquierda.
- ◆ *Árboles sintácticos:* Cada nodo en un árbol sintáctico representa a una construcción; los hijos del nodo representan los componentes significativos de la construcción.
- ◆ *Implementación de una definición dirigida por la sintaxis con atributos sintetizados:* Una definición con atributos sintetizados puede implementarse mediante un esquema de traducción orientado a la sintaxis, en el que todas las acciones se encuentran al final de la producción (un esquema de traducción orientado a la sintaxis “postfijo”). Las acciones calculan los atributos sintetizados del encabezado de la producción, en términos de los atributos sintetizados de los símbolos en el cuerpo. Si la gramática subyacente es LR, entonces este esquema de traducción orientado a la sintaxis puede implementarse en la pila del analizador sintáctico LR.
- ◆ *Eliminación de la recursividad por la izquierda de los esquemas de traducción orientados a la sintaxis:* Si un esquema de traducción orientado a la sintaxis sólo tiene efectos adicionales (no se calculan atributos), entonces el algoritmo estándar para eliminar la recursividad por la izquierda para las gramáticas nos permitirá realizar las acciones como si fueran terminales. Cuando los atributos sintetizados se calculan, de todas formas podemos eliminar la recursividad por la izquierda, si el esquema de traducción orientado a la sintaxis es postfijo.
- ◆ *Implementación de definiciones dirigidas por la sintaxis con atributos heredados por la izquierda mediante el análisis sintáctico de descenso recursivo:* Si tenemos una definición

con atributos heredados por la izquierda en una gramática que pueda analizarse de arriba hacia abajo, podemos construir un analizador de descenso recursivo sin rastreo hacia atrás para implementar la traducción. Los atributos heredados se convierten en argumentos de las funciones para sus no terminales, y los atributos sintetizados se devuelven mediante esa función.

- ◆ *Implementación de definiciones dirigidas por la sintaxis con atributos heredados por la izquierda en una gramática LL:* Toda definición con atributos heredados por la izquierda y una gramática LL subyacente puede implementarse junto con el análisis sintáctico. Los registros para guardar los atributos sintetizados para una no terminal se colocan debajo de esa no terminal en la pila, mientras que los atributos no heredados para un no terminal se almacenan con ese no terminal en la pila. Los registros de acción también se colocan en la pila para calcular los atributos en el tiempo apropiado.
- ◆ *Implementación de definiciones dirigidas por la sintaxis con atributos heredados por la izquierda en una gramática LL, de abajo hacia arriba:* Una definición con atributos heredados por la izquierda y una gramática LL subyacente puede convertirse en una traducción en una gramática LR, y la traducción se realiza en conexión con un análisis sintáctico ascendente. La transformación gramatical introduce no terminales “marcadores” que aparecen en la pila del analizador sintáctico ascendente y contienen atributos heredados del no terminal encima de ella en la pila. Los atributos sintetizados se mantienen con su no terminal en la pila.

5.7 Referencias para el capítulo 5

Las definiciones dirigidas por la sintaxis son una forma de definición inductiva, en la cual la inducción es sobre la estructura sintáctica. Como tales, se han utilizado desde hace mucho tiempo de manera informal en las matemáticas. Su aplicación a los lenguajes de programación se dio con el uso de una gramática para estructurar el reporte de Algol 60.

La idea de un analizador sintáctico que llama a las acciones semánticas puede encontrarse en Samelson y Bauer [8], y en Brooker y Morris [1]. Irons [2] construyó uno de los primeros compiladores orientados a la sintaxis, usando atributos sintetizados. La clase de definiciones con atributos heredados por la izquierda proviene de [6].

Los atributos heredados, los grafos de dependencias, y una prueba de circularidad de las definiciones dirigidas por la sintaxis (es decir, si hay o no algún árbol sintáctico sin orden, en el que puedan calcularse los atributos) son de Knuth [5]. Jazayeri, Ogden y Rounds [3] mostraron que para probar la circularidad se requiere un tiempo exponencial, como una función del tamaño de la definición dirigida por la sintaxis.

Los generadores de analizadores sintácticos como Yacc [4] (vea también las notas bibliográficas en el capítulo 4) soportan la evaluación de atributos durante el análisis sintáctico.

La encuesta realizada por Paakki [7] es un punto inicial para acceder a la extensa literatura sobre las definiciones y traducciones orientadas a la sintaxis.

1. Brooker, R. A. y D. Morris, “A general translation program for phrase structure languages”, *J. ACM* 9:1 (1962), pp. 1-10.

2. Irons, E. T., “A syntax directed compiler for Algol 60”, *Comm. ACM* **4**:1 (1961), pp. 51-55.
3. Jazayeri, M., W. F. Odgen y W. C. Rounds, “The intrinsic exponential complexity of the circularity problem for attribute grammars”, *Comm. ACM* **18**:12 (1975), pp. 697-706.
4. Johnson, S. C., “Yacc – Yet Another Compiler Compiler”, Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, 1975. Disponible en <http://dinousaur.compilertools.net/yacc/>.
5. Knuth, D. E., “Semantics of context-free languages”, *Mathematical Systems Theory* **2**:2 (1968), pp. 127-145. Vea también *Mathematical Systems Theory* **5**:1 (1971), pp. 95-96.
6. Lewis, P. M. II, D. J. Rosenkrantz y R. E. Stearns, “Attributed translations”, *J. Computer and System Sciences* **9**:3 (1974), pp. 279-307.
7. Paakkki, J., “Attribute grammar paradigms – a high-level methodology in language implementation”, *Computing Surveys* **27**:2 (1995), pp. 196-255.
8. Samelson, K. y F. L. Bauer, “Sequential formula translation”, *Comm. ACM* **3**:2 (1960), pp. 76-83.

Capítulo 6

Generación de código intermedio

En el modelo de análisis y síntesis de un compilador, el front-end analiza un programa fuente y crea una representación intermedia, a partir de la cual el back-end genera el código destino. Lo apropiado es que los detalles del lenguaje fuente se confinen al front-end, y los detalles de la máquina de destino al respaldo. Con una representación intermedia definida de manera adecuada, podemos construir un compilador para el lenguaje *i* y la máquina *j* mediante la combinación del front-end para el lenguaje *i* y el back-end para la máquina *j*. Este método para crear una suite de compiladores puede ahorrar una considerable cantidad de esfuerzo: podemos construir $m \times n$ compiladores con sólo escribir m front-ends y n back-ends.

Este capítulo trata las representaciones intermedias, la comprobación estática de tipos y la generación de código intermedio. Por cuestión de simplicidad, asumimos que el front-end de un compilador se organiza como en la figura 6.1, en donde el análisis sintáctico, la comprobación estática y la generación de código intermedio se realizan en forma secuencial; algunas veces pueden combinarse y mezclarse en el análisis sintáctico. Utilizaremos los formalismos orientados a la sintaxis de los capítulos 2 y 5 para especificar la comprobación y la traducción. Muchos de los esquemas de traducción pueden implementarse durante el análisis sintáctico de abajo-arriba o arriba-abajo, usando las técnicas del capítulo 5. Todos los esquemas pueden implementarse mediante la creación de un árbol sintáctico y después el recorrido de éste.

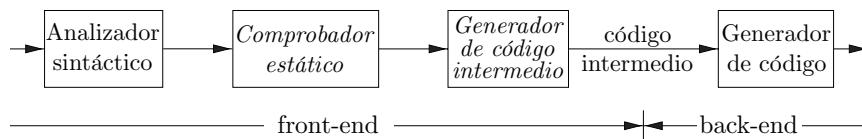


Figura 6.1: Estructura lógica del front-end de un compilador

La comprobación estática incluye la *comprobación de tipos*, la cual asegura que los operadores se apliquen a los operandos compatibles. También incluye cualquier comprobación sintáctica

que resulte después del análisis sintáctico. Por ejemplo, la comprobación estática asegura que una instrucción `break` en C esté encerrada dentro de una instrucción `while`, `for` o `switch`; se reporta un error si no existe dicha instrucción envolvente.

El método en este capítulo puede usarse para una gran variedad de representaciones intermedias, incluyendo los árboles sintácticos y el código de tres direcciones, los cuales se presentaron en la sección 2.8. El término “código de tres direcciones” proviene de las instrucciones de la forma general $x = y \ op \ z$ con tres direcciones: dos para los operandos y y z , y una para el resultado x .

Durante el proceso de traducir un programa en un lenguaje fuente dado al código para una máquina destino, un compilador puede construir una secuencia de representaciones intermedias, como en la figura 6.2. Las representaciones de alto nivel están cerca del lenguaje fuente y las representaciones de bajo nivel están cerca de la máquina destino. Los árboles sintácticos son de alto nivel; describen la estructura jerárquica natural del programa fuente y se adaptan bien a tareas como la comprobación estática de tipos.



Figura 6.2: Un compilador podría usar una secuencia de representaciones intermedias

Una representación de bajo nivel es adecuada para las tareas dependientes de la máquina, como la asignación de registros y la selección de instrucciones. El código de tres direcciones puede variar de alto a bajo nivel, dependiendo de la elección de operadores. Para las expresiones, las diferencias entre los árboles sintácticos y el código de tres direcciones es superficial, como veremos en la sección 6.2.3. Por ejemplo, para las instrucciones de ciclos un árbol sintáctico representa a los componentes de una instrucción, mientras que el código de tres direcciones contiene etiquetas e instrucciones de salto para representar el flujo de control, como en el lenguaje máquina.

La elección o diseño de una representación intermedia varía de un compilador a otro. Una representación intermedia puede ser un verdadero lenguaje, o puede consistir en estructuras de datos internas que se comparten mediante las fases del compilador. C es un lenguaje de programación, y aún así se utiliza con frecuencia como forma intermedia, ya que es flexible, se compila en código máquina eficiente y existe una gran variedad de compiladores. El compilador de C++ original consistía en una front-end que generaba C, y trataba a un compilador de C como back-end.

6.1 Variantes de los árboles sintácticos

Los nodos en un árbol sintáctico representan construcciones en el programa fuente; los hijos de un nodo representan los componentes significativos de una construcción. Un grafo acíclico dirigido (de aquí en adelante lo llamaremos *GDA*) para una expresión identifica a las *subexpresiones comunes* (subexpresiones que ocurren más de una vez) de la expresión. Como veremos en esta sección, pueden construirse GDAs mediante el uso de las mismas técnicas que construyen los árboles sintácticos.

6.1.1 Grafo dirigido acíclico para las expresiones

Al igual que el árbol sintáctico para una expresión, un GDA tiene hojas que corresponden a los operandos atómicos, y códigos interiores que corresponden a los operadores. La diferencia es que un nodo N en un GDA tiene más de un parente si N representa a una subexpresión común; en un árbol sintáctico, el árbol para la subexpresión común se replica tantas veces como aparezca la subexpresión en la expresión original. Por ende, un GDA no solo representa a las expresiones en forma más breve, sino que también proporciona pistas importantes al compilador, en la generación de código eficiente para evaluar las expresiones.

Ejemplo 6.1: La figura 6.3 muestra el GDA para la siguiente expresión:

$$a + a * (b - c) + (b - c) * d$$

La hoja para a tiene dos padres, ya que a aparece dos veces en la expresión. Lo más interesante es que dos ocurrencias de la subexpresión común $b - c$ se representan mediante un nodo, etiquetado como $-$. Ese nodo tiene dos padres, los cuales representan sus dos usos en las subexpresiones $a * (b - c)$ y $(b - c) * d$. Aun cuando b y c aparecen dos veces en la expresión completa, cada uno de sus nodos tiene un parente, ya que ambos usos se encuentran en la subexpresión común $b - c$. \square

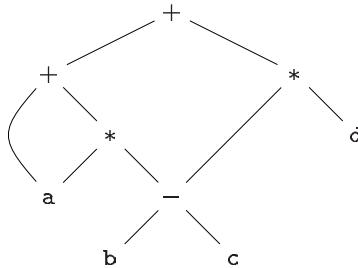


Figura 6.3: GDA para la expresión $a + a * (b - c) + (b - c) * d$

Las definiciones dirigidas por la sintaxis de la figura 6.4 puede construir árboles sintácticos o GDAs. Se utilizó para construir árboles sintácticos en el ejemplo 5.11, en donde las funciones *Hoja* y *Nodo* creaban un nodo nuevo cada vez que se les llamaba. Construirá un GDA si, antes de crear un nuevo nodo, estas funciones primero comprueban que ya existe un nodo idéntico. En caso de ser así, se devuelve el nodo existente. Por ejemplo, antes de construir un nuevo nodo, *Nodo(op, izq, der)*, comprobamos si ya hay un nodo con la etiqueta *op*, y los hijos *izq* y *der*, en ese orden. De ser así, *Nodo* devuelve el nodo existente; en caso contrario, crea uno nuevo.

Ejemplo 6.2: La secuencia de pasos que se muestra en la figura 6.5 construye el GDA de la figura 6.3, siempre y cuando *Nodo* y *Hoja* devuelvan un nodo existente, si es posible, como

PRODUCCIÓN	REGLAS SEMÁNTICAS
1) $E \rightarrow E_1 + T$	$E.\text{nodo} = \mathbf{new} \text{ Nodo}(' + ', E_1.\text{nodo}, T.\text{nodo})$
2) $E \rightarrow E_1 - T$	$E.\text{nodo} = \mathbf{new} \text{ Nodo}(' - ', E_1.\text{nodo}, T.\text{nodo})$
3) $E \rightarrow T$	$E.\text{nodo} = T.\text{nodo}$
4) $T \rightarrow (E)$	$T.\text{nodo} = E.\text{nodo}$
5) $T \rightarrow \text{id}$	$T.\text{nodo} = \mathbf{new} \text{ Hoja}(\text{id}, \text{id}.\text{entrada})$
6) $T \rightarrow \text{num}$	$T.\text{nodo} = \mathbf{new} \text{ Hoja}(\text{num}, \text{num}.\text{val})$

Figura 6.4: Definición orientada por la sintaxis para producir árboles sintácticos o GDAs

- 1) $p_1 = \text{Hoja}(\text{id}, \text{entrada-}a)$
- 2) $p_2 = \text{Hoja}(\text{id}, \text{entrada-}a) = p_1$
- 3) $p_3 = \text{Hoja}(\text{id}, \text{entrada-}b)$
- 4) $p_4 = \text{Hoja}(\text{id}, \text{entrada-}c)$
- 5) $p_5 = \text{Nodo}(' - ', p_3, p_4)$
- 6) $p_6 = \text{Nodo}(' * ', p_1, p_5)$
- 7) $p_7 = \text{Nodo}(' + ', p_1, p_6)$
- 8) $p_8 = \text{Hoja}(\text{id}, \text{entrada-}b) = p_3$
- 9) $p_9 = \text{Hoja}(\text{id}, \text{entrada-}c) = p_4$
- 10) $p_{10} = \text{Nodo}(' - ', p_3, p_4) = p_5$
- 11) $p_{11} = \text{Hoja}(\text{id}, \text{entrada-}d)$
- 12) $p_{12} = \text{Nodo}(' * ', p_5, p_{11})$
- 13) $p_{13} = \text{Nodo}(' + ', p_7, p_{12})$

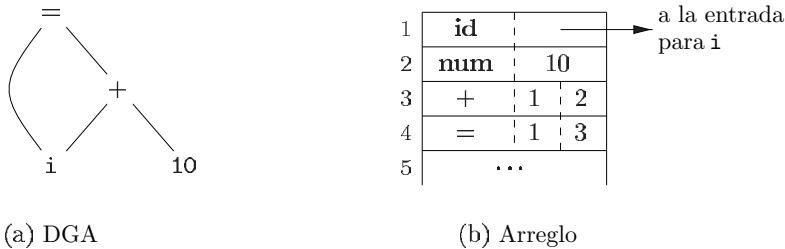
Figura 6.5: Pasos para construir el GDA de la figura 6.3

dijimos antes. Asumimos que *entrada-a* apunta a la entrada en la tabla de símbolos para **a**, y de manera similar para los demás identificadores.

Cuando la llamada a *Hoja* (**id**, *entrada-a*) se repite en el paso 2, se devuelve el nodo creado por la llamada anterior, por lo que $p_2 = p_1$. De manera similar, los nodos devueltos en los pasos 8 y 9 son iguales que los devueltos en los pasos 3 y 4 (es decir, $p_8 = p_3$ y $p_9 = p_4$). Por ende, el nodo devuelto en el paso 10 debe ser igual que el devuelto en el paso 5; es decir, $p_{10} = p_5$. \square

6.1.2 El método número de valor para construir GDAs

A menudo, los nodos de un árbol sintáctico o GDA se almacenan en un arreglo de registros, como lo sugiere la figura 6.6. Cada fila del arreglo representa a un registro y, por lo tanto, a un nodo. En cada registro, el primer campo es un código de operación, el cual indica la etiqueta del nodo. En la figura 6.6(b) las hojas tienen un campo adicional, el cual contiene el valor léxico (ya sea un apuntador a una tabla de símbolos o una constante, en este caso), y los nodos interiores tienen dos campos adicionales que indican a los hijos izquierdo y derecho.

Figura 6.6: Nodos de un GDA para $i = i + 10$, asignados en un arreglo

En este arreglo, para referirnos a los nodos proporcionamos el índice entero del registro para ese nodo dentro del arreglo. Con el tiempo, a este entero se le ha denominado el *número de valor* para el nodo, o para la expresión que éste representa. Por ejemplo, en la figura 6.6 el nodo etiquetado como $+$ tiene el número de valor 3, y sus hijos izquierdo y derecho tienen los números de valor 1 y 2, respectivamente. En la práctica, podríamos utilizar apuntadores a registros o referencias a objetos en vez de índices enteros, pero de todas formas nos referiremos a la referencia de un nodo como su “número de valor”. Si se almacenan en una estructura de datos apropiada, los números de valor nos ayudan a construir los GDAs de expresiones con eficiencia: el siguiente algoritmo muestra cómo.

Suponga que los nodos están almacenados en un arreglo, como en la figura 6.6, y que se hace referencia a cada nodo por su número de valor. Hagamos que la *firma* de un nodo interior sea el triple $\langle op, i, d \rangle$, en donde op es la etiqueta, i el número de valor de su hijo izquierdo y d el número de valor de su hijo derecho. Podemos suponer que un operador unario tiene $r = 0$.

Algoritmo 6.3: El método número de valor para construir los nodos de un GDA.

ENTRADA: Etiqueta op , nodo i y nodo d .

SALIDA: El número de valor de un nodo en el arreglo con la firma $\langle op, i, d \rangle$.

MÉTODO: Buscar en el arreglo un nodo M con la etiqueta op , el hijo izquierdo i y el hijo derecho r . Si hay un nodo así, devolver el número de valor M . Si no, crear un nuevo nodo N en el arreglo, con la etiqueta op , el hijo izquierdo i y el hijo derecho d , y devolver su número de valor. \square

Aunque el Algoritmo 6.3 produce la salida deseada, se requiere mucho tiempo para buscar en todo el archivo cada vez que necesitamos localizar un nodo, en especial si el arreglo contiene expresiones de todo un programa. Un método más eficiente es usar una tabla hash, en la cual los nodos se colocan en “baldes”, cada uno de los cuales tendrá, por lo general, sólo unos cuantos nodos. La tabla hash es una de varias estructuras de datos que soportan los *diccionarios* en forma eficiente.¹ Un diccionario es un tipo abstracto de datos que nos permite insertar y eliminar elementos de un conjunto, y determinar si un elemento dado se encuentra

¹Vea Aho, A. V., J. E. Hopcroft y J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983, para una explicación sobre las estructuras de datos con soporte para diccionarios.

en el conjunto. Una buena estructura de datos para diccionarios, como una tabla hash, realiza cada una de estas operaciones en un tiempo constante o casi constante, sin importar el tamaño del conjunto.

Para construir una tabla hash para los nodos de un GDA, necesitamos una *función hash* h para calcular el índice del balde para una firma $\langle op, i, d \rangle$, de tal forma que distribuya las firmas entre los baldes, para que sea poco probable que un balde obtenga mucho más de la porción justa de nodos. El índice de balde $h(op, i, d)$ se calcula en forma determinista a partir de op , i y r , de manera que podemos repetir el cálculo y siempre llegaremos al mismo índice de balde para el nodo $\langle op, i, d \rangle$.

Los baldes pueden implementarse como listas enlazadas, como en la figura 6.7. Un arreglo, indexado por un valor hash, contiene los *encabezados de baldes*, cada uno de los cuales apunta a la primera celda de una lista. Dentro de la lista enlazada para un balde, cada celda contiene el número de valor de uno de los nodos que se asignan a ese balde. Es decir, el nodo $\langle op, i, d \rangle$ puede encontrarse en la lista cuyo encabezado se encuentra en el índice $h(op, i, d)$ del arreglo.

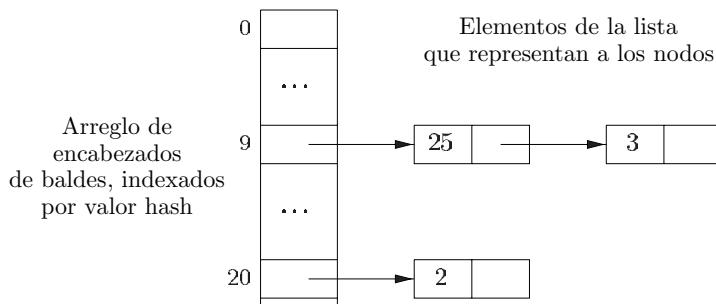


Figura 6.7: Estructura de datos para buscar en los baldes

Por lo tanto, dado el nodo de entrada op , i y d , calculamos el índice de balde $h(op, i, d)$ y buscamos el nodo de entrada dado en la lista de celdas en este balde. Por lo general, hay suficientes baldes de manera que no haya una lista con más de unas cuantas celdas. Sin embargo, tal vez tengamos que buscar en todas las celdas dentro de un balde, y para cada número de valor v que encontremos en una celda, debemos comprobar si la firma $\langle op, i, d \rangle$ del nodo de entrada coincide con el nodo que tiene el número de valor v en la lista de celdas (como en la figura 6.7). Si encontramos una coincidencia, devolvemos v . Si no la encontramos, sabemos que no puede existir un nodo de ese tipo en ningún otro balde, por lo que creamos una nueva celda, la agregamos a la lista de celdas para el índice de balde $h(op, i, d)$, y devolvemos el número de valor en esa nueva celda.

6.1.3 Ejercicios para la sección 6.1

Ejercicio 6.1.1: Construya el GDA para la siguiente expresión:

$$((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))$$

Ejercicio 6.1.2: Construya el GDA e identifique los números de valores para las subexpresiones de las siguientes expresiones, suponiendo que + asocia por la izquierda.

- a) $a + b + (a + b)$.
- b) $a + b + a + b$.
- c) $a + a + ((a + a + a + (a + a + a + a)))$.

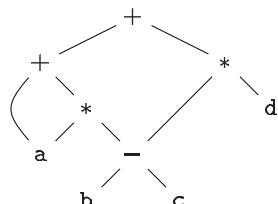
6.2 Código de tres direcciones

En el código de tres direcciones, hay máximo un operador en el lado derecho de una instrucción; es decir, no se permiten expresiones aritméticas acumuladas. Por ende, una expresión del lenguaje fuente como $x+y*z$ podría traducirse en la siguiente secuencia de instrucciones de tres direcciones:

$$\begin{aligned} t_1 &= y * z \\ t_2 &= x + t_1 \end{aligned}$$

en donde t_1 y t_2 son nombres temporales que genera el compilador. Este desenmarañamiento de expresiones aritméticas con varios operadores y de instrucciones de flujo de control anidadas hace que el código de tres direcciones sea conveniente para la generación y optimización de código destino, como veremos en los capítulos 8 y 9. El uso de nombres para los valores intermedios calculados por un programa permite reordenar el código de tres direcciones con facilidad.

Ejemplo 6.4: El código de tres direcciones es una representación lineal de un árbol sintáctico o de un GDA, en el cual los nombres explícitos corresponden a los nodos interiores del grafo. El GDA en la figura 6.3 se repite en la figura 6.8, junto con la secuencia de código de tres direcciones correspondiente. \square



(a) DAG

$$\begin{aligned} t_1 &= b - c \\ t_2 &= a * t_1 \\ t_3 &= a + t_2 \\ t_4 &= t_1 * d \\ t_5 &= t_3 + t_4 \end{aligned}$$

(b) Código de tres direcciones

Figura 6.8: Un GDA y su código correspondiente de tres direcciones

6.2.1 Direcciones e instrucciones

El código de tres direcciones se basa en dos conceptos: direcciones e instrucciones. En términos de orientación a objetos, estos conceptos corresponden a las clases y los diversos tipos de direcciones e instrucciones corresponden a subclases apropiadas. De manera alternativa, podemos implementar el código de tres direcciones usando registros con campos para las direcciones; en la sección 6.2.2 hablaremos sobre los registros, conocidos como cuádruplos y tripletas.

Una dirección puede ser una de las siguientes opciones:

- *Un nombre.* Por conveniencia, permitimos que los nombres de los programas fuente aparezcan como direcciones en código de tres direcciones. En una implementación, un nombre de origen se sustituye por un apuntador a su entrada en la tabla de símbolos, en donde se mantiene toda la información acerca del nombre.
- *Una constante.* En la práctica, un compilador debe tratar con muchos tipos distintos de constantes y variables. En la sección 6.5.2 veremos las conversiones de tipos dentro de las expresiones.
- *Un valor temporal generado por el compilador.* Es conveniente, en especial con los compiladores de optimización, crear un nombre distinto cada vez que se necesita un valor temporal. Estos valores temporales pueden combinarse, si es posible, cuando se asignan los registros a las variables.

Ahora consideraremos las tres instrucciones de tres direcciones comunes que utilizaremos en el resto de este libro. Las etiquetas simbólicas son para uso de las instrucciones que alteran el flujo de control. Una etiqueta simbólica representa el índice de una instrucción de tres direcciones en la secuencia de instrucciones. Los índices reales pueden sustituirse por las etiquetas, ya sea mediante una pasada separada o mediante la técnica de “parcheo de retroceso”, que veremos en la sección 6.7. He aquí una lista de las formas comunes de instrucciones de tres direcciones:

1. Instrucciones de asignación de la forma $x = y \ op \ z$, en donde op es una operación aritmética o lógica binaria, y x , y y z son direcciones.
2. Asignaciones de la forma $x = op \ y$, en donde op es una operación unaria. En esencia, las operaciones unarias incluyen la resta unaria, la negación lógica, los operadores de desplazamiento y los operadores de conversión que, por ejemplo, convierten un entero en un número de punto flotante.
3. *Instrucciones de copia* de la forma $x = y$, en donde a x se le asigna el valor de y .
4. Un salto incondicional *goto L*. La instrucción de tres direcciones con la etiqueta L es la siguiente que se va a ejecutar.
5. Saltos condicionales de la forma *if x goto L* e *ifFalse x goto L*. Estas instrucciones ejecutan a continuación la instrucción con la etiqueta L si x es verdadera y falsa, respectivamente. En cualquier otro caso, la siguiente instrucción en ejecutarse es la instrucción de tres direcciones que siga en la secuencia, como siempre.

6. Saltos condicionales como `if x relop y goto L`, que aplican un operador relacional ($<$, $==$, \geq , etc.) a x y y , y ejecutan a continuación la instrucción con la etiqueta L si x predomina en la relación $relop$ con y . Si no es así, se ejecuta a continuación la siguiente instrucción de tres direcciones que vaya después de `if x relop y goto L`, en secuencia.
7. Las llamadas a los procedimientos y los retornos se implementan mediante el uso de las siguientes instrucciones: `param x` para los parámetros; `call p, n` y `y = call p, n` para las llamadas a procedimientos y funciones, respectivamente; y `return y`, en donde y , que representa a un valor de retorno, es opcional. Su uso común es como la siguiente secuencia de instrucciones de tres direcciones:

```

param x1
param x2
...
param xn
call p, n

```

que se generan como parte de una llamada al procedimiento $p(x_1, x_2, \dots, x_n)$. El entero n , que indica el número de parámetros actuales en “`call p, n`” no es redundante, ya que las llamadas pueden anidarse. Es decir, algunas de las primeras instrucciones `param` podrían ser parámetros de una llamada que se realice después de que p devuelva su valor; ese valor se convierte en otro parámetro de la llamada anterior. En la sección 6.9 se describe la implementación de llamadas a procedimientos.

8. Instrucciones de copia indexadas, de la forma $x = y[i]$ y $x[i] = y$. La instrucción $x = y[i]$ establece x al valor en la ubicación que se encuentra a i unidades de memoria más allá de y . La instrucción $x[i] = y$ establece el contenido de la ubicación que se encuentra a i unidades más allá de x , con el valor de y .
9. Asignaciones de direcciones y apuntadores de la forma $x = \&y$, $x = *y$ y $*x = y$. La instrucción $x = \&y$ establece el *r-value* de x para que sea la ubicación (*l-value*) de y .² Se supone que y es un nombre, tal vez temporal, que denota a una expresión con un *l-value* tal como `A[i][j]`, y que x es el nombre de un apuntador o valor temporal. En la instrucción $x = *y$, se supone que y es un apuntador o un temporal cuyo *r-value* es una ubicación. El *r-value* de x se hace igual al contenido de esa ubicación. Por último, $*x = y$ establece el *r-value* del objeto al que apunta x con el *r-value* de y .

Ejemplo 6.5: Considere la siguiente instrucción:

```
do i = i+1; while (a[i] < v);
```

En la figura 6.9 se muestran dos posibles traducciones de esta instrucción. La traducción en la figura 6.9 utiliza una etiqueta simbólica L , que se adjunta a la primera instrucción. La traducción

²En la sección 2.8.3 vimos que los *l* y *r-value* son apropiados en los lados izquierdo y derecho de las asignaciones, respectivamente.

en (b) muestra números de posición para las instrucciones, empezando en forma arbitraria en la posición 100. En ambas traducciones, la última instrucción es un salto condicional a la primera instrucción. La multiplicación $i * 8$ es apropiada para un arreglo de elementos en el que cada uno de ellos ocupa 8 unidades de espacio. \square

L: $t_1 = i + 1$
 $i = t_1$
 $t_2 = i * 8$
 $t_3 = a [t_2]$
if $t_3 < v$ goto L

(a) Etiquetas simbólicas

100: $t_1 = i + 1$
101: $i = t_1$
102: $t_2 = i * 8$
103: $t_3 = a [t_2]$
104: if $t_3 < v$ goto 100

(b) Números de posición

Figura 6.9: Dos formas de asignar etiquetas a las instrucciones de tres direcciones

La elección de operadores permitidos es una cuestión importante en el diseño de una forma intermedia. Es evidente que el conjunto de operadores debe ser lo bastante amplio como para implementar las operaciones en el lenguaje fuente. Los operadores cercanos a las instrucciones de máquina facilitan la implementación de la forma intermedia en una máquina de destino. No obstante, si la front-end debe generar secuencias largas de instrucciones para algunas operaciones en lenguaje fuente, entonces el optimizador y el generador de código tal vez tengan que trabajar duro para redescubrir la estructura y generar buen código para estas operaciones.

6.2.2 Cuádruplos

La descripción de las instrucciones de tres direcciones especifica los componentes de cada tipo de instrucción, pero no especifica la representación de estas instrucciones en una estructura de datos. En un compilador, estas instrucciones pueden implementarse como objetos o como registros, con campos para el operador y los operandos. Tres de estas representaciones se conocen como “cuádruplos”, “tripletas” y “tripletas indirectas”.

Un *cuádruplo* tiene cuatro campos, a los cuales llamamos *op*, *arg₁*, *arg₂* y *resultado*. El campo *op* contiene un código interno para el operador. Por ejemplo, la instrucción de tres direcciones $x = y + z$ se representa colocando a $+$ en *op*, *y* en *arg₁*, *z* en *arg₂* y *x* en *resultado*. A continuación se muestran dos excepciones a esta regla:

1. Las instrucciones con operadores unarios como $x = \text{menos } y$ o $x = y$ no utilizan *arg₂*. Observe que para una instrucción de copia como $x = y$, *op* es $=$, mientras que para la mayoría de las otras operaciones, el operador de asignación es implícito.
2. Los operadores como *param* no utilizan *arg₂* ni *resultado*.
3. Los saltos condicionales e incondicionales colocan la etiqueta de destino en *resultado*.

Ejemplo 6.6: El código de tres direcciones para la asignación $a = b * - c + b * - c$; aparece en la figura 6.10(a). El operador especial *menos* se utiliza para distinguir al operador de resta

unario, como en $- c$, del operador de resta binario, como en $b - c$. Observe que la instrucción de “tres direcciones” de resta unaria sólo tiene dos direcciones, al igual que la instrucción de copia $a = t_5$.

Los cuádruplos en la figura 6.10(b) implementan el código de tres direcciones en (a). \square

$t_1 = \text{menos } c$	<i>op</i>	<i>arg₁</i>	<i>arg₂</i>	<i>resultado</i>
$t_2 = b * t_1$	0	menos	c	t_1
$t_3 = \text{menos } c$	1	*	b	t_1
$t_4 = b * t_3$	2	menos	c	t_3
$t_5 = t_2 + t_4$	3	*	b	t_3
$a = t_5$	4	+	t_2	t_4
	5	=	t_5	a
				...

(a) Código de tres direcciones

	<i>op</i>	<i>arg₁</i>	<i>arg₂</i>	<i>resultado</i>
0	menos	c		t_1
1	*	b	t_1	t_2
2	menos	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
				...

(b) Cuádruplos

Figura 6.10: Código de tres direcciones y su representación en cuádruplos

Por cuestión de legibilidad, usamos identificadores reales como a , b y c en los campos arg_1 , arg_2 y $resultado$ en la figura 6.10(b), en vez de apuntadores a sus entradas en la tabla de símbolos. Los nombres temporales pueden introducirse en la tabla de símbolos como nombres definidos por el programador, o pueden implementarse como objetos de una clase *Temp* con sus propios métodos.

6.2.3 Tripletas

Un *triple* sólo tiene tres campos, a los cuales llamamos op , arg_1 y arg_2 . Observe que el campo *resultado* de la figura 6.10(b) se utiliza principalmente para los nombres temporales. Al usar tripletas, nos referimos al resultado de una operación $x op y$ por su posición, en vez de usar un nombre temporal explícito. Por ende, en vez del valor temporal t_1 en la figura 6.10(b), una representación en tripletas se referiría a la posición (0). Los números entre paréntesis representan apuntadores a la misma estructura de las tripletas. En la sección 6.1.2, a las posiciones o apuntadores a las posiciones se les llamó números de valor.

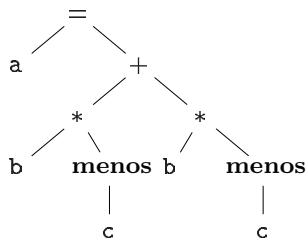
Las tripletas son equivalentes a las firmas en el Algoritmo 6.3. Por ende, el GDA y las representaciones en tripletas de las expresiones son equivalentes. La equivalencia termina con las expresiones, ya que las variantes del árbol sintáctico y el código de tres direcciones representan el flujo de control de una manera muy distinta.

Ejemplo 6.7: El árbol sintáctico y las tripletas en la figura 6.11 corresponden al código de tres direcciones y los cuádruplos en la figura 6.10. En la representación en tripletas de la figura 6.11(b), la instrucción de copia $a = t_5$ está codificada en la representación en tripletas, mediante la colocación de a en el campo arg_1 y (4) en el campo arg_2 . \square

Una operación ternaria como $x[i] = y$ requiere dos entradas en la estructura de las tripletas; por ejemplo, podemos colocar x e i en una tripleta y a y en la siguiente. De manera similar, podemos implementar $x = y[i]$ tratándola como si fuera las dos instrucciones $t = y[i]$ y $x = t$,

¿Por qué necesitamos instrucciones de copia?

Un algoritmo simple para traducir expresiones genera instrucciones de copia para las asignaciones, como en la figura 6.10(a), en donde copiamos t_5 en a , en vez de asignarle directamente $t_2 + t_4$. Por lo general, cada subexpresión obtiene su nuevo valor temporal propio para guardar su resultado, y sólo cuando se procesa el operador de asignación = es cuando aprendemos en dónde colocar el valor de la expresión completa. Una pasada de optimización de código, tal vez mediante el uso del GDA de la sección 6.1.1 como forma intermedia, puede descubrir que t_5 se puede sustituir por a .



(a) Árbol sintáctico

	op	arg ₁	arg ₂
0	menos	c	
1	*	b	(0)
2	menos	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

(b) Triples

Figura 6.11: Representaciones de $a + a * (b - c) + (b - c) * d$

en donde t es un valor temporal generado por el compilador. Observe que el valor temporal t en realidad no aparece en un triple, ya que se hace referencia a los valores temporales mediante su posición en la estructura de la misma.

En un compilador optimizador podemos ver un beneficio de los cuádruples, en comparación con las tripletas, en donde las instrucciones se mueven con frecuencia. Con los cuádruples, si movemos una instrucción que calcule un valor temporal t , entonces las instrucciones que usen a t no requerirán ninguna modificación. Con las tripletas, se hace referencia al resultado de una operación mediante su posición, por lo que al mover una instrucción tal vez tengamos que modificar todas las referencias a ese resultado. Este problema no ocurre con las tripletas indirectas, que veremos a continuación.

Las *tripletas indirectas* consisten en un listado de apuntadores a tripletas, en vez de ser un listado de las mismas tripletas. Por ejemplo, vamos a utilizar un arreglo llamado *instrucción* para listar apuntadores a tripletas en el orden deseado. Así, las tripletas de la figura 6.11(b) podrían representarse como en la figura 6.12.

Con las tripletas indirectas, un compilador de optimización puede mover una instrucción reordenando la lista en *instrucción*, sin afectar a las mismas tripletas. Si se implementa en Java, un arreglo de objetos de instrucciones es análogo a una representación en tripletas indirectas, ya que Java trata a los elementos del arreglo como referencias a objetos.

instrucción	op	arg ₁	arg ₂
35 (0)	menos	c	
36 (1)	*	b	(0)
37 (2)	menos	c	
38 (3)	*	b	(2)
39 (4)	+	(1)	(3)
40 (5)	=	a	(4)
...			...

Figura 6.12: Representación en tripletas indirectas del código de tres direcciones

6.2.4 Forma de asignación individual estática

La forma de asignación individual estática es una representación intermedia que facilita ciertas optimizaciones de código. Dos aspectos distinguen a la forma de asignación individual estática del código de tres direcciones. El primero es que todas las asignaciones en asignación individual estática son a variables con distintos nombres; de aquí que se utilice el término *asignación individual estática*. La figura 6.13 muestra el mismo programa intermedio en código de tres direcciones y en forma de asignación individual estática. Observe que los subíndices diferencian cada definición de variables p y q en la representación de asignación individual estática.

$$\begin{aligned} p &= a + b \\ q &= p - c \\ p &= q * d \\ p &= e - p \\ q &= p + q \end{aligned}$$

(a) Código de tres direcciones

$$\begin{aligned} p_1 &= a + b \\ q_1 &= p_1 - c \\ p_2 &= q_1 * d \\ p_3 &= e - p_2 \\ q_2 &= p_3 + q_1 \end{aligned}$$

(b) Forma de asignación individual estática

Figura 6.13: Programa intermedio en código de tres direcciones y asignación individual estática

La misma variable puede definirse en dos rutas de flujo de control distintas en un programa. Por ejemplo, el siguiente programa fuente:

```
if ( bandera ) x = -1; else x = 1;
y = x * a;
```

tiene dos rutas de flujo de control en las que se define la variable x. Si utilizamos distintos nombres para x en la parte verdadera y en la parte falsa de la instrucción condicional, entonces ¿qué nombre debemos usar en la asignación y = x * a? Aquí es donde entra en juego el segundo aspecto distintivo de asignación individual estática. Dicha asignación utiliza una convención de notación, conocida como función ϕ , para combinar las dos definiciones de x:

```
if ( bandera ) x1 = -1; else x2 = 1;
x3 =  $\phi$ (x1, x2);
```

Aquí, $\phi(x_1, x_2)$ tiene el valor x_1 si el flujo de control pasa a través de la parte verdadera de la condicional, y el valor x_2 si el flujo de control pasa a través de la parte falsa. Es decir, la función ϕ devuelve el valor de su argumento que corresponde a la ruta de flujo de control que se eligió para llegar a la instrucción de asignación que contiene la función ϕ .

6.2.5 Ejercicios para la sección 6.2

Ejercicio 6.2.1: Traduzca la expresión aritmética $a + -(b + c)$ en:

- a) Un árbol sintáctico.
- b) Cuádruplos.
- c) Tripletas.
- d) Tripletas indirectos.

Ejercicio 6.2.2: Repita el ejercicio 6.2.1 para las siguientes instrucciones de asignación:

- i. $a = b[i] + c[j]$.
- ii. $a[i] = b*c - b*d$.
- iii. $x = f(y+1) + 2$.
- iv. $x = *p + &y$.

! Ejercicio 6.2.3: Muestre cómo transformar una secuencia de código de tres direcciones en una en la que cada variable definida obtenga un nombre de variable único.

6.3 Tipos y declaraciones

Las aplicaciones de los tipos pueden agruparse mediante la comprobación y la traducción:

- La *comprobación de tipos* utiliza reglas lógicas para razonar acerca del comportamiento de un programa en tiempo de ejecución. En específico, asegura que los tipos de los operandos coincidan con el tipo esperado por un operador. Por ejemplo, el operador `&&` en Java espera que sus dos operandos sean booleanos; el resultado también es de tipo booleano.
- *Aplicaciones de traducción.* A partir del tipo de un nombre, un compilador puede determinar el almacenamiento necesario para ese nombre en tiempo de ejecución. La información del tipo también se necesita para calcular la dirección que denota la referencia a un arreglo, para insertar conversiones de tipo explícitas, y para elegir la versión correcta de un operador aritmético, entre otras cosas.

En esta sección, examinaremos los tipos y la distribución del almacenamiento para los nombres declarados dentro de un procedimiento o una clase. El almacenamiento actual para la llamada a un procedimiento o un objeto se asigna en tiempo de ejecución, cuando se llama al procedimiento o se crea el objeto. Sin embargo, al examinar las declaraciones locales en tiempo de compilación, podemos distribuir *direcciones relativas*, en donde la dirección relativa de un nombre o un componente de una estructura de datos es un desplazamiento a partir del inicio de un área de datos.

6.3.1 Expresiones de tipos

Los tipos tienen estructura, a la cual representamos mediante el uso de las *expresiones de tipos*: una expresión de tipos es un tipo básico o se forma mediante la aplicación de un operador llamado *constructor de tipos* a una expresión de tipos. Los conjuntos de tipos básicos y los constructores dependen del lenguaje que se va a comprobar.

Ejemplo 6.8: El tipo de arreglo `int[2][3]` puede leerse como “arreglo de 2 arreglos, de 3 enteros cada uno” y escribirse como una expresión de tipos `arreglo(2, arreglo(3, integer))`. Este tipo se representa mediante el árbol en la figura 6.14. El operador *arreglo* recibe dos parámetros, un número y un tipo. □

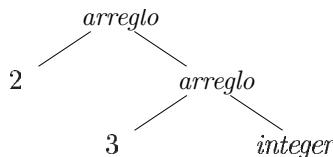


Figura 6.14: Expresión de tipos para `int[2][3]`

Vamos a usar la siguiente definición de expresiones de tipos:

- Un tipo básico es una expresión de tipos. Los tipos básicos comunes para un lenguaje incluyen `boolean`, `char`, `integer`, `float` y `void`; este último denota “la ausencia de un valor”.
- El nombre de un tipo es una expresión de tipos.
- Una expresión de tipos se puede formar mediante la aplicación del constructor de tipo *arreglo* a un número y a una expresión de tipos.
- Un registro es una estructura de datos con campos identificados por nombres. Una expresión de tipos se puede formar mediante la aplicación del constructor de tipo *registro* a los nombres de los campos y sus tipos. Los registros de los tipos se implementarán en la sección 6.3.6, mediante la aplicación del constructor *registro* a una tabla de símbolos que contiene entradas para los campos.
- Una expresión de tipos puede formarse mediante el uso del constructor de tipo \rightarrow para los tipos de funciones. Escribimos $s \rightarrow t$ para la “función del tipo s al tipo t ”. Los tipos de funciones serán útiles cuando hablamos sobre la comprobación de tipos en la sección 6.5.

Nombres de los tipos y tipos recursivos

Una vez que se define una clase, su nombre se puede usar como el nombre de un tipo en C++ o en Java; por ejemplo, considere a `Nodo` en el siguiente fragmento de programa:

```
public class Nodo { ... }
...
public Nodo n;
```

Pueden usarse nombres para definir tipos recursivos, los cuales son necesarios para las estructuras de datos como las listas enlazadas. El seudocódigo para el elemento de una lista:

```
class Celda { int info; Celda siguiente; ... }
```

define el tipo recursivo `Celda` como una clase que contiene un campo `info` y un campo `siguiente` de tipo `Celda`. Pueden definirse tipos recursivos similares en C, mediante el uso de registros y apuntadores. Las técnicas en este capítulo se enfocan en los tipos recursivos.

- Si s y t son expresiones de tipos, entonces su producto Cartesiano $s \times t$ es una expresión de tipos. Los productos se introducen por completitud; pueden usarse para representar una lista o tupla de tipos (por ejemplo, para los parámetros de funciones). Asumimos que \times se asocia a la izquierda y que tiene mayor precedencia que \rightarrow .
- Las expresiones de tipos pueden contener variables cuyos valores sean expresiones de tipos. En la sección 6.5.4 utilizaremos variables de tipos generados por el compilador.

Una manera conveniente de representar una expresión de tipos es mediante un grafo. El método número de valor de la sección 6.1.2 pudo adaptarse para construir un GDA para una expresión de tipos, con nodos interiores para los constructores de tipos y hojas para los tipos básicos, los nombres de los tipos y las variables de tipos; por ejemplo, vea el árbol de la figura 6.14.³

6.3.2 Equivalencia de tipos

¿Cuándo son equivalentes dos expresiones de tipos? Muchas reglas de comprobación de tipos tienen la forma: “**if** dos expresiones de tipos son iguales **then** devolver cierto tipo **else** error”. Surgen ambigüedades potenciales cuando se proporcionan nombres a las expresiones de tipos y después se utilizan los nombres en expresiones de tipos subsiguientes. La cuestión clave es si el nombre en una expresión de tipos se representa a sí mismo o si es una abreviación para otra expresión de tipos.

³Como los nombres de los tipos denotan expresiones de tipos, pueden establecer ciclos implícitos; vea el recuadro sobre “Nombres de tipos y tipos recursivos”. Si las flechas que van a los nombres de los tipos se redirigen a las expresiones de tipos denotadas por los nombres, entonces el grafo resultante puede tener ciclos debido a los tipos recursivos.

Cuando las expresiones de tipos se representan mediante grafos, dos tipos son *equivalentes en estructura* si y solo si una de las siguientes condiciones es verdadera:

- Son el mismo tipo básico.
- Se forman mediante la aplicación del mismo constructor de tipos equivalentes en estructura.
- Uno es el nombre de un tipo que denota al otro.

Si los nombres de los tipos se tratan como si se representaran a sí mismos, entonces las primeras dos condiciones en la definición anterior conducen a la *equivalencia de nombres* de las expresiones de tipos.

Las expresiones con nombres equivalentes reciben el mismo número de valor, si utilizamos el Algoritmo 6.3. Podemos probar la equivalencia estructural usando el algoritmo de unificación de la sección 6.5.5.

6.3.3 Declaraciones

Vamos a estudiar los tipos y las declaraciones mediante una gramática simplificada que declara sólo un nombre a la vez; las declaraciones con listas de nombres pueden manejarse como vimos en el ejemplo 5.10. La gramática es:

$$\begin{array}{lcl} D & \rightarrow & T \text{ id} ; D \mid \epsilon \\ T & \rightarrow & B \ C \mid \text{record} \ '{' \ D \ '}' \\ B & \rightarrow & \text{int} \mid \text{float} \\ C & \rightarrow & \epsilon \mid [\ \text{num} \] \ C \end{array}$$

El fragmento de la gramática anterior que maneja los tipos básicos y de arreglo se utilizó para ilustrar los atributos heredados en la sección 5.3.2. La diferencia en esta sección es que consideraremos la distribución del almacenamiento, así como los tipos.

La no terminal D genera una secuencia de declaraciones. La no terminal T genera tipos básicos, de arreglos o de registros. La no terminal B genera uno de los tipos básicos **int** y **float**. La no terminal C , de “componente”, genera cadenas de cero o más enteros, en donde cada entero se encierra entre llaves. Un tipo de arreglo consiste en un tipo básico especificado por B , seguido de los componentes del arreglo especificados por la no terminal C . Un tipo de registro (la segunda producción para T) es una secuencia de declaraciones para los campos del registro, todas encerradas entre llaves.

6.3.4 Distribución del almacenamiento para los nombres locales

Del tipo de un nombre podemos determinar la cantidad de almacenamiento que será necesaria para ese nombre en tiempo de ejecución. En tiempo de compilación, podemos usar estas cantidades para asignar a cada nombre una dirección relativa. El tipo y la dirección relativa se almacenan en esta entrada en la tabla de símbolos para el nombre. Los datos de longitud variable, como las cadenas, o los datos cuyo tamaño no puede determinarse sino hasta en tiempo de ejecución, como los arreglos dinámicos, se manejan reservando una cantidad fija conocida de almacenamiento para un apuntador a los datos. En el capítulo 7 hablaremos sobre el manejo del almacenamiento en tiempo de ejecución.

Alineación de direcciones

La distribución del almacenamiento para los objetos de datos tiene gran influencia de las restricciones de direccionamiento de la máquina de destino. Por ejemplo, las instrucciones para sumar enteros pueden esperar que los enteros estén *alineados*; es decir, que se coloquen en ciertas posiciones en memoria, como en una dirección divisible entre 4. Aunque un arreglo de diez caracteres sólo necesita suficientes bytes para guardar diez caracteres, un compilador podría, por lo tanto, asignar 12 bytes (el siguiente múltiplo de 4), dejando 2 bytes sin usar. El espacio que se deja sin utilizar debido a las consideraciones de alineación se conoce como *relleno*. Cuando el espacio es de extrema importancia, un compilador puede *empaquetar* los datos, para que no quede *relleno*; entonces tal vez haya que ejecutar instrucciones adicionales en tiempo de ejecución para posicionar los datos empaquetados, de manera que pueda operarse con ellos como si estuvieran alineados en forma apropiada.

Suponga que el almacenamiento se da en bloques de bytes contiguos, en donde un byte es la unidad más pequeña de memoria que puede direccionarse. Por lo general, un byte es de ocho bits y cierto número de bytes forman una palabra de máquina. Los objetos de varios bytes se almacenan en bytes consecutivos y reciben la dirección del primer byte.

La *anchura* de un tipo es el número de unidades de almacenamiento necesarias para los objetos de ese tipo. Un tipo básico, como un carácter, entero o número de punto flotante, requiere un número integral de bytes. Para facilitar el acceso, el espacio para los agregados como los arreglos y las clases se asigna en un bloque contiguo de bytes.⁴

El esquema de traducción (SDT) en la figura 6.15 calcula los tipos y sus tamaños para los tipos básicos y de arreglos; en la sección 6.3.6 veremos los tipos de registros. El SDT utiliza los atributos sintetizados *tipo* y *tamaño* para cada no terminal y dos variables, *t* y *w*, para pasar la información del tipo y el tamaño, de un nodo *B* en un árbol de análisis sintáctico al nodo para la producción $C \rightarrow \epsilon$. En una definición orientada por la sintaxis, *t* y *w* serían atributos heredados para *C*.

El cuerpo de la producción *T* consiste en el no terminal *B*, una acción y el no terminal *C*, que aparece en la siguiente línea. La acción entre *B* y *C* establece *t* a *B.tipo* y *w* a *B.tamaño*. Si $B \rightarrow \mathbf{int}$ entonces *B.tipo* se establece a *integer* y *B.tamaño* se establece a 4, el tamaño de un entero. De manera similar, si $B \rightarrow \mathbf{float}$ entonces *B.tipo* es *float* y *B.tamaño* es 8, el tamaño de un número de punto flotante.

Las producciones para *C* determinan si *T* genera un tipo básico o un tipo de arreglo. Si $C \rightarrow \epsilon$, entonces *t* se convierte en *C.tipo* y *w* se convierte en *C.tamaño*.

En cualquier otro caso, *C* especifica el componente de un arreglo. La acción para $C \rightarrow [\mathbf{num}] C_1$ forma a *C.tipo*, aplicando el constructor de tipos *arreglo* a los operandos *num.valor* y *C1.tipo*. Por ejemplo, el resultado de aplicar *arreglo* podría ser una estructura de árbol como la figura 6.14.

⁴La asignación del almacenamiento para los apuntadores en C y C++ es más simple si todos los apuntadores tienen la misma longitud. La razón es que el almacenamiento para un apuntador tal vez deba asignarse antes de que conozcamos el tipo de objetos a los que puede apuntar.

$T \rightarrow B$	$\{ t = B.tipo; w = B.anchura; \}$
C	
$B \rightarrow \mathbf{int}$	$\{ B.tipo = \mathbf{integer}; B.anchura = 4; \}$
$B \rightarrow \mathbf{float}$	$\{ B.tipo = \mathbf{float}; B.anchura = 8; \}$
$C \rightarrow \epsilon$	$\{ C.tipo = t; C.anchura = w; \}$
$C \rightarrow [\mathbf{num}] C_1$	$\{ \mathbf{arreglo}(\mathbf{num}.valor, C_1.tipo);$ $C.anchura = \mathbf{num}.valor \times C_1.anchura; \}$

Figura 6.15: Cálculo de los tipos y sus anchuras

El tamaño de un arreglo se obtiene al multiplicar el tamaño de un elemento por el número de elementos en el arreglo. Si las direcciones de enteros consecutivos difieren por 4, entonces los cálculos de las direcciones para un arreglo de enteros incluirán multiplicaciones por 4. Dichas multiplicaciones proporcionan oportunidades para la optimización, por lo cual es útil para el front-end hacerlas explícitas. En este capítulo, ignoraremos otras dependencias de la máquina, como la alineación de los objetos de datos sobre límites de palabras.

Ejemplo 6.9: El árbol de análisis sintáctico para el tipo $\mathbf{int}[2][3]$ se muestra mediante líneas punteadas en la figura 6.16. Las líneas sólidas muestran cómo se pasan el tipo y el tamaño desde B , descendiendo por la cadena de C s a través de las variables t y w , y ascendiendo por la cadena como los atributos sintetizados $tipo$ y $tamaño$. A las variables t y w se les asignan los valores de $B.tipo$ y $B.tamaño$, respectivamente, antes de examinar el subárbol con los nodos C . Los valores de t y w se utilizan en el nodo para $C \rightarrow \epsilon$, con el fin de iniciar la evaluación de los atributos sintetizados, ascendiendo por la cadena de nodos C . \square

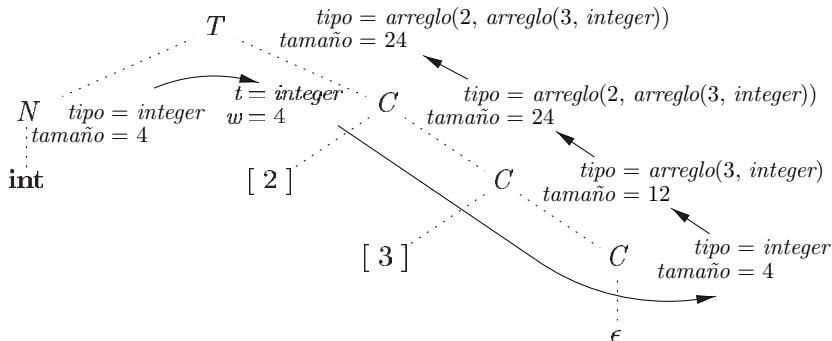


Figura 6.16: Traducción orientada por la sintaxis de los tipos de arreglos

6.3.5 Secuencias de las declaraciones

Los lenguajes como C y Java permiten que todas las declaraciones en un solo procedimiento se procesen como un grupo. Las declaraciones pueden distribuirse dentro de un procedimiento de Java, pero de todas formas pueden procesarse al momento de analizar el procedimiento. Por lo tanto, podemos usar una variable, por decir *desplazamiento*, para llevar el registro de la siguiente dirección relativa disponible.

El esquema de traducción de la figura 6.17 trata con una secuencia de declaraciones de la forma $T \mathbf{id}$, en donde T genera un tipo como en la figura 6.15. Antes de considerar la primera declaración, *desplazamiento* se establece a 0. A medida que se ve cada nuevo nombre x , este nombre se introduce en la tabla de símbolos y se establece su dirección relativa al valor actual de *desplazamiento*, que después se incrementa en base a la anchura del tipo de x .

$$\begin{aligned}
 P &\rightarrow & \{ \text{desplazamiento} = 0; \} \\
 &D & \\
 D &\rightarrow T \mathbf{id} ; & \{ \text{superior.put}(\mathbf{id}.lexema, T.tipo, \text{desplazamiento}); \\
 && \text{desplazamiento} = \text{desplazamiento} + T.\text{tamaño}; \} \\
 &D_1 & \\
 D &\rightarrow \epsilon &
 \end{aligned}$$

Figura 6.17: Cálculo de las direcciones relativas de los nombres declarados

La acción semántica dentro de la producción $D \rightarrow T \mathbf{id} ;$ D_1 crea una entrada en la tabla de símbolos mediante la ejecución de $\text{superior.put}(\mathbf{id}.lexema, T.\text{tipo}, \text{desplazamiento})$. Aquí, *superior* denota la tabla de símbolos actual. El método *superior.put* crea una entrada en la tabla de símbolos para $\mathbf{id}.lexema$, con el tipo $T.\text{tipo}$ y la dirección relativa *desplazamiento* en su área de datos.

La inicialización de *desplazamiento* en la figura 6.17 es más evidente si la primera producción aparece en una línea como:

$$P \rightarrow \{ \text{desplazamiento} = 0; \} D \quad (6.1)$$

Los no terminales que generan a ϵ , conocidas como no terminales marcadores, pueden usarse para rescribir las producciones, de manera que todas las acciones aparezcan al final de los lados derechos; vea la sección 5.5.4. Si utilizamos un no terminal como marcador M , la producción (6.1) puede redeclararse como:

$$\begin{aligned}
 P &\rightarrow M D \\
 M &\rightarrow \epsilon & \{ \text{desplazamiento} = 0; \}
 \end{aligned}$$

6.3.6 Campos en registros y clases

La traducción de las declaraciones en la figura 6.17 se transfiere a los campos en los registros y las clases. Podemos agregar tipos de registros a la gramática en la figura 6.15 si agregamos la siguiente producción:

$$T \rightarrow \mathbf{registro} \ '{' D '}'$$

Los campos en este tipo de registro se especifican mediante la secuencia de declaraciones generadas por D . El método de la figura 6.17 puede utilizarse para determinar los tipos y las direcciones relativas de los campos, siempre y cuando tengamos cuidado con dos cosas:

- Los nombres de los campos dentro de un registro deben ser distintos; es decir, puede aparecer un nombre máximo una vez en las declaraciones generadas por D .
- El desplazamiento o dirección relativa para el nombre de un campo es relativo al área de datos para ese registro.

Ejemplo 6.10: El uso de un nombre x para un campo dentro de un registro no entra en conflicto con los demás usos del nombre fuera del registro. Por ende, los tres usos de x en las siguientes declaraciones son distintos y no entran en conflicto unos con otros:

```
float x;
registro { float x; float y; } p;
registro { int etiqueta; float x; float y; } q;
```

Una siguiente asignación $x = p.x + q.x$; establece la variable x a la suma de los campos llamados x en los registros p y q . Observe que la dirección relativa de x en p difiere de la dirección relativa de x en q . \square

Por conveniencia, los tipos de registros codificarán tanto a los tipos como a las direcciones relativas de sus campos, usando una tabla de símbolos para el tipo de registro. Un tipo de registro tiene la forma $registro(t)$, en donde *registro* es el constructor de tipos y t es un objeto en la tabla de símbolos que contiene información acerca de los campos de este tipo de registro.

El esquema de traducción de la figura 6.18 consiste en una sola producción que se va a agregar a las producciones para T en la figura 6.15. Esta producción tiene dos acciones semánticas. La acción incrustada antes de D almacena la tabla de símbolos existente, denotada por *tope*, y establece *tope* a una nueva tabla de símbolos. También almacena el *desplazamiento* actual, y establece *desplazamiento* a 0. Las declaraciones generadas por D ocasionan que los tipos y las direcciones relativas se metan en la nueva tabla de símbolos. La acción después de D crea un tipo de registro usando *tope*, antes de restaurar la tabla de símbolos y el desplazamiento almacenados.

```
T → registro 'f' { Env.push(tope); tope = new Env();
                    Pila.push(desplazamiento); desplazamiento = 0; }
D '}' { T.tipo = registro(tope); T.anchura = desplazamiento;
          Tope = Env.pop(); desplazamiento = Pila.pop(); }
```

Figura 6.18: Manejo de los nombres de los campos en los registros

Para fines concretos, las acciones en la figura 6.18 proporcionan el seudocódigo para una implementación específica. Suponga que la clase *Env* implementa a las tablas de símbolos. La llamada *Env.push(tope)* mete la tabla de símbolos actual, denotada por *tope*, en una pila. Después, la variable *tope* se establece a una nueva tabla de símbolos. De manera similar, *desplazamiento* se mete en una pila llamada *Pila*. Después, la variable *desplazamiento* se establece a 0.

Una vez que se han traducido las declaraciones en D , la tabla de símbolos *tope* contiene los tipos y las direcciones relativas de los campos en este registro. Además, *desplazamiento* proporciona el almacenamiento necesario para todos los campos. La segunda acción establece $T.tipo$ a $registro(tope)$ y $T.tamaño$ a *desplazamiento*. Después, las variables *tope* y *desplazamiento* se restauran a los valores que se habían metido en la pila, para completar la traducción de este tipo de registro.

Esta explicación sobre el almacenamiento para los tipos de registros se transfiere a las clases, ya que no hay almacenamiento reservado para los métodos. Vea el ejercicio 6.3.2.

6.3.7 Ejercicios para la sección 6.3

Ejercicio 6.3.1: Determine los tipos y las direcciones relativas para los identificadores en la siguiente secuencia de declaraciones:

```
float x;
registro { float x; float y; } p;
registro { int etiqueta; float x; float y; } q;
```

! Ejercicio 6.3.2: Extienda el manejo de los nombres de los campos en la figura 6.18 a las clases y las jerarquías de clases con herencia simple.

- Proporcione una implementación de la clase *Env* que permita tablas de símbolos enlazadas, de manera que una subclase pueda redefinir el nombre de un campo o hacer referencia directa al nombre de un campo en una superclase.
- Proporcione un esquema de traducción que asigne un área de datos contigua para los campos en una clase, incluyendo los campos heredados. Estos campos deben mantener las direcciones relativas que se les asignaron en el esquema para la superclase.

6.4 Traducción de expresiones

El resto de este capítulo explora las cuestiones que surgen durante la traducción de expresiones e instrucciones. En esta sección empezamos con la traducción de expresiones en código de tres direcciones. Una expresión con más de un operador, como $a + b * c$, se traducirá en instrucciones con a lo más un operador por instrucción. La referencia a un arreglo $A[i][j]$ se expandirá en una secuencia de instrucciones de tres direcciones que calculan una dirección para la referencia. En la sección 6.5 consideraremos la comprobación de tipos de las expresiones y en la sección 6.6 veremos el uso de expresiones booleanas para dirigir el flujo de control a través de un programa.

6.4.1 Operaciones dentro de expresiones

La definición orientada por la sintaxis en la figura 6.19 construye el código de tres direcciones para una instrucción de asignación S , usando el atributo *codigo* para S y los atributos *dir* y *codigo* para una expresión E . Los atributos $S.codigo$ y $E.codigo$ denotan el código de tres direcciones

PRODUCCIÓN	REGLAS SEMÁNTICAS
$S \rightarrow \mathbf{id} = E ;$	$S.codigo = E.codigo \mid\mid$ $gen(tope.get(\mathbf{id}.lexema) ' =' E.dir)$
$E \rightarrow E_1 + E_2$	$E.dir = \mathbf{new} \ Temp()$ $E.codigo = E_1.codigo \mid\mid E_2.codigo \mid\mid$ $gen(E.dir ' =' E_1.dir ' +' E_2.dir)$
$ - E_1$	$E.dir = \mathbf{new} \ Temp()$ $E.codigo = E_1.codigo \mid\mid$ $gen(E.dir ' =' '\mathbf{menos}' E_1.dir)$
$ (E_1)$	$E.dir = E_1.dir$ $E.codigo = E_1.codigo$
$ \mathbf{id}$	$E.dir = tope.get(\mathbf{id}.lexema)$ $E.codigo = ''$

Figura 6.19: Código de tres direcciones para las expresiones

para S y E , respectivamente. El atributo $E.dir$ denota la dirección que contendrá el valor de E . En la sección 6.2.1 vimos que una dirección puede ser un nombre, una constante o un valor temporal generado por el compilador.

Considere la última producción, $E \rightarrow \mathbf{id}$, en la definición orientada por la sintaxis de la figura 6.19. Cuando una expresión es un solo identificador, por decir x , entonces el mismo x contiene el valor de la expresión. Las reglas semánticas para esta producción definen a $E.dir$ para que apunte a la entrada en la tabla de símbolos para esta instancia de \mathbf{id} . Suponga que $tope$ denota la tabla de símbolos actual. La función $tope.get$ obtiene la entrada cuando se aplica a la representación de cadena $\mathbf{id}.lexema$ de esta instancia \mathbf{id} . $E.code$ se establece a la cadena vacía.

Cuando $E \rightarrow (E_1)$, la traducción de E es la misma que la de la subexpresión E_1 . Por ende, $E.dir$ es igual a $E_1.dir$, y $E.codigo$ es igual a $E_1.codigo$.

Los operadores $+$ y $-$ unario en la figura 6.19 son representativos de los operadores en un lenguaje ordinario. Las reglas semánticas para $E \rightarrow E_1 + E_2$, generan código para calcular el valor de E a partir de los valores de E_1 y E_2 . Los valores se calculan y colocan en nombres temporales recién generados. Si E_1 se calcula y se coloca en $E_1.dir$, y E_2 en $E_2.dir$, entonces $E_1 + E_2$ se traduce en $t = E_1.addr + E_2.addr$, en donde t es un nuevo nombre temporal. A $E.addr$ se le asigna t . Para crear una secuencia de nombres temporales t_1, t_2, \dots distintos, se ejecuta $\mathbf{new} \ Temp()$ en forma sucesiva.

Por conveniencia, usamos la notación $gen(x ' =' y ' +' z)$ para representar la instrucción de tres direcciones $x = y + z$. Las expresiones que aparecen en lugar de variables como x , y y z se evalúan cuando se pasan a gen , y las cadenas entre comillas como $' ='$ se interpretan en forma literal.⁵ Otras instrucciones de tres direcciones se generarán de manera similar, aplicando gen a una combinación de expresiones y cadenas.

⁵En las definiciones dirigidas por la sintaxis, gen genera una instrucción y la devuelve. En los esquemas de traducción, gen genera una instrucción y la emite en forma incremental, colocándola en el flujo de instrucciones generadas.

Al traducir la producción $E \rightarrow E_1 + E_2$, las reglas semánticas en la figura 6.19 generan a $E.codigo$ mediante la concatenación de $E_1.codigo$, $E_2.codigo$ y una instrucción que suma los valores de E_1 y E_2 . La instrucción coloca el resultado de la suma en un nuevo nombre temporal para E , denotado por $E.dir$.

La traducción de $E \rightarrow -E_1$ es similar. Las reglas crean un nuevo nombre temporal para E y generan una instrucción para realizar la operación de resta unaria.

Por último, la producción $S \rightarrow \mathbf{id} = E$; genera instrucciones que asignan el valor de la expresión E al identificador \mathbf{id} . La regla semántica para esta producción utiliza la función `tope.get` para determinar la dirección del identificador representado por \mathbf{id} , como en las reglas para $E \rightarrow \mathbf{id}$. $S.codigo$ consiste en las instrucciones para calcular el valor de E y colocarlo en una dirección proporcionada por $E.dir$, seguida de una asignación a la dirección `tope.get(id.lexema)` para esta instancia de \mathbf{id} .

Ejemplo 6.11: La definición orientada por la sintaxis en la figura 6.19 traduce la instrucción de asignación $a = b + -c$; en la siguiente secuencia de código de tres direcciones:

```
t1 = menos c
t2 = b + t1
a = t2
```

□

6.4.2 Traducción incremental

Los atributos de código pueden ser cadenas largas, así que, por lo general, se generan en forma incremental, como vimos en la sección 5.5.2. Por ende, en vez de generar a $E.codigo$ como en la figura 6.19, podemos hacer que sólo se generen las nuevas instrucciones de tres direcciones, como en el esquema de traducción de la figura 6.20. En el método incremental, `gen` no sólo construye una instrucción de tres direcciones, sino que también adjunta la instrucción a la secuencia de instrucciones generadas hasta ahora. La secuencia puede retenerse en memoria para su posterior procesamiento, o puede enviarse como salida en forma incremental.

El esquema de traducción en la figura 6.20 genera el mismo código que la definición orientada por la sintaxis de la figura 6.19. Con el método incremental, el atributo `codigo` no se utiliza, ya que hay una sola secuencia de instrucciones que se crea mediante llamadas sucesivas a `gen`. Por ejemplo, la regla semántica para $E \rightarrow E_1 + E_2$ en la figura 6.20 simplemente llama a `gen` para generar la instrucción de suma; las instrucciones para calcular E_1 y colocarlo en $E_1.dir$, y para generar E_2 y colocarlo en $E_2.dir$ ya se han generado.

El método de la figura 6.20 también puede usarse para construir un árbol sintáctico. La nueva acción semántica para $E \rightarrow E_1 + E_2$ crea un nodo mediante el uso de un constructor, como en:

$$E \rightarrow E_1 + E_2 \{ E.dir = \mathbf{new} \ Nodo('+' , E_1.dir, E_2.dir); \}$$

Aquí, el atributo `dir` representa la dirección de un nodo, en vez de una variable o constante.

$$\begin{array}{lcl}
 S \rightarrow \mathbf{id} = E ; & \{ \text{gen}(tope.get}(\mathbf{id}.lexema) ' = ' E.dir); \}
 \\[10pt]
 E \rightarrow E_1 + E_2 & \{ E.dir = \mathbf{new} \text{ Temp}(); \\
 & \quad \text{gen}(E.dir ' = ' E_1.dir ' + ' E_2.dir); \}
 \\[10pt]
 | \quad - E_1 & \{ E.dir = \mathbf{new} \text{ Temp}(); \\
 & \quad \text{gen}(E.dir ' = ' ' \mathbf{menos} ' E_1.dir); \}
 \\[10pt]
 | \quad (E_1) & \{ E.dir = E_1.dir; \}
 \\[10pt]
 | \quad \mathbf{id} & \{ E.dir = tope.get(\mathbf{id}.lexema); \}
 \end{array}$$

Figura 6.20: Generación de código de tres direcciones para expresiones en forma incremental

6.4.3 Direcciónamiento de los elementos de un arreglo

Se puede tener acceso rápido a los elementos de un arreglo, si éstos se encuentran almacenados en un bloque de ubicaciones consecutivas. En C y Java, los elementos de un arreglo se enumeran desde $0, 1, \dots, n - 1$ para un arreglo con n elementos. Si la anchura de cada elemento del arreglo es w , entonces el i -ésimo elemento del arreglo A empieza en la siguiente ubicación:

$$base + i \times w \quad (6.2)$$

en donde $base$ es la dirección relativa del almacenamiento asignado para el arreglo. Es decir, $base$ es la dirección relativa de $A[0]$.

La fórmula (6.2) se generaliza para dos o más dimensiones. En dos dimensiones, escribimos $A[i_1][i_2]$ en C y Java para el elemento i_2 en la fila i_1 . Haga que w_1 sea el tamaño de una fila y que w_2 sea el tamaño de un elemento en una fila. Así, la dirección relativa de $A[i_1][i_2]$ se puede calcular mediante la siguiente fórmula:

$$base + i_1 \times w_1 + i_2 \times w_2 \quad (6.3)$$

En k dimensiones, la fórmula es:

$$base + i_1 \times w_1 + i_2 \times w_2 + \dots + i_k \times w_k \quad (6.4)$$

en donde w_j , para $1 \leq j \leq k$, es la generalización de w_1 y w_2 en (6.3).

De manera alternativa, la dirección relativa de una referencia a un arreglo puede calcularse en términos de los números de elementos n_j a lo largo de la dimensión j del arreglo y la anchura $w = w_k$ de un solo elemento del arreglo. En dos dimensiones (es decir, $k = 2$ y $w = w_2$), la ubicación para $A[i_1][i_2]$ se da mediante:

$$base + (i_1 \times n_2 + i_2) \times w \quad (6.5)$$

En k dimensiones, la siguiente fórmula calcula la misma dirección que (6.4):

$$base + ((\dots(i_1 \times n_2 + i_2) \times n_3 + i_3)\dots) \times n_k + i_k \times w \quad (6.6)$$

En forma más general, los elementos de un arreglo no tienen que enumerarse empezando en 0. En un arreglo unidimensional, los elementos del arreglo se enumeran como *inferior*, *inferior* + 1, ..., *superior* y *base* es la dirección relativa de $A[\text{inferior}]$. La fórmula (6.2) para la dirección de $A[i]$ se sustituye por:

$$\text{base} + (i - \text{inferior}) \times w \quad (6.7)$$

Las expresiones (6.2) y (6.7) pueden rescribirse como $i \times w + c$, en donde la subexpresión $c = \text{base} - \text{inferior} \times w$ puede calcularse previamente en tiempo de compilación. Observe que $c = \text{base}$ cuando *inferior* es 0. Asumimos que *c* se almacena en la entrada de la tabla de símbolos para A , por lo que la dirección relativa de $A[i]$ se obtiene con sólo sumar $i \times w$ a *c*.

El cálculo previo en tiempo de compilación también puede aplicarse a los cálculos de las direcciones para los elementos de arreglos multidimensionales; vea el ejercicio 6.4.5. No obstante, hay una situación en la que no podemos usar el cálculo previo en tiempo de compilación: cuando el tamaño del arreglo es dinámico. Si no conocemos los valores de *inferior* y *superior* (o sus generalizaciones en muchas dimensiones) en tiempo de compilación, entonces no podemos calcular constantes como *c*. Entonces, las fórmulas como (6.7) deben evaluarse como están escritas, cuando el programa se ejecuta.

Los anteriores cálculos de las direcciones se basan en la distribución por filas para los arreglos, la cual se utiliza en C y Java. Por lo general, un arreglo bidimensional se almacena en dos formas, ya sea en *orden por filas* (fila por fila) o *por columnas* (columna por columna). La figura 6.21 muestra la distribución de un arreglo A de 2×3 en (a) forma de orden por filas y (b) forma de orden por columnas. La forma de orden por columnas se utiliza en la familia de lenguajes Fortran.

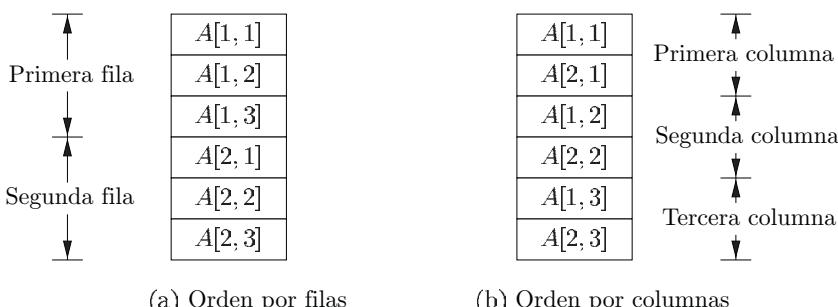


Figura 6.21: Distribuciones para un arreglo bidimensional

Podemos generalizar la forma de orden por filas o por columnas para muchas dimensiones. La generalización de la forma de orden por filas es para almacenar los elementos de tal forma que, a medida que exploramos en forma descendente un bloque de almacenamiento, los subíndices de más a la derecha parezcan variar más rápido, al igual que los números en un odómetro. La forma de orden por columnas se generaliza para la distribución opuesta, en donde los subíndices de más a la izquierda varían más rápido.

6.4.4 Traducción de referencias a arreglos

El principal problema al generar código para las referencias a arreglos es relacionar las fórmulas para calcular direcciones en la sección 6.4.3 a una gramática para referencias a arreglos. Suponga que el no terminal L genera un nombre de arreglo, seguido de una secuencia de expresiones de índices:

$$L \rightarrow L [E] \mid \mathbf{id} [E]$$

Como en C y Java, suponga que el elemento del arreglo con menor numeración es 0. Vamos a calcular las direcciones con base en los tamaños, mediante la fórmula (6.4) en vez de los números de elementos, como en (6.6). El esquema de traducción de la figura 6.22 genera código de tres direcciones para las expresiones con referencias a arreglos. Consiste en las producciones y las acciones semánticas de la figura 6.20, en conjunto con las producciones que involucran al no terminal L .

$$\begin{aligned}
 S \rightarrow & \mathbf{id} = E ; \quad \{ \text{gen}(\text{superior.get}(\mathbf{id}.lexema) ' = ' E.dir); \} \\
 & \mid L = E ; \quad \{ \text{gen}(L.dir.base '[' L.dir '] ' = ' E.dir); \} \\
 E \rightarrow & E_1 + E_2 \quad \{ E.dir = \mathbf{new} \text{ Temp}(); \\
 & \quad \text{gen}(E.dir ' = ' E_1.dir ' + ' E_2.dir); \} \\
 & \mid \mathbf{id} \quad \{ E.dir = \text{superior.get}(\mathbf{id}.lexema); \} \\
 & \mid L \quad \{ E.dir = \mathbf{new} \text{ Temp}(); \\
 & \quad \text{gen}(E.dir ' = ' L.arreglo.base '[' L.dir ']'); \} \\
 L \rightarrow & \mathbf{id} [E] \quad \{ L.arreglo = \text{superior.get}(\mathbf{id}.lexema); \\
 & \quad L.tipo = L.arreglo.tipo.elem; \\
 & \quad L.dir = \mathbf{new} \text{ Temp}(); \\
 & \quad \text{gen}(L.dir ' = ' E.dir ' * ' L.tipo.tamaño); \} \\
 & \mid L_1 [E] \quad \{ L.arreglo = L_1.arreglo; \\
 & \quad L.tipo = L_1.tipo.elem; \\
 & \quad t = \mathbf{new} \text{ Temp}(); \\
 & \quad L.dir = \mathbf{new} \text{ Temp}(); \\
 & \quad \text{gen}(t ' = ' E.dir ' * ' L.tipo.tamaño); \} \\
 & \quad \text{gen}(L.dir ' = ' L_1.dir ' + ' t); \}
 \end{aligned}$$

Figura 6.22: Acciones semánticas para las referencias a arreglos

El no terminal L tiene tres atributos sintetizados:

1. $L.dir$ denota un nombre temporal que se utiliza al calcular el desplazamiento para la referencia al arreglo, sumando los términos $i_j \times w_j$ en (6.4).

2. $L.\text{arreglo}$ es un apuntador a la entrada en la tabla de símbolos para el nombre del arreglo. La dirección base del arreglo, digamos $L.\text{arreglo}.base$, se usa para determinar el l -value actual de la referencia a un arreglo, después de analizar todas las expresiones de índices.
3. $L.\text{tipo}$ es el tipo del subarreglo generado por L . Para cualquier tipo t , asumimos que su tamaño se da mediante $t.\text{tamaño}$. Usamos tipos como atributos, en vez de tamaños, ya que los tipos se necesitan de todas formas para la comprobación de tipos. Para cualquier tipo de arreglo t , suponga que $t.\text{elem}$ proporciona el tipo del elemento.

La producción $S \rightarrow \mathbf{id} = E$; representa una asignación a una variable que no es un arreglo, la cual se maneja de la forma usual. La acción semántica para $S \rightarrow L = E$; genera una instrucción de copia indexada, para asignar el valor denotado por la expresión E a la ubicación denotada por la referencia al arreglo L . Recuerde que el atributo $L.\text{arreglo}$ proporciona la entrada en la tabla de símbolos para el arreglo. La dirección base del arreglo (la dirección de su 0-ésimo elemento) se proporciona mediante $L.\text{arreglo}.base$. El atributo $L.\text{dir}$ denota el nombre temporal que contiene el desplazamiento para la referencia al arreglo generada por L . Por lo tanto, la ubicación para la referencia al arreglo es $L.\text{arreglo}.base[L.\text{dir}]$. La instrucción generada copia el r -value de la dirección $E.\text{dir}$ en la ubicación para la referencia $L.\text{dir}$.

Las producciones $E \rightarrow E_1 + E_2$ y $E \rightarrow \mathbf{id}$ son las mismas que antes. La acción semántica para la nueva producción $E \rightarrow L$ genera código para copiar el valor de la ubicación denotada por L hacia un nuevo nombre temporal. Esta ubicación es $L.\text{arreglo}.base[L.\text{dir}]$, como vimos antes para la producción $S \rightarrow L = E$; . De nuevo, el atributo $L.\text{arreglo}$ proporciona el nombre del arreglo, y $L.\text{arreglo}.base$ proporciona su dirección base. El atributo $L.\text{dir}$ denota el nombre temporal que contiene el desplazamiento. El código para la referencia al arreglo coloca el r -value en la ubicación designada por la base y el desplazamiento en un nuevo nombre temporal, denotado por $E.\text{dir}$.

Ejemplo 6.12: Suponga que a denota un arreglo de 2×3 de enteros, y que c , i y j denotan enteros. Entonces, el tipo de a es $\text{arreglo}(2, \text{arreglo}(3, \text{integer}))$. Su anchura w es 24, asumiendo que la anchura de un entero es 4. El tipo de $a[i]$ es $\text{arreglo}(3, \text{integer})$, de anchura $w_1 = 12$. El tipo de $a[i][j]$ es integer .

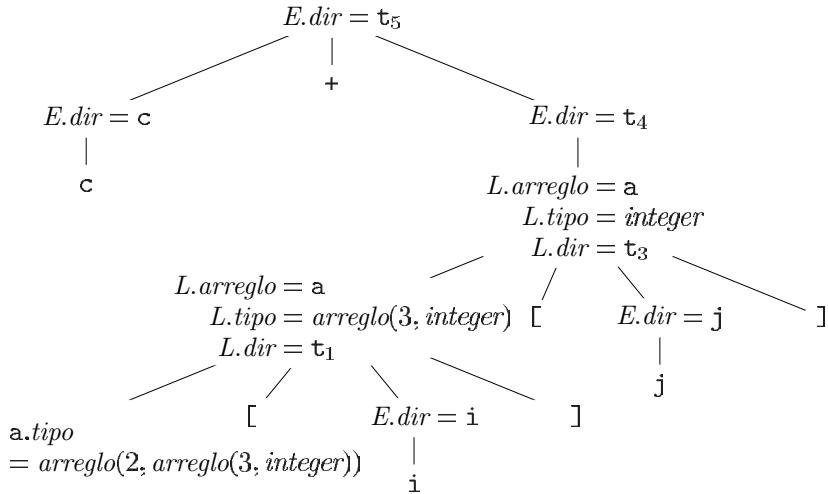
En la figura 6.23 se muestra un árbol de análisis sintáctico anotado para la expresión $c+a[i][j]$. La expresión se traduce en la secuencia de instrucciones de tres direcciones en la figura 6.24. Como siempre, hemos usado el nombre de cada identificador para referirnos a su entrada en la tabla de símbolos. \square

6.4.5 Ejercicios para la sección 6.4

Ejercicio 6.4.1: Agregue a la traducción de la figura 6.19 reglas para las siguientes producciones:

- a) $E \rightarrow E_1 * E_2$.
- b) $E \rightarrow + E_1$ (suma unaria).

Ejercicio 6.4.2: Repita el ejercicio 6.4.1 para la traducción incremental de la figura 6.20.

Figura 6.23: Árbol de análisis sintáctico anotado para $c + a[i] [j]$

```

 $t_1 = i * 12$ 
 $t_2 = j * 4$ 
 $t_3 = t_1 + t_2$ 
 $t_4 = a [ t_3 ]$ 
 $t_5 = c + t_4$ 

```

Figura 6.24: Código de tres direcciones para la expresión $c + a[i] [j]$

Ejercicio 6.4.3: Use la traducción de la figura 6.22 para traducir las siguientes asignaciones:

- $x = a[i] + b[j]$.
- $x = a[i] [j] + b[i] [j]$.
- $x = a[b[i] [j]] [c[k]]$.

! Ejercicio 6.4.4: Modifique la traducción de la figura 6.22 para referencias a arreglos del estilo Fortran; es decir, $\mathbf{id}[E_1, E_2, \dots, E_n]$ para un arreglo n -dimensional.

Ejercicio 6.4.5: Generalice la fórmula (6.7) para arreglos multidimensionales, e indique qué valores pueden almacenarse en la tabla de símbolos y utilizarse para calcular desplazamientos. Considere los siguientes casos:

- Un arreglo A de dos dimensiones, en forma de orden por filas. La primera dimensión tiene índices que van de i_1 a s_1 , y la segunda dimensión tiene índices de i_2 a s_2 . La anchura de un solo elemento del arreglo es w .

Anchuras de tipos simbólicos

El código intermedio debería ser relativamente independiente de la máquina destino, de manera que el optimizador no tenga que cambiar mucho si el generador de código se sustituye por uno para una máquina distinta. Sin embargo, como hemos descrito el cálculo de los tamaños de los tipos, hay una suposición en relación con los tipos básicos integrada en el esquema de traducción. Por ejemplo, el ejemplo 6.12 supone que cada elemento de un arreglo de enteros ocupa cuatro bytes. Algunos códigos intermedios, por ejemplo, P-code para Pascal, dejan al generador de código la función de llenar el tamaño de los elementos de los arreglos, por lo que el código intermedio es independiente del tamaño de una palabra de máquina. Podríamos haber hecho lo mismo en nuestro esquema de traducción si sustituyéramos el 4 (como la anchura de un entero) por una constante simbólica.

- b) Igual que (a), pero con el arreglo almacenado en forma de orden por columnas.
- ! c) Un arreglo A de k dimensiones, almacenado en forma de orden por filas, con elementos de tamaño w . La j -ésima dimensión tiene índices que van desde i_j hasta s_j .
- ! d) Igual que (c), pero con el arreglo almacenado en forma de orden por columnas.

Ejercicio 6.4.6: Un arreglo de enteros $A[i, j]$ tiene el índice i que varía de 1 a 10, y el índice j que varía de 1 a 20. Los enteros ocupan 4 bytes cada uno. Suponga que el arreglo a se almacena empezando en el byte 0. Encuentre la ubicación de:

- a) $A[4, 5]$
- b) $A[10, 8]$
- c) $A[3, 17]$.

Ejercicio 6.4.7: Repita el ejercicio 6.4.6 si A está almacenado en orden por columnas.

Ejercicio 6.4.8: Un arreglo de números reales $A[i, j, k]$ tiene el índice i que varía de 1 a 4, el índice j que varía de 0 a 4 y el índice k que varía de 5 a 10. Los números reales ocupan 8 bytes cada uno. Suponga que el arreglo A se almacena empezando en el byte 0. Encuentre la ubicación de:

- a) $A[3, 4, 5]$
- b) $A[1, 2, 7]$
- c) $A[4, 3, 9]$.

Ejercicio 6.4.9: Repita el ejercicio 6.4.8 si A está almacenado en orden por columnas.

6.5 Comprobación de tipos

Para realizar la *comprobación de tipos*, un compilador debe asignar una expresión de tipos a cada componente del programa fuente. Después, el compilador debe determinar que estas expresiones de tipos se conforman a una colección de reglas lógicas, conocida como el *sistema de tipos* para el lenguaje fuente.

La comprobación de tipos tiene el potencial de atrapar errores en los programas. En principio, cualquier comprobación puede realizarse en forma dinámica, si el código de destino lleva el

tipo de un elemento, junto con el valor del elemento. Un sistema de tipos *sólido* elimina la necesidad de la comprobación dinámica para los errores de tipos, ya que nos permite determinar en forma estática que estos errores no pueden ocurrir cuando se ejecuta el programa de destino. Una implementación de un lenguaje está *fuertemente tipificada* si un compilador garantiza que los programas que acepta se ejecutarán sin errores.

Además de su uso para la compilación, las ideas de la comprobación de tipos se han utilizado para mejorar la seguridad de los sistemas que permiten la importación y ejecución de módulos de software. Los programas en Java se compilan en bytecodes independientes de la máquina, que incluyen información detallada sobre los tipos, en relación con las operaciones en los bytecodes. El código importado se verifica antes de permitirle que se ejecute, para protegerse contra los errores inadvertidos y el comportamiento malicioso.

6.5.1 Reglas para la comprobación de tipos

La comprobación de tipos puede tomar dos formas: síntesis e inferencia. La *síntesis de tipos* construye el tipo de una expresión a partir de los tipos de sus subexpresiones. Requiere que se declaren los nombres antes de utilizarlos. El tipo de $E_1 + E_2$ se define en términos de los tipos de E_1 y E_2 . Una regla común para la síntesis de tipos tiene la siguiente forma:

if f tiene el tipo $s \rightarrow t$ **and** x tiene el tipo s ,
then la expresión $f(x)$ tiene el tipo t

(6.8)

Aquí, f y x denotan expresiones, y $s \rightarrow t$ denota una función de s a t . Esta regla para las funciones con un argumento se pasa a las funciones con varios argumentos. La regla (6.8) puede adaptarse para $E_1 + E_2$ si la vemos como una aplicación de la función $sumar(E_1, E_2)$.⁶

La *inferencia de tipos* determina el tipo de una construcción del lenguaje a partir de la forma en que se utiliza. Adelantándonos hasta los ejemplos en la sección 6.5.4, supongamos que *null* sea una función que evalúe si una lista está vacía. Entonces, del uso $null(x)$, podemos determinar que x debe ser una lista. No se conoce el tipo de los elementos de x ; todo lo que sabemos es que x debe ser una lista de elementos de algún tipo que hasta el momento se desconoce.

Las variables que representan expresiones de tipos nos permiten hablar acerca de los tipos desconocidos. Vamos a usar las letras griegas α, β, \dots para las variables de los tipos en las expresiones de tipos.

Una regla común para la inferencia de tipos tiene la siguiente forma:

if $f(x)$ es una expresión,
then para cierta α y β , f tiene el tipo $\alpha \rightarrow \beta$ **and** x tiene el tipo α

(6.9)

La inferencia de tipos es necesaria para lenguajes como ML, que comprueban los tipos pero no requieren la declaración de nombres.

⁶Vamos a usar el término “síntesis”, incluso aunque se utilice cierta información de contexto para determinar los tipos. Con la sobrecarga de funciones, en donde se da el mismo nombre a más de una función, tal vez también haya que considerar el contexto de $E_1 + E_2$ en algunos lenguajes.

En esta sección, consideraremos la comprobación de tipos de las expresiones. Las reglas para comprobar instrucciones son similares a las de las expresiones. Por ejemplo, tratamos la instrucción condicional “**if**(*E*)*S*;” como si fuera la aplicación de una función *if* a *E* y a *S*. Supongamos que el tipo especial *void* denota la ausencia de un valor. Entonces, la función *if* espera aplicarse a un valor *boolean* y a un *void*; el resultado de la aplicación es un *void*.

6.5.2 Conversiones de tipos

Considere las expresiones como *x* + *i*, en donde *x* es de tipo punto flotante, e *i* es de tipo entero. Como la representación de enteros y números de punto flotante es distinta dentro de una computadora, y se utilizan distintas instrucciones de máquina para las operaciones con enteros y números de punto flotante, tal vez el compilador tenga que convertir uno de los operandos de + para asegurar que ambos operandos sean del mismo tipo cuando ocurra la suma.

Suponga que los enteros se convierten a números de punto flotante cuando es necesario, usando un operador unario (*float*). Por ejemplo, el entero 2 se convierte a un número de punto flotante en el código para la expresión 2 * 3.14:

```
t1 = (float) 2
t2 = t1 * 3.14
```

Podemos extender dichos ejemplos para considerar versiones enteras y de punto flotante de los operadores; por ejemplo, **int*** para los operandos enteros y **float*** para los operandos de punto flotante.

En la sección 6.4.2 ilustraremos la síntesis de tipos, extendiendo el esquema para traducir expresiones. Presentaremos otro atributo *E.tipo*, cuyo valor es *integer* o *float*. La regla asociada con *E* → *E*₁ + *E*₂ se basa en el siguiente seudocódigo:

```
if ( E1.tipo = integer and E2.tipo = integer ) E.tipo = integer;
else if ( E1.tipo = float and E2.tipo = integer ) ...
...
```

A medida que se incrementa el número de tipos sujetos a conversión, el número de casos aumenta con rapidez. Por lo tanto, con números extensos de tipos, es importante una organización cuidadosa de las acciones semánticas.

Las reglas de conversión de tipos varían de un lenguaje a otro. Las reglas para Java en la figura 6.25 hacen distinciones entre las conversiones de *ampliación*, que tienen el propósito de preservar la información, y las conversiones de *reducción*, que pueden perder información. Las reglas de ampliación se proporcionan mediante la jerarquía en la figura 6.25(a): cualquier tipo con un nivel menor en la jerarquía puede ampliarse a un tipo con un nivel mayor. Por ende, un *char* puede ampliarse a un *int* o a un *float*, pero un *char* no puede ampliarse a un *short*. Las reglas de reducción se ilustran mediante el grafo en la figura 6.25(b): un tipo *s* puede reducirse a un tipo *t* si hay un camino de *s* a *t*. Observe que *char*, *short* y *byte* pueden convertirse entre sí.

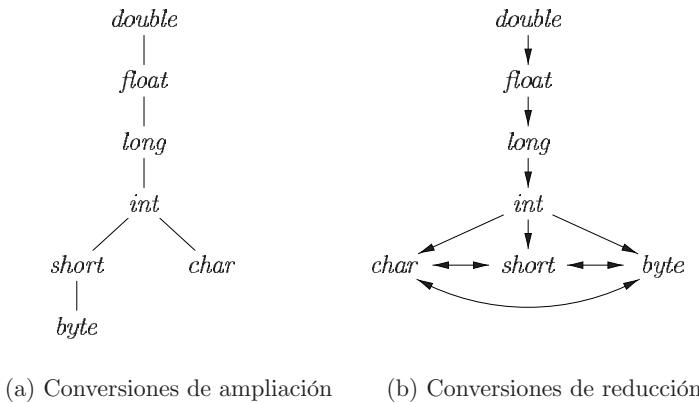


Figura 6.25: Conversiones entre tipos primitivos de Java

Se dice que la conversión de un tipo a otro es *implícita* si el compilador la realiza en forma automática. Las conversiones de tipo implícitas, también conocidas como *coerciones*, están limitadas en muchos lenguajes a las conversiones de ampliación. Se dice que la conversión es *explícita* si el programador debe escribir algo para provocar la conversión. A las conversiones explícitas también se les conoce como *conversiones* o *casts*.

La acción semántica para comprobar $E \rightarrow E_1 + E_2$ utiliza dos funciones:

1. $max(t_1, t_2)$ recibe dos tipos t_1 y t_2 , y devuelve el máximo (o el límite superior) de los dos tipos en la jerarquía de ampliación. Declara un error si t_1 o t_2 no se encuentran en la jerarquía; por ejemplo, si cualquiera de los dos tipos es un tipo de arreglo o de apuntador.
2. $ampliar(a, t, w)$ genera conversiones de tipos, si es necesario, para ampliar una dirección a de tipo t en una variable de tipo w . Devuelve la misma a si t y w son del mismo tipo. En cualquier otro caso, genera una instrucción para realizar la conversión y colocar el resultado en una t temporal, que se devuelve como el resultado. El seudocódigo para *ampliar*, suponiendo que los únicos tipos son *integer* y *float*, aparece en la figura 6.26.

```

Dir ampliar( Dir a, Tipo t, Tipo w)
  if ( t = w ) return a;
  else if ( t = integer and w = float ) {
    temp = new Temp();
    gen( temp '=' '(float)' a );
    return temp;
  }
  else error;
}

```

Figura 6.26: Seudocódigo para la función *ampliar*

La acción semántica para $E \rightarrow E_1 + E_2$ en la figura 6.27 ilustra cómo pueden agregarse conversiones de tipos al esquema en la figura 6.20 para traducir expresiones. En la acción semántica, la variable temporal a_1 es $E_1.dir$ si el tipo de E_1 no necesita convertirse al tipo de E , o una nueva variable temporal devuelta por *ampliar* si esta conversión es necesaria. De manera similar, a_2 es $E_2.dir$ o un nuevo valor temporal que contiene el valor de E_2 con el tipo convertido. Ninguna conversión es necesaria si ambos tipos son *integer* o ambos son *float*. Sin embargo, en general podríamos descubrir que la única forma de sumar valores de dos tipos distintos es convertir ambos en un tercer tipo.

```


$$E \rightarrow E_1 + E_2 \quad \{ \begin{aligned} E.tipo &= \max(E_1.tipo, E_2.tipo); \\ a_1 &= \text{ampliar}(E_1.dir, E_1.tipo, E.tipo); \\ a_2 &= \text{ampliar}(E_2.dir, E_2.tipo, E.tipo); \\ E_1.dir &= \mathbf{new} \text{ Temp}(); \\ \text{gen}(E.dir' = a_1' +' a_2); \end{aligned} \}$$


```

Figura 6.27: Introducción de conversiones de tipos en la evaluación de expresiones

6.5.3 Sobrecarga de funciones y operadores

Una *sobrecarga* de un símbolo tiene diferentes significados, dependiendo de su contexto. La sobrecarga se *resuelve* cuando se determina un significado único para cada ocurrencia de un nombre. En esta sección nos enfocaremos en la sobrecarga que puede resolverse con sólo ver los argumentos de una función, como en Java.

Ejemplo 6.13: El operador `+` en Java denota la concatenación de cadenas o la suma, dependiendo de los tipos de sus operandos. Las funciones definidas por el usuario pueden sobrecargarse también, como en

```

void err() { ... }
void err(String s) { ... }

```

Observe que podemos elegir una de estas dos versiones de una función `err` con sólo ver sus argumentos. \square

A continuación se muestra una regla de síntesis de tipos para funciones sobrecargadas:

if f puede tener el tipo $s_i \rightarrow t_i$ para $1 \leq i \leq n$, en donde $s_i \neq s_j$ para $i \neq j$
and x tiene el tipo s_k , para alguna $1 \leq k \leq n$
then la expresión $f(x)$ tiene el tipo t_k (6.10)

El método número de valor de la sección 6.1.2 puede aplicarse a las expresiones de tipos para resolver la sobrecarga con base en los tipos de los argumentos, de una forma eficiente. En un GDA que representa a una expresión de tipos, asignamos un índice entero, conocido como

número de valor, a cada nodo. Mediante el uso del Algoritmo 6.3 construimos una firma para un nodo, que consiste en su etiqueta y los números de valor de sus hijos, en orden de izquierda a derecha. La firma para una función consiste en el nombre de la función y los tipos de sus argumentos. La suposición de que podemos resolver la sobrecarga con base en los tipos de los argumentos es equivalente a decir que podemos resolver la sobrecarga con base en las firmas.

No siempre es posible resolver la sobrecarga con sólo ver los argumentos de una función. En Ada, en vez de un tipo simple, una subexpresión independiente puede tener un conjunto de posibles tipos, para los cuales el contexto debe proporcionar suficiente información para reducir las opciones a un tipo simple (vea el ejercicio 6.5.2).

6.5.4 Inferencia de tipos y funciones polimórficas

La inferencia de tipos es útil para un lenguaje como ML, el cual es fuertemente tipificado, pero no requiere que se declaren los nombres antes de utilizarlos. La inferencia de tipos asegura que los nombres se utilicen en forma consistente.

El término “polimórfico” se refiere a cualquier fragmento de código que puede ejecutarse con argumentos de distintos tipos. En esta sección veremos el *polimorfismo paramétricos*, en donde el polimorfismo se caracteriza por parámetros o variables de tipo. El ejemplo abierto es el programa de ML en la figura 6.28, el cual define la función *longitud*. El tipo de *longitud* puede describirse como, “para cualquier tipo α , *longitud* asigna una lista de elementos de tipo α a un entero”.

```
fun longitud(x) =
  if null(x) then 0 else longitud(tl(x)) + 1;
```

Figura 6.28: Programa de ML para la longitud de una lista

Ejemplo 6.14: En la figura 6.28, la palabra clave **fun** introduce la definición de una función; las funciones pueden ser recursivas. El fragmento del programa define la función *longitud* con un parámetro x . El cuerpo de la función consiste en una expresión condicional. La función predefinida *null* evalúa si una lista está vacía, y la función predefinida *tl* (abreviación de “cola” en inglés) devuelve el residuo de una lista, una vez que se elimina el primer elemento.

La función *longitud* determina la longitud o el número de elementos de una lista x . Todos los elementos de una lista deben tener el mismo tipo, pero *longitud* puede aplicarse a las listas cuyos elementos sean de cualquier tipo. En la siguiente expresión, *longitud* se aplica a dos tipos distintos de listas (los elementos de las listas se encierran entre “[” y “”]):

$$\text{longitud}([\text{"dom"}, \text{"lun"}, \text{"mar"}]) + \text{longitud}([10, 9, 8, 7]) \quad (6.11)$$

La lista de cadenas tiene longitud de 3 y la lista de enteros tiene longitud de 4, por lo que la expresión (6.11) se evalúa en 7. \square

Si utilizamos el símbolo \forall (que significa “para cualquier tipo”) y el constructor de tipos *lista*, el tipo de *longitud* puede escribirse como

$$\forall\alpha. \text{lista}(\alpha) \rightarrow \text{integer} \quad (6.12)$$

El símbolo \forall es el *cuantificador universal*, y se dice que la variable de tipo a la cual se aplica está *enlazada* por este símbolo. Las variables enlazadas pueden renombrarse a voluntad, siempre y cuando se renombren todas las ocurrencias de la variable. Por ende, la expresión de tipos

$$\forall\beta. \text{lista}(\beta) \rightarrow \text{integer}$$

es equivalente a (6.12). A una expresión de tipos con un símbolo \forall se le conoce de manera informal como un “tipo polimórfico”.

Cada vez que se aplica una función polimórfica, sus variables de tipo enlazadas pueden denotar un tipo distinto. Durante la comprobación de tipos, en cada uso de un tipo polimórfico sustituimos las variables enlazadas por variables nuevas y eliminamos los cuantificadores universales.

El siguiente ejemplo infiere de manera informal un tipo para *longitud*, usando en forma implícita las reglas de inferencia de tipos como (6.9), que repetimos a continuación:

if $f(x)$ es una expresión,
then para alguna α y β , f tiene el tipo $\alpha \rightarrow \beta$ **and** x tiene el tipo α

Ejemplo 6.15: El árbol sintáctico abstracto de la figura 6.29 representa la definición de *longitud* en la figura 6.28. La raíz del árbol, etiquetada como **fun**, representa a la definición de la función. El resto de los nodos que no son hojas pueden verse como aplicaciones de funciones. El nodo etiquetado como **+** representa la aplicación del operador **+** a un par de hijos. De manera similar, el nodo etiquetado como **if** representa la aplicación de un operador **if** a una tripleta formado por sus hijos (para la comprobación de tipos, no importa si se va a evaluar la parte **then** o **else**, pero no ambas).

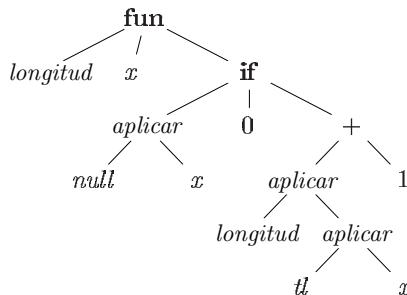


Figura 6.29: Árbol de sintaxis abstracto para la definición de la función de la figura 6.28

Del cuerpo de la función *longitud* podemos inferir su tipo. Considere los hijos del nodo etiquetado como **if**, de izquierda a derecha. Como *null* espera aplicarse a listas, *x* debe ser una lista. Vamos a usar la variable α como receptáculo para el tipo de los elementos de la lista; es decir, *x* tiene el tipo “lista de α ”.

Sustituciones, instancias y unificación

Si t es una expresión de tipos y S es una sustitución (una asignación de variables de tipos a expresiones de tipos), entonces escribimos $S(t)$ para el resultado de sustituir de manera consistente todas las ocurrencias de cada variable de tipo α en t por $S(\alpha)$. $S(t)$ se llama *instancia* de t . Por ejemplo, *lista(integer)* es una instancia de *lista(α)*, ya que es el resultado de sustituir *integer* por α en *lista(α)*. Sin embargo, observe que *integer → float* no es una instancia de $\alpha \rightarrow \alpha$, ya que una sustitución debe reemplazar todas las ocurrencias de α por la misma expresión de tipos.

La sustitución S es un *unificador* de las expresiones de tipos t_1 y t_2 si $S(t_1) = S(t_2)$. S es el *unificador más general* de t_1 y t_2 si para cualquier otro unificador de t_1 y t_2 , por decir S' , se da el caso de que para cualquier t , $S'(t)$ es una instancia de $S(t)$. En palabras, S' impone más restricciones sobre t que S .

Si *null(x)* es verdadera, entonces *longitud(x)* es 0. Por ende, el tipo de *longitud* debe ser “función de la lista de α a entero”. Este tipo inferido es consistente con el uso de *longitud* en la parte del *else*, *longitud(tl(x)) + 1*. \square

Como las variables pueden aparecer en las expresiones de tipos, tenemos que volver a examinar la noción de la equivalencia de tipos. Suponga que E_1 del tipo $s \rightarrow s'$ se aplica a E_2 del tipo t . En vez de sólo determinar la igualdad de s y t , debemos “unificarlos”. De manera informal, determinamos si s y t pueden hacerse equivalentes en estructura mediante la sustitución de las variables de tipos en s y t por expresiones de tipos.

Una *sustitución* es una asignación de las variables de tipos a las expresiones de tipos. Escribimos $S(t)$ para el resultado de aplicar la sustitución S a las variables en la expresión de tipos t ; consulte el recuadro titulado “Sustituciones, instancias y unificación”. Dos expresiones de tipos t_1 y t_2 se *unifican* si existe alguna sustitución S tal que $S(t_1) = S(t_2)$. En la práctica, estamos interesados en el unificador más general, el cual es una sustitución que impone la menor cantidad de restricciones sobre las variables en las expresiones. En la sección 6.5.5 podrá consultar un algoritmo de unificación.

Algoritmo 6.16: Inferencia de tipos para funciones polimórficas.

ENTRADA: Un programa que consiste en una secuencia de definiciones de funciones, seguidas por una expresión a evaluar. Una expresión está compuesta de aplicaciones de funciones y nombres, en donde los nombres pueden tener tipos polimórficos predefinidos.

SALIDA: Los tipos inferidos para los nombres en el programa.

MÉTODO: Por simplicidad, vamos a tratar sólo con funciones unarias. El tipo de una función $f(x_1, x_2)$ con dos parámetros puede representarse mediante una expresión de tipos $s_1 \times s_2 \rightarrow t$, en donde s_1 y s_2 son los tipos de x_1 y x_2 , respectivamente, y t es el tipo del resultado $f(x_1, x_2)$. Para comprobar una expresión $f(a, b)$ tenemos que relacionar el tipo de a con s_1 y el tipo de b con s_2 .

Compruebe las definiciones de funciones y la expresión en la secuencia de entrada. Use el tipo inferido de una función si se utiliza subsecuentemente en una expresión.

- Para una definición de función $\mathbf{fun} \, \mathbf{id}_1(\mathbf{id}_2) = E$, cree nuevas variables de tipo α y β . Asocie el tipo $\alpha \rightarrow \beta$ con la función \mathbf{id}_1 y el tipo α con el parámetro \mathbf{id}_2 . Después, infiera un tipo para la expresión E . Suponga que α denota al tipo s y β denota al tipo t después de la inferencia de tipos para E . El tipo inferido de la función \mathbf{id}_1 es $s \rightarrow t$. Enlace cualquier variable de tipo que permanezca libre en $s \rightarrow t$ mediante cuantificadores \forall .
- Para una aplicación de función $E_1 (E_2)$, infiera los tipos para E_1 y E_2 . Ya que E_1 se utiliza como una función, su tipo debe tener la forma $s \rightarrow s'$. Técnicamente, el tipo de E_1 debe unificarse con $\beta \rightarrow \gamma$, en donde β y γ son variables de tipos nuevas. Deje que t sea el tipo inferido de E_1 . Unifique a s y t . Si la unificación falla, la expresión tiene un error de tipo. En cualquier otro caso, el tipo inferido de $E_1(E_2)$ es s' .
- Para cada ocurrencia de una función polimórfica, sustituya las variables enlazadas en su tipo por variables nuevas distintas y elimine los cuantificadores \forall . La expresión de tipos resultante es el tipo inferido de esta ocurrencia.
- Para un nombre que se encuentre por primera vez, introduzca una nueva variable para su tipo.

□

Ejemplo 6.17: En la figura 6.30, inferimos un tipo para la función *longitud*. La raíz del árbol sintáctico en la figura 6.29 es para una definición de función, por lo que introducimos las variables β y γ , asociamos el tipo $\beta \rightarrow \gamma$ con la función *longitud* y el tipo β con x ; vea las líneas 1-2 de la figura 6.30.

En el hijo derecho de la raíz, vemos a **if** como una función polimórfica que se aplica a una tripleta, la cual consiste en un valor booleano y dos expresiones que representan las partes **then** y **else**. Su tipo es $\forall \alpha. \, \text{boolean} \times \alpha \times \alpha \rightarrow \alpha$.

Cada aplicación de una función polimórfica puede ser a un tipo distinto, por lo que creamos una nueva variable α_i (en donde i es de “if”) y eliminamos el \forall ; vea la línea 3 de la figura 6.30. El tipo del hijo izquierdo de **if** debe unificarse con *boolean*, y los tipos de sus otros dos hijos deben unificarse con α .

La función predefinida *null* tiene el tipo $\forall \alpha. \, \text{lista}(\alpha) \rightarrow \text{boolean}$. Usamos una nueva variable de tipos α_n (en donde n es de “null”) en vez de la variable enlazada α ; vea la línea 4. De la aplicación de *null* a x , podemos inferir que el tipo β de x debe coincidir con $\text{lista}(\alpha_n)$; vea la línea 5.

En el primer hijo de **if**, el tipo *boolean* para *null(x)* coincide con el tipo esperado por **if**. En el segundo hijo, el tipo α_i se unifica con *integer*; vea la línea 6.

Ahora, considere la subexpresión $\text{longitud}(\text{tl}(x)) + 1$. Creamos una nueva variable α_t (en donde t es de “tail” [cola]) para la variable enlazada α en el tipo de *tl*; vea la línea 8. De la aplicación *tl(x)*, inferimos que $\text{lista}(\alpha_t) = \beta = \text{lista}(\alpha_n)$; vea la línea 9.

Como $\text{longitud}(\text{tl}(x))$ es un operando de $+$, su tipo γ debe unificarse con *integer*; vea la línea 10. Se deduce que el tipo de *longitud* es $\text{lista}(\alpha_n) \rightarrow \text{integer}$. Una vez que se comprueba

LÍNEA	EXPRESIÓN : TIPO	UNIFICAR
1)	$longitud : \beta \rightarrow \gamma$	
2)	$x : \beta$	
3)	$\mathbf{if} : boolean \times \alpha_i \times \alpha_i \rightarrow \alpha_i$	
4)	$null : lista(\alpha_n) \rightarrow boolean$	
5)	$null(x) : boolean$	$lista(\alpha_n) = \beta$
6)	$0 : integer$	$\alpha_i = integer$
7)	$+ : integer \times integer \rightarrow integer$	
8)	$tl : lista(\alpha_t) \rightarrow lista(\alpha_t)$	
9)	$tl(x) : lista(\alpha_t)$	$lista(\alpha_t) = lista(\alpha_n)$
10)	$longitud(tl(x)) : \gamma$	$\gamma = integer$
11)	$1 : integer$	
12)	$longitud(tl(x)) + 1 : integer$	
13)	$\mathbf{if}(\dots) : integer$	

Figura 6.30: Inferencia de un tipo para la función *longitud* de la figura 6.28

la definición de la función, la variable de tipo α_n permanece en el tipo de *longitud*. Como no se hicieron suposiciones acerca de α_n , puede sustituir a cualquier tipo cuando se utiliza la función. Por lo tanto, creamos una variable enlazada y escribimos lo siguiente:

$$\forall \alpha_n. lista(\alpha_n) \rightarrow integer$$

para el tipo de *longitud*. \square

6.5.5 Un algoritmo para la unificación

De manera informal, la unificación es el problema de determinar si dos expresiones s y t pueden hacerse idénticas mediante la sustitución de expresiones por las variables en s y t . La prueba de igualdad de las expresiones es un caso especial de unificación; si s y t tienen constantes pero no variables, entonces s y t se unifican si y sólo si son idénticas. El algoritmo de unificación en esta sección se extiende a los grafos con ciclos, por lo que puede utilizarse para evaluar la equivalencia estructural de los tipos circulares.⁷

Vamos a implementar una formulación gráfica-teórica de unificación, en donde los tipos se representan mediante grafos. Las variables de tipos se representan por hojas y los constructores de tipos se representan mediante nodos interiores. Los nodos se agrupan en clases de equivalencia; si dos nodos se encuentran en la misma clase de equivalencia, entonces las expresiones de tipos que representan deben unificarse. Por ende, todos los nodos interiores en la misma clase deben ser para el mismo constructor de tipo, y sus correspondientes hijos deben ser equivalentes.

Ejemplo 6.18: Considere las dos expresiones de tipos siguientes:

⁷En algunas aplicaciones, es un error unificar una variable con una expresión que contenga a esa variable. El algoritmo 6.19 permite dichas sustituciones.

$$\begin{aligned} ((\alpha_1 \rightarrow \alpha_2) \times lista(\alpha_3)) &\rightarrow lista(\alpha_2) \\ ((\alpha_3 \rightarrow \alpha_4) \times lista(\alpha_3)) &\rightarrow \alpha_5 \end{aligned}$$

La siguiente sustitución S es el unificador más general para estas expresiones:

x	$S(x)$
α_1	α_1
α_2	α_2
α_3	α_1
α_4	α_2
α_5	$lista(\alpha_2)$

Esta sustitución asigna las dos expresiones de tipos a la siguiente expresión:

$$((\alpha_1 \rightarrow \alpha_2) \times lista(\alpha_1)) \rightarrow lista(\alpha_2)$$

Las dos expresiones se representan mediante los dos nodos etiquetados como $\rightarrow: 1$ en la figura 6.31. Los enteros en los nodos indican las clases de equivalencias a las que pertenecen los nodos, una vez que se unifican los nodos enumerados con el 1. \square

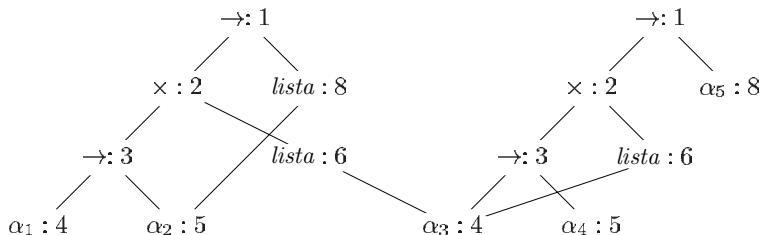


Figura 6.31: Clases de equivalencias después de la unificación

Algoritmo 6.19: Unificación de un par de nodos en un grafo de tipos.

ENTRADA: Un grafo que representa a un tipo y un par de nodos m y n que se van a unificar.

SALIDA: El valor booleano verdadero si las expresiones representadas por los nodos m y n se unifican; falso en cualquier otro caso.

MÉTODO: Se implementa un nodo mediante un registro con campos para un operador binario y apunadores a los hijos izquierdo y derecho. Los conjuntos de nodos equivalentes se mantienen usando el campo *set*. Se elige un nodo en cada clase de equivalencia como el único representante de la clase de equivalencia, haciendo que su campo *set* contenga un apuntador nulo. Los campos *set* de los nodos restantes en la clase de equivalencia apuntarán (quizá en forma indirecta a través de otros nodos en el conjunto) al representante. Al principio, cada nodo n es una clase de equivalencia por sí solo, con n como su propio nodo representativo.

```

boolean unificar(Nodo m, Nodo n) {
    s = buscar(m); t = buscar(n);
    if ( s = t ) return true;
    else if ( los nodos s y t representan el mismo tipo básico ) return true;
    else if ( s es un nodo op con los hijos s1 y s2 and
                t es un nodo op con los hijos t1 y t2 ) {
        union(s, t);
        return unificar(s1, t1) and unificar(s2, t2);
    }
    else if s o t representa a una variable {
        union(s, t);
        return true;
    }
    else return false;
}

```

Figura 6.32: Algoritmo de unificación

El algoritmo de unificación, que se muestra en la figura 6.32, utiliza las siguientes dos operaciones con los nodos:

- $buscar(n)$ devuelve el nodo representativo de la clase de equivalencia que contiene actualmente el nodo n .
- $union(m, n)$ combina las clases de equivalencias que contienen los nodos m y n . Si uno de los representantes para las clases de equivalencias de m y n es un nodo no variable, $union$ convierte a ese nodo no variable en el representante para la clase de equivalencia combinada; en caso contrario, $union$ convierte a uno de los dos representantes originales en el nuevo representante. Esta asimetría en la especificación de $union$ es importante, ya que una variable no puede utilizarse como representante de una clase de equivalencia para una expresión que contenga un constructor de tipos o un tipo básico. De lo contrario, dos expresiones equivalentes podrían unificarse a través de esa variable.

La operación $union$ en los conjuntos se implementa con sólo cambiar el campo *set* del representante de una clase de equivalencia para que apunte al representante del otro. Para encontrar la clase de equivalencia a la que pertenece un nodo, seguimos los apuntadores *set* de los nodos hasta llegar al representante (el nodo con un apuntador nulo en el campo *set*).

Observe que el algoritmo en la figura 6.32 utiliza $s = buscar(m)$ y $t = buscar(n)$ en vez de m y n , respectivamente. Los nodos representantes s y t son iguales si m y n están en la misma clase de equivalencia. Si s y t representan el mismo tipo básico, la llamada a $unificar(m, n)$ devuelve verdadero. Si s y t son nodos interiores para un constructor de tipos binarios, combinamos sus clases de equivalencias en base a la especulación y comprobamos en forma recursiva que sus respectivos hijos sean equivalentes. Al combinar primero, reducimos el número de clases de equivalencias antes de comprobar los hijos en forma recursiva, por lo que el algoritmo termina.

La sustitución de una expresión por una variable se implementa agregando la hoja para la variable a la clase de equivalencia que contiene el nodo para esa expresión. Suponga que m o n son una hoja para una variable. Suponga también que esta hoja se ha colocado en una clase de equivalencia con un nodo que representa a una expresión con un constructor de tipos o un tipo básico. Entonces, *buscar* devolverá un representante que refleje el constructor de tipos o el tipo básico, de manera que una variable no pueda unificarse con dos expresiones distintas. \square

Ejemplo 6.20: Suponga que las dos expresiones en el ejemplo 6.18 se representan mediante el grafo inicial en la figura 6.33, en donde cada nodo está en su propia clase de equivalencia. Cuando se aplica el Algoritmo 6.19 para calcular *unificar*(1, 9), observa que los nodos 1 y 9 representan al mismo operador. Por lo tanto, combina a 1 y 9 en la misma clase de equivalencia y llama *unificar*(2, 10) y a *unificar*(8, 14). El resultado de calcular *unificar*(1, 9) es el grafo que se mostró antes en la figura 6.31. \square

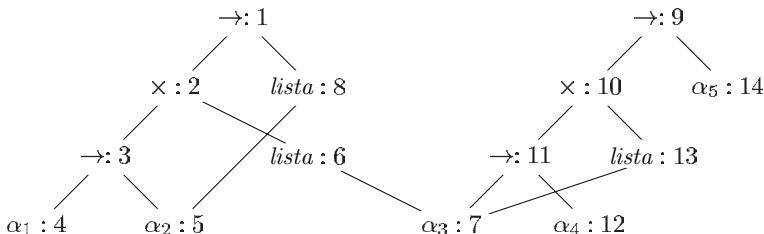


Figura 6.33: Grafo inicial con cada nodo en su propia clase de equivalencia

Si el Algoritmo 6.19 devuelve verdadero, podemos construir una sustitución S que actúe como el unificador, como se muestra a continuación. Para cada variable α , *buscar*(α) proporciona el nodo n que representa a la clase de equivalencia de α . La expresión representada por n es $S(\alpha)$. Por ejemplo, en la figura 6.31 podemos ver que el representante para α_3 es el nodo 4, que representa a α_1 . El representante para α_5 es el nodo 8, que representa a $lista(\alpha_2)$. La sustitución S resultante es como en el ejemplo 6.18.

6.5.6 Ejercicios para la sección 6.5

Ejercicio 6.5.1: Suponiendo que la función *ampliar* en la figura 6.26 pueda manejar cualquiera de los tipos en la jerarquía de la figura 6.25(a), traduzca las expresiones que se muestran a continuación. Suponga que c y d son caracteres, que s y t son enteros cortos, y que i y j son enteros, y x es un número de punto flotante.

- a) $x = s + c.$
- b) $i = s + c.$
- c) $x = (s + c) * (t + d).$

Ejercicio 6.5.2: Como en Ada, suponga que cada expresión debe tener un tipo único, pero que a partir de esa expresión por sí sola, todo lo que podemos deducir es un conjunto de posibles tipos. Es decir, la aplicación de la función E_1 al argumento E_2 , representada por $E \rightarrow E_1(E_2)$, tiene la siguiente regla asociada:

$$E.tipo = \{ t \mid \text{para alguna } s \text{ en } E_2.tipo, s \rightarrow t \text{ está en } E_1.tipo \}$$

Describa una definiciones dirigidas por la sintaxis que determine un tipo único para cada subexpresión, usando un atributo *tipo* para sintetizar un conjunto de posibles tipos de abajo hacia arriba y, una vez que se determine el tipo único de la expresión en general, proceda de arriba hacia abajo para determinar el atributo *único* para el tipo de cada subexpresión.

6.6 Flujo de control

La traducción de instrucciones como *if-else* y *while* está enlazada a la traducción de expresiones booleanas. En los lenguajes de programación, las expresiones booleanas se utilizan con frecuencia para:

1. *Alterar el flujo de control.* Las expresiones booleanas se utilizan como expresiones condicionales en las instrucciones que alteran el flujo de control. El valor de dichas expresiones booleanas es implícito en una posición a la que se llega en un programa. Por ejemplo, en **if** (E) S , la expresión E debe ser verdadera si se llega a la instrucción S .
2. *Calcular los valores lógicos.* Una expresión booleana puede representar a *true* o *false* como valores. Dichas expresiones booleanas pueden evaluarse en analogía con las expresiones aritméticas, mediante las instrucciones de tres direcciones con los operadores lógicos.

El uso que se pretende de las expresiones booleanas se determina mediante su contexto sintáctico. Por ejemplo, una expresión que va después de la palabra clave **if** se utiliza para alterar el flujo de control, mientras que una expresión del lado derecho de una asignación se utiliza para denotar un valor lógico. Dichos contextos sintácticos pueden especificarse en una variedad de formas: podemos usar dos no terminales distintos, usar los atributos heredados o establecer una bandera durante el análisis sintáctico. De manera alternativa, podemos construir un árbol sintáctico e invocar distintos procedimientos para los dos usos distintos de las expresiones booleanas.

Esta sección se concentra en el uso de las expresiones booleanas para alterar el flujo de control. Por claridad, presentaremos una nueva no terminal B para este fin. En la sección 6.6.6 consideraremos la forma en que un compilador puede permitir que las expresiones booleanas representen valores lógicos.

6.6.1 Expresiones booleanas

Las expresiones booleanas están compuestas de los operadores booleanos (que denotamos como **&&**, **||** y **!**, usando la convención de C para los operadores AND, OR y NOT, respectivamente) que se aplican a elementos que son variables booleanas o expresiones relacionales. Estas expresiones relacionales son de la forma $E_1 \text{ rel } E_2$, en donde E_1 y E_2 son expresiones

aritméticas. En esta sección, consideraremos las expresiones booleanas generadas por la siguiente gramática:

$$B \rightarrow B \mid\mid B \mid B \&\& B \mid !B \mid (B) \mid E \mathbf{rel} E \mid \mathbf{true} \mid \mathbf{false}$$

Utilizamos el atributo **rel.op** para indicar cuál de los seis operadores de comparación $<$, \leq , $=$, \neq , $>$ o \geq se representa mediante **rel**. Como es costumbre, suponemos que $\mid\mid$ y $\&\&$ son asociativos por la izquierda, y que $\mid\mid$ tiene la menor precedencia, después $\&\&$ y después $!$.

Dada la expresión $B_1 \mid\mid B_2$, si determinamos que B_1 es verdadera, entonces podemos concluir que toda la expresión es verdadera sin tener que evaluar B_2 . De manera similar, dada la expresión $B_1 \&\& B_2$, si B_1 es falsa, entonces toda la expresión es falsa.

La definición semántica del lenguaje de programación determina si deben evaluarse todas las partes de una expresión booleana. Si la definición de lenguaje permite (o requiere) que se queden sin evaluar partes de una expresión booleana, entonces el compilador puede optimizar la evaluación de expresiones booleanas calculando sólo lo suficiente de una expresión como para poder determinar su valor. Por ende, en una expresión como $B_1 \mid\mid B_2$, no necesariamente se evalúan por completo B_1 o B_2 . Si B_1 o B_2 es una expresión con efectos adicionales (por ejemplo, si contiene una función que modifique a una variable global), entonces puede obtenerse una respuesta inesperada.

6.6.2 Código de corto circuito

En el código de *corto circuito* (o de *salto*), los operadores booleanos $\&\&$, $\mid\mid$ y $!$ se traducen en saltos. Los mismos operadores no aparecen en el código; en vez de ello, el valor de una expresión booleana se representa mediante una posición en la secuencia de código.

Ejemplo 6.21: La instrucción

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

podría traducirse en el código de la figura 6.34. En esta traducción, la expresión booleana es verdadera si el control llega a la etiqueta L_2 . Si la expresión es falsa, el control pasa de inmediato a L_1 , ignorando a L_2 y la asignación $x = 0$. \square

```
if x < 100 goto L2
iffalse x > 200 goto L1
iffalse x != y goto L1
L2: x = 0
L1:
```

Figura 6.34: Código de salto

6.6.3 Instrucciones de flujo de control

Ahora consideraremos la traducción de expresiones booleanas en código de tres direcciones, en el contexto de instrucciones como las que se generan mediante la siguiente gramática:

$$\begin{aligned} S &\rightarrow \mathbf{if} (B) S_1 \\ S &\rightarrow \mathbf{if} (B) S_1 \mathbf{else} S_2 \\ S &\rightarrow \mathbf{while} (B) S_1 \end{aligned}$$

En estas producciones, el no terminal B representa a una expresión booleana y en la no terminal S representa a una instrucción.

Esta gramática generaliza el ejemplo abierto de las expresiones while que presentamos en el ejemplo 5.19. Como en ese ejemplo, tanto B como S tienen un atributo sintetizado llamado *codigo*, el cual proporciona la traducción en instrucciones de tres direcciones. Por simplicidad, generamos las traducciones $B.\text{codigo}$ y $S.\text{codigo}$ como cadenas, usando definiciones dirigidas por la sintaxis. Las reglas semánticas que definen los atributos *codigo* podrían implementarse en vez de generar árboles sintácticos y después emitir código durante el recorrido de un árbol, o mediante cualquiera de los métodos descritos en la sección 5.5.

La traducción de $\mathbf{if} (B) S_1$ consiste en $B.\text{codigo}$ seguida de $S_1.\text{codigo}$, como se muestra en la figura 6.35(a). Dentro de $B.\text{codigo}$ hay saltos con base en el valor de B . Si B es verdadera, el control fluye hacia la primera instrucción de $S_1.\text{codigo}$, y si B es falsa, el control fluye a la instrucción que sigue justo después de $S_1.\text{codigo}$.

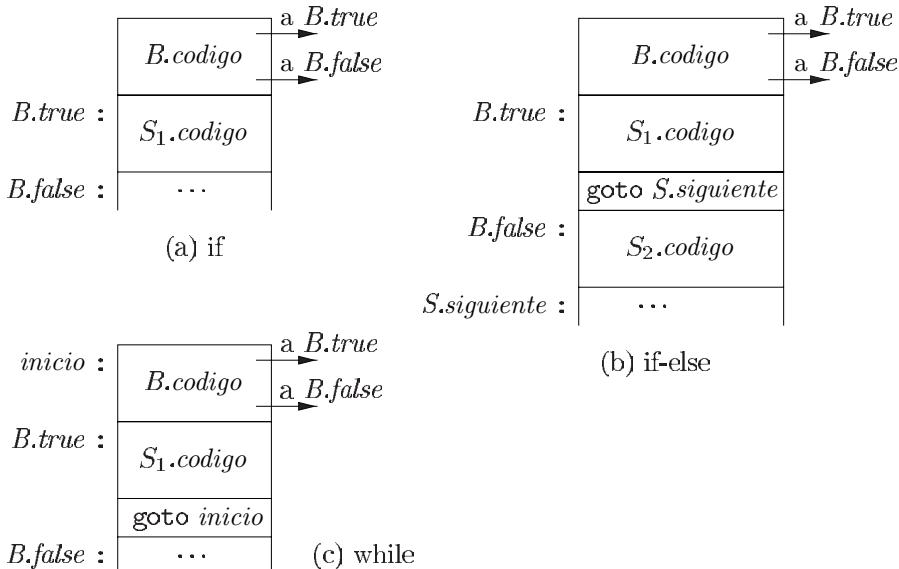


Figura 6.35: Código para las instrucciones if, if-else y while

Las etiquetas para los saltos en $B.\text{codigo}$ y $S.\text{codigo}$ se administran usando atributos heredados. Con una expresión booleana B , asociamos dos etiquetas: $B.\text{true}$, la etiqueta hacia la

cual fluye el control si B es verdadera, y $B.false$, la etiqueta hacia la cual fluye el control si B es falsa. Con una instrucción S , asociamos un atributo heredado $S.sigüiente$ que denota a una etiqueta para la instrucción que sigue justo después del código para S . En algunos casos, la instrucción que va justo después de $S.codigo$ es un salto hacia alguna etiqueta L . Un salto hacia un salto a L desde el interior de $S.codigo$ se evita mediante el uso de $S.sigüiente$.

La definición dirigida por la sintaxis en la figura 6.36-6.37 produce código de tres direcciones para las expresiones booleanas en el contexto de instrucciones if, if-else y while.

PRODUCCIÓN	REGLAS SEMÁNTICAS
$P \rightarrow S$	$S.sigüiente = nuevaetiqueta()$ $P.codigo = S.codigo \parallel etiqueta(S.sigüiente)$
$S \rightarrow \mathbf{assign}$	$S.codigo = \mathbf{assign}.codigo$
$S \rightarrow \mathbf{if} (B) S_1$	$B.true = nuevaetiqueta()$ $B.false = S_1.sigüiente = S.sigüiente$ $S.codigo = B.codigo \parallel etiqueta(B.true) \parallel S_1.codigo$
$S \rightarrow \mathbf{if} (B) S_1 \mathbf{else} S_2$	$B.true = nuevaetiqueta()$ $B.false = nuevaetiqueta()$ $S_1.sigüiente = S_2.sigüiente = S.sigüiente$ $S.codigo = B.codigo$ $\parallel etiqueta(B.true) \parallel S_1.codigo$ $\parallel gen('goto' S.sigüiente)$ $\parallel etiqueta(B.false) \parallel S_2.codigo$
$S \rightarrow \mathbf{while} (B) S_1$	$inicio = nuevaetiqueta()$ $B.true = nuevaetiqueta()$ $B.false = S.sigüiente$ $S_1.sigüiente = inicio$ $S.codigo = etiqueta(inicio) \parallel B.codigo$ $\parallel etiqueta(B.true) \parallel S_1.codigo$ $\parallel gen('goto' inicio)$
$S \rightarrow S_1 S_2$	$S_1.sigüiente = nuevaetiqueta()$ $S_2.sigüiente = S.sigüiente$ $S.codigo = S_1.codigo \parallel etiqueta(S_1.sigüiente) \parallel S_2.codigo$

Figura 6.36: Definición dirigida por la sintaxis para las instrucciones de flujo de control

Suponemos que $nuevaEtiqueta()$ crea una nueva etiqueta cada vez que se llama, y que $etiqueta(L)$ adjunta la etiqueta L a la siguiente instrucción de tres direcciones que se va a generar.⁸

⁸Si se implementan en sentido literal, las reglas semánticas generarán muchas etiquetas y pueden adjuntar más de una etiqueta a una instrucción de tres direcciones. El método de “parcheo de retroceso” de la sección 6.7 crea etiquetas

Un programa consiste en una instrucción generada por $P \rightarrow S$. Las reglas semánticas asociadas con esta producción inicializan $S.\text{siguiente}$ con una nueva etiqueta. $P.\text{codigo}$ consiste en $S.\text{codigo}$ seguido de la nueva etiqueta $S.\text{siguiente}$. El token **asigna** en la producción $S \rightarrow \text{asigna}$ es un receptor para las instrucciones de asignación. La traducción de las asignaciones es como se describió en la sección 6.4; para esta explicación del flujo de control, $S.\text{codigo}$ es simplemente **asigna.codigo**.

Al traducir $S \rightarrow \text{if } (B) S_1$, las reglas semánticas en la figura 6.36 crean una nueva etiqueta $B.\text{true}$ y la adjuntan a la primera instrucción de tres direcciones generada para la instrucción S_1 , como se muestra en la figura 6.35(a). Por ende, los saltos a $B.\text{true}$ dentro del código para B irán al código para S_1 . Además, al establecer $B.\text{false}$ a $S.\text{siguiente}$, aseguramos que el control ignore el código para S_1 , si B se evalúa como falsa.

Al traducir la instrucción if-else $S \rightarrow \text{if } (B) S_1 \text{ else } S_2$, el código para la expresión booleana B tiene saltos hacia fuera de ésta, que van a la primera instrucción del código para S_1 si B es verdadera, y a la primera instrucción del código para S_2 si B es falsa, como se ilustra en la figura 6.35(b). Además, el control fluye desde S_1 y S_2 hacia la instrucción de tres direcciones que va justo después del código para S ; su etiqueta se proporciona mediante el atributo heredado $S.\text{siguiente}$. Una instrucción **goto** $S.\text{siguiente}$ explícita aparece después del código de S_1 para ignorar el código de S_2 . No se necesita un goto después de S_2 , ya que $S_2.\text{siguiente}$ es igual que $S.\text{siguiente}$.

El código para $S \rightarrow \text{while } (B) S_1$ se forma a partir de $B.\text{codigo}$ y $S_1.\text{codigo}$ como se muestra en la figura 6.35(c). Utilizamos una variable local *inicio* para guardar una nueva etiqueta que se adjunta a la primera instrucción para esta instrucción while, que también es la primera instrucción para B . Usamos una variable en vez de un atributo, ya que *inicio* es local para las reglas semánticas de esta producción. La etiqueta heredada $S.\text{siguiente}$ marca la instrucción hacia la cual debe fluir el control si B es falsa; por ende, $B.\text{falsa}$ se establece para ser $S.\text{siguiente}$. Una nueva etiqueta $B.\text{true}$ se adjunta a la primera instrucción para S_1 ; el código para B genera un salto hacia esta etiqueta si B es verdadera. Después del código para S_1 , colocamos la instrucción **goto** *inicio*, la cual produce un salto de regreso al principio del código para la expresión booleana. Observe que $S_1.\text{siguiente}$ se establece a esta etiqueta *inicio*, por lo que los saltos desde el interior de $S_1.\text{codigo}$ pueden ir directo hacia *inicio*.

El código para $S \rightarrow S_1 S_2$ consiste en el código para S_1 seguido del código para S_2 . Las reglas semánticas administran las etiquetas; la primera instrucción después del código para S_1 es el principio del código para S_2 ; y la instrucción después del código para S_2 es también la instrucción después del código para S .

En la sección 6.7 hablaremos sobre la traducción de las instrucciones de flujo de control. Ahí veremos un método alternativo, conocido como “parcheo de retroceso”, el cual emite código para las instrucciones en una sola pasada.

6.6.4 Traducción del flujo de control de las expresiones booleanas

Las reglas semánticas para las expresiones booleanas en la figura 6.37 complementan a las reglas semánticas para las instrucciones en la figura 6.36. Como en la distribución de código de la figura 6.35, una expresión booleana B se traduce en instrucciones de tres direcciones que

sólo cuando se necesitan. De manera alternativa, las etiquetas innecesarias pueden eliminarse durante una fase de organización siguiente.

evalúan a B mediante saltos condicionales e incondicionales hacia una de dos etiquetas: $B.\text{true}$ si B es verdadera, y $B.\text{false}$ si B es falsa.

PRODUCCIÓN	REGLAS SEMÁNTICAS
$B \rightarrow B_1 \text{ } B_2$	$B_1.\text{true} = B.\text{true}$ $B_1.\text{false} = \text{nuevaetiqueta}()$ $B_2.\text{true} = B.\text{true}$ $B_2.\text{false} = B.\text{false}$ $B.\text{codigo} = B_1.\text{codigo} \parallel \text{etiqueta}(B_1.\text{false}) \parallel B_2.\text{codigo}$
$B \rightarrow B_1 \&\& B_2$	$B_1.\text{true} = \text{nuevaetiqueta}()$ $B_1.\text{false} = B.\text{false}$ $B_2.\text{true} = B.\text{true}$ $B_2.\text{false} = B.\text{false}$ $B.\text{codigo} = B_1.\text{codigo} \parallel \text{etiqueta}(B_1.\text{true}) \parallel B_2.\text{codigo}$
$B \rightarrow !B_1$	$B_1.\text{true} = B.\text{false}$ $B_1.\text{false} = B.\text{true}$ $B.\text{codigo} = B_1.\text{codigo}$
$B \rightarrow E_1 \text{ rel } E_2$	$B.\text{codigo} = E_1.\text{codigo} \parallel E_2.\text{codigo}$ $\parallel \text{gen('if' } E_1.\text{dir rel.op } E_2.\text{dir 'goto' } B.\text{true})$ $\parallel \text{gen('goto' } B.\text{false})$
$B \rightarrow \text{true}$	$B.\text{codigo} = \text{gen('goto' } B.\text{true})$
$B \rightarrow \text{false}$	$B.\text{codigo} = \text{gen('goto' } B.\text{false})$

Figura 6.37: Generación del código de tres direcciones para valores booleanos

La cuarta producción en la figura 6.37, $B \rightarrow E_1 \text{ rel } E_2$, se traduce directamente en una instrucción de tres direcciones de comparación, con saltos hacia los lugares apropiados. Por ejemplo, B de la forma $a < b$ se traduce en:

```
if a < b goto B.true
goto B.false
```

El resto de las producciones para B se traducen de la siguiente manera:

1. Suponga que B es de la forma $B_1 \text{ || } B_2$. Si B_1 es verdadera, entonces sabemos de inmediato que B en sí es verdadera, por lo que $B_1.\text{true}$ es igual que $B.\text{true}$. Si B_1 es falsa, entonces hay que evaluar B_2 , para hacer que $B_1.\text{false}$ sea la etiqueta de la primera instrucción en el código para B_2 . Las salidas verdadera y falsa de B_2 son iguales que las salidas verdadera y falsa de B , respectivamente.

2. La traducción de $B_1 \&\& B_2$ es similar.
3. No se necesita código para una expresión B de la forma $!B_1$: sólo se intercambian las salidas verdadera y falsa de B para obtener las salidas verdadera y falsa de B_1 .
4. Las constantes **true** y **false** se traducen en saltos hacia $B.true$ y $B.false$, respectivamente.

Ejemplo 6.22: Considere de nuevo la siguiente instrucción del ejemplo 6.21:

if (x < 100 || x > 200 && x != y) x = 0; (6.13)

Si utilizamos las definiciones dirigidas por la sintaxis en las figuras 6.36 y 6.37, obtendremos el código de la figura 6.38.

```

if x < 100 goto L2
goto L3
L3: if x > 200 goto L4
      goto L1
L4: if x != y goto L2
      goto L1
L2: x = 0
L1:

```

Figura 6.38: Traducción del flujo de control de una instrucción if simple

La instrucción (6.13) constituye un programa generado por $P \rightarrow S$ de la figura 6.36. Las reglas semánticas para la producción generan una nueva etiqueta L_1 para la instrucción después del código para S . La instrucción S tiene la forma $\text{if } (B) S_1$, en donde S_1 es $x = 0;$, por lo que las reglas en la figura 6.36 generan una nueva etiqueta L_2 y la adjuntan a la primera instrucción (y sólo a ésta, en este caso) en $S_1.codigo$, que es $x = 0$.

Como $||$ tiene menor precedencia que $\&\&$, la expresión booleana en (6.13) tiene la forma $B_1 || B_2$, en donde B_1 es $x < 100$. Siguiendo las reglas en la figura 6.37, $B_1.true$ es L_2 , la etiqueta de la asignación $x = 0;$. $B_1.false$ es una nueva etiqueta L_3 , que se adjunta a la primera instrucción en el código para B_2 .

Observe que el código generado no es óptimo, en cuanto a que la traducción tiene tres instrucciones más (goto) que el código en el ejemplo 6.21. La instrucción `goto L3` es redundante, ya que L_3 es la etiqueta de la siguiente instrucción. Las dos instrucciones `goto L1` pueden eliminarse mediante el uso de `ifFalse` en vez de instrucciones `if`, como en el ejemplo 6.21. \square

6.6.5 Evitar gotos redundantes

En el ejemplo 6.22, la comparación $x > 200$ se traduce en el siguiente fragmento de código:

```

        if x > 200 goto L4
        goto L1
L4:   ...

```

En vez de ello, considere la siguiente instrucción:

```

        ifFalse x > 200 goto L1
L4:   ...

```

Esta instrucción `ifFalse` aprovecha el flujo natural de una instrucción a la siguiente en secuencia, por lo que el control simplemente “pasa” hacia la etiqueta L_4 si $x > 200$ es falsa, con lo cual se evita un salto.

En los esquemas de código para las instrucciones `if` y `while` de la figura 6.35, el código para la instrucción S_1 sigue justo después del código para la expresión booleana B . Mediante el uso de una etiqueta especial llamada *pasar* (es decir, “no generar ningún salto”), podemos adaptar las reglas semánticas de las figuras 6.36 y 6.37 para permitir que el control pase del código para B al código para S_1 . Las nuevas reglas para $S \rightarrow \text{if } (B) S_1$ en la figura 6.36 establecen $B.\text{true} \rightarrow \text{pasar}$:

$$\begin{aligned}
 B.\text{true} &= \text{pasar} \\
 B.\text{false} &= S_1.\text{siguiente} = S.\text{siguiente} \\
 S.\text{codigo} &= B.\text{codigo} \parallel S_1.\text{codigo}
 \end{aligned}$$

De manera similar, las reglas para las instrucciones `if-else` y `while` también establecen $B.\text{true} \rightarrow \text{pasar}$.

Ahora vamos a adaptar las reglas semánticas a las expresiones booleanas, para permitir que el control pase a través de un código al otro, siempre que sea posible. Las nuevas reglas para $B \rightarrow E_1 \text{ rel } E_2$ en la figura 6.39 generan dos instrucciones, como en la figura 6.37, si $B.\text{true}$ y $B.\text{false}$ son etiquetas explícitas; es decir, que ninguna sea igual a *pasar*. En caso contrario, si $B.\text{true}$ es una etiqueta explícita, entonces $B.\text{false}$ debe ser *pasar*, para que generen una instrucción `if` que permita que el control pase de un código a otro si la condición es falsa. En cambio, si $B.\text{true}$ es una etiqueta explícita, entonces generan una instrucción `ifFalse`. En el caso restante, tanto $B.\text{true}$ como $B.\text{false}$ son *pasar*, por lo que no se genera ningún salto.⁹

En las nuevas reglas para $B \rightarrow B_1 \parallel B_2$ en la figura 6.40, observe que el significado de la etiqueta *pasar* para B es distinto del significado para B_1 . Suponga que $B.\text{true}$ es *pasar*; es decir, el control pasa a través de B , si B se evalúa como verdadera. Aunque B se evalúa como verdadera si B_1 lo hace, $B_1.\text{true}$ debe asegurar que el control salte sobre el código para B_2 , para llegar a la siguiente instrucción después de B .

Por otro lado, si B_1 se evalúa como falsa, el valor verdadero de B se determina mediante el valor de B_2 , por lo que las reglas en la figura 6.40 aseguran que $B_1.\text{false}$ correspondan al control que pasa desde B_1 hasta el código para B_2 .

Las reglas semánticas para $B \rightarrow B_1 \& B_2$ son similares a las de la figura 6.40. Las dejaremos como un ejercicio.

Ejemplo 6.23: Con las nuevas reglas que utilizan la etiqueta especial *pasar*, el programa (6.13) del ejemplo 6.21

⁹En C y Java, las expresiones pueden contener asignaciones en su interior, por lo que debe generarse código para las subexpresiones E_1 y E_2 , incluso si tanto $B.\text{true}$ como $B.\text{false}$ son *pasar*. Si se desea, el código muerto puede eliminarse durante una fase de optimización.

$$\begin{aligned}
 prueba &= E_1.dir \text{ rel.} op E_2.dir \\
 s &= \text{if } B.true \neq \text{pasar} \text{ and } B.false \neq \text{pasar} \text{ then} \\
 &\quad \text{gen('if' prueba 'goto' } B.true) \parallel \text{gen('goto' } B.false) \\
 &\quad \text{else if } B.true \neq \text{pasar} \text{ then} \text{gen('if' prueba 'goto' } B.true) \\
 &\quad \text{else if } B.false \neq \text{pasar} \text{ then} \text{gen('ifFalse' prueba 'goto' } B.false) \\
 &\quad \text{else } '' \\
 B.codigo &= E_1.codigo \parallel E_2.codigo \parallel s
 \end{aligned}$$
Figura 6.39: Reglas semánticas para $B \rightarrow E_1 \text{ rel } E_2$

$$\begin{aligned}
 B_1.true &= \text{if } B.true \neq \text{pasar} \text{ then } B.true \text{ else } \text{nuevaetiqueta}() \\
 B_1.false &= \text{pasar} \\
 B_2.true &= B.true \\
 B_2.false &= B.false \\
 B.codigo &= \text{if } B.true \neq \text{pasar} \text{ then } B_1.codigo \parallel B_2.codigo \\
 &\quad \text{else } B_1.codigo \parallel B_2.codigo \parallel \text{etiqueta}(B_1.true)
 \end{aligned}$$
Figura 6.40: Reglas semánticas para $B \rightarrow B_1 \parallel B_2$

$$\text{if (x } < 100 \parallel x > 200 \&\& x \neq y \text{) } x = 0;$$

se traduce en el código de la figura 6.41.

$$\begin{aligned}
 &\text{if } x < 100 \text{ goto L}_2 \\
 &\text{ifFalse } x > 200 \text{ goto L}_1 \\
 &\text{ifFalse } x \neq y \text{ goto L}_1 \\
 \text{L}_2: &\quad x = 0 \\
 \text{L}_1:
 \end{aligned}$$

Figura 6.41: Instrucción if traducida mediante el uso de la técnica de pasar de un código a otro

Como en el ejemplo 6.22, las reglas para $P \rightarrow S$ crean la etiqueta L_1 . La diferencia del ejemplo 6.22 es que el atributo heredado $B.true$ es *pasar* cuando se aplican las reglas semánticas para $B \rightarrow B_1 \parallel B_2$ ($B.false$ es L_1). Las reglas en la figura 6.40 crean una nueva etiqueta L_2 para permitir un salto sobre el código para B_2 , si B_1 se evalúa como verdadera. Por ende, $B_1.true$ es L_2 . y $B_1.false$ es *pasar*, ya que B_2 debe evaluarse si B_1 es falsa.

Por lo tanto, llegamos a la producción $B \rightarrow E_1 \text{ rel } E_2$ que genera a $x < 100$ con $B.true = L_2$ y $B.false = \text{pasar}$. Con estas etiquetas heredadas, las reglas en la figura 6.39 generan, por lo tanto, una sola instrucción $\text{if } x < 100 \text{ goto L}_2$. \square

6.6.6 Valores booleanos y código de salto

Esta sección se ha enfocado en el uso de expresiones booleanas para alterar el flujo de control en las instrucciones. Una expresión booleana también puede evaluarse en base a su valor, como en las instrucciones de asignación como `x = true`; o `x = a < b`;

Una forma limpia de manejar ambas funciones de las expresiones booleanas es primero construir un árbol sintáctico para las expresiones, usando cualquiera de los siguientes métodos:

1. *Usar dos pasadas.* Construya un árbol sintáctico completo para la entrada, y después recorra el árbol en orden por profundidad, calculando las transacciones especificadas por las reglas semánticas.
2. *Usar una pasada para las instrucciones, pero dos pasadas para las expresiones.* Con este método, traduciríamos E en `while (E) S1` antes de examinar S_1 . Sin embargo, la traducción de E se realizaría construyendo su árbol sintáctico y después recorriendo ese árbol.

La siguiente gramática tiene un solo no terminal E para las expresiones:

$$\begin{aligned} S &\rightarrow \mathbf{id} = E ; \mid \mathbf{if} (E) S \mid \mathbf{while} (E) S_1 \mid S S \\ E &\rightarrow E \mid E \mid E \&& E \mid E \text{ rel } E \mid E + E \mid (E) \mid \mathbf{id} \mid \mathbf{true} \mid \mathbf{false} \end{aligned}$$

El no terminal E gobierna el flujo de control en $S \rightarrow \mathbf{while} (E) S_1$. El mismo no terminal E denota un valor en $S \rightarrow \mathbf{id} = E$; y $E \rightarrow E + E$.

Podemos manejar estas dos funciones de las expresiones mediante el uso de funciones separadas para generación de código. Suponga que el atributo $E.n$ denota el nodo del árbol sintáctico para una expresión E y que los nodos son objetos. Suponga que el método *salto* genera código de salto en un nodo de una expresión, y que el método *r-value* genera código para calcular el valor del nodo en un nombre temporal.

Cuando E aparece en $S \rightarrow \mathbf{while} (E) S_1$, se hace una llamada al método *salto* en el nodo $E.n$. La implementación de *salto* se basa en las reglas para las expresiones booleanas en la figura 6.37. En específico, se genera el código de salto mediante la llamada a $E.n.\text{salto}(t, f)$, en donde t es una nueva etiqueta para la primera instrucción de $S_1.\text{codigo}$ y f es la etiqueta $S.\text{siguiente}$.

Cuando E aparece en $S \rightarrow \mathbf{id} = E ;$, se hace una llamada al método *r-value* en el nodo $E.n$. Si E tiene la forma $E_1 + E_2$, la llamada al método $E.n.\text{r-value}()$ genera código como se vio en la sección 6.4. Si E tiene la forma $E_1 \&& E_2$, primero generamos código de salto para E y después asignamos verdadero o falso a un nuevo nombre temporal t en las salidas de verdadero y falso, respectivamente, desde el código de salto.

Por ejemplo, la asignación `x = a < b && c < d` puede implementarse mediante el código en la figura 6.42.

6.6.7 Ejercicios para la sección 6.6

Ejercicio 6.6.1: Agregue reglas a la definición dirigida por la sintaxis de la figura 6.36 para las siguientes construcciones de flujo de control:

- a) Una instrucción repeat: `repeat S while B`.

```

        ifFalse a < b goto L1
        ifFalse c > d goto L1
        t = true
        goto L2
L1:  t = false
L2:  x = t

```

Figura 6.42: Traducción de una asignación booleana mediante el cálculo del valor de un nombre temporal

! b) Un ciclo for: **for** ($S_1; B; S_2$) S_3 .

Ejercicio 6.6.2: Las máquinas modernas tratan de ejecutar muchas instrucciones al mismo tiempo, incluyendo instrucciones de bifurcación. Por ende, hay un severo costo si la máquina sigue en forma especulativa a una bifurcación, cuando el control en realidad se va por otro lado (todo el trabajo especulativo se tira a la basura). Por lo tanto, es conveniente disminuir al mínimo el número de bifurcaciones. Observe que la implementación de un ciclo while en la figura 6.35(c) tiene dos bifurcaciones por cada iteración: una para entrar al cuerpo desde la condición B y la otra para saltar de vuelta al código para B . Como resultado, por lo general, es preferible implementar **while** (B) S como si fuera **if** (B) { **repeat** S **until** !(B) }. Muestre cuál sería la apariencia del esquema de código para esta traducción, y modifique la regla para los ciclos while en la figura 6.36.

! Ejercicio 6.6.3: Suponga que hubiera un operador “or exclusivo” (verdadero si y sólo si uno de sus argumentos es verdadero) en C. Escriba la regla para este operador como en la figura 6.37.

Ejercicio 6.6.4: Traduzca las siguientes expresiones, mediante el esquema de traducción para evitar gotos de la sección 6.6.5:

- a) **if** (a==b **&&** c==d **||** e==f) x == 1;
- b) **if** (a==b **||** c==d **||** e==f) x == 1;
- c) **if** (a==b **&&** c==d **&&** e==f) x == 1;

Ejercicio 6.6.5: Proporcione un esquema de traducción con base en la definición dirigida por la sintaxis de las figuras 6.36 y 6.37.

Ejercicio 6.6.6: Adapte las reglas semánticas en las figuras 6.36 y 6.37 para permitir que el control pase de un código a otro, usando reglas como las de las figuras 6.39 y 6.40.

! Ejercicio 6.6.7: Las reglas semánticas para las instrucciones en el ejercicio 6.6.6 generan etiquetas innecesarias. Modifique las reglas para las instrucciones en la figura 6.36, para crear etiquetas según sea necesario, usando una etiqueta especial *diferida*, para indicar que la etiqueta no se ha creado todavía. Sus reglas deben generar código similar al del ejemplo 6.21.

!! Ejercicio 6.6.8: La sección 6.6.5 habla acerca de usar código de paso (fall-through) para disminuir al mínimo el número de saltos en el código intermedio generado. Sin embargo, no aprovecha la opción de sustituir una condición por su complemento; por ejemplo, sustituir `if a < b goto L1; goto L2` por `if b >= a goto L2; goto L1`. Desarrolle definiciones dirigidas por la sintaxis que aproveche esta opción cuando sea necesario.

6.7 Parcheo de retroceso (backpatch)

Un problema clave al generar código para expresiones booleanas e instrucciones de flujo de control es el de relacionar una instrucción de salto con el destino del mismo. Por ejemplo, la traducción de la expresión booleana B en `if (B) S` contiene un salto, para cuando B es falsa, a la instrucción que sigue del código para S . En una traducción de una sola pasada, B debe traducirse antes de examinar S . ¿Cuál sería entonces el destino del `goto` que salta sobre el código para S ? En la sección 6.6 resolvimos este problema mediante el paso de etiquetas como atributos heredados hacia donde se generaban las instrucciones de salto relevantes. Pero entonces se requiere una pasada aparte para enlazar las etiquetas a las direcciones.

Esta sección se enfoca en un método complementario, conocido como *parcheo de retroceso*, en el cual se pasan listas de saltos como atributos sintetizados. En específico, cuando se genera un salto, el destino de éste se deja temporalmente sin especificar. Cada salto de este tipo se coloca en una lista de saltos cuyas etiquetas deben llenarse cuando se pueda determinar la etiqueta apropiada. Todos los saltos en una lista tienen la misma etiqueta de destino.

6.7.1 Generación de código de una pasada, mediante parcheo de retroceso

La técnica de parcheo de retroceso puede usarse para generar código para las expresiones booleanas y las instrucciones de flujo de control en una pasada. Las traducciones que generemos serán de la misma forma que las de la sección 6.6, excepto por la forma en que se manejan las etiquetas.

En esta sección, los atributos sintetizados *listatrue* y *listafalse* del no terminal B se utilizan para manejar las etiquetas en el código de salto para las expresiones booleanas. En especial, $B.listatrue$ será una lista de instrucciones de salto o de salto condicional, en las cuales debemos insertar la etiqueta a la que se pasa el control si B es verdadera. De igual forma, $B.listafalse$ es la lista de instrucciones que en un momento dado obtienen la etiqueta a la que se pasa el control cuando B es falsa. A medida que se genera código para B , los saltos a las salidas de verdadero se dejan incompletos, sin llenar el campo de la etiqueta. Estos saltos incompletos se colocan en listas a las que apuntan *B.listatrue* y *B.listafalse*, según sea apropiado. De manera similar, una instrucción S tiene un atributo sintetizado llamado *S.siglista*, el cual denota una lista de saltos a la instrucción que va justo después del código para S .

Para especificar, generamos instrucciones en un arreglo de instrucciones, y las etiquetas serán los índices para este arreglo. Para manipular las listas de saltos, usamos tres funciones:

1. *crearLista(i)* crea una nueva lista que sólo contiene a i , un índice que apunta al arreglo de instrucciones; *crearLista* devuelve un apuntador a la lista recién creada.

2. $combinar(p_1, p_2)$ concatena las listas a las que apuntan p_1 y p_2 , y devuelve un apuntador a la lista concatenada.
3. $backpatch(p, i)$ inserta a i como la etiqueta de destino para cada una de las instrucciones en la lista a la que apunta p .

6.7.2 Técnica de parcheo de retroceso para las expresiones booleanas

Ahora vamos a construir un esquema de traducción adecuado para generar código para las expresiones booleanas, durante el análisis sintáctico ascendente. Un no terminal marcador M en la gramática hace que una acción semántica obtenga, en tiempos apropiados, el índice de la siguiente instrucción que se va a generar. La gramática es la siguiente:

$$\begin{array}{l} B \rightarrow B_1 \mid\mid M B_2 \mid B_1 \&\& M B_2 \mid ! B_1 \mid (B_1) \mid E_1 \text{ rel } E_2 \mid \text{true} \mid \text{false} \\ M \rightarrow \epsilon \end{array}$$

El esquema de traducción se encuentra en la figura 6.43.

- 1) $B \rightarrow B_1 \mid\mid M B_2 \quad \{ \text{backpatch}(B_1.listafalse, M.instr);$
 $B.listatru = \text{combinar}(B_1.listatru, B_2.listatru);$
 $B.listafalse = B_2.listafalse; \}$
- 2) $B \rightarrow B_1 \&\& M B_2 \quad \{ \text{backpatch}(B_1.listatru, M.instr);$
 $B.listatru = B_2.listatru;$
 $B.listafalse = \text{combinar}(B_1.listafalse, B_2.listafalse); \}$
- 3) $B \rightarrow ! B_1 \quad \{ B.listatru = B_1.listafalse;$
 $B.listafalse = B_1.listatru; \}$
- 4) $B \rightarrow (B_1) \quad \{ B.listatru = B_1.listatru;$
 $B.listafalse = B_1.listafalse; \}$
- 5) $B \rightarrow E_1 \text{ rel } E_2 \quad \{ B.listatru = \text{crearlista}(siginstr);$
 $B.listafalse = \text{crearlista}(siginstr + 1);$
 $\text{emitir}('if' E_1.dir \text{ rel.op } E_2.dir 'goto _');$
 $\text{emitir}('goto _'); \}$
- 6) $B \rightarrow \text{true} \quad \{ B.listatru = \text{crearlista}(siginstr);$
 $\text{emitir}('goto _'); \}$
- 7) $B \rightarrow \text{false} \quad \{ B.listafalse = \text{crearlista}(siginstr);$
 $\text{emitir}('goto _'); \}$
- 8) $M \rightarrow \epsilon \quad \{ M.instr = siginstr; \}$

Figura 6.43: Esquema de traducción para las expresiones booleanas

Considere la acción semántica (1) para la producción $B \rightarrow B_1 \mid\mid M B_2$. Si B_1 es verdadera, entonces B también es verdadera, por lo que los saltos en $B_1.listatru$ se convierten en parte de $B.listatru$. No obstante, si B_1 es falsa, debemos probar B_2 a continuación, por lo

que el destino para los saltos $B_1.listafalse$ debe ser el principio del código generado para B_2 . Este destino se obtiene usando el no terminal marcador M . Ese no terminal produce, como un atributo sintetizado $M.instr$, el índice de la siguiente instrucción, justo antes de que el código de B_2 empiece a generarse.

Para obtener ese índice de instrucción, asociamos con la producción $M \rightarrow \epsilon$ la siguiente acción semántica:

$$\{ M.instr = siginstr; \}$$

La variable $siginstr$ contiene el índice de la siguiente instrucción a seguir. A este valor se le aplicará un parcheo de retroceso hacia $B_1.listafalse$ (es decir, cada instrucción en la lista $B_1.listafalse$ recibirá a $M.instr$ como su etiqueta de destino) cuando hayamos visto el resto de la producción $B \rightarrow B_1 \parallel M B_2$.

La acción semántica (2) para $B \rightarrow B_1 \&& M B_2$ es similar a (1). La acción (3) para $B \rightarrow !B$ intercambia las listas de verdadero y falso. La acción (4) ignora los paréntesis.

Por simplicidad, la acción semántica (5) genera dos instrucciones: goto condicional y goto incondicional. En ninguna de las dos se llena su destino. Estas instrucciones se colocan en listas nuevas, a las que apuntan $B.listatrue$ y $B.listafalse$, respectivamente.

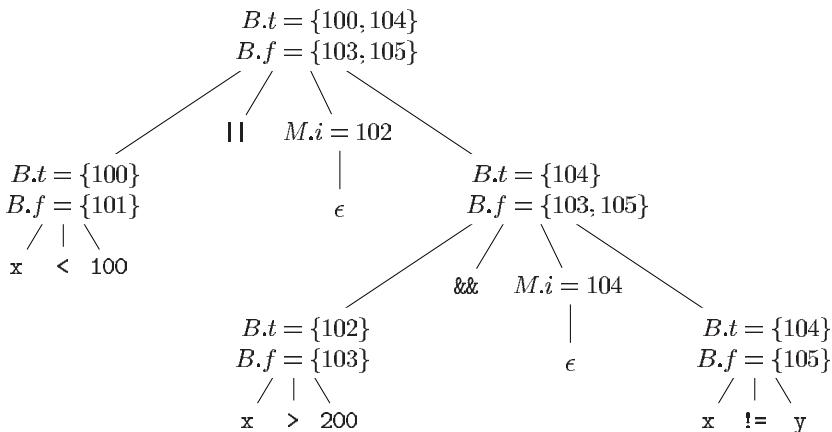


Figura 6.44: Árbol de análisis sintáctico anotado para $x < 100 \parallel x > 200 \&& x \neq y$

Ejemplo 6.24: Considere de nuevo la siguiente expresión:

$$x < 100 \parallel x > 200 \&& x \neq y$$

En la figura 6.44 se muestra un árbol de análisis sintáctico anotado; por cuestión de legibilidad, los atributos *listatrue*, *listafalse* e *instr* se representan mediante las letras *t*, *f* e *i*, respectivamente. Las acciones se realizan durante un recorrido del árbol en orden por profundidad. Como todas las acciones aparecen en los extremos de los lados derechos, pueden realizarse en conjunto con las reducciones durante un análisis sintáctico ascendente. En respuesta a la reducción de $x < 100$ a B mediante la producción (5), se generan las siguientes dos instrucciones:

```
100: if x < 100 goto _
101: goto _
```

(Empezamos de manera arbitraria los números de las instrucciones en 100). El marcador no terminal M en la producción

$$B \rightarrow B_1 \mid\mid M B_2$$

registra el valor de $siginstr$, que en este momento es 102. La reducción de $x > 200$ a B mediante la producción (5) genera las siguientes instrucciones:

```
102: if x > 200 goto _
103: goto _
```

La subexpresión $x > 200$ corresponde a B_1 en la producción

$$B \rightarrow B_1 \&\& M B_2$$

El marcador no terminal M registra el valor actual de $siginstr$, que ahora es 104. Al reducir $x != y$ a B mediante la producción (5), se genera lo siguiente:

```
104: if x != y goto _
105: goto _
```

Ahora reducimos mediante $B \rightarrow B_1 \&\& M B_2$. La acción semántica correspondiente llama a $backpatch(B_1.listatru, M.instr)$ para enlazar la salida de verdadero de B_1 con la primera instrucción de B_2 . Como $B_1.listatru$ es $\{102\}$ y $M.instr$ es 104, esta llamada a $backpatch$ completa la instrucción 102 con el 104. Las seis instrucciones generadas hasta ahora se muestran en la figura 6.45(a).

La acción semántica asociada con la reducción final mediante $B \rightarrow B_1 \mid\mid M B_2$ llama a $backpatch(\{101\}, 102)$, y las instrucciones quedan como se muestra en la figura 6.45(b).

Toda la expresión es verdadera si y sólo si se llega a los gotos de las instrucciones 100 o 104, y es falsa si y sólo si se llega a los gotos de las instrucciones 103 o 105. Más adelante en el proceso de compilación se llenarán los destinos de estas instrucciones, cuando veamos lo que debemos hacer dependiendo de si la expresión es verdadera o falsa. \square

6.7.3 Instrucciones de flujo de control

Ahora usaremos la técnica de parcheo de retroceso para traducir las instrucciones de flujo de control en una pasada. Considere las instrucciones generadas por la siguiente gramática:

$$\begin{aligned} S &\rightarrow \mathbf{if}(B)S \mid \mathbf{if}(B)S \, \mathbf{else} \, S \mid \mathbf{while}(B)S \mid \{L\} \mid A ; \\ L &\rightarrow LS \mid S \end{aligned}$$

Aquí, S denota una instrucción, L una lista de instrucciones, A una instrucción de asignación y B una expresión booleana. Observe que debe haber otras producciones, como las que se utilizan

```

100: if x < 100 goto -
101: goto -
102: if x > 200 goto 104
103: goto -
104: if x != y goto -
105: goto -

```

(a) Después de aplicar parcheo de retroceso para agregar la instrucción 104 a la 102.

```

100: if x < 100 goto -
101: goto 102
102: if y > 200 goto 104
103: goto -
104: if x != y goto -
105: goto -

```

(b) Después de aplicar parcheo de retroceso para agregar la instrucción 102 a la 101.

Figura 6.45: Pasos en el proceso de parcheo de retroceso

para las instrucciones de asignación. Sin embargo, las producciones dadas son suficientes para ilustrar las técnicas usadas para traducir instrucciones de flujo de control.

El esquema de código para las instrucciones if, if-else y while es el mismo que en la sección 6.6. Hacemos la suposición tácita de que la secuencia de código en el arreglo de instrucciones refleja el flujo natural de control de una instrucción a la siguiente. Si no es así, entonces deben insertarse saltos explícitos para implementar el flujo de control secuencial natural.

El esquema de traducción en la figura 6.46 mantiene listas de saltos que se llenan cuando se encuentran sus destinos. Al igual que en la figura 6.43, las expresiones booleanas generadas por el no terminal B tienen dos listas de saltos, $B.listatru$ e y $B.listafal$ se, las cuales corresponden a las salidas de verdadero y falso del código para B , respectivamente. Las instrucciones generadas por los no terminales S y L tienen una lista de saltos sin llenar, proporcionadas por el atributo *siglista*, que deben completarse en un momento dado mediante la técnica de parcheo de retroceso. $S.siglista$ es una lista de todos los saltos condicionales e incondicionales a la instrucción que va después del código para la instrucción S en orden de ejecución. $L.siglista$ se define de manera similar.

Considera la acción semántica (3) en la figura 6.46. El esquema de código para la producción $S \rightarrow \mathbf{while} \ B \ S_1$ es como el de la figura 6.35(c). Las dos ocurrencias del marcador no terminal M en la producción

$$S \rightarrow \mathbf{while} \ M_1 \ (\ B \) \ M_2 \ S_1$$

registran los números de instrucción del principio del código para B y del principio del código para S_1 . Las etiquetas correspondientes en la figura 6.35(c) son *inicio* y *B.true*, respectivamente.

- 1) $S \rightarrow \mathbf{if}(B) M S_1 \{ \text{backpatch}(B.\text{listatru}, M.\text{instr});$
 $S.\text{siglista} = \text{combinar}(B.\text{listafalse}, S_1.\text{siglista}); \}$
- 2) $S \rightarrow \mathbf{if}(B) M_1 S_1 \mathbf{else} M_2 S_2$
 $\{ \text{backpatch}(B.\text{listatru}, M_1.\text{instr});$
 $\text{backpatch}(B.\text{listafalse}, M_2.\text{instr});$
 $\text{temp} = \text{combinar}(S_1.\text{siglista}, N.\text{siglista});$
 $S.\text{siglista} = \text{combinar}(\text{temp}, S_2.\text{siglista}); \}$
- 3) $S \rightarrow \mathbf{while} M_1 (B) M_2 S_1$
 $\{ \text{backpatch}(S_1.\text{siglista}, M_1.\text{instr});$
 $\text{backpatch}(B.\text{listatru}, M_2.\text{instr});$
 $S.\text{siglista} = B.\text{listafalse};$
 $\text{emitir}(\text{'goto'} M_1.\text{instr}); \}$
- 4) $S \rightarrow \{ L \} \quad \{ S.\text{siglista} = L.\text{siglista}; \}$
- 5) $S \rightarrow A ; \quad \{ S.\text{siglista} = \mathbf{null}; \}$
- 6) $M \rightarrow \epsilon \quad \{ M.\text{instr} = \text{siginstr}; \}$
- 7) $N \rightarrow \epsilon \quad \{ N.\text{siglista} = \text{crearlista}(siginstr);$
 $\text{emitir}(\text{'goto'} \text{ '}); \}$
- 8) $L \rightarrow L_1 M S \quad \{ \text{backpatch}(L_1.\text{siglista}, M.\text{instr});$
 $L.\text{siglista} = S.\text{siglista}; \}$
- 9) $L \rightarrow S \quad \{ L.\text{siglista} = S.\text{siglista}; \}$

Figura 6.46: Traducción de instrucciones

De nuevo, la única producción para M es $M \rightarrow \epsilon$. La acción (6) en la figura 6.46 establece el atributo $M.\text{inst}$ con el número de la siguiente instrucción. Una vez que se ejecuta el cuerpo S_1 de la instrucción `while`, el control fluye hacia el principio. Por lo tanto, cuando reducimos $\mathbf{while} M_1 (B) M_2 S_1$ a S , aplicamos la técnica de parcheo de retroceso a $S_1.\text{siglista}$ para hacer que todos los destinos en esa lista sean $M_1.\text{instr}$. Se adjunta un salto explícito hacia el principio del código para B después del código para S_1 , ya que el control también puede “caer hasta el fondo”. Se aplica la técnica de parcheo de retroceso a $B.\text{listatru}$ para ir al principio de S_1 , haciendo que los saltos en $B.\text{listatru}$ vayan a $M_2.\text{instr}$.

Un argumento más convincente para el uso de $S.\text{siglista}$ y $L.\text{siglista}$ se presenta cuando se genera código para la instrucción condicional $\mathbf{if}(B) S_1 \mathbf{else} S_2$. Si el control “cae hasta el fondo” de S_1 , como cuando S_1 es una asignación, debemos incluir al final del código para S_1 un salto sobre el código para S_2 . Utilizamos otro marcador no terminal para generar este salto después de S_1 . Dejemos que el no terminal N sea este marcador con la producción $N \rightarrow \epsilon$.

N tiene el atributo $N.siglista$, que será una lista consistente en el número de instrucción del salto `goto _` que se genera mediante la acción semántica (7) para N .

La acción semántica (2) en la figura 6.46 trata con las instrucciones if-else con la siguiente sintaxis:

$$S \rightarrow \mathbf{if} (B) M_1 S_1 N \mathbf{else} M_2 S_2$$

Aplicamos la técnica de parcheo de retroceso a los saltos cuando B es verdadera para la instrucción $M_1.instr$; esta última es el principio del código para S_1 . De manera similar, aplicamos parcheo de retroceso a los saltos cuando B es falsa para ir al principio del código para S_2 . La lista $S.siglista$ incluye todos los saltos hacia fuera de S_1 y S_2 , así como el salto generado por N . La variable $temp$ es un nombre temporal que se utiliza sólo para combinar listas.

Las acciones semánticas (8) y (9) manejan secuencias de instrucciones. En

$$L \rightarrow L_1 M S$$

la instrucción que va después del código para L_1 en orden de ejecución es el principio de S . Por ende, se aplica parcheo de retroceso a la lista $L_1.siglista$, al principio del código de S , lo cual se da mediante $M.instr$. En $L \rightarrow S$, $L.siglista$ es igual que $S.siglista$.

Observe que no se generan nuevas instrucciones en ninguna parte de estas reglas semánticas, excepto por las reglas (3) y (7). Todo el resto del código se genera por las acciones semánticas asociadas con las instrucciones de asignación y las expresiones. El flujo de control produce el parcheo de retroceso apropiado, de manera que las asignaciones y las evaluaciones de las expresiones booleanas se conecten en forma apropiada.

6.7.4 Instrucciones break, continue y goto

La construcción más elemental de un lenguaje de programación para cambiar el flujo de control en un programa es la instrucción `goto`. En C, una instrucción como `goto L` envía el control a la instrucción etiquetada como `L`; debe haber precisamente una instrucción con la etiqueta `L` en este alcance. Las instrucciones `goto` pueden implementarse mediante el mantenimiento de una lista de saltos sin llenar para cada etiqueta, y después se aplica el parcheo de retroceso al destino, cuando éste se conoce.

Java elimina el uso de las instrucciones `goto`; sin embargo, permite saltos disciplinados conocidos como instrucciones `break`, las cuales envían el control hacia fuera de una construcción envolvente, y las instrucciones `continue`, que activan la siguiente iteración de un ciclo envolvente. El siguiente extracto de un analizador léxico muestra el uso de las instrucciones `break` y `continue` simples:

```

1)  for ( ; ; readch() ) {
2)      if( peek == ' ' || peek == '\t' ) continue;
3)      else if ( peek == '\n' ) linea = linea + 1;
4)      else break;
5)  }
```

El control salta de la instrucción `break` en la línea 4 a la siguiente instrucción después del ciclo `for` envolvente. El control salta de la instrucción `continue` en la línea 2 al código para evaluar `readch()` y después a la instrucción `if` en la línea 2.

Si S es la construcción envolvente, entonces una instrucción break es un salto a la primera instrucción después del código para S . Podemos generar código para la instrucción break si hacemos lo siguiente: (1) llevar el registro de la instrucción S envolvente, (2) generar un salto sin llenar para la instrucción break, y (3) colocar este salto sin llenar en $S.siglista$, en donde $siglista$ es como vimos en la sección 6.7.3.

En una front-end de dos pasadas que construye árboles sintácticos, $S.siglista$ puede implementarse como un campo en el nodo para S . Podemos llevar el registro de S utilizando la tabla de símbolos para asignar un identificador especial **break** al nodo para la instrucción S envolvente. Este método también se encarga de las instrucciones break etiquetadas en Java, ya que la tabla de símbolos se puede utilizar para asignar la etiqueta al nodo del árbol sintáctico para la construcción envolvente.

De manera alternativa, en vez de usar la tabla de símbolos para acceder al nodo para S , podemos colocar un apuntador a $S.siglista$ en la tabla de símbolos. Ahora, al llegar a una instrucción break, generamos un salto sin llenar, buscamos $siglista$ a través de la tabla de símbolos y agregamos el salto a la lista, en donde se le aplicará parcheo de retroceso como vimos en la sección 6.7.3.

Las instrucciones continue se pueden manejar de una forma análoga a la instrucción break. La principal diferencia entre las dos es que el destino del salto generado es distinto.

6.7.5 Ejercicios para la sección 6.7

Ejercicio 6.7.1: Usando la traducción de la figura 6.43, traduzca cada una de las siguientes expresiones. Muestre las listas de verdadero y falso para cada subexpresión. Puede suponer que la dirección de la primera instrucción que se genera es 100.

- a) $a==b \&& (c==d \mid\mid e==f)$
- b) $(a==b \mid\mid c==d) \mid\mid e==f$
- c) $(a==b \&& c==d) \&& e==f$

Ejercicio 6.7.2: En la figura 6.47(a) está el esquema de un programa, y la figura 6.47(b) muestra un bosquejo de la estructura del código de tres direcciones que se genera, usando la traducción con parcheo de retroceso de la figura 6.46. Aquí, los símbolos i_1 a i_8 son las etiquetas de las instrucciones generadas que empiezan cada una de las secciones de “Código”. Cuando implementamos esta traducción, mantenemos para cada expresión booleana E dos listas de posiciones en el código para E , las cuales denotamos mediante $E.true$ y $E.false$. Las posiciones en la lista $E.true$ son aquellas posiciones en las que, en un momento dado, colocamos la etiqueta de la instrucción a la cual debe fluir el control cada vez que E es verdadera; $E.false$ lista de manera similar las posiciones en donde debemos colocar la etiqueta hacia la que fluye el control cuando se descubre que E es falsa. Además, para cada instrucción S mantenemos una lista de posiciones en donde debemos colocar la etiqueta hacia la cual fluye el control cuando S termina. Proporcione el valor (uno de i_1 a i_8) que en un momento dado sustituye a cada posición en cada una de las siguientes listas:

- (a) $E_3.false$ (b) $S_2.siguinte$ (c) $E_4.false$ (d) $S_1.siguinte$ (e) E_2true

```

while ( $E_1$ ) {
    if ( $E_2$ )
        while ( $E_3$ )
             $S_1$ ;
    else {
        if ( $E_4$ )
             $S_2$ ;
         $S_3$ 
    }
}

```

(a)

i_1 : Código para E_1
 i_2 : Código para E_2
 i_3 : Código para E_3
 i_4 : Código para S_1
 i_5 : Código para E_4
 i_6 : Código para S_2
 i_7 : Código para S_3
 i_8 : ...

(b)

Figura 6.47: Estructura de flujo de control del programa para el ejercicio 6.7.2

Ejercicio 6.7.3: Al realizar la traducción de la figura 6.47 usando el esquema de la figura 6.46, creamos listas $S_i.sigiente$ para cada instrucción, empezando con las instrucciones de asignación S_1 , S_2 y S_3 , y continuando con instrucciones if, if-else, while y bloques de instrucciones cada vez más grandes. En la figura 6.47 hay cinco instrucciones construidas de este tipo:

S_4 : **while** (E_3) S_1 .

S_5 : **if** (E_4) S_2 .

S_6 : El bloque que consiste en S_5 y S_3 .

S_7 : La instrucción **if** S_4 **else** S_6 .

S_8 : El programa completo.

Para cada una de estas instrucciones construidas, hay una regla que nos permite construir a $S_i.sigiente$ en términos de otras listas $S_j.sigiente$, y las listas $E_k.true$ y $E_k.false$ para las expresiones en el programa. Proporcione las reglas para

- (a) $S_4.sigiente$ (b) $S_5.sigiente$ (c) $S_6.sigiente$ (d) $S_7.sigiente$ (e) $S_8.sigiente$

6.8 Instrucciones switch

La instrucción “switch” o “case” está disponible en una variedad de lenguajes. Nuestra sintaxis de la instrucción switch aparece en la figura 6.48. Hay una expresión selectora E , que se va a evaluar, seguida de n valores constantes V_1 , V_2 , ..., V_n que la expresión podría recibir, tal vez incluyendo un “valor” *predeterminado (default)*, que siempre coincide con la expresión si ningún otro valor lo hace.

```

switch ( E ) {
    case V1: S1
    case V2: S2
    ...
    case Vn-1: Sn-1
    default: Sn
}

```

Figura 6.48: Sintaxis de la instrucción switch

6.8.1 Traducción de las instrucciones switch

La traducción que se pretende de una instrucción switch es el código para:

1. Evaluar la expresión *E*.
2. Buscar el valor *V_j* en la lista de casos que sea igual al valor de la expresión. Recuerde que el valor predeterminado coincide con la expresión si no coincide ninguno de los valores que se mencionaron explícitamente en los casos.
3. Ejecutar la instrucción *S_j* asociada con el valor encontrado.

El paso (2) es una bifurcación de *n* vías, la cual se puede implementar en una de varias formas. Si el número de casos es pequeño, por decir 10 como máximo, entonces es razonable utilizar una secuencia de saltos condicionales, cada uno de los cuales evalúa un valor individual y nos transfiere al código para la instrucción correspondiente.

Una manera compacta de implementar esta secuencia de saltos condicionales es mediante la creación de una tabla de pares, en donde cada par consiste en un valor y una etiqueta para el correspondiente código de la instrucción. El valor de la misma expresión, a la par con la etiqueta de la instrucción predeterminada, se coloca al final de la tabla en tiempo de ejecución. Un ciclo simple generado por el compilador compara el valor de la expresión con cada valor en la tabla, con lo cual se asegura que si no se encuentra otra coincidencia, la última entrada (predeterminada) coincidirá sin duda.

Si el número de valores excede a 10 o más, es más eficiente construir una tabla de hash para los valores, con las etiquetas de las diversas instrucciones como entradas. Si no se encuentra una entrada para el valor que posee la expresión switch, se genera un salto a la instrucción predeterminada.

Hay un caso especial común que puede implementarse incluso en forma más eficiente que mediante una bifurcación de *n* vías. Si todos los valores se encuentran en cierto rango pequeño, por decir de *mín* a *máx*, y el número de valores distintos es una fracción razonable de *máx* – *mín*, entonces podemos construir un arreglo de *máx* – *mín* “baldes”, en donde el balde *j* – *mín* contiene la etiqueta de la instrucción con el valor *j*; cualquier balde que de otra forma permanecería sin llenar contiene la etiqueta predeterminada.

Para ejecutar el switch, evalúe la expresión para obtener el valor *j*; compruebe que se encuentre en el rango de *mín* a *máx* y transfiera de manera indirecta a la entrada de la tabla, en el desplazamiento *j* – *mín*. Por ejemplo, si la expresión es de tipo carácter, se podría crear una

tabla de, por decir, 128 entradas (dependiendo del conjunto de caracteres) y se podrían realizar transferencias a través de la misma sin comprobación de rangos.

6.8.2 Traducción orientada por la sintaxis de las instrucciones switch

El código intermedio en la figura 6.49 es una traducción conveniente de la instrucción switch en la figura 6.48. Todas las pruebas aparecen al final, de manera que un simple generador de código puede reconocer la bifurcación de varias vías y generar código eficiente para la misma, usando la implementación más apropiada que se sugiere al principio de esta sección.

```

código para evaluar E y colocarla en t
goto prueba
L1:    código para S1
goto siguiente
L2:    código para S2
goto
...
Ln-1:  código para Sn-1
goto siguiente
Ln:    código para Sn
goto siguiente
test:    if t = V1 goto L1
          if t = V2 goto L2
          ...
          if t = Vn-1 goto Ln-1
          goto Ln
siguiente:

```

Figura 6.49: Traducción de una instrucción switch

La secuencia más directa que se muestra en la figura 6.50 requeriría que el compilador realizara un análisis extenso para encontrar la implementación más eficiente. Observe que no es conveniente en un compilador de una pasada colocar las instrucciones de bifurcación al principio, ya que el compilador entonces no podría emitir código para cada una de las instrucciones S_i como las vimos.

Para traducir a la forma de la figura 6.49, cuando vemos la palabra clave **switch**, generamos dos nuevas etiquetas **prueba** y **siguiente**, y un nuevo nombre temporal t . Después, al analizar sintácticamente la expresión E , generamos código para evaluar E y colocarla en t . Después de procesar E , generamos el salto **goto prueba**.

Después, al analizar cada palabra clave **case**, creamos una nueva etiqueta L_i y la introducimos en la tabla de símbolos. Colocamos en una cola, que se utiliza sólo para guardar casos, un par valor-etiqueta que consiste en el valor V_i de la constante del caso y L_i (o un apuntador a la entrada en la tabla de símbolos para L_i). Procesamos cada instrucción **case** $V_i: S_i$ emitiendo la etiqueta L_i que se adjunta al código para S_i , seguida por el salto **goto siguiente**.

```

código para evaluar  $E$  y colocarla en  $t$ 
if  $t \neq V_1$  goto  $L_1$ 
código para evaluar  $S_1$ 
goto siguiente
L1: if  $t \neq V_2$  goto  $L_2$ 
código para evaluar  $S_2$ 
goto siguiente
L2: ...
Ln-2: if  $t \neq V_{n-1}$  goto  $L_{n-1}$ 
código para evaluar  $S_{n-1}$ 
goto siguiente
Ln-1: código para evaluar  $S_n$ 
siguiente:

```

Figura 6.50: Otra traducción de una instrucción switch

Al llegar al final de la instrucción switch, estamos listos para generar el código para la bifurcación de n vías. Al leer la cola de pares valor-etiqueta, podemos generar una secuencia de instrucciones de tres direcciones de la forma que se muestra en la figura 6.51. Ahí, t es el nombre temporal que contiene el valor de la expresión selectora E , y L_n es la etiqueta para la instrucción predeterminada.

```

case  $t = V_1$  L1
case  $t = V_2$  L2
...
case  $t = V_{n-1}$  Ln-1
case  $t = t$  Ln
label siguiente

```

Figura 6.51: Instrucciones case con código de tres direcciones, utilizadas para traducir una instrucción switch

La instrucción `case $t = V_i$ Li` es un sinónimo para `if $t = V_i$ goto Li` en la figura 6.49, pero la instrucción `case` es más fácil de detectar para el generador de código final, como candidato para un tratamiento especial. En la fase de generación de código, estas secuencias de instrucciones `case` pueden traducirse en una bifurcación de n vías del tipo más eficiente, dependiendo de cuántas haya y de si los valores se encuentran dentro de un rango pequeño.

6.8.3 Ejercicios para la sección 6.8

! Ejercicio 6.8.1: Para poder traducir una instrucción switch en una secuencia de instrucciones case como en la figura 6.51, el traductor necesita crear la lista de pares valor-etiqueta, a medida

que procesa el código fuente para la instrucción `switch`. Para hacer esto, usamos una traducción adicional que acumula sólo los pares. Bosqueje una definición orientada por la sintaxis que produzca la lista de pares, y que a la vez emita código para las instrucciones S_i que sean las acciones para cada caso.

6.9 Código intermedio para procedimientos

En el capítulo 7 hablaremos con detalle sobre los procedimientos y su implementación, junto con la administración en tiempo de ejecución del almacenamiento para los nombres. Utilizamos el término función en esta sección para un procedimiento que devuelve un valor. Hablaremos brevemente sobre las declaraciones de las funciones y el código de tres direcciones para las llamadas a funciones. En el código de tres direcciones, una llamada a una función se desglosa en la evaluación de parámetros en preparación para una llamada, seguida de la llamada en sí. Por simplicidad vamos a suponer que los parámetros se pasan por valor; en la sección 1.6.6 hablamos sobre los métodos de paso por valor.

Ejemplo 6.25: Suponga que `a` es un arreglo de enteros, y que `f` es una función de enteros a enteros. Entonces, la siguiente asignación:

```
n = f(a[i]);
```

podría traducirse en el siguiente código de tres direcciones:

- 1) $t_1 = i * 4$
- 2) $t_2 = a [t_1]$
- 3) param t_2
- 4) $t_3 = \text{call } f, 1$
- 5) $n = t_3$

Las primeras dos líneas calculan el valor de la expresión `a[i]` y lo colocan en el nombre temporal `t2`, como vimos en la sección 6.4. La línea 3 convierte a `t2` en un parámetro real para la llamada en la línea 4 de `f` con un parámetro. La línea 5 asigna a `t3` el valor devuelto por la llamada a la función. La línea 6 asigna el valor devuelto a `n`. \square

Las producciones en la figura 6.52 permiten definiciones de funciones y llamadas a funciones. (La sintaxis genera comas no deseadas después del último parámetro, pero es suficiente como para ilustrar la traducción). Los no terminales `D` y `T` generan declaraciones y tipos, en forma respectiva, como en la sección 6.3. La definición de una función generada por `D` consiste en la palabra clave **define**, un tipo de valor de retorno, el nombre de la función, parámetros formales entre paréntesis y el cuerpo de una función que consiste en una instrucción. El no terminal `F` genera cero o más parámetros formales, en donde un parámetro formal consiste en un tipo, seguido de un identificador. Los no terminales `S` y `E` generan instrucciones y expresiones, respectivamente. La producción para `S` agrega una instrucción que devuelve el valor de una expresión. La producción para `E` agrega llamadas a funciones, con parámetros reales generados por `A`. Un parámetro real es una expresión.

$$\begin{aligned}
 D &\rightarrow \mathbf{define} \ T \ \mathbf{id} \ (\ F \) \ \{ \ S \ \} \\
 F &\rightarrow \epsilon \mid T \ \mathbf{id} \ , \ F \\
 S &\rightarrow \mathbf{return} \ E \ ; \\
 E &\rightarrow \mathbf{id} \ (\ A \) \\
 A &\rightarrow \epsilon \mid E \ , \ A
 \end{aligned}$$

Figura 6.52: Agregar funciones al lenguaje fuente

Las definiciones de funciones y las llamadas a funciones pueden traducirse mediante el uso de conceptos que ya hemos introducido en este capítulo.

- *Tipos de funciones.* El tipo de una función debe codificar el tipo de valor de retorno y los tipos de los parámetros formales. Sea *void* un tipo especial que representa a ningún parámetro o ningún tipo de valor de retorno. El tipo de una función *pop()* que devuelve un entero es, por lo tanto, una “función de *void* a *integer*”. Los tipos de las funciones pueden representarse mediante el uso de un constructor *fun* aplicado al tipo de valor de retorno y una lista ordenada de tipos para los parámetros.
- *Tablas de símbolos.* Sea *s* la tabla de símbolos superior cuando se llega a la definición de la función. El nombre de la función se introduce en *s* para usarse en el resto del programa. Los parámetros formales de una función pueden manejarse en analogía con los nombres de los campos en un registro (vea la figura 6.18). En la producción para *D*, después de ver **define** y el nombre de la función, metemos a *s* y establecemos una nueva tabla de símbolos:

Env.push(superior); superior = new Env(tope);

Llamemos a la nueva tabla de símbolos, *t*. Observe que *superior* se pasa como parámetro en **new Env(tope)**, por lo que la nueva tabla de símbolos *t* puede enlazarse con la anterior, *s*. La nueva tabla *t* se utiliza para traducir el cuerpo de la función. Regresamos a la tabla de símbolos anterior *s* una vez que se traduce el cuerpo de la función.

- *Comprobación de tipos.* Dentro de las expresiones, una función se trata como cualquier otro operador. Por lo tanto, se aplica la explicación sobre la comprobación de tipos en la sección 6.5.2, incluyendo las reglas para las coerciones. Por ejemplo, si *f* es una función con un parámetro de tipo real, entonces el entero 2 se coerciona con un real en la llamada *f(2)*.
- *Llamadas a funciones.* Al generar instrucciones de tres direcciones para la llamada a una función **id**(*E, E, ..., E*), basta con generar las instrucciones de tres direcciones para evaluar o reducir los parámetros *E* a direcciones, seguidos de una instrucción **param** para cada parámetro. Si no queremos combinar las instrucciones para evaluar parámetros con las instrucciones **param**, podemos guardar el atributo *E.dir* para cada expresión *E* en una estructura de datos tal como una cola. Una vez que se traducen todas las expresiones, pueden generarse instrucciones **param** a medida que se vacía la cola.

El procedimiento es una construcción de programación tan importante y con un uso tan frecuente, que es imperativo para un compilador generar buen código para las llamadas y los retornos de los procedimientos. Las rutinas en tiempo de ejecución que manejan el paso de parámetros, las llamadas y los retornos de los procedimientos, forman parte del paquete de soporte en tiempo de ejecución. En el capítulo 7 hablaremos sobre los mecanismos para el soporte en tiempo de ejecución.

6.10 Resumen del capítulo 6

Las técnicas que vimos en este capítulo se pueden combinar para construir un front-end simple de un compilador, como el del apéndice A. Ésta se puede construir en forma incremental:

- ◆ *Elegir una representación intermedia:* Por lo general, ésta es cierta combinación de una notación gráfica y código de tres direcciones. Al igual que en los árboles sintácticos, un nodo en una notación gráfica representa a una construcción; los hijos del nodo representan a sus subconstrucciones. El código de tres direcciones recibe su nombre debido a las instrucciones de la forma $x = y \text{ op } z$, con un operador por instrucción como máximo. Hay instrucciones adicionales para el flujo de control.
- ◆ *Traducir expresiones:* Las expresiones con operaciones integradas pueden desglosarse en una secuencia de operaciones individuales, adjuntando acciones a cada producción de la forma $E \rightarrow E_1 \text{ op } E_2$. La acción crea un nodo para E con los nodos para E_1 y E_2 como hijos, o genera una instrucción de tres direcciones que aplica **op** a las direcciones para E_1 y E_2 , y coloca el resultado en un nuevo nombre temporal, el cual se convierte en la dirección para E .
- ◆ *Comprobar tipos:* El tipo de una expresión $E_1 \text{ op } E_2$ se determina mediante el operador **op** y los tipos de E_1 y E_2 . Una coerción es una conversión de tipos implícita, como la conversión de *integer* a *flota*. El código intermedio contiene conversiones de tipos explícitas para asegurar una coincidencia exacta entre los tipos de los operandos y los tipos que espera un operador.
- ◆ *Usar una tabla de símbolos para implementar las declaraciones:* Una declaración especifica el tipo de un nombre. El tamaño de un tipo es la cantidad de almacenamiento necesario para un nombre con ese tipo. Con el uso de tamaños la dirección relativa de un nombre en tiempo de ejecución puede calcularse como un desplazamiento a partir del inicio de una nueva área de datos. El tipo y la dirección relativa de un nombre se colocan en la tabla de símbolos debido a una declaración, para que el traductor pueda obtenerlas más adelante, cuando el nombre aparece en una expresión.
- ◆ *Apilar arreglos:* Para un acceso rápido, los elementos de los arreglos se almacenan en ubicaciones consecutivas. Los arreglos de arreglos se apilan, de manera que puedan tratarse como un arreglo unidimensional de elementos individuales. El tipo de un arreglo se utiliza para calcular la dirección de un elemento del arreglo, relativo a la base del mismo.
- ◆ *Generar código de salto para expresiones booleanas:* En el código de corto circuito o de salto, el valor de una expresión booleana está implícito en la posición a la que se llega en el código. El código de salto es útil, ya que, por lo general, una expresión booleana B

se utiliza para el flujo de control, como en **if** (B) S . Los valores booleanos pueden calcularse mediante el salto hacia $t = \text{true}$ o $t = \text{false}$, según sea apropiado, en donde t es un nombre temporal. Usando etiquetas para los saltos, una expresión booleana puede traducirse al heredar etiquetas que correspondan con sus salidas de verdadero y de falso. Las constantes *true* y *false* se traducen en un salto hacia las salidas de verdadero y de falso, respectivamente.

- ◆ *Implementar instrucciones mediante el uso del flujo de control:* Las instrucciones pueden traducirse heredando una etiqueta *siguiente*, en donde *siguiente* marca la primera instrucción después del código para esta instrucción. La condicional $S \rightarrow \text{if } (B) S_1$ puede traducirse adjuntando una nueva etiqueta que marque el principio del código para S_1 y pasando la nueva etiqueta junto con $S.\text{siguiente}$ a las salidas de verdadero y de falso, respectivamente, de B .
- ◆ *Como alternativa, usar parcheo de retroceso:* Es una técnica para generar código en las expresiones booleanas y las instrucciones de una sola pasada. La idea es mantener listas de saltos incompletos, en donde todas las instrucciones de salto en una lista tienen el mismo destino. Cuando el destino se vuelve conocido, todas las instrucciones en su lista se completan llenando el destino.
- ◆ *Implementar registros:* Los nombres de los campos en un registro o en una clase pueden tratarse como una secuencia de declaraciones. Un tipo de registro codifica los tipos y las direcciones relativas de los campos. Podemos usar un objeto de tabla de símbolos para este propósito.

6.11 Referencias para el capítulo 6

La mayoría de las técnicas en este capítulo provienen de la oleada de actividad de diseño e implementación alrededor de Algol 60. La traducción orientada por la sintaxis al código intermedio estaba bien establecida para cuando se crearon Pascal [11] y C [6, 9].

UNCOL (Lenguaje universal orientado a compiladores) es un lenguaje intermedio universal mítico, que existe desde mediados de la década de 1950. Mediante un UNCOL podían construirse compiladores al enlazar un front-end para cierto lenguaje fuente con el back-end para cierto lenguaje destino [10]. Las técnicas autosuficientes que se proporcionan en el reporte [10] se utilizan de manera rutinaria para cambiar el destino de los compiladores.

El ideal de UNCOL en cuanto a combinar y relacionar interfaces de usuario con back-ends se ha logrado en varias formas. Un compilador redirigible consiste en un front-end que puede unirse con varios back-ends para implementar cierto lenguaje en varias máquinas. Neliac fue uno de los primeros ejemplos de un lenguaje con un compilador redirigible [5] escrito en su propio lenguaje. Otro método es modernizar un front-end para un nuevo lenguaje con un compilador existente. Feldman [2] describe cómo agregar un front-end de Fortran 77 a los compiladores de C [6] y [9]. GCC, la Colección de compiladores GNU [3], soporta front-ends para C, C++, Objective-C, Fortran, Java y Ada.

Los números de valor y su implementación mediante tablas de hash provienen de Ershov [1].

El uso de la información de los tipos para mejorar la seguridad de los códigos de byte de Java se describe por Gosling [4].

La inferencia de tipos mediante el uso de la unificación para resolver conjuntos de ecuaciones se ha vuelto a describir varias veces; Milner [7] describe su aplicación para ML. Vea Pierce [8] para un tratamiento detallado sobre los tipos.

1. Ershov, A. P., “On programming of arithmetic operations”, *Comm. ACM* **1**:8 (1958), pp. 3-6. Vea también *Comm. ACM* **1**:9 (1958), p. 16.
2. Fieldman, S. L., “Implementation of a portable Fortran 77 compiler using modern tools”, *ACM SIGPLAN Notices* **14**:8 (1979), pp. 98-106.
3. Página inicial de GCC: <http://gcc.gnu.org/>, Fundación de software libre.
4. Gosling, J., “Java intermediate bytecodes”, *Proc. ACM SIGPLAN Workshop on Intermediate Representations* (1995), pp. 111-118.
5. Huskey, H. D., M. H. Halstead y R. McArthur, “Neliac: un dialecto de Algol”, *Comm. ACM* **3**:8 (1960), pp. 463-468.
6. Johnson, S. C., “A tour through the portable C compiler”, Bell Telephone Laboratories, Inc., Murray Hill, N. J., 1979.
7. Milner, R., “A theory of type polymorphism in programming”, *J. Computer and System Sciences* **17**:3 (1978), pp. 348-375.
8. Pierce, B. C., *Types and Programming Languages*, MIT Press, Cambridge, Mass., 2002.
9. Ritchie, D. M., “A tour through the UNIX C compiler”, Bell Telephone Laboratories, Inc., Murray Hill, N. J., 1979.
10. Strong, J., J. Wegstein, A. Tritter, J. Olsztyn, O. Mock y T. Steel, “The problem of programming communication with changing machines: a proposed solution”, *Comm. ACM* **1**:8 (1958), pp. 12-18. Parte 2: **1**:9 (1958), pp. 9-15. Informe del comité Share Ad-Hoc sobre Lenguajes Universales.
11. Wirth, N. “The design of a Pascal compiler”, *Software—Practice and Experience* **1**:4 (1971), pp. 309-333.

Capítulo 7

Entornos en tiempo de ejecución

Un compilador debe implementar con precisión las abstracciones expresadas en la definición del lenguaje fuente. Por lo general, estas abstracciones incluyen los conceptos que vimos en la sección 1.6, como los nombres, alcances, enlaces, tipos de datos, operadores, procedimientos, parámetros y construcciones de flujo de control. El compilador debe cooperar con el sistema operativo y demás software del sistema para dar soporte a estas abstracciones en la máquina destino.

Para ello, el compilador crea y administra un *entorno en tiempo de ejecución*, en el cual supone que se están ejecutando sus programas destino. Este entorno se encarga de varias cuestiones como la distribución y la asignación de las ubicaciones de almacenamiento para los objetos nombrados en el programa fuente, los mecanismos que usa el programa destino para acceder a las variables, los enlaces entre los procedimientos, los mecanismos para el paso de parámetros y las interfaces para el sistema operativo, los dispositivos de entrada/salida, y otros programas.

Los dos temas de este capítulo son la asignación de ubicaciones de almacenamiento y el acceso a las variables y datos. Vamos a hablar con detalle sobre la administración de la memoria, incluyendo la asignación de la pila, la administración del montículo de datos y la recolección de basura. En el siguiente capítulo, presentaremos las técnicas para generar código destino para muchas construcciones comunes del lenguaje.

7.1 Organización del almacenamiento

Desde la perspectiva del desarrollador de compiladores, el programa destino se ejecuta en su propio espacio de direcciones lógicas, en donde cada valor del programa tiene una ubicación. La administración y organización de este espacio de direcciones lógicas se comparte entre el compilador, el sistema operativo y la máquina destino. El sistema operativo asigna las direcciones lógicas en direcciones físicas que, por lo general, están esparcidas a lo largo de la memoria.

La representación en tiempo de ejecución de un programa objeto en el espacio de direcciones lógicas consiste en datos y área del programa, como se muestra en la figura 7.1. Un compilador

para un lenguaje como C++ en un sistema operativo como Linux podría subdividir la memoria de esta forma:

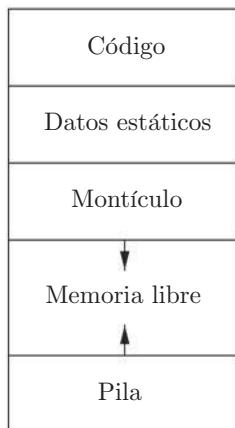


Figura 7.1: Subdivisión típica de la memoria en tiempo de ejecución, en áreas de código y de datos

A lo largo de este libro vamos a suponer que el almacenamiento en tiempo de ejecución viene en bloques de bytes contiguos, en donde un byte es la unidad más pequeña de memoria direccionable. Un byte es de ocho bits y cuatro bytes forman una palabra de máquina. Los objetos multibyte se almacenan en bytes consecutivos y reciben la dirección del primer byte.

Como vimos en el capítulo 6, la cantidad de almacenamiento necesario para un nombre se determina en base a su tipo. Un tipo de datos elemental, como un carácter, entero o punto flotante, puede almacenarse en un número integral de bytes. El almacenamiento para un tipo agregado, como un arreglo o estructura, debe ser lo bastante grande como para que pueda contener todos sus componentes.

La distribución del almacenamiento para los objetos de datos recibe una fuerte influencia por parte de las restricciones de direccionamiento de la máquina destino. En muchas máquinas, las instrucciones para sumar enteros podrían esperar que los enteros estuvieran *alineados*, es decir, que se coloquen en una dirección que pueda dividirse entre 4. Aunque un arreglo de diez caracteres sólo necesita suficientes bytes para guardar diez caracteres, un compilador podría asignar 12 bytes para obtener la alineación apropiada, dejando 2 bytes sin usar. El espacio que queda sin usarse debido a las consideraciones de alineación se conoce como *relleno*. Cuando el espacio es primordial, un compilador puede *empaquetar* los datos de manera que no quede relleno; así, tal vez haya que ejecutar instrucciones adicionales en tiempo de ejecución para posicionar los datos empaquetados, de manera que puedan manejarse como si estuvieran alineados en forma apropiada.

El tamaño del código destino generado es fijo en tiempo de compilación, por lo que el compilador puede colocar el código destino ejecutable en un área determinada en forma estática llamada *Código*, por lo general, en la parte inferior de la memoria. De manera similar, el tamaño de ciertos objetos de datos del programa, como las constantes globales, y los datos generados por el compilador, como la información para soportar la recolección de basura, pueden conocerse en tiempo de compilación, y estos objetos de datos pueden colocarse en otra área determinada

en forma estática, llamada *Estática*. Una razón para asignar en forma estática todos los objetos de datos posibles es que las direcciones de estos objetos pueden compilarse en el código destino. En las primeras versiones de Fortran, todos los objetos de datos podían asignarse en forma estática.

Para incrementar al máximo el uso de espacio en tiempo de ejecución, las otras dos áreas, *Pila* y *Montículo*, se encuentran en extremos opuestos del resto del espacio de direcciones. Estas áreas son dinámicas; su tamaño puede cambiar a medida que se ejecuta el programa. Estas áreas crecen una hacia la otra, según sea necesario. La pila se utiliza para almacenar estructuras de datos conocidas como registros de activación, que se generan durante las llamadas a procedimientos.

En la práctica, la pila crece hacia las direcciones inferiores, y el montículo hacia las superiores. Sin embargo, a lo largo de este capítulo y del siguiente vamos a suponer que la pila crece hacia las direcciones superiores, de manera que podamos usar desplazamientos positivos por conveniencia de notación en todos nuestros ejemplos.

Como veremos en la siguiente sección, un registro de activación se utiliza para almacenar información sobre el estado de la máquina, como el valor del contador del programa y los registros de la máquina, cuando ocurre una llamada a un procedimiento. Cuando el control regresa de la llamada, la activación del procedimiento que se llamó puede reiniciarse después de restaurar los valores de los registros relevantes y establecer el contador del programa al punto justo después de la llamada. Los objetos de datos cuyos tiempos de vida están contenidos en el de una activación pueden asignarse en la pila, junto con la demás información asociada con ella.

Muchos lenguajes de programación permiten al programador asignar y desasignar datos bajo el control del programa. Por ejemplo, C tiene las funciones `malloc` y `free` que pueden usarse para obtener y devolver pedazos arbitrarios de almacenamiento. El montículo se utiliza para administrar este tipo de datos de larga duración. En la sección 7.4 veremos varios algoritmos de administración de memoria que pueden usarse para mantener el montículo.

7.1.1 Asignación de almacenamiento estática y dinámica

La distribución y asignación de los datos a ubicaciones de memoria en el entorno de ejecución son cuestiones clave en la administración del almacenamiento. Estas cuestiones son engañosas, ya que el mismo nombre en el texto de un programa puede hacer referencia a varias ubicaciones en tiempo de ejecución. Los dos adjetivos *estático* y *dinámico* diferencian entre el tiempo de compilación y el tiempo de ejecución, respectivamente. Decimos que una decisión de asignación de almacenamiento es *estática* si el compilador la puede realizar con sólo ver el texto del programa, no lo que el programa hace cuando se ejecuta. Por el contrario, una decisión es *dinámica* si puede decidirse sólo cuando el programa se está ejecutando. Muchos compiladores utilizan cierta combinación de las dos estrategias para la asignación de almacenamiento dinámica:

1. *Almacenamiento de la pila.* A los nombres locales para un procedimiento se les asigna espacio en una pila. A partir de la sección 7.2 veremos la “pila en tiempo de ejecución”. La pila soporta la política normal de llamadas/retornos de los procedimientos.
2. *Almacenamiento del montículo.* Los datos que pueden durar más que la llamada al procedimiento que los creó se asignan, por lo general, en un “montículo” de almacenamiento reutilizable. A partir de la sección 7.4 veremos la administración del montículo. El montículo es

un área de memoria que permite que los objetos u otros elementos de datos obtengan almacenamiento cuando se crean, y que devuelvan ese almacenamiento cuando se invalidan.

Para dar soporte a la administración de la pila, la “recolección de basura” (garbage collection) permite que el sistema en tiempo de ejecución detecte los elementos de datos inútiles y reutilice su almacenamiento, incluso si el programador no devuelve su espacio en forma explícita. La recolección automática de basura es una característica esencial de muchos lenguajes modernos, a pesar de ser una operación difícil de realizarse con eficiencia; tal vez no sea posible para ciertos lenguajes.

7.2 Asignación de espacio en la pila

Casi todos los compiladores que utilizan procedimientos, funciones o métodos como unidades de acciones definidas por el usuario administran por lo menos una parte de su memoria en tiempo de ejecución como una pila. Cada vez que se hace una llamada a un procedimiento,¹ el espacio para sus variables locales se mete en una pila y, cuando el procedimiento termina, ese espacio se saca de la pila. Como veremos más adelante, este arreglo no sólo permite compartir el espacio entre las llamadas a procedimientos cuyas duraciones no se traslapan en el tiempo, sino que también nos permite compilar código para un procedimiento de tal forma que las direcciones relativas de sus variables no locales siempre sean las mismas, sin importar la secuencia de las llamadas a los procedimientos.

7.2.1 Árboles de activación

La asignación de la pila no sería posible si las llamadas a los procedimientos, o las *activaciones* de los procedimientos, no se anidaran en el tiempo. El siguiente ejemplo muestra el anidamiento de las llamadas a procedimientos.

Ejemplo 7.1: La figura 7.2 contiene un bosquejo de un programa que lee nueve enteros, los coloca en un arreglo *a* y los ordena mediante el algoritmo de quicksort recursivo.

La función principal tiene tres tareas. Llama a *leerArreglo*, establece los centinelas y después llama a *quicksort* para que opere con todo el arreglo de datos completo. La figura 7.3 sugiere una secuencia de llamadas que podrían resultar de una ejecución del programa. En esta ejecución, la llamada a *particion(1, 9)* devuelve 4, por lo que *a[1]* a *a[3]* contienen elementos menores que su valor pivote *v* elegido, mientras que los elementos más grandes están en *a[5]* hasta *a[9]*. □

En este ejemplo, y en general, las activaciones de los procedimientos se anidan en el tiempo. Si una activación del procedimiento *p* llama al procedimiento *q*, entonces esa activación de *q* debe terminar antes de que pueda terminar la activación de *p*. Hay tres casos comunes:

1. La activación de *q* termina en forma normal. Entonces en casi cualquier lenguaje, el control se reanuda justo después del punto de *p* en el cual se hizo una llamada a *q*.

¹Recuerde que utilizamos “procedimiento” como un término genérico para función, procedimiento, método o subrutina.

```

int a[11];
void leerArreglo() { /* Lee 9 enteros y los coloca en a[1],..., a[9]. */
    int i;
    ...
}
int particion(int m, int n) {
    /* Elije un valor pivote  $v$  y partitiona  $a[m .. n]$  de manera que
        $a[m .. p - 1]$  sea menor que  $v$ ,  $a[p] = v$  y  $a[p + 1 .. n]$  sea igual
       o mayor que  $v$ . Devuelve  $p$ . */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = particion(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    leerArreglo();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}

```

Figura 7.2: Bosquejo de un programa de quicksort

2. La activación de q , o algún procedimiento q que se haya llamado, ya sea en forma directa o indirecta, aborta; es decir, es imposible que continúe la ejecución. En este caso, p termina en forma simultánea con q .
3. La activación de q termina debido a una excepción que q no puede manejar. El procedimiento p puede manejar la excepción, en cuyo caso la activación de q ha terminado, mientras que la activación de p continúa, aunque no necesariamente desde el punto en el que se hizo la llamada a q . Si p no puede manejar la excepción, entonces esta activación de p termina al mismo tiempo que la activación de q , y se supone que la excepción se manejará por alguna otra activación abierta de un procedimiento.

Por lo tanto, podemos representar las activaciones de los procedimientos durante la ejecución de un programa completo mediante un árbol, conocido como *árbol de activación*. Cada nodo corresponde a una activación, y la raíz es la activación del procedimiento “principal” (main) que inicia la ejecución del programa. En un nodo para una activación del procedimiento p , los hijos corresponden a las activaciones de los procedimientos a los cuales se llama mediante esta activación de p . Mostramos estas activaciones en el orden en el que se llaman, de izquierda a derecha. Observe que un hijo debe terminar antes de que la activación a su derecha pueda empezar.

Una versión de Quicksort

El bosquejo de un programa de ordenamiento rápido (quicksort) en la figura 7.2 utiliza dos funciones auxiliares: *leerArreglo* y *particion*. La función *leerArreglo* se utiliza sólo para cargar los datos en el arreglo *a*. Los elementos primero y último de *a* no se utilizan para datos, sino para “centinelas” que se establecen en la función principal. Asumimos que *a*[0] se establece a un valor menor que cualquier valor de datos posible, y que *a*[10] se establece a un valor mayor que cualquier valor de datos.

La función *particion* divide una porción del arreglo, delimitado por los argumentos *m* y *n*, por lo que los elementos inferiores de *a*[*m*] hasta *a*[*n*] están al principio y los elementos superiores al final, aunque ninguno de los grupos se encuentra necesariamente ordenado. No vamos a profundizar en la forma en que trabaja la función *particion*; basta con decir que puede depender de la existencia de centinelas. Un posible algoritmo para *particion* se sugiere mediante el código más detallado en la figura 9.1.

El procedimiento recursivo *quicksort* primero decide si necesita ordenar más de un elemento del arreglo. Observe que un elemento siempre está “ordenado”, por lo que *quicksort* no tiene nada que hacer en ese caso. Si hay elementos qué ordenar, *quicksort* llama primero a *particion*, la cual devuelve un índice *i* para separar los elementos inferior y superior. Luego, estos dos grupos de elementos se ordenan mediante dos llamadas recursivas a *quicksort*.

Ejemplo 7.2: En la figura 7.4 se muestra un posible árbol de activación que completa la secuencia de llamadas y retornos sugerida en la figura 7.3. Las funciones se representan mediante las primeras letras de sus nombres. Recuerde que este árbol sólo es una posibilidad, ya que los argumentos de las llamadas siguientes, y también el número de llamadas a lo largo de cualquier bifurcación, se ven influenciadas por los valores que devuelve *particion*. \square

El uso de una pila en tiempo de ejecución se permite mediante varias relaciones útiles entre el árbol de activación y el comportamiento del programa:

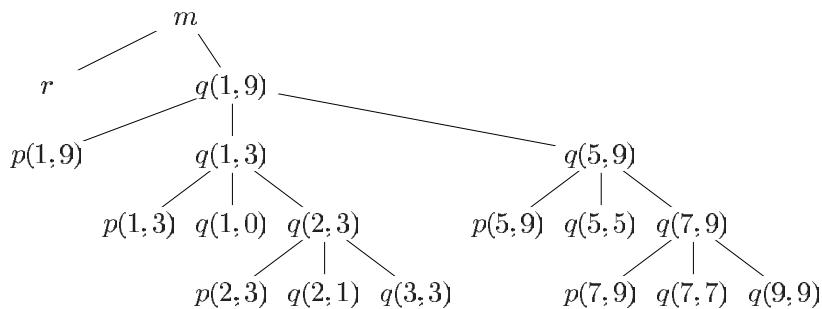
1. La secuencia de llamadas a procedimientos corresponde a un recorrido en preorden del árbol de activación.
2. La secuencia de retornos corresponde a un recorrido en postorden del árbol de activación.
3. Suponga que el control se encuentra dentro de una activación específica de cierto procedimiento, que corresponde a un nodo *N* del árbol de activación. Entonces las activaciones que se encuentran abiertas (*vivas*) son aquellas que corresponden al nodo *N* y sus ancestros. El orden en el que se llamaron estas activaciones es el orden en el que aparecen a lo largo de la ruta hacia *N*, empezando en la raíz, y regresarán en el orden inverso.

```

entrar a main()
entrar a leerArreglo()
salir de leerArreglo()
entrar a quicksort(1,9)
    entrar a particion(1,9)
    salir de particion(1,9)
    entrar a quicksort(1,3)
    ...
    salir de quicksort(1,3)
    entrar a quicksort(5,9)
    ...
    salir de quicksort(5,9)
    salir de quicksort(1,9)
salir de main()

```

Figura 7.3: Activaciones posibles para el programa de la figura 7.2

Figura 7.4: Árbol de activación que representa las llamadas durante una ejecución de *quicksort*

7.2.2 Registros de activación

Por lo general, las llamadas a los procedimientos y los retornos de los mismos se manejan mediante una pila en tiempo de ejecución, conocida como *pila de control*. Cada activación en vivo tiene un *registro de activación* (algunas veces se le conoce como *marco*) en la pila de control, con la raíz del árbol de activación en la parte inferior, y toda la secuencia de registros de activación en la pila que corresponde a la ruta en el árbol de activación que va hacia la activación en la que reside el control en ese momento. Esta última activación tiene su registro en la parte superior de la pila.

Ejemplo 7.3: Si el control se encuentra actualmente en la activación $q(2, 3)$ del árbol de la figura 7.4, entonces el registro de activación para $q(2, 3)$ se encuentra en la parte superior de la pila de control. Justo debajo se encuentra el registro de activación para $q(1, 3)$, el padre de $q(2, 3)$ en el árbol. Debajo de ése se encuentra el registro de activación $q(1, 9)$, y en la parte inferior se encuentra el registro de activación para m , la función principal y raíz del árbol de activación. \square

Vamos a dibujar en forma convencional pilas de control, con la parte inferior de la pila más arriba que la parte superior, por lo que los elementos en un registro de activación que aparecen más abajo en la página en realidad estarán más cerca de la parte superior de la pila.

El contenido de los registros de activación varía con el lenguaje que se esté implementando. He aquí una lista de los tipos de datos que podrían aparecer en un registro de activación (en la figura 7.5 podrá ver un resumen y el orden posible para estos elementos):

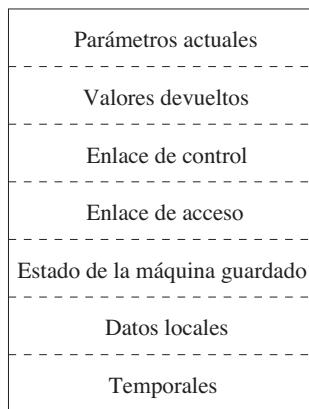


Figura 7.5: Un registro de activación general

1. Los valores temporales, como los que surgen de la evaluación de expresiones, en casos en los que dichas temporales no pueden guardarse en registros.
2. Los datos locales que pertenecen al procedimiento cuyo registro de activación sea éste.
3. Un estado de la máquina guardado, con información acerca del estado de la máquina justo antes de la llamada al procedimiento. Por lo general, esta información incluye la *dirección de retorno* (valor del contador del programa, al que debe regresar el procedimiento al que se llamó) y el contenido de los registros que utiliza el procedimiento al que se llamó y que debe restaurarse cuando ocurra el retorno.
4. Puede ser necesario un “enlace de acceso” para localizar los datos que necesita el procedimiento al que se llamó pero que se encuentran en otra parte; por ejemplo, en otro registro de activación. En la sección 7.3.5 hablaremos sobre los enlaces de acceso.
5. Un *enlace de control*, que apunta al registro de activación del procedimiento que hizo la llamada.
6. Espacio para el valor de retorno de la función a la que se llamó, si lo hay. De nuevo, no todos los procedimientos a los que se llama devuelven un valor, y si uno lo hace, tal vez sea mejor colocar ese valor en un registro, por cuestión de eficiencia.
7. Los parámetros actuales que utiliza el procedimiento que hizo la llamada. Por lo común, estos valores no se colocan en el registro de activación sino en registros, siempre que sea posible, para aumentar la eficiencia. No obstante, mostramos un espacio para ellos, para que sean completamente generales.

Ejemplo 7.4: La figura 7.6 muestra imágenes de la pila en tiempo de ejecución, a medida que el control fluye a través del árbol de activación de la figura 7.4. Las líneas punteadas en los árboles parciales van a las activaciones que han terminado. Como el arreglo a es global, se asigna espacio para éste antes de que empiece la ejecución con una activación del procedimiento *main*, como se muestra en la figura 7.6(a).

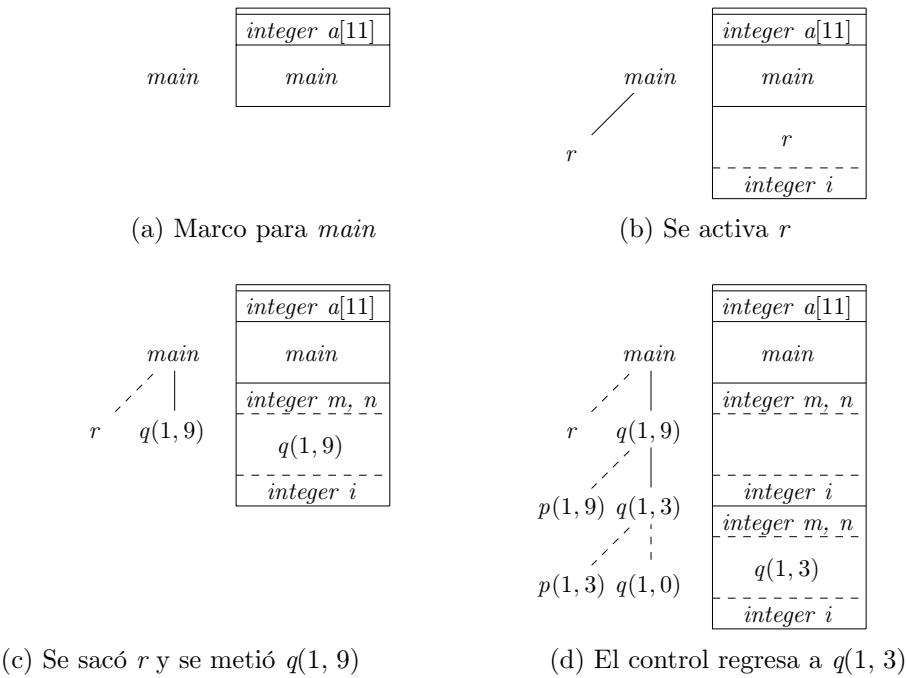


Figura 7.6: Pila de registros de activación que crece hacia abajo

Cuando el control llega a la primera llamada en el cuerpo de *main*, el procedimiento *r* se activa y su registro de activación se mete en la pila (figura 7.6(b)). El registro de activación para *r* contiene espacio para la variable local *i*. Recuerde que la parte superior de la pila se encuentra en la parte inferior de los diagramas. Cuando el control regresa de esta activación, su registro se saca de la pila, dejando sólo el registro para *main* en la pila.

Después, el control llega a la llamada a *q* (quicksort) con los parámetros actuales 1 y 9, y se coloca un registro de activación para esta llamada en la parte superior de la pila, como en la figura 7.6(c). El registro de activación para *q* contiene espacio para los parámetros *m* y *n* junto con la variable local *i*, siguiendo el esquema general en la figura 7.5. Observe que el espacio que una vez utilizó la llamada de *r* se reutiliza en la pila. No habrá rastros de datos locales de *r* disponibles para *q(1, 9)*. Cuando *q(1, 9)* regresa, la pila tiene de nuevo sólo el registro de activación para *main*.

Ocurren varias activaciones entre las últimas dos imágenes de la figura 7.6. Se hizo una llamada recursiva a *q(1, 3)*. Las activaciones *p(1, 3)* y *q(1, 0)* empezaron y terminaron durante el tiempo de vida de *q(1, 3)*, dejando el registro de activación para *q(1, 3)* en la parte superior

(figura 7.6(d)). Observe que cuando un procedimiento es recursivo, es común tener al mismo tiempo varios de sus registros de activación en la pila. \square

7.2.3 Secuencias de llamadas

Las llamadas a los procedimientos se implementan mediante lo que conocemos como *secuencias de llamadas*, las cuales consisten en código que asigna un registro de activación en la pila e introduce información en sus campos. Una *secuencia de retorno* es código similar para restaurar el estado de la máquina, de manera que el procedimiento que hace la llamada pueda continuar su ejecución después de la llamada.

Las secuencias de llamadas y la distribución de los registros de activación pueden diferir en forma considerable, incluso entre implementaciones del mismo lenguaje. A menudo, el código en una secuencia de llamadas se divide entre el procedimiento que hace la llamada (el “emisor”) y el procedimiento al que llama (el “receptor”). No hay una división exacta de las tareas en tiempo de ejecución entre el emisor y el receptor; el lenguaje fuente, la máquina destino y el sistema operativo imponen requerimientos que pueden dar preferencia a una solución sobre otra. En general, si un procedimiento se llama desde n puntos diferentes, entonces la porción de la secuencia de llamadas que se asigna al emisor se genera n veces. Sin embargo, la porción asignada al receptor se genera sólo una vez. Por ende, es conveniente colocar lo más que se pueda de la secuencia de llamadas en el receptor; todo lo que podamos confiar en que el receptor sepa. Sin embargo, más adelante veremos que el receptor no puede saberlo todo.

Al diseñar las secuencias de llamadas y la distribución de los registros de activación, los siguientes principios son útiles:

1. Los valores que se comunican entre el emisor y el receptor por lo general se colocan al principio del registro de activación del receptor, de manera que se encuentren lo más cerca posible del registro de activación del emisor. La motivación es que el emisor puede calcular los valores de los parámetros actuales de la llamada y colocarlos encima de su propio registro de activación, sin tener que crear el registro de activación completo del receptor, o incluso sin tener que conocer la distribución de ese registro. Además, permite el uso de procedimientos que no siempre reciben el mismo número o tipo de argumentos, como la función `printf` de C. El receptor sabe en dónde colocar el valor de retorno, relativo a su propio registro de activación, mientras que los argumentos que haya presentes aparecerán en secuencia debajo de esa posición en la pila.
2. Por lo general, los elementos de longitud fija se colocan en la parte media. En la figura 7.5 podemos ver que dichos elementos por lo general incluyen los campos del enlace de control, el enlace de acceso y el estado de la máquina. Si se guardan exactamente los mismos componentes del estado de la máquina para cada llamada, entonces el mismo código puede realizar los procesos de guardar y restaurar cada uno. Además, si estandarizamos la información de estado de la máquina, entonces será más fácil para los programas como los depuradores descifrar el contenido de la pila, en caso de que ocurra un error.
3. Los elementos cuyo tamaño no se conozca con la suficiente anticipación se colocan al final del registro de activación. La mayoría de las variables locales tienen una longitud fija, que el compilador puede determinar examinando el tipo de la variable. Sin embargo, algunas

variables locales tienen un tamaño que no puede determinarse sino hasta que se ejecuta el programa; el ejemplo más común es un arreglo de tamaño dinámico, en donde el valor de uno de los parámetros del receptor determina la longitud del arreglo. Además, la cantidad de espacio necesario para los valores temporales depende por lo general de qué tan exitosa es la fase de generación de código para mantener los valores temporales en los registros. Por ende, mientras que en un momento dado el compilador conoce el espacio necesario para los valores temporales, tal vez no sepa cuando se genera por primera vez el código intermedio.

4. Debemos localizar con cuidado el apuntador de la parte superior de la pila. Un método común es hacer que apunte al final de los campos de longitud fija en el registro de activación. Así, podemos acceder a los datos de longitud fija mediante desplazamientos fijos, conocidos para el generador de código intermedio, relativos al apuntador de la parte superior de la pila. Una consecuencia de este método es que los campos de longitud variable en los registros de activación se encuentran en realidad “encima” de la parte superior de la pila. Sus desplazamientos deben calcularse en tiempo de ejecución, pero también se puede acceder a ellos desde el apuntador de la parte superior de la pila, mediante el uso de un desplazamiento positivo.

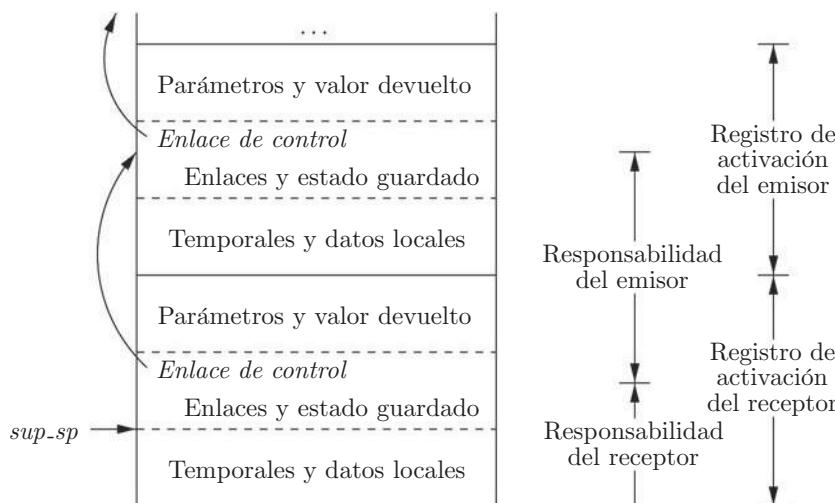


Figura 7.7: División de tareas entre el procedimiento que llama (emisor) y el que recibe la llamada (receptor)

En la figura 7.7 se sugiere un ejemplo de la forma en que pueden cooperar el emisor y el receptor en el manejo de la pila. Un registro *sup-sp* apunta al final del campo de estado de la máquina en el registro de activación superior actual. Esta posición dentro del registro de activación del receptor es conocida para el emisor, por lo que este último puede hacerse responsable de establecer *sup-sp* antes de pasar el control al receptor. La secuencia de llamadas y su división entre emisor y receptor es la siguiente:

1. El emisor evalúa los parámetros actuales.

2. El emisor almacena una dirección de retorno y el valor antiguo de *sup_sp* en el registro de activación del receptor. Después, el emisor incrementa *sup_sp* a la posición que se muestra en la figura 7.7. Es decir, *sup_sp* se mueve más allá de los datos locales del emisor y los temporales, y de los parámetros y campos de estado del receptor.
3. El receptor guarda los valores de los registros y demás información de estado.
4. El receptor inicializa sus datos locales y empieza la ejecución.

Una secuencia de retorno correspondiente y adecuada sería:

1. El receptor coloca el valor de retorno enseguida de los parámetros, como en la figura 7.5.
2. Mediante la información en el campo de estado de la máquina, el receptor restaura a *sup_sp* y a otros registros, y después se bifurca hacia la dirección de retorno que el emisor colocó en el campo de estado.
3. Aunque *sup_sp* se ha decrementado, el emisor sabe en dónde está el valor de retorno, relativo al valor actual de *sup_sp*; por lo tanto, el emisor puede usar ese valor.

Las secuencias anteriores de llamado y retorno permiten que el número de argumentos del procedimiento receptor varíen de llamada en llamada (por ejemplo, como en la función `printf` de C). Observe que en tiempo de compilación, el código destino del emisor conoce el tamaño del área de parámetros. Sin embargo, el código destino del receptor debe estar preparado para manejar otras llamadas también, por lo que espera hasta ser llamado y después examina el campo de parámetros. Usando la organización de la figura 7.7, la información que describe a los parámetros debe colocarse enseguida del campo de estado, para que el receptor pueda encontrarla. Por ejemplo, en la función `printf` de C el primer argumento describe al resto de los argumentos, por lo que una vez que se ha localizado el primer argumento, el emisor puede averiguar en dónde se encuentran los demás.

7.2.4 Datos de longitud variable en la pila

Con frecuencia, el sistema de administración de memoria en tiempo de ejecución debe manejar la asignación de espacio para los objetos cuyo tamaño no se conoce en tiempo de compilación, pero que son locales para un procedimiento y por ende se les puede asignar a la pila. En los lenguajes modernos, a los objetos cuyo tamaño no puede determinarse en tiempo de compilación se les asigna espacio en el montículo de datos, la estructura de almacenamiento que veremos en la sección 7.4. Sin embargo, también es posible asignar objetos, arreglos u otras estructuras de tamaño desconocido en la pila, y aquí veremos cómo hacerlo. La razón de preferir colocar objetos en la pila, si es posible, es que evitamos el proceso de utilizar la recolección de basura para recuperar su espacio. La pila sólo puede usarse para un objeto si éste es local para un procedimiento, y se vuelve inaccesible cuando el procedimiento regresa.

Una estrategia común para asignar arreglos de longitud variable (es decir, arreglos cuyo tamaño depende del valor de uno o más parámetros del procedimiento al que se llamó) se muestra en la figura 7.8. El mismo esquema funciona para los objetos de cualquier tipo, si éstos son locales para el procedimiento al que se llamó y tienen un tamaño que depende de los parámetros de la llamada.

En la figura 7.8 el procedimiento p tiene tres arreglos locales, cuyos tamaños suponemos que no pueden determinarse en tiempo de compilación. El almacenamiento para estos arreglos no forma parte del registro de activación para p , aunque aparece en la pila. Sólo aparece un apuntador al principio de cada arreglo en el mismo registro de activación. Por ende, cuando se ejecuta p , estos apuntadores se encuentran en desplazamientos conocidos desde el apuntador de la parte superior de la pila, por lo que el código destino puede acceder a los elementos a través de estos apuntadores.

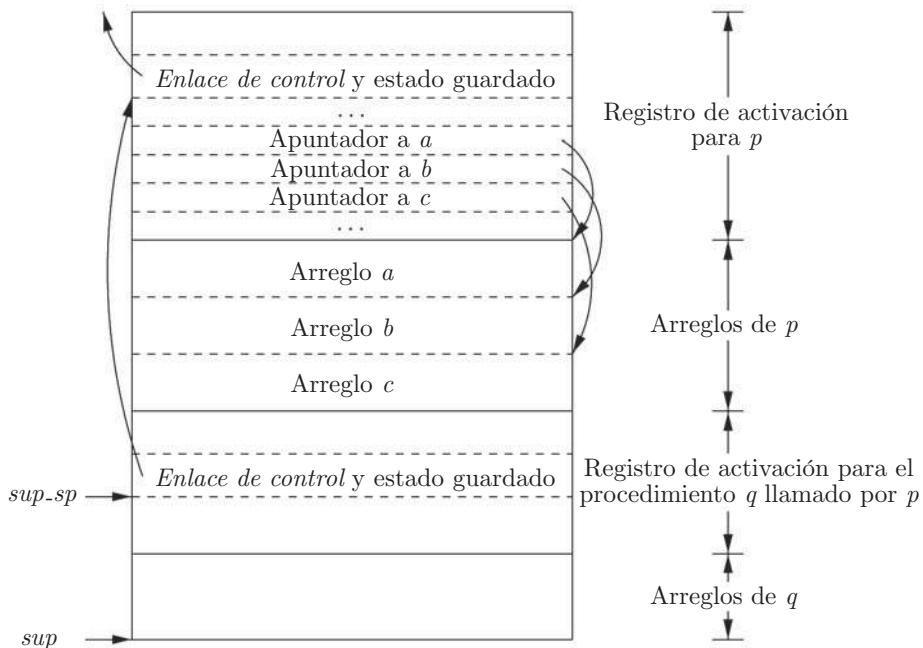


Figura 7.8: Acceso a los arreglos asignados en forma dinámica

En la figura 7.8 también se muestra el registro de activación para un procedimiento q , llamado por p . El registro de activación para q empieza después de los arreglos de p , y cualquier arreglo de longitud variable de q se localiza más allá de eso.

El acceso a los datos en la pila es a través de dos apuntadores, $tope$ y $sup-sp$. Aquí, $tope$ marca la parte superior actual de la pila; apunta a la posición en la que empezará el siguiente registro de activación. El segundo, $sup-sp$, se utiliza para buscar campos locales de longitud fija del registro de activación. Para tener coherencia con la figura 7.7, vamos a suponer que $sup-sp$ apunta al final del campo de estado de la máquina. En la figura 7.8, $sup-sp$ apunta al final de este campo en el registro de activación para q . De ahí podemos encontrar el campo del enlace de control para q , que nos conduce a la posición en el registro de activación para p hacia donde apuntaba $sup-sp$ cuando p estaba en la parte superior.

El código para reposicionar a $tope$ y $sup-sp$ puede generarse en tiempo de compilación, en términos de los tamaños que se conocerán en tiempo de ejecución. Cuando regresa q , $sup-sp$

puede restaurarse desde el enlace de control guardado en el registro de activación para q . El nuevo valor de *tope* es (el antiguo valor sin restaurar de) *sup_sp* menos la longitud de los campos de estado de la máquina, de enlace de control y de acceso, de valor de retorno y de parámetros (como en la figura 7.5) en el registro de activación de q . El emisor conoce esta longitud en tiempo de compilación, aunque puede depender de éste, si el número de parámetros puede variar entre una llamada a q y otra.

7.2.5 Ejercicios para la sección 7.2

Ejercicio 7.2.1: Suponga que el programa de la figura 7.2 utiliza una función *particion* que siempre elige $a[m]$ como el *pivote*. Además, cuando el arreglo $a[m], \dots, a[n]$ se reordena, suponga que el orden se preserva lo más posible. Es decir, primero van todos los elementos menores que v en su orden original, después todos los elementos iguales a v y, por último, todos los elementos mayores que v , en su orden original.

- Dibuje el árbol de activación cuando se ordenan los números 9, 8, 7, 6, 5, 4, 3, 2, 1.
- ¿Cuál es el número mayor de registros de activación que pueden aparecer en un momento dado en la pila?

Ejercicio 7.2.2: Repita el ejercicio 7.2.1 cuando el orden inicial de los números es 1, 3, 5, 7, 9, 2, 4, 6, 8.

Ejercicio 7.2.3: En la figura 7.9 hay código en C para calcular los números de Fibonacci en forma recursiva. Suponga que el registro de activación para f incluye los siguientes elementos en orden: (valor de retorno, argumento n , s local, t local); por lo regular también habrá otros elementos en el registro de activación. Las siguientes preguntas asumen que la llamada inicial es $f(5)$.

- Muestre el árbol de activación completo.
- ¿Cuál es la apariencia de la pila y sus registros de activación, la primera vez que $f(1)$ está a punto de regresar?
- ¿Cuál es la apariencia de la pila y sus registros de activación, la quinta vez que $f(1)$ está a punto de regresar?

Ejercicio 7.2.4: He aquí un bosquejo de dos funciones f y g en C:

```
int f(int x) { int i; ... return i+1; ... }
int g(int y) { int j; ... f(j+1) ... }
```

Es decir, la función g llama a f . Dibuje la parte superior de la pila, empezando con el registro de activación para g , después de que g llama a f , y cuando f está a punto de retornar. Puede considerar sólo valores de retorno, parámetros, enlaces de control y el espacio para las variables locales; no tiene que considerar los valores del estado guardado, temporales o locales que no se muestren en el bosquejo de código. Sin embargo, debe indicar lo siguiente:

```

int f(int n) {
    int t, s;
    if (n < 2) return 1;
    s = f(n-1);
    t = f(n-2);
    return s+t;
}

```

Figura 7.9: Programa de Fibonacci para el ejercicio 7.2.3

- a) ¿Qué función crea el espacio en la pila para cada elemento?
- b) ¿Qué función escribe el valor de cada elemento?
- c) ¿A qué registro de activación pertenece el elemento?

Ejercicio 7.2.5: En un lenguaje que pasa los parámetros por referencia, hay una función $f(x, y)$ que realiza lo siguiente:

```
x = x + 1; y = y + 2; return x+y;
```

Si a recibe el valor de 3, y después se hace una llamada a $f(a, a)$, ¿qué es lo que se devuelve?

Ejercicio 7.2.6: La función f en C se define mediante lo siguiente:

```

int f(int x, *py, ***ppz) {
    **ppz += 1; *py += 2; x += 3; return x+y+z;
}

```

La variable a es un apuntador a b ; la variable b es un apuntador a c , y c es un entero que contiene actualmente el valor de 4. Si llamamos a $f(c, b, a)$, ¿qué se devuelve?

7.3 Acceso a los datos no locales en la pila

En esta sección veremos la forma en que los procedimientos acceden a sus datos. Es muy importante el mecanismo para buscar los datos que se utilizan dentro de un procedimiento p , pero que no pertenecen a p . El acceso se vuelve más complicado en los lenguajes en los que pueden declararse procedimientos dentro de otros procedimientos. Por lo tanto, ahora empezaremos con el caso simple de las funciones en C, y después presentaremos un lenguaje llamado ML, el cual permite declaraciones de funciones anidadas y funciones como “objetos de primera clase”; es decir, las funciones pueden recibir funciones como argumentos y devolver funciones como valores. Esta capacidad se puede realizar mediante la modificación de la implementación de la pila en tiempo de ejecución, por lo que consideraremos varias opciones para modificar los marcos de pila de la sección 7.2.

7.3.1 Acceso a los datos sin procedimientos anidados

En la familia de lenguajes C, todas las variables se definen ya sea dentro de una sola función o fuera de cualquier función (“globalmente”). Lo más importante es que es imposible declarar un procedimiento cuyo alcance esté por completo dentro de otro procedimiento. En vez de ello, una variable global v tiene un alcance que consiste en todas las funciones que van después de la declaración de v , excepto en donde haya una definición local del identificador v . Las variables que se declaran dentro de una función tienen un alcance que consiste sólo en esa función, o parte de ella, si la función tiene bloques anidados, como vimos en la sección 1.6.3.

Para los lenguajes que no permiten declaraciones de procedimientos, la asignación del almacenamiento para las variables y el acceso a esas variables es simple:

1. A las variables globales se les asigna un almacenamiento estático. Las ubicaciones de estas variables permanecen fijas y son conocidas en tiempo de compilación. Así que, para acceder a cualquier variable que no sea local para el procedimiento actual en ejecución, sólo utilizamos la dirección que se determina en forma estática.
2. Cualquier otro nombre debe ser local para la activación en la parte superior de la pila. Podemos acceder a estas variables a través del apuntador sup_sp de la pila.

Un beneficio importante de la asignación estática para las variables globales es que los procedimientos declarados se pueden pasar como parámetros o pueden devolverse como resultados (en C, se pasa un apuntador a la función), sin cambios considerables en la estrategia de acceso a los datos. Con la regla de alcance estático de C y sin procedimientos anidados, cualquier nombre que no sea local para un procedimiento no será local para todos los procedimientos, sin importar cómo se activen. De manera similar, si se devuelve un procedimiento como resultado, entonces cualquier nombre no local hace referencia al almacenamiento que se le asigna en forma estática.

7.3.2 Problemas con los procedimientos anidados

El acceso se complica aún más cuando un lenguaje permite anidar las declaraciones de los procedimientos y también utiliza la regla de alcance estático normal; es decir, un procedimiento puede acceder a variables del procedimiento cuyas declaraciones rodean a su propia declaración, siguiendo la regla de alcance anidado que se describe para los bloques en la sección 1.6.3. La razón es que saber en tiempo de compilación que la declaración de p está inmediatamente anidada dentro de q no nos indica las posiciones relativas de sus registros de activación en tiempo de ejecución. De hecho, como p o q (o ambas) pueden ser recursivas, puede haber varios registros de activación de p y q en la pila.

Encontrar la declaración que se aplique a un nombre no local x en un procedimiento anidado p es una decisión estática; puede realizarse mediante una extensión de la regla de alcance estático para los bloques. Suponga que x se declara en el procedimiento circundante q . Encontrar la activación relevante de q a partir de una activación de p es una decisión dinámica; requiere información adicional en tiempo de ejecución sobre las activaciones. Una posible solución a este problema es utilizar los “enlaces de acceso”, que presentaremos en la sección 7.3.5.

7.3.3 Un lenguaje con declaraciones de procedimientos anidados

La familia C de lenguajes y muchos otros lenguajes conocidos no tienen soporte para los procedimientos anidados, así que vamos a presentar uno que sí lo tiene. La historia de los procedimientos anidados en los lenguajes es larga. Algol 60, un ancestro de C, tenía esta capacidad, al igual que su descendiente Pascal, un lenguaje de enseñanza que fue popular durante algún tiempo. De los lenguajes más recientes con procedimientos anidados, uno de los más influyentes es ML; de este lenguaje tomaremos prestada la sintaxis y la semántica (vea el recuadro titulado “Más acerca de ML” para conocer algunas de las características interesantes de ML):

- ML es un *lenguaje funcional*, lo cual significa que las variables, una vez que se declaran y se inicializan, no se modifican. Sólo hay unas cuantas excepciones como el arreglo, cuyos elementos pueden modificarse mediante llamadas a funciones especiales.
- Las variables se definen, y se inicializan sus valores inmodificables mediante una instrucción de la siguiente forma:

`val <nombre> = <expresión>`

- Las funciones se definen mediante la siguiente sintaxis:

`fun <nombre> (<argumentos>) = <cuerpo>`

- Para los cuerpos de las funciones, utilizaremos instrucciones “let” de la siguiente forma:

`let <lista de definiciones> in <instrucciones> end`

Por lo general, las definiciones son instrucciones `val` o `fun`. El alcance de cada definición consiste en todas las siguientes definiciones hasta `in`, y de todas las instrucciones hasta `end`. Lo más importante es que las definiciones de las funciones pueden anidarse. Por ejemplo, el cuerpo de una función *p* puede contener una instrucción `let` que incluya la definición de otra función *q* (anidada). De manera similar, *q* puede tener definiciones de funciones dentro de su propio cuerpo, lo cual conduce al anidamiento profundo arbitrario de las funciones.

7.3.4 Profundidad de anidamiento

Vamos a dar la *profundidad de anidamiento* 1 a los procedimientos que no estén anidados dentro de otro procedimiento. Por ejemplo, todas las funciones en C se encuentran en la profundidad de anidamiento 1. No obstante, si un procedimiento *p* se define de inmediato dentro de un procedimiento en el nivel de anidamiento *i*, entonces damos a *p* la profundidad de anidamiento *i* + 1.

Ejemplo 7.5: La figura 7.10 contiene un bosquejo en ML de nuestro ejemplo de quicksort. La única función en la profundidad de anidamiento 1 es la más externa, *ordenar*, la cual lee un arreglo *a* de 9 enteros y los ordena mediante el algoritmo quicksort. En la línea (2) se encuentra

Más acerca de ML

Además de ser casi por completo funcional, ML presenta otras sorpresas al programador que está acostumbrado a C y su familia de lenguajes.

- ML soporta las *funciones de mayor orden*. Es decir, una función puede recibir funciones como argumentos, y puede construir y devolver otras. A su vez, esas funciones pueden recibir funciones como argumentos, hasta cualquier nivel.
- En esencia, ML no tiene iteración, como las instrucciones `for` y `while` de C. En vez de ello, el efecto de la iteración se logra mediante la recursividad. Este método es esencial en un lenguaje funcional, ya que no podemos cambiar el valor de una variable de iteración como `i` en la instrucción “`for(i=0; i<10; i++)`” de C, sino que ML convertiría a `i` en un argumento de función, y la función se llamaría a sí misma con valores cada vez mayores de `i`, hasta llegar al límite.
- ML soporta listas y estructuras de árbol etiquetadas como tipos de datos primitivos.
- ML no requiere la declaración de los tipos de las variables. En vez de ello, deduce los tipos en tiempo de compilación, y si no puede lo trata como un error. Por ejemplo, `val x = 1` sin duda hace que `x` tenga el tipo entero, y si también vemos `val y = 2*x`, entonces sabemos que `y` es también un entero.

el arreglo `a`, definido dentro de `ordenar`. Observe la forma de la declaración de ML. El primer argumento de `arreglo` dice que deseamos que el arreglo tenga 11 elementos; todos los arreglos de ML se indexan mediante enteros que empiezan desde 0, por lo que este arreglo es bastante similar al arreglo `a` en C de la figura 7.2. El segundo argumento de `arreglo` dice que al principio, todos los elementos del arreglo `a` contienen el valor 0. Esta elección del valor inicial permite al compilador de ML deducir que `a` es un arreglo de enteros, ya que 0 es un entero, por lo que no tenemos que declarar un tipo para `a`.

También hay varias funciones declaradas dentro de `ordenar`: `leerArreglo`, `intercambiar` y `quicksort`. En las líneas (4) y (6) sugerimos que tanto `leerArreglo` como `intercambiar` acceden al arreglo `a`. Observe que en ML, los accesos a los arreglos pueden violar la naturaleza funcional del lenguaje, y ambas de estas funciones en realidad modifican los valores de los elementos de `a`, como en la versión de `quicksort` en C. Como cada una de estas tres funciones se define de inmediato dentro de una función en el nivel 1 de profundidad de anidamiento, todas sus profundidades de anidamiento son 2.

Las líneas (7) a (11) muestran parte del detalle de `quicksort`. El valor local `v`, el pivote para la partición, se declara en la línea (8). La función `particion` se define en la línea (9). En la línea (10) sugerimos que `particion` accede tanto al arreglo `a` como al valor de pivote `v`, y que también llama a la función `intercambiar`. Como `particion` se define inmediatamente dentro de una función en la profundidad de anidamiento 2, se encuentra en la profundidad 3. La línea (11) sugiere que `quicksort` accede a las variables `a` y `v`, a la función `particion` y a sí misma en forma recursiva.

```

1) fun ordenar(archEntrada, archSalida) =
    let
2)      val a = array(11,0);
3)      fun leerArreglo(archEntrada) = ... ;
4)          ... a ... ;
5)      fun intercambiar(i,j) =
6)          ... a ... ;
7)      fun quicksort(m,n) =
        let
8)          val v = ... ;
9)          fun particion(y,z) =
10)              ... a ... v ... intercambiar ...
11)              in
12)                  ... a ... v ... particion ... quicksort
13)              end
14)          in
15)              ... a ... leerArreglo ... quicksort ...
16)          end;

```

Figura 7.10: Una versión de quicksort, en estilo ML, usando funciones anidadas

La línea (12) sugiere que la función exterior *ordenar* accede a *a* y llama a los dos procedimientos *leerArreglo* y *quicksort*. \square

7.3.5 Enlace de acceso

Podemos obtener una implementación directa de la regla de alcance estático normal para funciones anidadas si agregamos un apuntador, conocido como el *enlace de acceso*, a cada registro de activación. Si el procedimiento *p* está inmediatamente anidado dentro del procedimiento *q* en el código fuente, entonces el enlace de acceso en cualquier activación de *p* apunta a la activación más reciente de *q*. Observe que la profundidad de anidamiento de *q* debe ser exactamente una menos que la profundidad de anidamiento de *p*. Los enlaces de acceso forman una cadena desde el registro de activación en la parte superior de la pila, hasta una secuencia de activaciones a profundidades de anidamiento cada vez menores. A lo largo de esta cadena se encuentran todas las activaciones cuyos datos y procedimientos son accesibles para el procedimiento actual en ejecución.

Suponga que el procedimiento *p* en la parte superior de la pila se encuentra en la profundidad de anidamiento n_p , y que *p* tiene que acceder a *x*, que es un elemento definido dentro de cierto procedimiento *q* que rodea a *p*, y tiene la profundidad de anidamiento n_q . Observe que $n_q \leq n_p$, y sólo hay igualdad si *p* y *q* son el mismo procedimiento. Para buscar *x*, empezamos en el registro de activación para *p* en la parte superior de la pila y seguimos el enlace de acceso $n_p - n_q$ veces, de un registro de activación al otro. Por último, terminamos en un registro de activación para *q*, y éste siempre será el más reciente (mayor) para *q* que aparezca en ese momento en la pila. Este registro de activación contiene el elemento *x* que queremos. Como

el compilador conoce la distribución de los registros de activación, x se encontrará en cierto desplazamiento fijo a partir de la posición en el registro de activación de q a la que podamos llegar siguiendo el último enlace de acceso.

Ejemplo 7.6: La figura 7.11 muestra una secuencia de pilas que podrían resultar de la ejecución de la función *ordenar* de la figura 7.10. Como antes, representamos los nombres de las funciones mediante sus primeras letras, y mostramos algunos de los datos que podrían aparecer en los diversos registros de activación, así como el enlace de acceso para cada activación. En la figura 7.11(a), vemos la situación después de que *ordenar* llama a *leerArreglo* para cargar la entrada en el arreglo a , y después llama a *quicksort(1, 9)* para ordenar el arreglo. El enlace de acceso de *quicksort(1, 9)* apunta al registro de activación para *ordenar*, no debido a que *ordenar* llamó a *quicksort*, sino porque *ordenar* es la función anidada más cercana que rodea a *quicksort* en el programa de la figura 7.10.

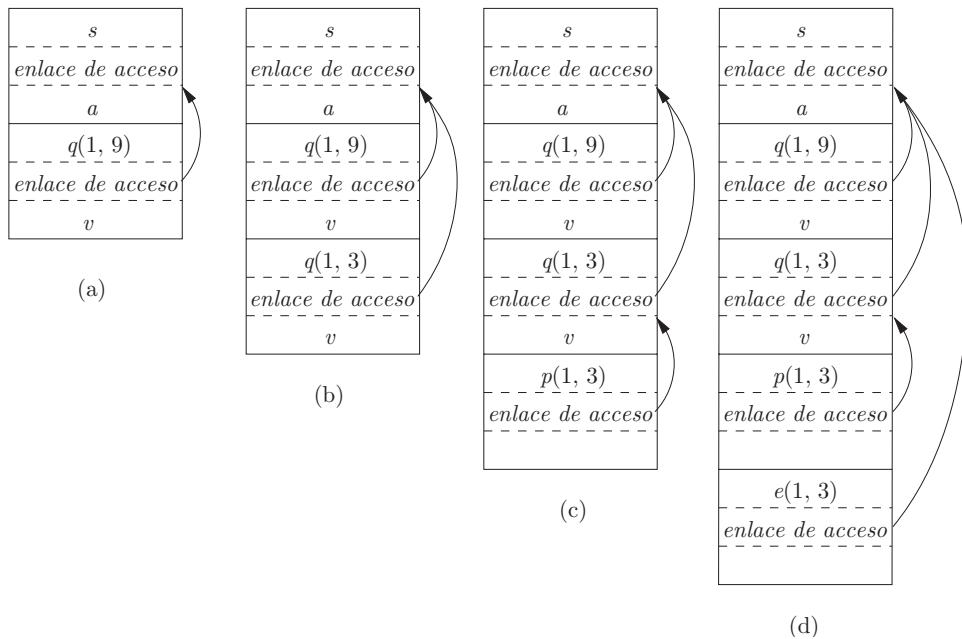


Figura 7.11: Enlaces de acceso para buscar datos que no sean locales

En los pasos siguientes de la figura 7.11 podemos ver una llamada recursiva a *quicksort(1, 3)*, seguida de una llamada a *particion*, que llama a *intercambiar*. Observe que el enlace de acceso de *quicksort(1, 3)* apunta a *ordenar*, por la misma razón que lo hace el enlace de acceso de *quicksort(1, 9)*.

En la figura 7.11(d), el enlace de acceso para *intercambiar* pasa por alto los registros de activación para *quicksort* y *particion*, ya que *intercambiar* está anidada justo dentro de *ordenar*. Ese arreglo está bien, ya que *intercambiar* sólo tiene que acceder al arreglo a , y los dos elementos que debe intercambiar se indican mediante sus propios parámetros i y j . \square

7.3.6 Manipulación de los enlaces de acceso

¿Cómo se determinan los enlaces de acceso? El caso simple ocurre cuando se da una llamada a un procedimiento específico, cuyo nombre se proporciona de manera explícita en la llamada al procedimiento. El caso más difícil es cuando la llamada es a un parámetro del procedimiento; en ese caso, el procedimiento específico que se está llamando no se conoce sino hasta en tiempo de ejecución, y la profundidad de anidamiento del procedimiento llamado puede diferir en distintas ejecuciones de la llamada. Por ende, vamos primero a considerar lo que debería ocurrir cuando un procedimiento q llama a un procedimiento p en forma explícita. Existen tres casos:

1. El procedimiento p se encuentra en una profundidad de anidamiento mayor que la de q . Entonces, p debe definirse inmediatamente dentro de q , o la llamada realizada por q no estaría en una posición dentro del alcance del nombre del procedimiento p . Por ende, la profundidad de anidamiento de p es exactamente uno más que la de q , y el enlace de acceso que proviene de p debe conducir a q . Es fácil para la secuencia de llamadas incluir un paso que coloque en el enlace de acceso para p un apuntador al registro de activación de q . Algunos ejemplos incluyen la llamada que *ordenar* hace a *quicksort* para establecer la figura 7.11(a), y la llamada que *quicksort* hace a *particion* para crear la figura 7.11(c).
2. La llamada es recursiva, es decir, $p = q$.² Entonces, el enlace de acceso para el nuevo registro de activación es el mismo que el del registro de activación que está debajo. Un ejemplo es la llamada que *quicksort*(1, 9) hace a *quicksort*(1, 3), para establecer la figura 7.11(b).
3. La profundidad de anidamiento n_p de p es menor que la profundidad de anidamiento n_q de q . Para que la llamada dentro de q pueda estar en el alcance del nombre p , el procedimiento q debe anidarse dentro de cierto procedimiento r , mientras que p es un procedimiento definido inmediatamente dentro de r . Por lo tanto, para encontrar el registro de activación superior para r hay que seguir la cadena de enlaces de acceso, empezando en el registro de activación para q , para $n_q - n_p + 1$ saltos. Entonces, el enlace de acceso para p debe ir a esta activación de r .

Ejemplo 7.7: Para un ejemplo del caso (3), observe cómo pasamos de la figura 7.11(c) a la 7.11(d). La profundidad de anidamiento 2 de la función *intercambiar* a la que se llamó es uno menos que la profundidad 3 de la función *particion*, que la llamó. Por ende, empezamos en el registro de activación para *particion* y seguimos $3 - 2 + 1 = 2$ enlaces de acceso, lo cual nos lleva del registro de activación de *particion* al de *quicksort*(1, 3), y después al de *ordenar*. Por lo tanto, el enlace de acceso para *intercambiar* va hacia el registro de activación para *ordenar*, como podemos ver en la figura 7.11(d).

Una forma equivalente de descubrir este enlace de acceso es sólo seguir los enlaces de acceso para $n_q - n_p$ saltos, y copiar el enlace de acceso que se encuentre en ese registro. En nuestro ejemplo, daríamos un salto al registro de activación de *quicksort*(1, 3) y copiaríamos su enlace de acceso a *ordenar*. Observe que este enlace de acceso es correcto para *intercambiar*, aun cuando esta función no se encuentra en el alcance de *quicksort*, siendo éstas funciones hermanas anidadas dentro de *ordenar*. \square

²ML permite las funciones mutuamente recursivas, que deben manejarse de la misma forma.

7.3.7 Enlaces de acceso para los parámetros de procedimientos

Cuando se pasa un procedimiento p a otro procedimiento q como parámetro, y después q llama a su parámetro (y por lo tanto llama a p en esta activación de q), es posible que q no conozca el contexto en el que aparece p en el programa. De ser así, es imposible para q saber cómo establecer el enlace de acceso para p . La solución a este problema es la siguiente: cuando se utilizan procedimientos como parámetros, el emisor debe pasar, junto con el nombre del procedimiento que es el parámetro, el enlace de acceso apropiado para ese parámetro.

El emisor conoce el enlace, ya que si el procedimiento r pasa a p como un parámetro actual, entonces p debe ser un nombre accesible para r y, por lo tanto, r puede determinar el enlace de acceso para p de la misma forma que si r hubiera llamado directamente a p . Es decir, usamos las reglas para construir enlaces de acceso que vimos en la sección 7.3.6.

Ejemplo 7.8: En la figura 7.12 podemos ver un bosquejo de una función a en ML, que tiene anidadas a las funciones b y c . La función b tiene un parámetro con valor de función llamado f , al cual llama. La función c define dentro de ella a una función d , y después c llama a b con el parámetro actual d .

```
fun a(x) =
  let
    fun b(f) =
      ... f ... ;
    fun c(y) =
      let
        fun d(z) = ...
      in
        ... b(d) ...
      end
    in
      ... c(1) ...
    end;
```

Figura 7.12: Bosquejo de un programa en ML que utiliza funciones como parámetros

Vamos a rastrear lo que ocurre cuando se ejecuta a . Primero, a llama a c , por lo que colocamos un registro de activación para c encima del de a en la pila. El enlace de acceso para c apunta al registro para a , ya que c se define justo dentro de a . Después, c llama a $b(d)$. La secuencia de llamada establece un registro de activación para b , como se muestra en la figura 7.13(a).

Dentro de este registro de activación se encuentra el parámetro actual d y su enlace de acceso, que en conjunto forman el valor del parámetro formal f en el registro de activación para b . Observe que c sabe sobre d , ya que d está definida dentro de c , y por lo tanto c pasa un apuntador a su propio registro de activación como el enlace de acceso. Sin importar en dónde se definió d ,

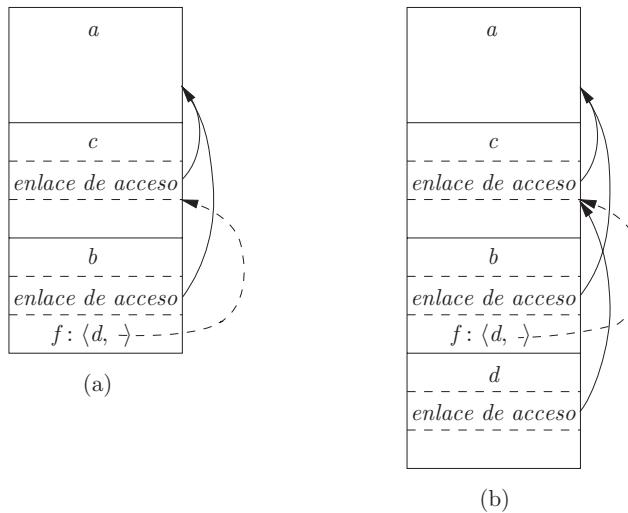


Figura 7.13: Los parámetros actuales llevan su enlace de acceso con ellos

si c se encuentra en el alcance de esa definición, entonces debe aplicarse una de las tres reglas de la sección 7.3.6, y c puede proporcionar el enlace.

Ahora vamos a ver lo que hace b . Sabemos que en cierto punto utiliza su parámetro f , que tiene el efecto de llamar a d . En la pila aparece un registro de activación para d , como se muestra en la figura 7.13(b). El enlace de acceso apropiado para colocar en este registro de activación se encuentra en el valor para el parámetro f ; el enlace es hacia el registro de activación para c , ya que c rodea inmediatamente a la definición de d . Observe que b es capaz de establecer el enlace apropiado, aun cuando b no se encuentra en el alcance de la definición de c . \square

7.3.8 Estructura de datos Display

Un problema con el método del enlace de acceso para los datos no locales es que si aumenta la profundidad de anidamiento, tal vez tengamos que seguir largas cadenas de enlaces para llegar a los datos que requerimos. Una implementación más eficiente utiliza un arreglo auxiliar d , conocido como *display*, el cual consiste en un apuntador para cada profundidad de anidamiento. Arreglamos todo de forma que, en todo momento, $d[i]$ sea un apuntador al registro de activación más alto en la pila, para cualquier procedimiento en la profundidad de anidamiento i . En la figura 7.14 se muestran ejemplos de un display. Por ejemplo, en la figura 7.14(d) podemos ver el display d , en donde $d[1]$ contiene un apuntador al registro de activación para *ordenar*, el registro de activación más alto (y único) para una función en la profundidad de anidamiento 1. Además, $d[2]$ contiene un apuntador al registro de activación para *intercambiar*, el registro más alto en la profundidad 2, y $d[3]$ apunta a *particion*, el registro más alto en la profundidad 3.

La ventaja de utilizar un display es que si el procedimiento p se está ejecutando, y debe acceder al elemento x que pertenece a cierto procedimiento q , sólo debemos buscar en $d[i]$, en donde i es la profundidad de anidamiento de q ; seguimos el apuntador $d[i]$ hacia el registro de activación para q , en donde x se encuentra en un desplazamiento conocido. El compilador sabe

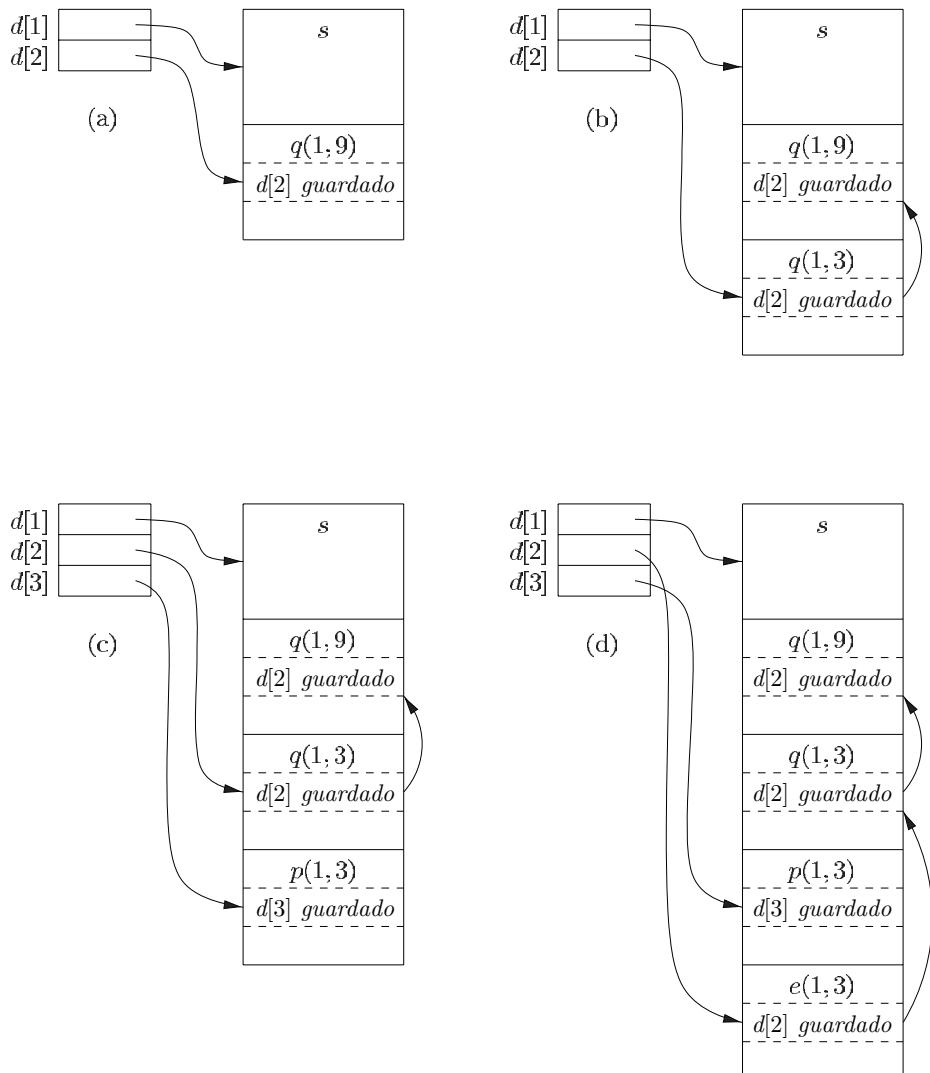


Figura 7.14: Mantenimiento del display

lo que es i , de forma que puede generar código para acceder a x usando $d[i]$, y el desplazamiento de x a partir de la parte superior del registro de activación para q . Por ende, el código nunca tendrá que seguir una cadena larga de enlaces de acceso.

Para poder mantener el display en forma correcta, debemos guardar los valores anteriores de las entradas de visualización en nuevos registros de activación. Si se hace una llamada al procedimiento p en la profundidad n_p , y su registro de activación no es el primero en la pila para un procedimiento en la profundidad n_p , entonces el registro de activación para p debe contener el valor anterior de $d[n_p]$, mientras que $d[n_p]$ en sí se establece para apuntar a esta activación de p . Cuando p regresa y su registro de activación se elimina de la pila, restauramos $d[n_p]$ para que tenga su valor anterior a la llamada de p .

Ejemplo 7.9: En la figura 7.14 se ilustran varios pasos de manipulación del display. En la figura 7.14(a), *ordenar* en la profundidad 1 ha llamado a *quicksort(1, 9)* en la profundidad 2. El registro de activación para *quicksort* tiene un lugar para almacenar el valor anterior de $d[2]$, el cual se indica como $d[2]$ *guardado*, aunque en este caso y como no hubo un registro de activación anterior en la profundidad 2, este apuntador es nulo.

En la figura 7.14(b), *quicksort(1, 9)* llama a *quicksort(1, 3)*. Como los registros de activación para ambas llamadas se encuentran en la profundidad 2, debemos almacenar el apuntador a *quicksort(1, 9)*, que se encontraba en $d[2]$, en el registro para *quicksort(1, 3)*. Después, hacemos que $d[2]$ apunte a *quicksort(1, 3)*.

A continuación se hace una llamada a *particion*. Esta función está en la profundidad 3, por lo que usamos la posición $d[3]$ en el display por primera vez, y hacemos que apunte al registro de activación para *particion*. El registro para *particion* tiene una posición para un valor anterior de $d[3]$, pero en este caso no hay ninguno, por lo que el apuntador sigue siendo nulo. En la figura 7.14(c) se muestra el display y la pila en este momento.

Después, *particion* llama a *intercambiar*. Esa función está en la profundidad 2, por lo que su registro de activación almacena el apuntador anterior $d[2]$, que va hacia el registro de activación para *quicksort(1, 3)*. Observe que los apuntadores de visualización se “cruzan”; es decir, $d[3]$ apunta más debajo de la pila que $d[2]$. No obstante, es una situación apropiada; *intercambiar* sólo puede acceder a sus propios datos y los de *ordenar* a través de $d[1]$. \square

7.3.9 Ejercicios para la sección 7.3

Ejercicio 7.3.1: En la figura 7.15 hay una función `main` en ML, la cual calcula números de Fibonacci de una manera no estándar. La función `fib0` calcula el n -ésimo número de Fibonacci para cualquier $n \geq 0$. Dentro de esta función se anida la función `fib1`, que calcula el n -ésimo número de Fibonacci asumiendo que $n \geq 2$, y dentro de `fib1` se anida `fib2`, que supone que $n \geq 4$. Observe que ni `fib1` ni `fib2` tienen que comprobar los casos básicos. Muestre la pila de registros de activación que resultan de una llamada a `main`, hasta el momento en el que la primera llamada (a `fib0(1)`) está a punto de regresar. Muestre el enlace de acceso en cada uno de los registros de activación en la pila.

Ejercicio 7.3.2: Suponga que implementamos las funciones de la figura 7.15 mediante un display. Muestre el display en el momento en el que está a punto de regresar la primera llamada a `fib0(1)`. Además, indique la entrada guardada del display en cada uno de los registros de activación en la pila, en ese momento.

```

fun main () {
    let
        fun fib0(n) =
            let
                fun fib1(n) =
                    let
                        fun fib2(n) = fib1(n-1) + fib1(n-2)
                    in
                        if n >= 4 then fib2(n)
                        else fib0(n-1) + fib0(n-2)
                    end
                in
                    if n >= 2 then fib1(n)
                    else 1
                end
            in
                fib0(4)
            end;
}

```

Figura 7.15: Funciones anidadas para calcular números de Fibonacci

7.4 Administración del montículo

El montículo es la porción del almacenamiento que se utiliza para los datos que tienen una vida indefinida, o hasta que el programa los elimina en forma explícita. Mientras que, por lo general, las variables locales se vuelven inaccesibles cuando terminan sus procedimientos, muchos lenguajes nos permiten crear objetos u otro tipo de datos cuya existencia no se ve atada a la activación del procedimiento que los crea. Por ejemplo, tanto C++ como Java cuentan con la palabra clave `new` para que el programador cree objetos (o apuntadores a ellos) que puedan pasarse de un procedimiento a otro, de manera que sigan existiendo aun después de que haya desaparecido el procedimiento que los creó. Dichos objetos se almacenan en un montículo.

En esta sección hablaremos sobre el *administrador de memoria*, el subsistema que asigna y desasigna espacio dentro del montículo; sirve como una interfaz entre los programas de aplicaciones y el sistema operativo. Para los lenguajes como C o C++, que desasignan trozos de almacenamiento en forma *manual* (es decir, mediante instrucciones explícitas del programa, como `free` o `delete`), el administrador de memoria también es responsable de implementar la desasignación.

En la sección 7.5 hablaremos sobre la *recolección de basura*, que es el proceso de buscar espacios dentro del montículo que el programa ya no utilice y que, por lo tanto, puedan reasignarse para alojar otros elementos de datos. Para los lenguajes como Java, el recolector de basura es el que desasigna la memoria. Cuando se le requiere, el recolector de basura es un subsistema importante del administrador de memoria.

7.4.1 El administrador de memoria

El administrador de memoria lleva el registro de todo el espacio libre en el almacenamiento del montículo, en todo momento. Realiza dos funciones básicas:

- *Asignación.* Cuando un programa solicita memoria para una variable u objeto,³ el administrador de memoria produce un trozo de memoria contigua del montículo del tamaño requerido. Si es posible, satisface una solicitud de asignación usando espacio libre en el montículo; si no hay disponible un trozo del tamaño necesario, busca incrementar el espacio de almacenamiento del montículo, obteniendo bytes consecutivos de memoria virtual del sistema operativo. Si el espacio se agota, el administrador de memoria devuelve esa información al programa de aplicación.
- *Desasignación.* El administrador de memoria devuelve el espacio desasignado a la reserva de espacio libre, de manera que pueda reutilizar ese espacio para satisfacer otras solicitudes de asignación. Por lo general, los administradores de memoria no devuelven la memoria al sistema operativo, aun cuando disminuya el uso del montículo por parte del programa.

La administración de memoria sería más simple si (a) todas las solicitudes de asignación fueran para trozos del mismo tamaño, y (b) el almacenamiento se liberara en forma predecible, por ejemplo, que el primero en asignarse fuera el primero en desasignarse. Hay algunos lenguajes como Lisp, para los cuales se aplica la condición (a); el lenguaje Lisp puro utiliza sólo un elemento de datos (una celda de dos apuntadores) a partir de la cual se construyen todas las estructuras de datos. La condición (b) también se aplica en ciertas situaciones, de las cuales la más común es con los datos que pueden asignarse en la pila en tiempo de ejecución. Sin embargo, en la mayoría de los lenguajes no se aplican (a) ni (b) en general. En vez de ello se asignan elementos de distintos tamaños, y no hay una buena forma de predecir los tiempos de vida de todos los objetos asignados.

Por ende, el administrador de memoria debe estar preparado para atender, en cualquier orden, las solicitudes de asignación y desasignación de cualquier tamaño, que varían desde un byte hasta la misma longitud de todo el espacio de direcciones del programa.

He aquí las propiedades que deseamos de los administradores de memoria:

- *Eficiencia en el espacio.* Un administrador de memoria debe disminuir al mínimo el espacio total del montículo que necesita un programa. Al hacer esto, los programas más grandes pueden ejecutarse en un espacio de direcciones virtuales fijas. La eficiencia en el espacio se logra al reducir la “fragmentación”, como veremos en la sección 7.4.4.
- *Eficiencia del programa.* Un administrador de memoria debe hacer un buen uso del subsistema de memoria para permitir que los programas se ejecuten con más rapidez. Como veremos en la sección 7.4.2, el tiempo requerido para ejecutar una instrucción puede variar en gran medida, según la posición que tengan los objetos en la memoria. Por fortuna, los programas tienden a exhibir “localidad”, un fenómeno que veremos en la sección 7.4.3, el cual se refiere a la forma no aleatoria mediante clústers, en la que los programas acceden por lo general a la memoria. Al poner atención a la colocación de los objetos en la memoria, el administrador de memoria puede hacer un mejor uso del espacio y, con suerte, hacer que el programa se ejecute con más rapidez.

³ A partir de este momento, nos referiremos a las cosas que requieren espacio de memoria como “objetos”, inclusive aunque no sean verdaderos objetos, en el sentido de la “programación orientada a objetos”.

- *Baja sobrecarga.* Debido a que las asignaciones y desasignaciones de memoria son operaciones frecuentes en muchos programas, es importante que estas operaciones sean lo más eficientes que sea posible. Es decir, deseamos disminuir al mínimo la *sobrecarga*: la fracción de tiempo de ejecución que se invierte en realizar la asignación y la desasignación. Observe que el costo de las asignaciones está dominado por las solicitudes pequeñas; la sobrecarga de administrar objetos grandes es menos importante, ya que por lo general se puede amortizar a través de una cantidad mayor de cálculos.

7.4.2 La jerarquía de memoria de una computadora

La administración de la memoria y la optimización del compilador deben realizarse teniendo en mente la manera en que se comporta la memoria. Las máquinas modernas están diseñadas de forma que los programadores puedan escribir programas correctos sin preocuparse por los detalles del subsistema de memoria. Sin embargo, la eficiencia de un programa se determina no sólo en base al número de instrucciones ejecutadas, sino también por el tiempo que se requiere para ejecutar cada una de estas instrucciones. El tiempo necesario para ejecutar una instrucción puede variar en forma considerable, ya que el tiempo requerido para acceder a distintas partes de la memoria puede ir desde nanosegundos hasta milisegundos. Por lo tanto, los programas que hacen uso intensivo de los datos pueden beneficiarse de manera considerable de las optimizaciones que hacen un buen uso del subsistema de memoria. Como veremos en la sección 7.4.3, pueden aprovechar el fenómeno de “localidad”: el comportamiento no aleatorio de los programas comunes.

La gran discrepancia en los tiempos de acceso a la memoria se debe a la limitación fundamental en la tecnología de hardware; podemos construir almacenamiento más pequeño y rápido, o almacenamiento grande y lento, pero no almacenamiento que sea tanto grande como rápido. Simplemente es imposible hoy en día construir gigabytes de almacenamiento con tiempos de acceso en nanosegundos, que es la velocidad con la que operan los procesadores de alto rendimiento. Por lo tanto, casi todas las computadoras modernas organizan su almacenamiento como una *jerarquía de memoria*. Como se muestra en la figura 7.16, una jerarquía de memoria consiste en una serie de elementos de almacenamiento, en donde los más pequeños y rápidos están “más cerca” del procesador, y los más lentos y grandes están más alejados.

Por lo general, un procesador tiene un pequeño número de registros, cuyo contenido se encuentra bajo el control del software. A continuación, tiene uno o más niveles de caché, que por lo general se fabrica a partir de RAM estática, con un tamaño desde varios kilobytes hasta varios megabytes. El siguiente nivel de la jerarquía es la memoria física (principal), compuesta de cientos de megabytes o gigabytes de RAM dinámica. La memoria física se respalda mediante la memoria virtual, que se implementa mediante gigabytes de discos. Al momento en que ocurre un acceso a la memoria, la máquina primero busca los datos en el almacenamiento más cercano (de menor nivel) y, si los datos no están ahí, busca en el siguiente nivel más alto, y así sucesivamente.

Los registros son escasos, por lo que el uso de ellos se destina a las aplicaciones específicas y se administran mediante el código que genera un compilador. Todos los demás niveles de la jerarquía se administran en forma automática; de esta manera no sólo se simplifica la tarea de programación, sino que el mismo programa puede funcionar en forma efectiva en distintas máquinas con diferentes configuraciones de memoria. Con cada acceso a la memoria, la máquina busca en cada nivel de memoria en sucesión, empezando con el nivel más bajo, hasta que localiza los datos. Las cachés se administran de manera exclusiva en el hardware, para poder

Tamaños típicos	Tiempos de acceso típicos
> 2GB	Memoria virtual (Disco) 3 - 15 ms
256MB - 2GB	Memoria física 100 - 150 ns
128KB - 4MB	Caché de 2do. nivel 40 - 60 ns
16 - 64KB	Caché de 1er. nivel 5 - 10 ns
32 palabras	Registros (Procesador) 1 ns

Figura 7.16: Configuraciones típicas de jerarquía de memoria

estar a la par con los tiempos de acceso relativamente rápidos de la RAM. Como los discos son muy lentos, la memoria virtual se administra mediante el sistema operativo, con la ayuda de una estructura de hardware conocida como “búfer de traducción adelantada”.

Los datos se transfieren como bloques de almacenamiento contiguo. Para amortizar el costo del acceso, se utilizan bloques más grandes con los niveles menores de la jerarquía. Entre la memoria principal y la caché, los datos se transfieren en bloques conocidos como *líneas de caché* que, por lo general, son de 32 a 256 bytes de largo. Entre la memoria virtual (disco) y la memoria principal, los datos se transfieren en bloques conocidos como *páginas*, por lo general, entre 4K y 64K bytes de tamaño.

7.4.3 Localidad en los programas

La mayoría de los programas exhiben un alto nivel de *localidad*; es decir, invierten la mayoría de su tiempo ejecutando una fracción relativamente pequeña del código y sólo acceden a una pequeña fracción de los datos. Decimos que un programa tiene *localidad temporal* si es probable que las ubicaciones de memoria a las que accede vuelva a utilizarlas dentro de un periodo corto. Decimos que un programa tiene *localidad espacial* si es probable que las ubicaciones de memoria cercanas a la ubicación a la que accede también las vaya a utilizar dentro de un periodo corto.

Se dice que los programas invierten el 90% de su tiempo ejecutando el 10% del código. He aquí el porqué:

- A menudo, los programas contienen muchas instrucciones que nunca se ejecutan. Los programas creados con componentes y bibliotecas sólo utilizan una pequeña fracción de la funcionalidad que se provee. Además, a medida que los requerimientos cambian y los programas evolucionan, los sistemas heredados con frecuencia contienen muchas instrucciones que ya no se utilizan.

RAM estática y dinámica

La mayoría de la memoria de acceso aleatorio es *dinámica*, lo cual significa que está formada por circuitos electrónicos bastante simples, que pierden su carga (y por ende “olvidan” el bit que estaban almacenando) en un periodo corto. Estos circuitos deben actualizarse (es decir, hay que leer sus bits y volver a escribirlos) en forma periódica. Por otro lado, la RAM *estática* está diseñada con un circuito más complejo para cada bit, y en consecuencia el bit almacenado puede permanecer en forma indefinida, hasta que se modifique. Es evidente que un chip puede almacenar más bits si utiliza los circuitos de RAM dinámica en vez de los circuitos de RAM estática, por lo que tendremos a ver memorias principales extensas de la variedad dinámica, mientras que las memorias más pequeñas, como las cachés, están formadas de circuitos estáticos.

- En realidad, sólo una pequeña fracción del código que podría invocarse es la que se ejecuta durante el funcionamiento normal del programa. Por ejemplo, las instrucciones para manejar entradas ilegales y casos excepcionales, aunque son críticas para que el programa sea correcto, raras veces se invocan durante una ejecución.
- El programa ordinario invierte la mayoría de su tiempo ejecutando los ciclos más internos y los ciclos con uso intenso de la recursividad.

La localidad nos permite sacar provecho de la jerarquía de memoria de una computadora moderna, como se muestra en la figura 7.16. Al colocar las instrucciones y los datos más comunes en el almacenamiento más rápido pero a la vez más pequeño, mientras dejamos el resto en el almacenamiento más lento, pero a la vez más grande, podemos reducir en forma considerable el tiempo de acceso promedio a la memoria de un programa.

Se ha descubierto que muchos programas exhiben tanto localidad temporal como espacial, en cuanto a la forma en que acceden a las instrucciones y los datos. Sin embargo, los patrones de acceso a los datos muestran en general una mayor discrepancia que los patrones de acceso a las instrucciones. Las políticas como mantener los datos que hayan tenido un uso más reciente en la jerarquía más veloz funcionan bien para los programas comunes, pero tal vez no funcionen bien para algunos programas que hacen un uso intensivo de los datos; por ejemplo, los que iteran a través de arreglos muy extensos.

A menudo no podemos saber, con sólo ver el código, qué secciones del programa se utilizarán con más frecuencia, en especial para cierta entrada. Aún si conocemos qué instrucciones se ejecutarán con más frecuencia, la caché más rápida, por lo general, no es lo bastante grande como para almacenarlas a todas al mismo tiempo. Por lo tanto, debemos ajustar el contenido del almacenamiento más rápido en forma dinámica y lo utilizaremos para almacenar instrucciones que quizás se utilicen con mucha frecuencia en un futuro cercano.

Optimización mediante la jerarquía de memoria

La política de mantener las instrucciones de uso más reciente en la caché tiende a funcionar bien; en otras palabras, por lo general, el pasado es un buen vaticinador del uso a futuro de la memoria. Cuando se ejecuta una nueva instrucción, hay una alta probabilidad de que la siguiente

Arquitecturas de caché

¿Cómo sabemos si una línea de caché está en una caché? Sería demasiado costoso revisar cada una de las líneas en la caché, por lo que una práctica común es restringir la colocación de una línea de caché dentro de la misma caché. A esta restricción se le conoce como *asociatividad de conjuntos*. Una caché tiene *asociatividad de conjuntos de k vías* si una línea de caché sólo puede residir en k ubicaciones. La caché más simple es la que tiene una asociatividad de 1 vía, a la cual también se le conoce como caché con *asignación directa*. En una caché con asignación directa, los datos con la dirección de memoria n sólo se pueden colocar en la dirección de caché representada por $n \bmod s$, en donde s es el tamaño de la caché. De manera similar, una caché con asociatividad de conjuntos de k vías se divide en k conjuntos, en donde un dato con la dirección n se puede asignar sólo a la ubicación representada por $n \bmod (s/k)$ en cada conjunto. La mayoría de las cachés de instrucciones y de datos tienen una asociatividad entre 1 y 8. Cuando una línea de caché se lleva a la caché, y todas las posibles ubicaciones que pueden contener esa línea están ocupadas, es común desalojar la última línea que se haya utilizado.

instrucción también se vaya a ejecutar. Este fenómeno es un ejemplo de localidad espacial. Una técnica efectiva para mejorar la localidad espacial de las instrucciones es hacer que el compilador coloque los bloques básicos (secuencias de instrucciones que siempre se ejecutan en forma secuencial) que tengan más probabilidad de seguirse unos a otros en forma contigua; en la misma página, o incluso en la misma línea de caché, si es posible. Las instrucciones que pertenecen al mismo ciclo o a la misma función también tienen una alta probabilidad de ejecutarse en conjunto.⁴

También podemos mejorar la localidad temporal y espacial de los accesos a los datos en un programa, cambiando la distribución de los datos o el orden de los cálculos. Por ejemplo, los programas que visitan grandes cantidades de datos en forma repetida, realizando cada vez una pequeña cantidad de cálculos, no tienen un buen desempeño. Es mejor si podemos llevar ciertos datos de un nivel lento de la jerarquía de memoria a un nivel más rápido una vez (por ejemplo, de disco a memoria principal), y realizamos todos los cálculos necesarios sobre estos datos mientras residen en el nivel más rápido. Podemos aplicar este concepto en forma recursiva, para reutilizar los datos en la memoria física, en las cachés y en los registros.

7.4.4 Reducción de la fragmentación

Al principio de la ejecución del programa, el montículo es una unidad contigua de espacio libre. A medida que el programa asigna y desasigna memoria, este espacio se divide en trozos de memoria libres y usados, y los trozos libres no tienen que residir en un área contigua del montículo. Nos referimos a los trozos libres de memoria como huecos. Con cada solicitud de asignación, el administrador de memoria debe *colocar* el trozo de memoria solicitado en un hueco que sea lo bastante grande. A menos que se encuentre un agujero del tamaño exacto, debemos *dividir* cierto hueco, creando uno aún más pequeño.

⁴Cuando una máquina obtiene una palabra en la memoria, es muy simple realizar también una *preobtención* de las siguientes palabras contiguas de memoria. Por ende, una característica común de la jerarquía de memoria es que se obtiene un bloque de varias palabras de un nivel de memoria cada vez que se accede a ese nivel.

Con cada solicitud de desasignación, los trozos liberados de memoria se agregan de vuelta a la reserva de espacio libre. *Unimos* los agujeros contiguos para formar huecos más grandes, ya que de otra forma sólo pueden hacerse más pequeños. Si no tenemos cuidado, la memoria puede terminar *fragmentada*, consistente en grandes cantidades de agujeros pequeños, no contiguos. Entonces es posible que ningún agujero sea lo bastante grande como para satisfacer una solicitud futura, aun cuando puede haber suficiente espacio libre agregado.

Colocación de objetos en base al mejor ajuste y al siguiente ajuste

Para reducir la fragmentación, debemos controlar la forma en que el administrador de memoria coloca objetos nuevos en el montículo. Se ha descubierto en la práctica que una buena estrategia para disminuir al mínimo la fragmentación en los programas reales es asignar la memoria solicitada en el hueco más pequeño disponible, que sea lo suficientemente grande. Este algoritmo del *mejor ajuste* tiende a dejar los agujeros grandes para satisfacer las solicitudes siguientes más grandes. Una alternativa conocida como *primer ajuste*, en donde un objeto se coloca en el primer agujero (dirección más baja) en el que pueda ajustarse, requiere menos tiempo para colocar los objetos, pero ha resultado ser inferior al mejor ajuste en cuanto al rendimiento general.

Para implementar la colocación en base al mejor ajuste con más eficiencia, podemos separar el espacio libre en *contenedores*, de acuerdo con sus tamaños. Una idea práctica es tener muchos más contenedores para los tamaños más pequeños, ya que por lo general hay muchos más objetos pequeños. Por ejemplo, el administrador de memoria Lea que se utiliza en el compilador de GNU C llamado `gcc`, alinea todos los trozos en límites de 8 bytes. Hay un contenedor para cada múltiplo de bloques de 8 bytes, desde 16 bytes hasta 512 bytes. A los contenedores de mayor tamaño se les asigna espacio en forma logarítmica (es decir, el tamaño mínimo para cada contenedor es el doble del tamaño del contenedor anterior), y dentro de cada uno de estos contenedores los bloques se ordenan según su tamaño. Siempre hay un bloque de espacio libre que puede extenderse mediante la acción de solicitar más páginas al sistema operativo. Conocido como *trozo wilderness*, Lea trata a este trozo como el contenedor de mayor tamaño, debido a su extensibilidad.

El uso de contenedores facilita la búsqueda del trozo con el mejor ajuste.

- Si, en los tamaños pequeños que solicita el administrador de memoria de Lea, hay un contenedor para trozos de ese tamaño solamente, podemos tomar cualquier trozo de ese contenedor.
- Para los tamaños que no tienen un contenedor privado, buscamos el que tenga permitido incluir trozos del tamaño deseado. Dentro de ese contenedor, podemos usar la estrategia del primer ajuste o la del mejor ajuste; es decir, buscamos y seleccionamos el primer trozo que sea lo suficiente grande, o invertimos más tiempo y buscamos el trozo más pequeño que sea lo suficiente grande. Observe que cuando el ajuste no es exacto, por lo general hay que colocar el resto del trozo en un contenedor de tamaño más pequeño.
- Sin embargo, puede ser que el contenedor de destino esté vacío, o que todos los trozos en ese contenedor sean demasiado pequeños como para satisfacer la solicitud de espacio. En ese caso, sólo repetimos la búsqueda, utilizando el contenedor para el (los) siguiente(s) tamaño(s) más grande(s). En un momento dado, o encontramos un trozo que podamos usar, o llegamos al trozo “wilderness”, a partir del cual podemos obtener con seguridad el espacio necesario, tal vez yendo al sistema operativo, para obtener páginas adicionales para el montículo.

Aunque la colocación en base al mejor ajuste tiende a mejorar la utilización del espacio, tal vez no sea lo mejor en términos de localidad espacial. Los trozos que un programa asigna casi al mismo tiempo tienden a tener patrones de referencia y tiempos de vida similares. Al colocarlos juntos, se mejora la localidad espacial del programa. Una adaptación útil del algoritmo del mejor ajuste es modificar la colocación en el caso en el que no puede encontrarse un trozo del tamaño exacto solicitado. En este caso, utilizamos la estrategia del *siguiente ajuste*, tratando de asignar el objeto en el último trozo que se haya dividido, siempre que haya suficiente espacio en el trozo para el nuevo objeto. La estrategia del siguiente ajuste también tiende a aumentar la velocidad de la operación de asignación.

Administración y coalescencia del espacio libre

Cuando se desasigna un objeto en forma manual, el administrador de memoria debe liberar su trozo, para que pueda asignarse de nuevo. En ciertas circunstancias tal vez también sea posible combinar (*coalescencia*) ese trozo con trozos adyacentes del montículo, para formar un trozo más grande. Hay una ventaja en esto, ya que siempre es posible usar un trozo grande para que realice el trabajo de varios trozos pequeños de un tamaño total equivalente, aunque muchos trozos pequeños no pueden contener un objeto grande, como es el caso con el trozo combinado.

Si mantenemos un contenedor para trozos de un tamaño fijo, como lo hace Lea para los tamaños pequeños, entonces tal vez sea preferible no combinar bloques adyacentes de ese tamaño en un trozo del doble del tamaño. Es más simple mantener todos los trozos de un solo tamaño en todas las páginas que sean necesarias, y nunca combinarlos. Así, un esquema simple de asignación y desasignación sería mantener un mapa de bits, con un bit para cada trozo en el contenedor. Un 1 indica que el trozo está ocupado; un 0 indica que está libre. Cuando se desasigna un trozo, cambiamos su 1 por un 0. Cuando tenemos que asignar un trozo, buscamos cualquier trozo que tenga un bit 0, cambiamos ese bit por 1 y usamos el trozo correspondiente. Si no hay trozos libres, obtenemos una nueva página, la dividimos en trozos del tamaño apropiado y extendemos el vector de bits.

Las cosas se vuelven más complejas cuando el montículo se administra como un todo, sin contenedores, o si estamos dispuestos a combinar los trozos adyacentes y desplazar el trozo resultante hacia un contenedor distinto, si es necesario. Hay dos estructuras de datos que son útiles para soportar la coalescencia de bloques libres adyacentes:

- *Etiquetas delimitadoras.* En los extremos superior e inferior de cada trozo, ya sea que esté libre o asignado, mantenemos información vital. En ambos extremos, tenemos un bit libre/usado que indica si el bloque se encuentra asignado (usado) o disponible (libre). Junto a cada bit libre/usado hay un conteo del número total de bytes en el trozo.
- *Una lista libre incrustada, doblemente enlazada.* Los trozos libres (pero no los asignados) también se enlazan en una lista doblemente enlazada. Los apuntadores para esta lista se encuentran dentro de los mismos bloques, por decir adyacentes a las marcas delimitadoras en cada extremo. Por ende, no se necesita espacio adicional para la lista libre, aunque su existencia coloca un límite inferior acerca de cómo se pueden obtener los trozos pequeños; deben acomodar dos marcas delimitadoras y dos apuntadores, incluso si el objeto es de un solo byte. El orden de los trozos en la lista libre se deja sin especificar.

Por ejemplo, la lista podría ordenarse por tamaño, con lo cual se facilita la colocación en base al mejor ajuste.

Ejemplo 7.10: La figura 7.17 muestra parte de un montículo con tres trozos adyacentes, *A*, *B* y *C*. El trozo *B*, con un tamaño de 100, acaba de ser desasignado y devuelto a la lista libre. Como conocemos el inicio (extremo izquierdo) de *B*, también conocemos el extremo del trozo que se encuentra justo a la izquierda de *B*, que en este ejemplo es *A*. El bit libre/usado en el extremo derecho de *A* es 0 en este momento, por lo que también *A* está libre. Por lo tanto, podemos combinar *A* y *B* en un solo trozo de 300 bytes.

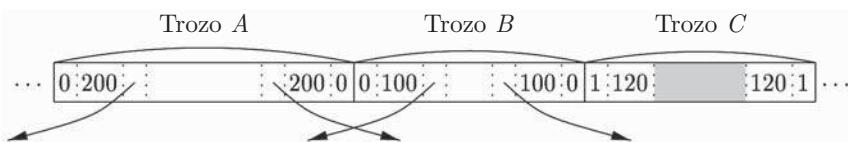


Figura 7.17: Parte de un montículo y una lista libre doblemente enlazada

Podría darse el caso de que el trozo *C*, que se encuentra justo a la derecha de *B*, también esté libre, en cuyo caso podemos combinar *A*, *B* y *C*. Observe que si combinamos trozos siempre que sea posible, entonces nunca podrá haber dos trozos libres adyacentes, por lo que nunca tendremos que buscar más allá de los dos trozos adyacentes al que se está desasignando. En el caso actual, buscamos el inicio de *C* empezando en el extremo izquierdo de *B*, que conocemos, y buscamos el número total de bytes en *B*, que se encuentra en la etiqueta delimitadora izquierda de *B* y es de 100 bytes. Con esta información, buscamos el extremo derecho de *B* y el inicio del trozo a su derecha. En ese punto, examinamos el bit libre/usado de *C* y descubrimos que es 1, lo cual significa que está usado; por ende, *C* no está disponible para combinarse.

Como debemos combinar a *A* y *B*, tenemos que eliminar uno de ellos de la lista libre. La estructura tipo lista libre doblemente enlazada nos permite buscar los trozos que se encuentran antes y después de *A* y de *B*. Hay que tener en cuenta que no debemos suponer que los vecinos físicos *A* y *B* también son adyacentes en la lista libre. Si conocemos los trozos que van antes y después de *A* y *B* en la lista libre, es muy sencillo manipular los apuntadores en la lista para sustituir a *A* y *B* por un solo trozo combinado. □

La recolección automática de basura puede eliminar la fragmentación por completo, si se desplazan todos los objetos asignados a un almacenamiento contiguo. En la sección 7.6.4 veremos con detalle la interacción entre la recolección de basura y la administración de memoria.

7.4.5 Solicitudes de desasignación manual

Cerraremos esta sección con la administración manual de la memoria, en donde el programador debe arreglar en forma explícita la desasignación de datos, como en C y C++. Lo ideal es que se elimine cualquier almacenamiento que ya no vayamos a usar. Por el contrario, cualquier almacenamiento al que vayamos a hacer referencia no debe eliminarse. Por desgracia, es difícil

hacer valer cualquiera de estas propiedades. Además de considerar las dificultades con la desasignación manual, vamos a describir algunas de las técnicas que utilizan los programadores para ayudarse con estas dificultades.

Problemas con la desasignación manual

La administración manual de la memoria está propensa a errores. Los errores comunes toman dos formas: el no poder eliminar datos a los que no se puede referenciar es un error de *fuga de memoria*, y el hacer referencia a datos eliminados es un error de *desreferencia de apuntador colgante*.

Es difícil para los programadores saber si un programa *nunca* hará referencia a cierto almacenamiento en el futuro, por lo que el primer error común es no eliminar el almacenamiento al que nunca se hará referencia. Observe que, aunque las fugas de memoria pueden reducir la velocidad de ejecución de un programa debido a un incremento en el uso de la memoria, no afectan el hecho de que un programa sea correcto, siempre y cuando la máquina no se quede sin memoria. Muchos programas pueden tolerar las fugas de memoria, en especial si la fuga es lenta. Sin embargo, para los programas que se ejecutan por períodos largos, y en especial los programas que nunca se detienen, como los sistemas operativos o el código de servidor, es imprescindible que no tengan fugas.

La recolección automática de basura se deshace de las fugas de memoria, desasignando toda la basura. Aún con la recolección automática de basura, un programa puede utilizar de todas formas más memoria de la necesaria. Un programador puede saber si nunca se hará referencia a un objeto, aun cuando existan referencias a ese objeto en alguna parte. En ese caso, el programador debe eliminar deliberadamente estas referencias, para que puedan desasignarse de manera automática.

Si se pone demasiado empeño en la eliminación de objetos, se pueden producir problemas aún peores que las fugas de memoria. El segundo error común es eliminar cierto almacenamiento y después tratar de hacer referencia a los datos en el almacenamiento desasignado. Los apuntadores a almacenamiento que se ha desasignado se conocen como *apuntadores colgantes*. Una vez que el almacenamiento liberado se reasigna a una nueva variable, cualquier operación de lectura, escritura o desasignación a través del apuntador colgante puede producir efectos que parecen aleatorios. A cualquier operación, como lectura, escritura o desasignación, que siga a un apuntador y trate de utilizar el objeto al que apunta, se le conoce como *desreferenciar* el apuntador.

Observe que, al leer a través de un apuntador colgante se puede llegar a obtener un valor arbitrario. Al escribir a través de un apuntador colgante se modifica en forma arbitraria el valor de la nueva variable. Desasignar el almacenamiento de un apuntador colgante significa que el almacenamiento de la nueva variable puede asignarse a otra variable más, y las acciones sobre la variable nueva y la anterior pueden entrar en conflicto unas con otras.

A diferencia de las fugas de memoria, al desreferenciar un apuntador colgante después de que se reasigna el almacenamiento liberado, casi siempre se produce un error en el programa que es difícil de depurar. Como resultado, los programadores se inclinan más por no desasignar una variable, si no están seguros de que no se pueda hacer referencia a ella.

Una forma relacionada de error de programación es el acceso a una dirección ilegal. Algunos ejemplos comunes de dichos errores incluyen el desreferenciar apuntadores nulos y acceder a

Un ejemplo: Purify

Purify de Rational es una de las herramientas comerciales más populares que ayuda a los programadores a encontrar errores de acceso a memoria y fugas de memoria en los programas. Purify instrumenta el código binario agregando instrucciones adicionales para comprobar errores a medida que el programa se ejecuta. Mantiene un mapa de la memoria para indicar en dónde se encuentran todos los espacios liberados y usados. Cada objeto asignado se agrupa con espacio adicional; los accesos a ubicaciones no asignadas o a espacios entre los objetos se marcan como errores. Este método encuentra algunas referencias a apuntadores colgantes, pero no cuando la memoria se ha reasignado y hay un objeto válido ocupando ese lugar. Este método también encuentra algunos de los accesos a arreglos fuera de límites, si caen dentro del espacio insertado al final de los objetos.

Purify también encuentra las fugas de memoria al final de la ejecución de un programa. Analiza el contenido de todos los objetos asignados, en busca de posibles valores de apuntadores. Cualquier objeto que no tenga un apuntador es un trozo de memoria que se considera como fuga. Purify reporta la cantidad de memoria que se fugó y las ubicaciones de los objetos fugados. Podemos comparar a Purify con un “recolector de basura conservador”, lo cual veremos en la sección 7.8.3.

un elemento de un arreglo fuera de los límites. Es mejor detectar dichos errores que hacer que el programa corrompa los resultados en silencio. De hecho, muchas violaciones de seguridad explotan errores de programación de este tipo, en donde ciertas entradas del programa permiten el acceso no intencional a los datos, con lo cual un “hacker” podría tomar el control del programa y de la máquina. Un antídoto es hacer que el compilador inserte comprobaciones con cada acceso, para asegurar que se encuentre dentro de los límites. El optimizador del compilador puede descubrir y eliminar esas comprobaciones que no sean realmente necesarias, ya que el optimizador puede deducir que el acceso debe estar dentro de los límites.

Convenciones y herramientas de programación

Ahora presentaremos algunas de las convenciones y herramientas más populares que se han desarrollado para ayudar a los programadores a lidiar con la complejidad en la administración de la memoria:

- La *propiedad de los objetos* es útil cuando podemos razonar de manera estática acerca del tiempo de vida de un objeto. La idea es asociar a un *propietario* con cada objeto en todo momento. El propietario es un apuntador a ese objeto, que supuestamente pertenece a cierta invocación de una función. El propietario (es decir, su función) es responsable de eliminar el objeto o de pasarlo a otro propietario. Es posible tener otros apuntadores que no sean propietarios al mismo objeto; estos apuntadores se pueden sobreescibir en cualquier momento, y no deben aplicarse eliminaciones a través de ellos. Esta convención elimina las fugas de memoria, así como los intentos de eliminar el mismo objeto dos veces. Sin embargo, no ayuda a resolver el problema de la referencia a un apuntador colgante, ya que es posible seguir un apuntador que no sea propietario a un objeto que se haya eliminado.

- El *conteo de referencias* es útil cuando el tiempo de vida de un objeto debe determinarse en forma dinámica. La idea es asociar de manera dinámica un conteo con cada objeto asignado. Cada vez que se crea una referencia al objeto, incrementamos el conteo de referencias; cada vez que se elimina una referencia, decrementamos el conteo. Cuando el conteo llega a cero, ya no se puede hacer referencia al objeto y, por lo tanto, puede eliminarse. Sin embargo, esta técnica no atrapa las estructuras de datos circulares sin usar, en donde no se puede acceder a una colección de objetos, pero sus conteos de referencia no son cero, ya que se hacen referencia unas a otras. Para una ilustración de este problema, vea el ejemplo 7.11. El conteo de referencias erradica todas las referencias de apuntadores colgantes, ya que no hay referencias pendientes a objetos eliminados. El conteo de referencia es costoso, ya que impone una sobrecarga en cada operación que almacena a un apuntador.
- La *asignación basada en regiones* es útil para las colecciones de objetos cuyos tiempos de vida están enlazados a fases específicas en un cálculo. Cuando se crean objetos para usarse sólo dentro de cierto paso de un cálculo, podemos asignarlos a todos en la misma región. Después eliminamos la región completa, una vez que termina el paso del cálculo. Esta técnica de *asignación basada en regiones* tiene un grado de aplicación limitado. No obstante, cuando se puede usar es muy eficiente; en vez de desasignar un objeto a la vez, los elimina a todos en la región de una vez por todas.

7.4.6 Ejercicios para la sección 7.4

Ejercicio 7.4.1: Suponga que el montículo consiste en siete trozos, empezando en la dirección 0. Los tamaños de los trozos en orden son: 80, 30, 60, 50, 70, 20, 40 bytes. Al colocar un objeto en un trozo, lo colocamos en el extremo superior si hay suficiente espacio restante para formar un trozo más pequeño (de manera que el trozo más pequeño pueda permanecer fácilmente en la lista enlazada de espacio libre). Sin embargo, no podemos tolerar trozos menores de 8 bytes, por lo que si un objeto es casi tan grande como el trozo seleccionado, le proporcionamos todo el trozo completo y colocamos el objeto en el extremo inferior del trozo. Si solicitamos espacio para objetos de los siguientes tamaños: 32, 64, 48, 16, en ese orden, ¿cuál es la apariencia de la lista de espacio libre después de satisfacer las solicitudes, si el método para seleccionar trozos es:

- a) ¿Primer ajuste?
- b) ¿Mejor ajuste?

7.5 Introducción a la recolección de basura

Los datos a los que no se puede hacer referencia se conocen, por lo general, como *basura*. Muchos lenguajes de programación de alto nivel evitan al programador tener que administrar la memoria en forma manual, al ofrecer la recolección automática de basura, la cual desasigna los datos inalcanzables. La recolección de basura existe desde la implementación inicial de Lisp, en 1958. Otros lenguajes importantes que ofrecen recolección de basura son: Java, Perl, ML, Modula-3, Prolog y Smalltalk.

En esta sección presentaremos muchos de los conceptos de la recolección de basura. La noción de que un objeto sea “alcanzable” es tal vez intuitiva, pero debemos ser precisos; en la sección 7.5.2 hablaremos sobre las reglas exactas. Además, en la sección 7.5.3 veremos un método simple pero imperfecto de recolección automática de basura: el conteo de referencias, que se basa en la idea de que, una vez que un programa ha perdido todas las referencias a un objeto, simplemente no puede y, por lo tanto, no hará referencia al almacenamiento.

La sección 7.6 cubre los recolectores basados en el rastreo, que son algoritmos para descubrir todos los objetos que siguen siendo útiles, y después convierten todos los demás trozos del montículo en espacio libre.

7.5.1 Metas de diseño para los recolectores de basura

La recolección de basura es el proceso de reclamar los trozos de almacenamiento que contiene objetos a los que un programa ya no puede tener acceso. Debemos suponer que los objetos tienen un tipo que el recolector de basura puede determinar en tiempo de ejecución. Con la información sobre el tipo podemos determinar qué tan grande es el objeto, y qué componentes del objeto contienen referencias (apuntadores) a otros objetos. También suponemos que las referencias a los objetos siempre son a la dirección del inicio del objeto, y nunca apuntadores a lugares dentro del objeto. Por ende, todas las referencias a un objeto tienen el mismo valor y pueden identificarse con facilidad.

Un programa de usuario, al cual nos referiremos como el *mutador*, modifica la colección de objetos en el montículo. El mutador crea objetos adquiriendo espacio del administrador de memoria, y el mutador puede introducir y quitar referencias a objetos existentes. Los objetos se convierten en basura cuando el programa mutador no puede “alcanzarlos”, en el sentido que precisaremos en la sección 7.5.2. El recolector de basura busca estos objetos inalcanzables y reclama su espacio enviándolos al administrador de memoria, que lleva el registro del espacio libre.

Un requerimiento básico: seguridad en los tipos

No todos los lenguajes son buenos candidatos para la recolección automática de basura. Para que un recolector de basura pueda funcionar, debe indicar en dónde se encuentra cualquier elemento de datos o componente de un elemento de datos, o si podría usarse como apuntador a un trozo de espacio de memoria asignado. Se dice que un lenguaje en el que se puede determinar el tipo de cualquier componente de datos tiene *seguridad en los tipos*. Hay lenguajes con seguridad en los tipos como ML, para el cual podemos determinar los tipos en tiempo de compilación. Existen otros lenguajes con seguridad en los tipos como Java, cuyos tipos no pueden determinarse en tiempo de compilación, sino en tiempo de ejecución. A éstos se les llama lenguajes con *tipos dinámicos*. Si un lenguaje no tiene seguridad estática o dinámica en los tipos, entonces decimos que es *inseguro*.

Los lenguajes inseguros, en los que por desgracia se incluyen algunos de los lenguajes más importantes como C y C++, son malos candidatos para la recolección automática de basura. En los lenguajes inseguros, las direcciones de memoria se pueden manipular en forma arbitraria: pueden aplicarse operaciones aritméticas arbitrarias a los apuntadores para crear nuevos apuntadores, y se pueden convertir enteros arbitrarios en apuntadores. Por ende, en teoría un programa podría hacer referencia a cualquier ubicación en memoria, en cualquier momento. En

consecuencia, ninguna ubicación de memoria se puede considerar como inaccesible, y ningún almacenamiento puede reclamarse con seguridad.

En la práctica, la mayoría de los programas en C y C++ no generan apuntadores en forma arbitraria, y se ha utilizado un recolector de basura que en teoría es poco sólido, pero funciona bien en la práctica. En la sección 7.8.3 hablaremos sobre la recolección de basura conservadora para C y C++.

Métrica de rendimiento

A menudo, la recolección de basura es tan costosa que, aunque se inventó hace varias décadas y previene por completo las fugas de memoria, todavía no la adoptan muchos de los principales lenguajes de programación. A través de los años se han propuesto muchos métodos distintos, y no hay un algoritmo de recolección de basura que sea sin duda el mejor. Antes de explorar las opciones, primero vamos a enumerar las medidas de rendimiento que debemos considerar al diseñar un recolector de basura.

- *Tiempo de ejecución total.* La recolección de basura puede ser un proceso muy lento. Es importante que no aumente en forma considerable el tiempo total de ejecución de una aplicación. Como es necesario que el recolector de basura interactúe con muchos datos, en gran parte su rendimiento se determina en base a la forma en que impulsa el subsistema de memoria.
- *Uso de espacio.* Es importante que la recolección de basura evite la fragmentación y que haga el mejor uso de la memoria disponible.
- *Tiempo de pausa.* Los recolectores de basura simples son notables por hacer que los programas (los mutadores) se detengan en forma repentina durante un tiempo extremadamente largo, ya que la recolección de basura entra en acción sin advertencia previa. Por ende, además de minimizar el tiempo total de ejecución, es conveniente que se disminuya el tiempo de pausa máximo. Como un caso especial importante, las aplicaciones en tiempo real requieren completar ciertos cálculos dentro de un límite de tiempo. Debemos suprimir la recolección de basura mientras se realizan tareas en tiempo real, o restringir el tiempo de pausa máximo. Por ende, la recolección de basura se utiliza raras veces en las aplicaciones en tiempo real.
- *Localidad de los programas.* No podemos evaluar la velocidad de un recolector de basura sólo en base a su tiempo de ejecución. El recolector de basura controla la colocación de los datos y, por ende, ejerce una influencia sobre la localidad de los datos del programa mutador. Puede mejorar la localidad temporal de un mutador al liberar espacio y reutilizarlo; puede mejorar la localidad espacial del mutador al reubicar los datos que se utilizan en conjunto en la misma caché, o en las mismas páginas.

Algunas de estas metas de diseño entran en conflicto unas con otras, por lo que debemos realizar concesiones con cuidado, considerando la forma en que se comportan los programas comúnmente. Además, los objetos de distintas características pueden favorecer distintos tratamientos, para lo cual un recolector requiere usar distintas técnicas para distintos tipos de objetos.

Por ejemplo, el número de objetos asignados lo dominan los objetos pequeños, por lo que la asignación de objetos pequeños no debe suceder en una gran sobrecarga. Por otro lado, hay que considerar los recolectores de basura que reubicán objetos alcanzables. La reubicación es un proceso costoso al tratar con objetos grandes, pero es menos problemático con los objetos pequeños.

Como otro ejemplo, en general, entre más tengamos que esperar a recolectar la basura en un recolector basado en rastreo, mayor será la fracción de objetos que puedan recolectarse. La razón es que a menudo los objetos “mueren siendo jóvenes”, por lo que si esperamos un poco, muchos de los objetos recién asignados quedarán inalcanzables. Por lo tanto, dicho recolector es menos costoso en promedio, por cada objeto inalcanzable recolectado. Por otra parte, la recolección poco frecuente aumenta el uso de la memoria de un programa, disminuye su localidad en los datos y aumenta la duración de las pausas.

En contraste, un recolector de conteo de referencias, al introducir una sobrecarga constante en muchas de las operaciones del mutador, puede reducir la velocidad de ejecución general de un programa en forma considerable. Por otro lado, el conteo de referencias no crea pausas largas, y es eficiente en el uso de la memoria, ya que busca la basura tan pronto como ésta se produce (con la excepción de ciertas estructuras cíclicas que veremos en la sección 7.5.3).

El diseño del lenguaje también puede afectar las características del uso de la memoria. Algunos lenguajes fomentan un estilo de programación que genera mucha basura. Por ejemplo, los programas en los lenguajes de programación funcionales, o casi funcionales, crean más objetos para evitar mutar los ya existentes. En Java, todos los objetos exceptuando los tipos base como enteros y referencias, se asignan en el montículo y no en la pila, aun cuando sus tiempos de vida estén confinados al de la invocación de una función. Este diseño libera al programador de tener que preocuparse sobre los tiempos de vida de las variables, a expensas de generar más basura. Se han desarrollado optimizaciones para los compiladores, para analizar los tiempos de vida de las variables y asignarlos en la pila, siempre que sea posible.

7.5.2 Capacidad de alcance

Nos referimos a todos los datos a los que un programa puede acceder en forma directa, sin tener que desreferenciar ningún apuntador, como el *conjunto raíz*. Por ejemplo, en Java el conjunto raíz de un programa consiste en todos los miembros de campo estáticos y todas las variables en su pila. Es obvio que un programa puede alcanzar cualquier miembro de su conjunto raíz en cualquier momento. De manera recursiva, cualquier objeto con una referencia que se almacene en los miembros de campo o en los elementos de un arreglo de cualquier objeto alcanzable, puede alcanzarse a sí mismo.

La capacidad de alcance se vuelve un poco más compleja cuando el compilador ha optimizado el programa. En primer lugar, un compilador puede mantener las variables de referencia en los registros. Estas referencias también deben considerarse como parte del conjunto raíz. En segundo lugar, aun cuando en un lenguaje con seguridad de tipos los programadores no puedan manipular las direcciones de memoria en forma directa, a menudo un compilador lo hace con el fin de agilizar el código. Por ende, los registros en el código compilado pueden apuntar a la parte media de un objeto o de un arreglo, o pueden contener un valor al cual se aplica un desplazamiento para calcular una dirección legal. He aquí algunas cosas que puede hacer un compilador optimizador para permitir que el recolector de basura encuentre el conjunto raíz correcto:

- El compilador puede restringir la invocación de la recolección de basura a sólo ciertos puntos de código en el programa, cuando no existen referencias “ocultas”.
- El compilador puede escribir información que el recolector de basura puede utilizar para recuperar todas las referencias, como el especificar qué registros contienen referencias, o cómo calcular la dirección base de un objeto que recibe una dirección interna.
- El compilador puede asegurar que haya una referencia a la dirección base de todos los objetos alcanzables, cada vez que pueda invocarse el recolector de basura.

El conjunto de objetos alcanzables cambia a medida que se ejecuta un programa. Crece a medida que se crean nuevos objetos y se reduce a medida que los objetos se vuelven inalcanzables. Es importante recordar que, una vez que un objeto se vuelve inalcanzable, no puede volverse alcanzable de nuevo. Hay cuatro operaciones básicas que realiza un mutador para modificar el conjunto de objetos alcanzables:

- *Asignaciones de objetos.* El administrador de memoria realiza estas asignaciones, devolviendo una referencia a cada trozo de memoria recién asignado. Esta operación agrega miembros al conjunto de objetos alcanzables.
- *Paso de parámetros y valores de retorno.* Las referencias a objetos se pasan del parámetro de entrada actual al correspondiente parámetro formal, y del resultado devuelto al procedimiento que se llamó. Los objetos a los que apuntan estas referencias permanecen alcanzables.
- *Asignaciones de referencias.* Las asignaciones de la forma $u = v$, en donde u y v son referencias, tienen dos efectos. En primer lugar, u es ahora una referencia al objeto al que v hace referencia. Mientras que u sea alcanzable, el objeto al que hace referencia es sin duda alcanzable. En segundo lugar, se pierde la referencia original en u . Si esta referencia es la última para cierto objeto alcanzable, entonces ese objeto se vuelve inalcanzable. Cada vez que un objeto se vuelve inalcanzable, todos los objetos que pueden alcanzarse sólo a través de referencias contenidas en ese objeto también se vuelven inalcanzables.
- *Retornos de procedimientos.* Cuando un procedimiento termina, el marco que contiene sus variables locales se saca de la pila. Si el marco contiene la única referencia alcanzable a cualquier objeto, ese objeto se vuelve inalcanzable. De nuevo, si los objetos que ahora son inalcanzables contienen las únicas referencias a otros objetos, éstos también se vuelven inalcanzables, y así en lo sucesivo.

En resumen, los nuevos objetos se introducen a través de las asignaciones de objetos. El paso de parámetros y las asignaciones pueden propagar la capacidad de alcance; las asignaciones y los finales de procedimientos pueden terminar la capacidad de alcance. Cuando un objeto se vuelve inalcanzable, puede hacer que más objetos lo sean también.

Hay dos formas básicas de buscar objetos inalcanzables. Podemos atrapar las transiciones en el momento en el que los objetos alcanzables se vuelven inalcanzables, o podemos localizar en forma periódica a todos los objetos alcanzables y después inferir que todos los demás objetos no lo son. El *conteo de referencia*, que presentamos en la sección 7.4.5, es una aproximación reconocida para

Supervivencia de los objetos en la pila

Cuando se hace una llamada a un procedimiento, una variable v , cuyo objeto se asigna en la pila, puede tener apuntadores a v colocados en variables no locales. Estos apuntadores seguirán existiendo después de que el procedimiento regrese, aunque el espacio para v desaparece, lo cual produce una situación de referencia colgante. ¿Debemos asignar un valor local como v en la pila, como lo hace C, por ejemplo? La respuesta es que la semántica de muchos lenguajes *requiere* que las variables locales dejen de existir cuando su procedimiento regresa. Retener una referencia a dicha variable es un error de programación, y no se requiere que el compilador corrija el error en el programa.

el primer método. Mantenemos un conteo de las referencias a un objeto, a medida que el mutador realiza acciones que pueden modificar el conjunto de capacidad de alcance. Cuando el conteo llega a cero, el objeto se vuelve inalcanzable. En la sección 7.5.3 hablaremos sobre este método con más detalle.

El segundo método calcula la capacidad de alcance mediante un rastreo de todas las referencias en forma transitiva. Un recolector de basura *basado en el rastreo* empieza etiquetando (“marcando”) todos los objetos en el conjunto raíz como “alcanzables”, examina en forma iterativa todas las referencias en los objetos alcanzables para buscar más objetos alcanzables y los etiqueta como tales. Este método debe rastrear todas las referencias antes de poder determinar que cierto objeto es inalcanzable. Pero una vez que se calcula el conjunto alcanzable, puede buscar muchos objetos inalcanzables a la vez y localizar una buena cantidad de almacenamiento libre al mismo tiempo. Como todas las referencias deben analizarse al mismo tiempo, tenemos la opción de reasignar los objetos alcanzables y, por lo tanto, reducir la fragmentación. Hay muchos algoritmos distintos basados en rastreo; en las secciones 7.6 y 7.7.1 hablaremos sobre las opciones disponibles.

7.5.3 Recolectores de basura con conteo de referencias

Ahora consideraremos un recolector de basura simple, aunque imperfecto, basado en el conteo de referencias, el cual identifica la basura a medida que un objeto cambia de alcanzable a inalcanzable; el objeto puede eliminarse cuando su conteo disminuye hasta cero. Con un recolector de basura con conteo de referencias, cada objeto debe tener un campo para el conteo de referencia. Estos conteos pueden mantenerse de la siguiente forma:

1. *Asignación de objetos.* El conteo de referencia del nuevo objeto se establece en 1.
2. *Paso de parámetros.* El conteo de referencia de cada objeto que se pasa a un procedimiento se incrementa.
3. *Asignaciones de referencias.* Para la instrucción $u = v$, en donde u y v son referencias, el conteo de referencias del objeto al que v hace referencia aumenta en uno, y el conteo para el objeto anterior al que u hacía referencia se reduce en uno.

4. *Retornos de procedimientos.* Cuando un procedimiento termina, todas las referencias contenidas por las variables locales del registro de activación de ese procedimiento deben también decrementarse. Si varias variables locales contienen referencias al mismo objeto, el conteo de ese objeto debe decrementarse una vez para cada una de esas referencias.
5. *Pérdida transitiva de la capacidad de alcance.* Cada vez que el conteo de referencias de un objeto se vuelve cero, también debemos decrementar el conteo de cada objeto al que apunta una referencia dentro del objeto.

El conteo de referencias tiene dos desventajas importantes: no puede recolectar las estructuras de datos cíclicas inalcanzables, y es costoso. Las estructuras de datos cíclicas son bastante plausibles; a menudo, estas referencias apuntan de vuelta a sus nodos padres, o se apuntan entre sí como referencias cruzadas.

Ejemplo 7.11: La figura 7.18 muestra tres objetos con referencias entre sí, pero sin referencias de ninguna otra parte. Si ninguno de estos objetos es parte del conjunto raíz, entonces todos son basura, pero cada uno de sus conteos de referencia son mayores que 0. Dicha situación es equivalente a una fuga de memoria, si utilizamos el conteo de referencia para la recolección de basura, ya que entonces esta basura y cualquier estructura como ésta nunca se desasignarán. \square

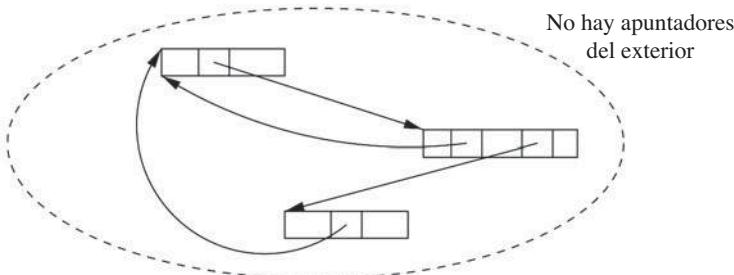


Figura 7.18: Una estructura de datos cíclica, inalcanzable

La sobrecarga del conteo de referencia es alta, ya que se introducen operaciones adicionales con cada asignación de referencias, y en las entradas y salidas de los procedimientos. Esta sobrecarga es proporcional a la cantidad de cálculos en el programa, y no sólo para el número de objetos en el sistema. De especial interés son las actualizaciones que se realizan a las referencias en el conjunto raíz de un programa. El concepto de *conteo de referencias diferidas* se ha propuesto como un medio para eliminar la sobrecarga asociada con la actualización de los conteos de referencias, debido a los accesos locales a la pila. Es decir, los conteos de referencias no incluyen las referencias del conjunto raíz del programa. Un objeto no se considera como basura sino hasta que se explora todo el conjunto raíz y no se encuentran referencias a ese objeto.

Por otra parte, la ventaja del conteo de referencias es que la recolección de basura se realiza en forma *incremental*. Aun cuando la sobrecarga total puede ser extensa, las operaciones se esparcen a través de los cálculos del mutador. Aunque al eliminar una referencia muchos

objetos pueden quedar inalcanzables, la operación de modificar en forma recursiva los conteos de referencias puede diferirse con facilidad, y realizarse en forma gradual con el tiempo. Por ende, el conteo de referencias es un algoritmo muy atractivo cuando deben cumplirse límites de tiempo, así como para las aplicaciones interactivas en las que son inaceptables las pausas extensas y repentinas. Otra ventaja es que la basura se recolecta de inmediato, manteniendo bajo el uso del espacio.

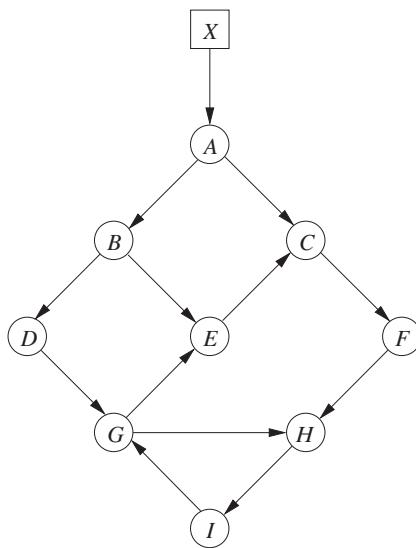


Figura 7.19: Una red de objetos

7.5.4 Ejercicios para la sección 7.5

Ejercicio 7.5.1: ¿Qué ocurre con los conteos de referencia de los objetos en la figura 7.19, si:

- Se elimina el apuntador de A a B ?
- Se elimina el apuntador de X a A ?
- Se elimina el nodo C ?

Ejercicio 7.5.2: ¿Qué ocurre con los conteos de referencia cuando se elimina el apuntador de A a D en la figura 7.20?

7.6 Introducción a la recolección basada en el rastreo

En vez de recolectar la basura a medida que se va creando, los recolectores basados en el rastreo se ejecutan en forma periódica para buscar objetos inalcanzables y reclamar su espacio.

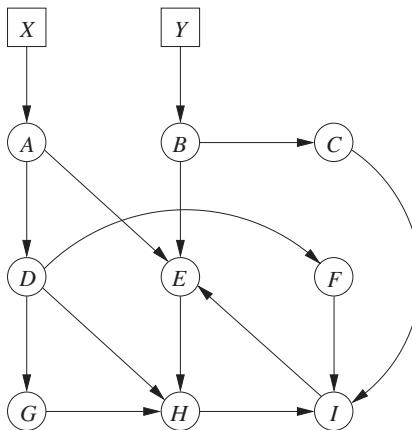


Figura 7.20: Otra red de objetos

Por lo general, ejecutamos el recolector basado en el rastreo cada vez que se agota el espacio libre o cuando su cantidad disminuye por debajo de cierto valor del umbral.

Empezaremos esta sección con la presentación del algoritmo de recolección de basura “marcar y limpiar” más simple. Después describiremos la variedad de algoritmos basados en el rastreo, en términos de cuatro estados en los que se pueden colocar los trozos de memoria. Esta sección también contiene una variedad de mejoras al algoritmo básico, incluyendo aquellas en las cuales la reubicación de objetos forma parte de la función de recolección de basura.

7.6.1 Un recolector básico “marcar y limpiar”

Los algoritmos de recolección de basura *marcar y limpiar* son algoritmos simples que buscan todos los objetos inalcanzables y los colocan en la lista de espacio libre. El algoritmo 7.12 visita y “marca” todos los objetos alcanzables en el primer paso del rastreo y después “limpia” todo el montículo para liberar los objetos inalcanzables. El algoritmo 7.14, que veremos después de presentar un marco de trabajo general para los algoritmos basados en el rastreo, es una optimización del algoritmo 7.12. Al usar una lista adicional para guardar todos los objetos asignados, visita los objetos alcanzables sólo una vez.

Algoritmo 7.12: Recolección de basura marcar y limpiar.

ENTRADA: Un conjunto raíz de objetos, un montículo y una *lista libre*, llamada *Libre*, con todos los trozos desasignados del montículo. Al igual que en la sección 7.4.4, todos los trozos de espacio se marcan con etiquetas delimitadoras para indicar su estado libre/usado y el tamaño.

SALIDA: Una lista *Libre* modificada, después de haber eliminado toda la basura.

MÉTODO: El algoritmo, que se muestra en la figura 7.21, usa varias estructuras de datos simples. La lista *Libre* guarda los objetos que sabemos que están libres. Una lista llamada *SinExplorar* contiene objetos que hemos determinado que son alcanzables, pero cuyos sucesores no hemos

```

1) /* fase de marcado */
2) establecer el bit alcanzado en 1 y agregar a la lista SinExplorar cada objeto
   referenciado por el conjunto raíz;
3) while (SinExplorar ≠  $\emptyset$ ) {
4)     eliminar cierto objeto o de SinExplorar;
5)     for (cada objeto o' referenciado en o) {
6)         if (o' no se alcanzó; es decir, su bit alcanzado es 0) {
7)             establecer el bit alcanzado de o' a 1;
8)             colocar o' en SinExplorar;
9)         }
10}
11}

```

Figura 7.21: Un recolector de basura “marcar y limpiar”

considerado todavía. Es decir, no hemos explorado estos objetos para ver qué otros objetos se pueden alcanzar a través de ellos. La lista *SinExplorar* está vacía al principio. Además, cada objeto incluye un bit para indicar si se ha alcanzado (el *bit alcanzado*). Antes de que empiece el algoritmo, todos los objetos asignados tienen el bit “alcanzado” establecido en 0.

En la línea (1) de la figura 7.21, inicializamos la lista *SinExplorar* colocando ahí todos los objetos a los que el conjunto raíz hace referencia. El bit “alcanzado” para estos objetos también se establece en 1. Las líneas (2) a (7) son un ciclo en el que, a su vez, examina cada objeto *o* que se coloca en la lista *SinExplorar*.

El ciclo for de las líneas (4) a (7) implementa la exploración del objeto *o*. Examinamos cada objeto *o'* para el cual buscamos una referencia dentro de *o*. Si *o'* ya se ha alcanzado (su bit alcanzado es 1), entonces no hay necesidad de hacer algo más acerca de *o'*; o se ha explorado antes, o se encuentra en la lista *SinExplorar* para explorarlo después. No obstante, si *o'* no se ha alcanzado ya, entonces necesitamos establecer su bit alcanzado a 1 en la línea (6) y agregarlo a la lista *SinExplorar* en la línea (7). La figura 7.22 ilustra este proceso. Muestra una lista *SinExplorar* con cuatro objetos. El primer objeto en esta lista, que corresponde al objeto *o* en la explicación anterior, se encuentra en el proceso de ser explorado. Las líneas punteadas corresponden a los tres tipos de objetos que se podrían alcanzar desde *o*:

1. Un objeto explorado con anterioridad, que no necesita explorarse otra vez.
2. Un objeto que se encuentra en la lista *SinExplorar*.
3. Un elemento que es alcanzable, pero que antes se consideraba como inalcanzable.

Las líneas (8) a (11), la fase de limpieza, reclaman el espacio de todos los objetos que permanecen sin alcanzar al final de la fase de marcado. Observe que aquí se incluyen todos los objetos

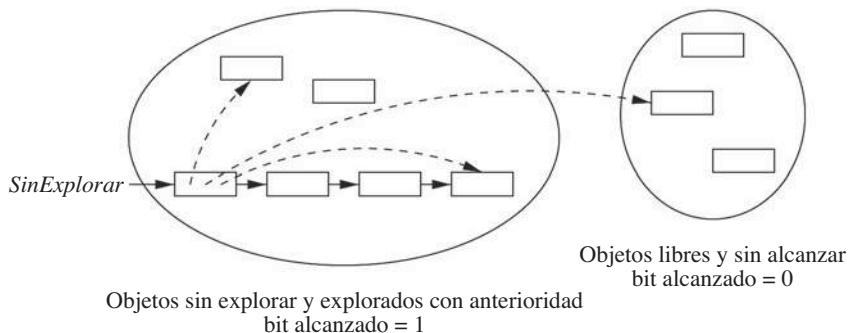


Figura 7.22: Las relaciones entre los objetos durante la fase de marcado de un recolector de basura “marcar y limpiar”

que estaban en un principio en la lista *Libre*. Como no podemos enumerar en forma directa el conjunto de objetos sin alcanzar, el algoritmo realiza una limpieza en todo el montículo completo. La línea (10) coloca los objetos libres y sin alcanzar en la lista *Libre*, uno a la vez. La línea (11) se encarga de los objetos alcanzables. Establecemos su bit alcanzado a 0, para poder mantener las condiciones previas apropiadas para la siguiente ejecución del algoritmo de recolección de basura. \square

7.6.2 Abstracción básica

Todos los algoritmos basados en el rastreo calculan el conjunto de objetos alcanzables y después obtienen el complemento de este conjunto. Por lo tanto, la memoria se recicla de la siguiente manera:

- El programa o mutador se ejecuta y realiza las solicitudes de asignación.
- El recolector de basura descubre la capacidad de alcance mediante el rastreo.
- El recolector de basura reclama el almacenamiento para los objetos inalcanzables.

Este ciclo se ilustra en la figura 7.23, en términos de cuatro estados para los trozos de memoria: *Libre*, *SinAlcanzar*, *SinExplorar* y *Explorado*. El estado de un trozo podría almacenarse en el mismo trozo, o podría estar implícito en las estructuras de datos utilizadas por el algoritmo de recolección de basura.

Aunque los algoritmos basados en rastreo pueden diferir en su implementación, todos pueden describirse en términos de los siguientes estados:

1. *Libre*. Un trozo se encuentra en el estado *Libre* si está listo para ser reasignado. Por ende, un trozo *Libre* no debe contener un objeto alcanzable.
2. *SinAlcanzar*. Los trozos se consideran inalcanzables, a menos que el rastreo demuestre lo contrario. Un trozo se encuentra en el estado *SinAlcanzar* en cualquier punto durante la recolección de basura, si aún no se ha establecido su capacidad de alcance. Cada vez que

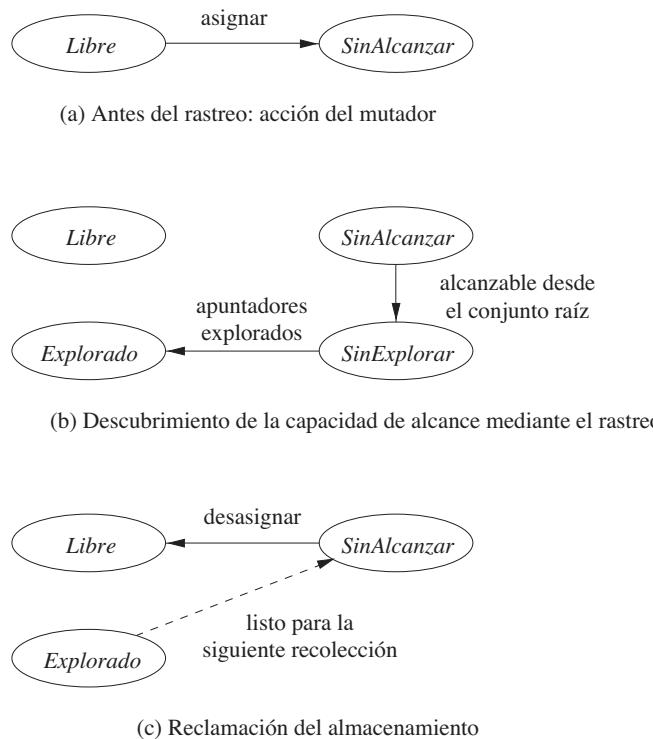


Figura 7.23: Estados de la memoria en el ciclo de recolección de basura

el administrador de memoria asigna un trozo, su estado se establece a *SinAlcanzar*, como se muestra en la figura 7.23(a). Además, después de una ronda de recolección de basura, el estado de un objeto alcanzable se restablece a *SinAlcanzar*, para prepararlo para la siguiente ronda; vea la transición de *Explorado* a *SinAlcanzar*, que se muestra punteada para enfatizar que se prepara para la siguiente ronda.

3. *SinExplorar*. Los trozos que sabemos son alcanzables se encuentran en el estado *SinExplorar*, o en el estado *Explorado*. Un trozo se encuentra en el estado *SinExplorar* si se sabe que es alcanzable, pero no se han explorado todavía sus apuntadores. La transición a *SinExplorar* desde *SinAlcanzar* ocurre cuando descubrimos que un trozo es alcanzable; vea la figura 7.23(b).
4. *Explorado*. Todo objeto *SinExplorar* se explorará en un momento dado, y cambiará al estado *Explorado*. Para explorar un objeto, examinamos cada uno de los apuntadores dentro de éste y seguimos esos apuntadores hacia los objetos a los que hacen referencia. Si una referencia es a un objeto *SinAlcanzar*, entonces ese objeto se coloca en el estado *SinExplorar*. Cuando se completa la exploración de un objeto, ese objeto se coloca en el estado *Explorado*; vea la transición inferior en la figura 7.23(b). Un objeto *Explorado* sólo puede contener referencias a otros objetos *Explorado* o *SinExplorar*, pero nunca a objetos *SinAlcanzar*.

Cuando no quedan objetos en el estado *SinExplorar*, el cálculo de la capacidad de alcance está completo. Los objetos que quedan en el estado *SinAlcanzar* al final son verdaderamente inalcanzables. El recolector de basura reclama el espacio que ocupan y coloca los trozos en el estado *Libre*, como se muestra mediante la transición sólida en la figura 7.23(c). Para prepararse para el siguiente ciclo de recolección de basura, los objetos en el estado *Explorado* se devuelven al estado *SinAlcanzar*; vea la transición punteada en la figura 7.23(c). De nuevo, recuerde que estos objetos en realidad son alcanzables en este momento. El estado *Inalcanzable* es apropiado porque es conveniente empezar con todos los objetos en este estado cuando empieza la recolección de basura, ya que para entonces cualquiera de los objetos que en ese momento podían alcanzarse, pueden sin duda haber quedado inalcanzables.

Ejemplo 7.13: Vamos a ver cómo se relacionan las estructuras de datos del algoritmo 7.12 con los cuatro estados que presentamos antes. Usando el bit “alcanzado” y la membresía en las listas *Libre* y *SinExplorar*, podemos diferenciar cada uno de los cuatro estados. La tabla de la figura 7.24 sintetiza la caracterización de los cuatro estados, en términos de la estructura de datos para el Algoritmo 7.12. \square

ESTADO	EN <i>Libre</i>	EN <i>SinExplorar</i>	BIT ALCANZADO
<i>Libre</i>	Sí	No	0
<i>SinAlcanzar</i>	No	No	0
<i>SinExplorar</i>	No	Sí	1
<i>Explorado</i>	No	No	1

Figura 7.24: Representación de los estados en el Algoritmo 7.12

7.6.3 Optimización de “marcar y limpiar”

El paso final en el algoritmo básico de marcar y limpiar es costoso, ya que no hay una manera fácil de buscar sólo los objetos inalcanzables sin examinar todo el montículo. Un algoritmo mejorado, creado por Baker, mantiene una lista de todos los objetos asignados. Para buscar el conjunto de objetos inalcanzables, que debemos devolver al espacio libre, tomamos la diferencia del conjunto de los objetos asignados y los objetos alcanzados.

Algoritmo 7.14: Recolector “marcar y limpiar” de Baker.

ENTRADA: Un conjunto raíz de objetos, un montículo, una lista llamada *Libre* de espacio libre, y una lista de objetos asignados, a los cuales nos referimos como *SinAlcanzar*.

SALIDA: Las listas modificadas *Libre* y *SinAlcanzar*, que contienen los objetos asignados.

MÉTODO: En este algoritmo, que se muestra en la figura 7.25, la estructura de datos para la recolección de basura consta de cuatro listas llamadas *Libre*, *SinAlcanzar*, *SinExplorar* y *Explorado*, cada una de las cuales contiene todos los objetos en el estado del mismo nombre. Estas listas pueden implementarse mediante listas incrustadas, doblemente enlazadas, como vimos en la sección 7.4.4. No se utiliza un bit “alcanzado” en los objetos, pero suponemos que

cada objeto contiene bits que indican en cuál de los cuatro estados se encuentra. Al principio, *Libre* es la lista libre que mantiene el administrador de memoria, y todos los objetos asignados se encuentran en la lista *SinAlcanzar* (que también mantiene el administrador de memoria, a medida que asigna trozos a los objetos).

- 1) $Explorado = \emptyset$;
- 2) $SinExplorar =$ conjunto de objetos referenciados en el conjunto raíz;
- 3) **while** ($SinExplorar \neq \emptyset$) {
- 4) mover el objeto o de $SinExplorar$ a $Explorado$;
- 5) **for** (cada objeto o' referenciado en o) {
- 6) **if** (o' está en *SinAlcanzar*)
- 7) mover o' de *SinAlcanzar* a *SinExplorar*;
- 8) }
- 9) }
- 10) $Libre = Libre \cup SinAlcanzar$;
- 11) $SinAlcanzar = Explorado$;

Figura 7.25: Algoritmo para marcar y limpiar de Baker

Las líneas (1) y (2) inicializan a *Explorado* para que sea la lista vacía, y a *SinExplorar* para que sólo tenga los objetos que se alcanzan desde el conjunto raíz. Observe que estos objetos se encontraban supuestamente en la lista *SinAlcanzar* y deben eliminarse de ahí. Las líneas (3) a (7) son una implementación simple del algoritmo básico de marcado, que utiliza estas listas. Es decir, el ciclo for de las líneas (5) a (7) examina las referencias en un objeto sin explorar o , y si no se ha alcanzado todavía cualquiera de esas referencias o' , la línea (7) cambia o' al estado *SinExplorar*.

Al final, la línea (8) toma esos objetos que se encuentran aún en la lista *SinAlcanzar* y desasigna sus trozos, moviéndolos a la lista *Libre*. Después, la línea (9) toma todos los objetos en el estado *Explorado*, que son los objetos alcanzables, y reinicializa la lista *SinAlcanzar* para que contenga exactamente esos objetos. Se supone que, a medida que el administrador de memoria cree nuevos objetos, esos también se agregarán a la lista *SinAlcanzar* y se eliminarán de la lista *Libre*. \square

En ambos algoritmos de esta sección hemos asumido que los trozos devueltos a la lista libre quedan como estaban antes de la desasignación. Sin embargo, como vimos en la sección 7.4.4, a menudo es conveniente combinar los trozos libres adyacentes en trozos más grandes. Si deseamos hacer esto, entonces cada vez que devolvamos un trozo a la lista libre, ya sea en la línea (10) de la figura 7.21 o en la línea (8) de la figura 7.25, examinaremos los trozos a su izquierda y a su derecha, y los combinaremos si hay uno libre.

7.6.4 Recolectores de basura “marcar y compactar”

Los recolectores de *reubicación* desplazan los objetos alcanzables alrededor del montículo para ayudar a eliminar la fragmentación de memoria. Es común que el espacio ocupado por los objetos alcanzables sea mucho menor que el espacio liberado. Por ende, después de identificar todos los huecos, en vez de liberarlos en forma individual, una alternativa atractiva es reubicar

todos los objetos alcanzables en un extremo del montículo, dejando todo el resto del montículo como un solo trozo libre. Después de todo, el recolector de basura ya ha analizado cada referencia dentro de los objetos alcanzables, por lo que no se requiere de mucho trabajo para actualizarlas, de manera que apunten a las nuevas ubicaciones. Estas referencias, más las del conjunto raíz, son todas las referencias que necesitamos modificar.

Al tener todos los objetos alcanzables en ubicaciones contiguas se reduce la fragmentación del espacio de memoria, facilitando el alojamiento de objetos grandes. Además, al hacer que los datos ocupen menos líneas y páginas de caché, la reubicación mejora la localidad temporal y espacial de un programa, ya que se crean objetos nuevos casi al mismo tiempo que se asignan los trozos cercanos. Los objetos en trozos cercanos pueden beneficiarse de la preobtención, si se utilizan en conjunto. Además, la estructura de datos para mantener el espacio libre se simplifica; en vez de una lista libre, todo lo que necesitamos es un apuntador *libre* al inicio del único bloque libre.

Los recolectores de reubicación varían en cuanto a si realizan la reubicación en el lugar, o si reservan el espacio anticipándose al tiempo para la reubicación:

- Un *recolector de marcar y compactar*, que se describe en esta sección, compacta los objetos en su lugar. La reubicación en el lugar reduce el uso de la memoria.
- El *recolector de copia* de la sección 7.6.5, más eficiente y popular, desplaza los objetos de una región de la memoria a otra. Al reservar espacio adicional para la reubicación, los objetos alcanzables pueden moverse a medida que se van descubriendo.

El recolector marcar y compactar en el Algoritmo 7.15 tiene tres fases:

1. La primera es una fase de marcado, similar a la de los algoritmos marcar y limpiar que se describieron antes.
2. En segundo lugar, el algoritmo explora la sección asignada del montículo y calcula una nueva dirección para cada uno de los objetos alcanzables. Se asignan nuevas direcciones desde el extremo inferior del montículo, por lo que no hay huecos entre los objetos alcanzables. La nueva dirección para cada objeto se registra en una estructura llamada *NuevaUbicacion*.
3. Por último, el algoritmo copia los objetos a sus nuevas ubicaciones, actualizando todas las referencias en los objetos para que apunten a las nuevas ubicaciones correspondientes. Las direcciones necesarias se encuentran en *NuevaUbicacion*.

Algoritmo 7.15: Un recolector de basura marcar y compactar.

ENTRADA: Un conjunto raíz de objetos, un montículo y *libre*, un apuntador que marca el inicio del espacio libre.

SALIDA: El nuevo valor del apuntador *libre*.

MÉTODO: El algoritmo está en la figura 7.26; utiliza las siguientes estructuras de datos:

1. Una lista *SinExplorar*, como en el Algoritmo 7.12.

2. Los bits “alcanzado” en todos los objetos, al igual que en el Algoritmo 7.12. Para mantener nuestra descripción simple, nos referimos a los objetos como “alcanzados” o “sin alcanzar”, cuando queremos indicar que su bit “alcanzado” es 1 o 0, en forma respectiva. Al principio, ningún objeto se ha alcanzado.
3. El apuntador *libre*, que marca el inicio del espacio sin asignar en el montículo.
4. La tabla *NuevaUbicacion*. Esta estructura podría ser una tabla hash, árbol de búsqueda, u otra estructura que implemente las siguientes dos operaciones:
 - (a) Establecer *NuevaUbicacion*(*o*) a una nueva dirección para el objeto *o*.
 - (b) Dado el objeto *o*, obtener el valor de *NuevaUbicacion*(*o*).

No debemos preocuparnos por la estructura exacta utilizada, aunque podemos asumir que *NuevaUbicacion* es una tabla hash y, por lo tanto, las operaciones “set” y “get” se realizan en tiempo promedio constante, sin importar cuántos objetos hay en el montículo.

La primera fase (de marcado) de las líneas (1) a la (7) es en esencia la misma que la primera fase del Algoritmo 7.12. La segunda fase, de las líneas (8) a (12), visita cada trozo en la parte asignada del montículo, a partir del extremo izquierdo, o inferior. Como resultado, a los trozos se les asignan nuevas direcciones que se incrementan en el mismo orden que sus direcciones anteriores. Este ordenamiento es importante, ya que cuando reubicamos objetos, podemos hacerlo de una forma que asegure que sólo movemos los objetos a la izquierda, en el espacio que estaba antes ocupado por los objetos que ya desplazamos de antemano.

La línea (8) inicia el apuntador *libre* en el extremo inferior del montículo. En esta fase, utilizamos a *libre* para indicar la primera dirección nueva disponible. Creamos una nueva dirección sólo para aquellos objetos *o* que se marcan como alcanzables. El objeto *o* recibe la siguiente dirección disponible en la línea (10), y en la línea (11) incrementamos *libre* por la cantidad de almacenamiento que el objeto *o* requiere, por lo que *libre* apunta de nuevo al inicio del espacio libre.

En la fase final, las líneas (13) a (17), visitamos de nuevo los objetos alcanzables, en el mismo orden a partir de la izquierda que en la segunda fase. Las líneas (15) y (16) sustituyen a todos los apuntadores internos de un objeto alcanzable *o* por sus propios nuevos valores, usando la tabla *NuevaUbicacion* para determinar el sustituto. Después, la línea (17) desplaza el objeto *o*, con las referencias internas modificadas, a su nueva ubicación. Por último, las líneas (18) y (19) redirigen a los apuntadores en los elementos del conjunto raíz que no son objetos del montículo; por ejemplo, los objetos asignados en forma estática o asignados por la pila. La figura 7.27 sugiere cómo los objetos alcanzables (los que no están sombreados) se desplazan hacia abajo del montículo, mientras que los apuntadores internos se modifican para apuntar a las nuevas ubicaciones de los objetos alcanzados. □

7.6.5 Recolectores de copia

Un recolector de copia reserva, con anticipación, el espacio al cual se pueden desplazar los objetos, con lo que se interrumpe la dependencia entre el rastreo y la búsqueda de espacio libre.

```

  /* marcar */
1)  SinExplorar = conjunto de objetos referenciados por el conjunto raíz;
2)  while (SinExplorar ≠  $\emptyset$ ) {
3)      eliminar el objeto o de SinExplorar;
4)      for (cada objeto o' referenciado en o) {
5)          if (o' no se ha alcanzado) {
6)              marcar o' como alcanzado;
7)              colocar o' en la lista SinExplorar;
8)      }
9)      /* calcular nuevas ubicaciones */
10)     libre = ubicación inicial del almacenamiento del montículo;
11)     for (cada trozo de memoria o en el montículo, a partir del extremo inferior) {
12)         if (o es alcanzado {
13)             NuevaUbicacion(o) = libre;
14)             libre = libre + sizeof(o);
15)         }
16)         /* redirigir referencias y mover objetos alcanzables */
17)         for (cada trozo de memoria o en el montículo, a partir del extremo inferior) {
18)             if (o es alcanzado) {
19)                 for (cada referencia o.r en o)
20)                     o.r = NuevaUbicacion(o.r);
21)                 copiar o a NuevaUbicacion(o);
22)             }
23)         }
24)     }
25)     for (cada referencia r en el conjunto raíz)
26)         r = NuevaUbicacion(r);

```

Figura 7.26: Un recolector marcar y compactar

El espacio en memoria se partitiona en dos *semiespacios*, *A* y *B*. El mutador asigna la memoria en un semiespacio, por decir *A*, hasta que se llena, punto en el cual el mutador se detiene y el recolector de basura copia los objetos alcanzables al otro espacio, por decir *B*. Cuando se completa la recolección de basura, se invierten las funciones de los semiespacios. Al mutador se le permite reanudar y asigna los objetos en el espacio *B*, y la siguiente ronda de la recolección de basura mueve los objetos alcanzables al espacio *A*. El siguiente algoritmo se debe a C. J. Cheney.

Algoritmo 7.16: Recolector de copia de Cheney.

ENTRADA: Un conjunto raíz de objetos, y un montículo que consiste en el semiespacio *Desde*, que contiene los objetos asignados, y el semiespacio *Hacia*, todo el cual está libre.

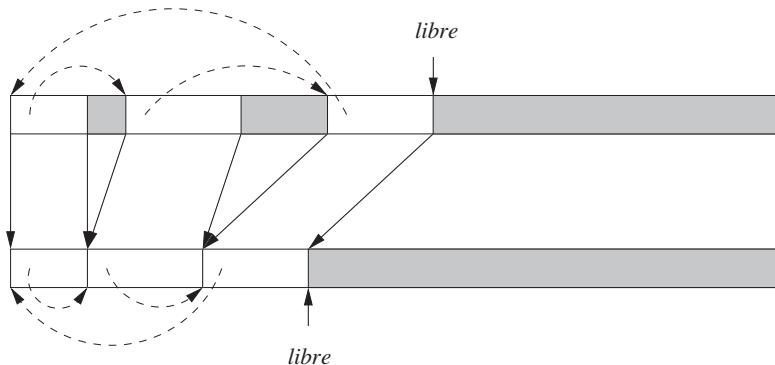


Figura 7.27: Desplazamiento de los objetos alcanzados hacia la parte frontal del montículo, preservando los apuntadores internos

SALIDA: Al final, el semiespacio *Hacia* contiene los objetos asignados. Un apuntador *libre* indica el inicio del espacio libre restante en el semiespacio *Hacia*. El semiespacio *Desde* está completamente libre.

MÉTODO: El algoritmo se muestra en la figura 7.28. El algoritmo de Cheney busca los objetos alcanzables en el semiespacio *Desde* y los copia, tan pronto como se alcanzan, hacia el semiespacio *Hacia*. Esta colocación agrupa los objetos relacionados y puede mejorar la localidad espacial.

Antes de examinar el mismo algoritmo, que es la función *RecolectorCopia* en la figura 7.28, vamos a considerar la función auxiliar *BuscarNuevaUbicacion* en las líneas (11) a (16). Esta función recibe un objeto *o* y le busca una nueva ubicación en el espacio *Hacia*, si *o* no tiene todavía una ubicación ahí. Todas las nuevas ubicaciones se registran en una estructura llamada *NuevaUbicacion*, y un valor de NULL indica que *o* no tiene una ubicación asignada.⁵ Al igual que en el Algoritmo 7.15, la forma exacta de la estructura *NuevaUbicacion* puede variar, pero está bien suponer que es una tabla hash.

Si descubrimos en la línea (12) que *o* no tiene ubicación, entonces se le asigna el inicio del espacio libre dentro del semiespacio *Hacia*, en la línea (13). La línea (14) incrementa el apuntador *libre* por la cantidad de espacio que ocupa *o*, y en la línea (15) copiamos *o* del espacio *Desde* al espacio *Hacia*. Por ende, el movimiento de los objetos de un semiespacio al otro ocurre como un efecto adicional, la primera vez que buscamos la nueva ubicación para el objeto. Sin importar que la ubicación de *o* se haya establecido antes o no, la línea (16) devuelve la ubicación de *o* en el espacio *Hacia*.

Ahora podemos considerar el algoritmo en sí. La línea (2) establece que ninguno de los objetos en el espacio *Desde* tienen nuevas direcciones todavía. En la línea (3), inicializamos dos apuntadores, *sinexplorar* y *libre*, para que apunten al inicio del semiespacio *Hacia*. El apuntador *libre* siempre indicará el inicio del espacio libre dentro del espacio *Hacia*. A medida que agregamos objetos al espacio *Hacia*, aquellos con las direcciones debajo de *sinexplorar* se

⁵En una estructura de datos ordinaria, como una tabla hash, si a *o* no se le asigna una ubicación, entonces simplemente no habría mención de este objeto en la estructura.

```

1)  RecolectorCopia () {
2)      for (todos los objetos o en el espacio Desde) NuevaUbicacion(o) =NULL;
3)      sinexplorar = libre = dirección inicial del espacio Desde;
4)      for (cada referencia r en el conjunto raíz)
5)          sustituir r con BuscarNuevaUbicacion(r);
6)      while (sinexplorar ≠ libre) {
7)          o = objeto en la ubicación sinexplorar;
8)          for (cada referencia o.r dentro de o)
9)              o.r = BuscarNuevaUbicacion(o.r);
10)             sinexplorar = sinexplorar + sizeof(o);
11)      }
12)  }
13)  /* Buscar la nueva ubicación para el objeto, si éste se ha movido. */
14)  /* En cualquier otro caso, colocar el objeto en el estado SinExplorar. */
15)  BuscarNuevaUbicacion(o) {
16)      if (NuevaUbicacion(o) = NULL) {
17)          NuevaUbicacion(o) = libre;
18)          libre = libre + sizeof(o);
19)          copiar o a NuevaUbicacion(o);
20)      }
21)      return NuevaUbicacion(o);
22)  }

```

Figura 7.28: Un recolector de basura de copia

encontrarán en el estado *Explorado*, mientras que los que se encuentran entre *sinexplorar* y *libre* están en el estado *SinExplorar*. Por ende, *libre* siempre dirige a *sinexplorar*, y cuando este último se empareja con el anterior, no hay más objetos *SinExplorar* y terminamos con la recolección de basura. Observe que realizamos nuestro trabajo dentro del espacio *Hacia*, aunque todas las referencias dentro de los objetos examinados en la línea (8) nos conducen de vuelta al espacio *Desde*.

Las líneas (4) y (5) manejan los objetos alcanzables desde el conjunto raíz. Observe que como efecto adicional, algunas de las llamadas a *BuscarNuevaUbicacion* en la línea (5) incrementarán a *libre*, a medida que se asignan trozos para estos objetos dentro de *Hacia*. Por ende, se entrará al ciclo de las líneas (6) a la (10) la primera vez a la que se llegue, a menos que no haya objetos referenciados por el conjunto raíz (en cuyo caso todo el montículo es basura). Después, este ciclo explora cada uno de los objetos que se han agregado a *Hacia* y se encuentra en el estado *SinExplorar*. La línea (7) recibe el siguiente objeto *sinexplorar*, *o*. Después, en las líneas (8) y (9), cada referencia dentro de *o* se traduce desde su valor en el semiespacio *Desde*, hacia su valor en el espacio *Hacia*. Observe que, como efecto adicional, si una referencia dentro de *o* es a un objeto que no hemos alcanzado con anterioridad, entonces la llamada a *BuscarNuevaUbicacion* en la línea (9) crea espacio para ese objeto en el espacio *Hacia*, y desplaza ahí a ese objeto. Por último, la línea (10) incrementa a *sinexplorar* para que apunte al siguiente objeto, justo debajo de *o* en el espacio *Hacia*. \square

7.6.6 Comparación de costos

El algoritmo de Cheney tiene la ventaja de que no toca a ninguno de los objetos inalcanzables. Por otro lado, un recolector de basura de copia debe desplazar el contenido de todos los objetos alcanzables. En especial, este proceso es costoso para los objetos grandes y para los objetos de larga vida que sobreviven a las distintas rondas de la recolección de basura. Podemos resumir el tiempo de ejecución de cada uno de los cuatro algoritmos que se describen en esta sección, como se muestra a continuación. Cada estimación ignora el costo de procesar el conjunto raíz.

- *Marcar y limpiar básico* (Algoritmo 7.12): Proporcional al número de trozos en el montículo.
- *Marcar y limpiar de Baker* (Algoritmo 7.14): Proporcional al número de objetos alcanzables.
- *Marcar y compactar básico* (Algoritmo 7.15): Proporcional al número de trozos en el montículo, más el tamaño total de los objetos alcanzables.
- *Recolector de copia de Cheney* (Algoritmo 7.16): Proporcional al tamaño total de los objetos alcanzables.

7.6.7 Ejercicios para la sección 7.6

Ejercicio 7.6.1: Muestre los pasos de un recolector de basura marcar y limpiar en:

- La figura 7.19, con el apuntador $A \rightarrow B$ eliminado.
- La figura 7.19, con el apuntador $A \rightarrow C$ eliminado.
- La figura 7.20, con el apuntador $A \rightarrow D$ eliminado.
- La figura 7.20, con el objeto B eliminado.

Ejercicios 7.6.2: El algoritmo marcar y limpiar de Baker desplaza los objetos entre cuatro listas: *Libre*, *SinAlcanzar*, *SinExplorar* y *Explorado*. Para cada una de las redes de objetos del ejercicio 7.6.1, indique para cada objeto la secuencia de listas en la que se encuentra a sí misma, justo antes de que empiece la recolección de basura, y hasta justo después de que termine.

Ejercicio 7.6.3: Suponga que realizamos una recolección de basura marcar y compactar en cada una de las redes del ejercicio 7.6.1. Suponga además que

- Cada objeto tiene un tamaño de 100 bytes.
- Al principio, los nueve objetos en el montículo se ordenan en forma alfabética, empezando en el byte 0 del montículo.

¿Cuál es la dirección de cada objeto, después de la recolección de basura?

Ejercicio 7.6.4: Suponga que ejecutamos el algoritmo de recolección de copia en cada una de las redes del ejercicio 7.6.1. Además, suponga que

- i. Cada objeto tiene un tamaño de 100 bytes.
- ii. La lista sinexplorar se maneja como una cola, y cuando un objeto tiene más de un apuntador, los objetos alcanzables se agregan a la cola en orden alfabético.
- iii. El semiespacio *Desde* empieza en la ubicación 0, y el semiespacio *Hacia* empieza en la ubicación 10,000.

¿Cuál es el valor de *NuevaUbicacion*(*o*) para cada objeto *o* que permanece después de la recolección de basura?

7.7 Recolección de basura de pausa corta

Los recolectores simples basados en el rastreo realizan la recolección de basura con el estilo de parar el mundo lo cual puede introducir largas pausas en la ejecución de los programas de usuario. Podemos reducir la longitud de las pausas al realizar la recolección de basura una parte a la vez. Podemos dividir el trabajo en el tiempo, intercalando la recolección de basura con la mutación, o podemos dividir el trabajo en el espacio, recolectando un subconjunto de la basura a la vez. La primera se conoce como *recolección incremental* y la segunda como *recolección parcial*.

Un recolector incremental divide el análisis de capacidad de alcance en unidades más pequeñas, lo cual permite al mutador ejecutarse entre estas unidades de ejecución. El conjunto alcanzable cambia a medida que se ejecuta el mutador, por lo que la recolección incremental es compleja. Como veremos en la sección 7.7.1, buscar una respuesta un poco conservadora puede hacer que el rastreo sea más eficiente.

El más conocido de los algoritmos de recolección parcial es la *recolección de basura generacional*; partitiona los objetos de acuerdo al tiempo transcurrido desde que se han asignado, y recolecta los objetos recién creados con más frecuencia, ya que suelen tener un tiempo de vida más corto. Un algoritmo alternativo, el *algoritmo del tren*, también recolecta un subconjunto de basura a la vez, y se aplica mejor a los objetos más maduros. Estos dos algoritmos pueden utilizarse en conjunto para crear un recolector parcial que maneje los objetos más nuevos y más antiguos de manera distinta. En la sección 7.7.3 hablaremos sobre el algoritmo básico de la recolección parcial, y después describiremos con más detalle la forma en que funcionan los algoritmos generacional y del tren.

Las ideas de la recolección incremental y parcial pueden adaptarse para crear un algoritmo que recolecte los objetos en paralelo en un microprocesador; vea la sección 7.8.1.

7.7.1 Recolección de basura incremental

Los recolectores incrementales son conservadores. Aunque un recolector de basura no debe recolectar objetos que no sean basura, no tiene que recolectar toda la basura en cada ronda. A la basura que queda después de la recolección se le denomina *basura flotante*. Desde luego que

es conveniente disminuir al mínimo la basura flotante. En especial, un recolector incremental no debe dejar la basura que no haya sido alcanzable al principio de un ciclo de recolección. Si podemos estar seguros de una garantía de recolección así, entonces cualquier basura que no se recolecte en una ronda se recolectará en la siguiente, y no habrá fuga de memoria debido a este método para la recolección.

En otras palabras, los recolectores incrementales se van a lo seguro, al sobreestimar el conjunto de objetos alcanzables. Primero procesan el conjunto raíz del programa en forma atómica, sin interferencia por parte del mutador. Despues de buscar el conjunto inicial de objetos sin explorar, las acciones del mutador se intercalan con el paso de rastreo. Durante este periodo, todas las acciones del mutador que pueden cambiar su capacidad de alcance se registran brevemente, en una tabla lateral, de manera que el recolector pueda realizar los ajustes necesarios cuando reanude su ejecución. Si el espacio se agota antes de que se complete el rastreo, el recolector completa el proceso de rastreo, sin permitir que el mutador se ejecute. En cualquier caso, al terminar el rastreo, el espacio se reclama en forma atómica.

Precisión de la recolección incremental

Una vez que un objeto se vuelve inalcanzable, no es posible para ese objeto volverse alcanzable otra vez. Por ende, a medida que proceden la recolección de basura y la mutación, el conjunto de objetos alcanzables sólo puede:

1. Crecer debido a la asignación de nuevos objetos una vez que empieza la recolección de basura.
2. Reducirse al perder referencias a objetos asignados.

Hagamos que el conjunto de objetos alcanzables al principio de la recolección de basura sea R ; hagamos que $Nuevo$ sea el conjunto de objetos asignados durante la recolección de basura, y que $Perdido$ sea el conjunto de objetos que se han vuelto inalcanzables debido a las referencias perdidas desde que empezó el rastreo. El conjunto de objetos alcanzables cuando se completa el rastreo es:

$$(R \cup Nuevo) - Perdido.$$

Es costoso restablecer la capacidad de alcance de un objeto cada vez que un mutador pierde una referencia al objeto, por lo que los recolectores incrementales no tratan de recolectar toda la basura al final del rastreo. Cualquier basura que quede como residuo (basura flotante) debe ser un subconjunto de los objetos $Perdido$. Si se expresa con formalidad, el conjunto S de objetos encontrados por el rastreo debe satisfacer la siguiente condición:

$$(R \cup Nuevo) - Perdido \subseteq S \subseteq (R \cup Nuevo)$$

Rastreo incremental simple

Primero vamos a describir un algoritmo de rastreo simple para encontrar el límite superior $R \cup Nuevo$. El comportamiento del mutador se modifica durante el rastreo de la siguiente manera:

- Todas las referencias que existían antes de la recolección de basura se preservan; es decir, antes de que el mutador sobre escriba una referencia, se recuerda su valor anterior y se trata como un objeto adicional sin explorar que sólo contiene esa referencia.
- Todos los objetos creados se consideran alcanzables de inmediato, y se colocan en el estado *SinExplorar*.

Este esquema es conservador pero correcto, ya que busca a R , el conjunto de todos los objetos alcanzables antes de la recolección de basura, más *Nuevo*, el conjunto de todos los objetos recién asignados. Sin embargo, el costo es alto, ya que el algoritmo intercepta todas las operaciones de escritura y recuerda todas las referencias sobreescritas. Parte de este trabajo es innecesario, ya que puede involucrar objetos que son inalcanzables al final de la recolección de basura. Podríamos evitar parte de este trabajo y también mejorar la precisión del algoritmo, si pudiéramos detectar cuando las referencias sobreescritas apuntan a objetos que son inalcanzables cuando termina esta ronda de recolección de basura. El siguiente algoritmo avanza de manera considerable hacia estas dos direcciones.

7.7.2 Análisis de capacidad alcance incremental

Si intercalamos el mutador con un algoritmo de rastreo básico, como el Algoritmo 7.12, entonces ciertos objetos alcanzables pueden clasificarse por error como inalcanzables. El problema es que las acciones del mutador pueden violar una invariante clave del algoritmo; es decir, un objeto *Explorado* sólo puede contener referencias a otros objetos *Explorado* o *SinExplorar*, nunca a objetos *SinAlcanzar*. Considere el siguiente escenario:

1. El recolector de basura encuentra que el objeto o_1 es alcanzable y explora los apuntadores dentro de o_1 , con lo cual o_1 se coloca en el estado *Explorado*.
2. El mutador almacena una referencia a un objeto *SinAlcanzar* (*pero alcanzable*) o en el objeto *Explorado* o_1 . Para ello, copia una referencia a o desde un objeto o_2 que se encuentra actualmente en el estado *SinAlcanzar* o *SinExplorar*.
3. El mutador pierde la referencia a o en el objeto o_2 . Pudo haber sobreescrito la referencia de o_2 a o antes de explorar la referencia, o tal vez o_2 pudo haberse vuelto inalcanzable y nunca haber llegado al estado *SinAlcanzar* para que se exploraran sus referencias.

Ahora, o es alcanzable a través del objeto o_1 , pero tal vez el recolector de basura no haya visto la referencia a o en o_1 , ni la referencia a o en o_2 .

La clave para un rastreo incremental más preciso y que siga estando correcto, es que debemos observar todas las copias de las referencias a objetos actualmente inalcanzables desde un objeto que no se ha explorado, hacia uno que sí se ha explorado. Para interceptar las transferencias problemáticas de las referencias, el algoritmo puede modificar la acción del mutador durante el rastreo, en cualquiera de las siguientes formas:

- *Barreras de escritura*. Interceptar escrituras de referencias hacia un objeto *Explorado* o_1 , cuando la referencia es hacia un objeto *SinAlcanzar* o . En este caso, clasificamos a o como

alcanzable y lo colocamos en el conjunto *SinExplorar*. De manera alternativa, podemos colocar el objeto escrito o_1 de vuelta en el conjunto *SinExplorar*, para poder volver a explorarlo.

- *Barreras de lectura.* Interceptar las lecturas de referencias en los objetos *SinAlcanzar* o *SinExplorar*. Cada vez que el mutador lee una referencia a un objeto o desde un objeto que se encuentre en el estado *SinAlcanzar* o en *SinExplorar*, clasificamos a o como alcanzable y lo colocamos en el conjunto *SinExplorar*.
- *Barreras de transferencia.* Interceptar la pérdida de la referencia original en un objeto *SinAlcanzar* o *SinExplorar*. Cada vez que el mutador sobrescribe una referencia en un objeto *SinAlcanzar* o *SinExplorar*, guardamos la referencia que se está sobrescribiendo, la clasificamos como inalcanzable y colocamos esta misma referencia en el conjunto *SinExplorar*.

Ninguna de las opciones anteriores encuentra el conjunto más pequeño de objetos alcanzables. Si el proceso de rastreo determina que un objeto es alcanzable, permanece alcanzable aun cuando todas las referencias a éste se sobrescriben antes de que termine el rastreo. Es decir, el conjunto de objetos alcanzables encontrados está entre $(R \cup \text{Nuevo}) - \text{Perdido}$ y $(R \cup \text{Nuevo})$.

Las barreras de escritura son la opción más eficiente de todas las anteriores descritas. Las barreras de lectura son más costosas, ya que por lo general hay muchas más lecturas que escrituras. Las barreras de transferencia no son competitivas; debido a que muchos objetos “mueren jóvenes”, este método retendría muchos objetos inalcanzables.

Implementación de barreras de escritura

Podemos implementar las barreras de escritura de dos maneras. El primer método es recordar, durante una fase de mutación, todas las nuevas referencias que se escribieron en los objetos *Explorado*. Podemos colocar todas estas referencias en una lista; el tamaño de la lista es proporcional al número de operaciones de escritura para los objetos *Explorado*, a menos que se eliminen duplicados de la lista. Hay que tener en cuenta que las referencias en la lista pueden sobrescribirse más adelante por sí mismas, y podrían también ignorarse.

El segundo método, que es más eficiente, es recordar las ubicaciones en donde ocurrieron las escrituras. Podemos recordarlas como una lista de ubicaciones escritas, posiblemente con los duplicados eliminados. Debemos tener en cuenta que no es importante señalar las ubicaciones exactas en las que se realizó la escritura, siempre y cuando todas las ubicaciones en las que se escribió se hayan vuelto a explorar. Por ende, existen varias técnicas que nos permiten recordar menos detalles acerca de la posición exacta de las ubicaciones que se escribieron.

- En vez de recordar la dirección exacta o el objeto y campo que se escribe, podemos recordar sólo los objetos que contienen los campos en los que se escribió.
- Podemos dividir el espacio de direcciones en bloques de tamaño fijo, conocidos como *tarjetas*, y utilizar un arreglo de bits para recordar las tarjetas en las que se ha escrito.

- Podemos optar por recordar las páginas que contienen las ubicaciones en las que se escribió. Podemos simplemente proteger las páginas que contienen objetos *Explorado*. Así, cualquier escritura en objetos *Explorado* se detectará sin ejecutar ninguna instrucción explícita, debido a que producirán una violación a la protección, y el sistema operativo generará una excepción en el programa.

En general, entre menos refinados sean los detalles con los que recordemos las ubicaciones escritas, se requerirá menos espacio, a expensas de incrementar la cantidad de reexploraciones realizadas. En el primer esquema, todas las referencias en los objetos modificados tendrán que volverse a explorar, sin importar qué referencia se haya modificado en realidad. En los últimos dos esquemas, todos los objetos alcanzables en las tarjetas o en las páginas modificadas deben volverse a explorar al final del proceso de rastreo.

Combinación de las técnicas incremental y de copiado

Los métodos anteriores son suficientes para la recolección de basura marcar y limpiar. La recolección de copia es un poco más complicada, debido a su interacción con el mutador. Los objetos en los estados *Explorado* o *SinExplorar* tienen dos direcciones, una en el semiespacio *Desde* y otra en el semiespacio *Hacia*. Al igual que en el Algoritmo 7.16, debemos mantener una asignación de las direcciones anteriores de un objeto, a su dirección reubicada.

Hay dos opciones en la forma en que podemos actualizar las referencias. En primer lugar, podemos hacer que el mutador realice todas las modificaciones en el espacio *Desde*, y sólo al final de la recolección de basura actualizamos todos los apuntadores y copiamos todo el contenido hacia el espacio *Hacia*. En segundo lugar, podemos realizar modificaciones a la representación en el espacio *Hacia*. Cada vez que el mutador desreferencie un apuntador al espacio *Desde*, este apuntador se traducirá a una nueva ubicación en el espacio *Hacia*, si es que hay uno. Al final, todos los apuntadores deberán traducirse para que apunten al espacio *Hacia*.

7.7.3 Fundamentos de la recolección parcial

El hecho fundamental es que por lo general los objetos “mueren jóvenes”. Se ha descubierto que, por lo común, entre el 80 y 98% de todos los objetos recién asignados mueren antes de unos cuantos millones de instrucciones, o antes de que se asigne otro megabyte. Es decir, con frecuencia los objetos se vuelven inalcanzables antes de invocar cualquier tipo de recolección de basura. Por ende, es muy eficiente aplicar con frecuencia la recolección de basura en los objetos nuevos.

Aun así, es probable que los objetos que sobreviven una vez a una recolección puedan sobrevivir a muchas recolecciones más. Con los recolectores de basura que hemos descrito hasta ahora, encontraremos que los mismos objetos maduros serán alcanzables una y otra vez y, en el caso de los recolectores de copia, se copiarán una y otra vez, en cada ronda de recolección de basura. La recolección de basura generacional funciona con más frecuencia en el área del montículo que contiene a los objetos más jóvenes, por lo que tiende a recolectar mucha basura con relativamente poco trabajo. Por otro lado, el algoritmo del tren no invierte una gran proporción de tiempo en los objetos jóvenes, pero sí limita las pausas debido a la recolección de basura. Por ende, una buena combinación de estrategias es utilizar la recolección generacional

para los objetos jóvenes, y una vez que un objeto madura lo suficiente, hay que “promoverlo” hacia un montículo separado, administrado por el algoritmo del tren.

Al conjunto de objetos que se van a recolectar en una ronda de recolección parcial se le conoce como el conjunto *destino*, y al resto de los objetos como el conjunto *estable*. Lo ideal sería que un recolector parcial debiera reclamar todos los objetos en el conjunto destino que sean inalcanzables desde el conjunto raíz del programa. Sin embargo, para ello se requeriría rastrear a todos los objetos, que es lo que hemos tratado de evitar desde un principio. En vez de ello, los recolectores parciales reclaman en forma conservadora a todos esos objetos que no pueden alcanzarse, ya sea a través del conjunto raíz del programa, o del conjunto estable. Como algunos objetos en el conjunto estable pueden ser por sí solos inalcanzables, es posible que tratemos a algunos objetos en el conjunto destino como alcanzables, aunque en realidad no tengan una ruta proveniente del conjunto raíz.

Podemos adaptar los recolectores de basura descritos en las secciones 7.6.1 y 7.6.4 para que trabajen en forma parcial, modificando la definición del “conjunto raíz”. En vez de hacer referencia sólo a los objetos contenidos en los registros, la pila y las variables globales, el conjunto raíz ahora incluye también todos los objetos en el conjunto estable que apuntan a los objetos en el conjunto de destino. Las referencias de los objetos destino a otros objetos destino se rastrean como antes, para encontrar a todos los objetos alcanzables. Podemos ignorar todos los apuntadores a objetos estables, ya que todos estos objetos se consideran alcanzables en esta ronda de recolección parcial.

Para identificar los objetos estables que hacen referencia a los objetos destino, podemos adoptar técnicas similares a las que se utilizan en la recolección de basura incremental. En la recolección incremental debemos recordar todas las escrituras de referencias de los objetos explorados a los objetos sin alcanzar, durante el proceso de rastreo. Aquí debemos recordar todas las escrituras de referencias de los objetos estables a los objetos destino, a lo largo de la ejecución del mutador. Cada vez que el mutador almacena en un objeto estable una referencia a un objeto en el conjunto destino, recordamos la referencia o la ubicación de la escritura. Al conjunto de objetos que contienen referencias de los objetos estables a los objetos destino se le conoce como el *conjunto recordado* para este conjunto de objetos. Como vimos en la sección 7.7.2, podemos comprimir la representación de un conjunto recordado si sólo recordamos la tarjeta o página en la que se encuentra el objeto escrito.

A menudo, los recolectores de basura parciales se implementan como recolectores de basura de copia. Los recolectores que no son de copia también pueden implementarse mediante el uso de listas enlazadas, para llevar la cuenta de los objetos alcanzables. El esquema “generacional” que se describe a continuación es un ejemplo de cómo puede combinarse la recolección de copia con la recolección parcial.

7.7.4 Recolección de basura generacional

La recolección de basura generacional es una manera efectiva de explotar la propiedad de que la mayoría de los objetos mueren jóvenes. El almacenamiento del montículo en una recolección de basura generacional se separa en una serie de particiones. Vamos a usar la convención de enumerarlas como $0, 1, 2, \dots, n$, en donde las particiones con menor numeración contienen a los objetos más jóvenes. Primero, los objetos se crean en la partición 0. Cuando se llena esta

partición, se realiza la recolección de basura y sus objetos alcanzables se mueven a la partición 1. Ahora, con la partición 0 vacía otra vez, reanudamos la asignación de nuevos objetos en esa partición. Cuando la partición 0 se vuelve a llenar,⁶ se recolecta la basura y sus objetos alcanzables se copian a la partición 1, en donde se unen a los objetos que se copiaron antes. Este patrón se repite hasta que la partición 1 también se llena, momento en el cual se aplica la recolección de basura a las particiones 0 y 1.

En general, cada ronda de recolección de basura se aplica a todas las particiones enumeradas con i o un número menor, para cierta i ; la i apropiada a elegir es la partición con la numeración más alta que se encuentre llena. Cada vez que un objeto sobrevive a una recolección (es decir, queda como alcanzable), se promueve de la partición que ocupa a la siguiente partición más alta, hasta que llega a la partición más antigua, la que está enumerada con n .

Utilizando la terminología presentada en la sección 7.7.3, cuando se recolecta la basura en las particiones i y menores, las particiones desde 0 hasta i componen el conjunto destino, y todas las particiones encima de i componen el conjunto estable. Para apoyar la búsqueda de conjuntos raíz para todas las recolecciones parciales posibles, en cada partición i mantenemos un *conjunto recordado*, el cual consiste en todos los objetos en las particiones encima de i que apuntan a los objetos en el conjunto i . El conjunto raíz para una recolección parcial invocada sobre el conjunto i incluye a los conjuntos recordados para las particiones i y menores.

En este esquema, todas las particiones debajo de i se recolectan cada vez que recolectamos a i . Hay dos razones para esta política:

1. Como las generaciones más jóvenes contienen más basura y se recolectan con más frecuencia, de todas formas, podemos recolectarlas junto con una generación más antigua.
2. Si seguimos esta estrategia, tenemos que recordar sólo las referencias que apuntan desde una generación más antigua, hacia una generación más reciente. Es decir, ni las escrituras a los objetos en la generación más joven, ni la promoción de objetos a la siguiente generación provocan actualizaciones en ninguno de los conjuntos recordados. Si recolectamos basura en una partición sin una partición más joven, la generación más joven se volvería parte del conjunto estable, y tendríamos que recordar las referencias que apunten de las generaciones más jóvenes a las más antiguas también.

En resumen, este esquema recolecta las generaciones más jóvenes con más frecuencia, y las recolecciones de estas generaciones son en especial más efectivas, ya que los “objetos mueren jóvenes”. La recolección de basura de generaciones más antiguas requiere más tiempo, ya que incluye la recolección de todas las generaciones más jóvenes, y contiene menos basura en forma proporcional. Sin embargo, las generaciones antiguas tienen que recolectarse de vez en cuando para eliminar los objetos inalcanzables. La generación más antigua contiene los objetos más maduros; su recolección es costosa, ya que equivale a una recolección completa. Es decir, en ocasiones los recolectores generacionales requieren que se realice el paso completo de rastreo y, por lo tanto, pueden introducir largas pausas en la ejecución de un programa. A continuación veremos una alternativa para manejar sólo los objetos maduros.

⁶Técnicamente, las particiones no se “llenan”, ya que el administrador de memoria puede expandirlas con bloques de disco adicionales, si lo desea. No obstante, por lo general existe un límite en el tamaño de una partición, aparte de la última. Nos referiremos a la acción de llegar a este límite como “llenar” la partición.

7.7.5 El algoritmo del tren

Aunque el método generacional es muy eficiente para manejar los objetos inmaduros, es menos eficiente para los maduros, ya que éstos se mueven cada vez que hay una recolección en la que se vean implicados, y es muy poco probable que sean basura. Un método distinto a la recolección incremental, conocido como el *algoritmo del tren*, se desarrolló para mejorar el manejo de los objetos maduros. Puede usarse para recolectar toda la basura, pero tal vez sea mejor usar el método generacional para los objetos inmaduros y, sólo después de que hayan sobrevivido unas cuantas rondas de recolección, “promoverlos” a otro montículo, administrado por el algoritmo del tren. Otra ventaja para el algoritmo del tren es que no tenemos que realizar una recolección de basura completa, como se requiere en ocasiones para la recolección de basura generacional.

Para motivar el algoritmo del tren, vamos a analizar un ejemplo simple de por qué es necesario, en el método generacional, tener rondas ocasionales de recolección de basura en las que se incluya todo. La figura 7.29 muestra dos objetos mutuamente enlazados en dos particiones i y j , en donde $j > i$. Como ambos objetos tienen apuntadores que provienen desde el exterior de su partición, si realizáramos una recolección de sólo la partición i o sólo la partición j , nunca podríamos recolectar uno de estos objetos. Aún así, pueden de hecho formar parte de una estructura de basura cíclica, sin enlaces provenientes del exterior. En general, los “enlaces” entre los objetos mostrados pueden involucrar a muchos objetos y largas cadenas de referencias.

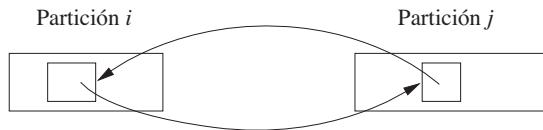


Figura 7.29: Una estructura cíclica a través de particiones que puede ser basura cíclica

En la recolección de basura generacional, en un momento dado recolectamos la partición j , y como $i < j$, también recolectamos a i en ese momento. Así, la estructura cíclica estará contenida por completo en la porción del montículo que se está recolectando, y podemos saber si en realidad es basura. No obstante, si nunca tenemos una ronda de recolección que incluya a i y a j , tendríamos un problema con la basura cíclica, al igual que con el conteo de referencias para la recolección de basura.

El algoritmo del tren utiliza particiones de longitud fija, llamadas *vagones*; un vagón podría ser un solo bloque de disco, siempre y cuando no haya objetos más grandes que los bloques de disco, o el tamaño del vagón podría ser mayor, pero debe ser fijo de forma permanente. Los vagones se organizan en *trenes*. No hay un límite para el número de vagones en un tren, y tampoco para el número de trenes. Hay un orden lexicográfico para los vagones: primero se ordenan por número de tren, y dentro de un tren, se ordenan por número de vagón, como en la figura 7.30.

Hay dos formas en que el algoritmo del tren puede recolectar la basura:

- El primer vagón en orden lexicográfico (es decir, el primer vagón restante del primer tren restante) se recolecta en un paso de recolección de basura incremental. Este paso es similar a la recolección de la primera partición en el algoritmo generacional, ya que mantenemos

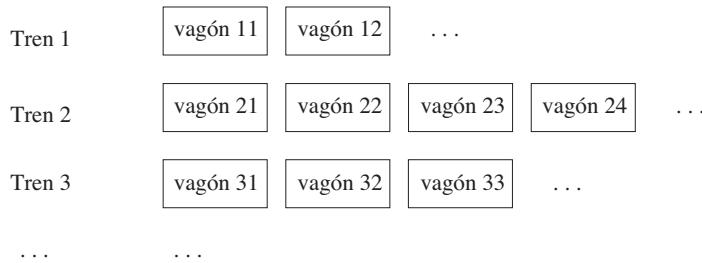


Figura 7.30: Organización del montículo para el algoritmo del tren

una lista “recordados” de todos los apuntadores que provienen del exterior del vagón. Aquí identificamos a los objetos sin referencias, así como los ciclos de basura que están contenidos por completo dentro de este vagón. Los objetos alcanzables en el vagón siempre se mueven hacia algún otro vagón, de manera que cada vagón en el que se haya recolectado la basura se queda vacío y puede eliminarse del tren.

- Algunas veces, el primer tren no tiene referencias externas. Es decir, no hay apuntadores desde el conjunto raíz hacia ningún vagón del tren, y los conjuntos recordados para los vagones sólo contienen referencias provenientes de otros vagones en el tren, no de otros trenes. En esta situación, el tren es una enorme colección de basura cíclica, y eliminamos el tren completo.

Conjuntos recordados

Ahora vamos a proporcionar los detalles del algoritmo del tren. Cada vagón tiene un conjunto recordado, el cual consiste en todas las referencias a los objetos en el vagón que provienen de:

- a) Objetos en los vagones de numeración más alta en el mismo tren.
 - b) Objetos en los trenes de numeración más alta.

Además, cada tren tiene un conjunto recordado que consiste de todas las referencias provenientes de los trenes de mayor numeración. Es decir, el conjunto recordado para un tren es la unión de los conjuntos recordados para sus vagones, excepto aquellas referencias que son internas para el tren. Por lo tanto, es posible representar ambos tipos de conjuntos recordados, dividiendo los conjuntos recordados para los vagones en porciones “internas” (mismo tren) y “externas” (otros trenes).

Observe que las referencias a los objetos pueden venir de cualquier parte, no sólo desde los vagones con un orden lexicográfico mayor. Sin embargo, los dos procesos de recolección de basura tratan con el primer vagón del primer tren, y con todo el primer tren completo, respectivamente. Por ende, cuando es momento de usar los conjuntos recordados en una recolección de basura, no hay nada anterior desde donde pudieran venir las referencias, y por consecuencia no tiene caso recordar las referencias a los vagones con mayor numeración en ningún momento. Desde luego que debemos tener cuidado de administrar los conjuntos recordados en forma apropiada, modificándolos cada vez que el mutador modifique las referencias en cualquier objeto.

Administración de los trenes

Nuestro objetivo es sacar del primer tren todos los objetos que no son basura cíclica. Así, el primer tren se convierte en nada más que basura cíclica y, por lo tanto, se recolecta en la siguiente ronda de recolección de basura, o si la basura no es cíclica, entonces sus vagones pueden recolectarse uno a la vez.

Por ende, debemos empezar nuevos trenes una que otra vez, aun cuando no haya un límite en el número de vagones en un tren, y podríamos en principio sólo agregar nuevos vagones a un solo tren, cada vez que necesitáramos más espacio. Por ejemplo, podríamos empezar un nuevo tren después de cada k creaciones de objetos, para cierta k . Es decir, en general, un nuevo objeto se coloca en el último vagón del último tren, si hay espacio, o en un nuevo vagón que se agrega al final del último tren, si no hay espacio. Sin embargo, en forma periódica debemos empezar mejor un nuevo tren con un vagón, y colocar el nuevo objeto ahí.

Recolección de basura en un vagón

El corazón del algoritmo del tren es la forma en que procesamos el primer vagón del primer tren, durante una ronda de recolección de basura. Al principio, el conjunto alcanzable se considera como los objetos de ese vagón que tienen referencias del conjunto raíz, y los que tienen referencias en el conjunto recordado para ese vagón. Después exploramos estos objetos como en un recolector marcar y limpiar, pero no exploramos los objetos alcanzados que están fuera del vagón que se está recolectando. Después de este rastreo, algunos objetos en el vagón pueden llegar a identificarse como basura. No hay necesidad de reclamar su espacio, ya que todo el vagón va a desaparecer de todas formas.

Sin embargo, es probable que haya algunos objetos alcanzables en el vagón, los cuales deben moverse hacia algún otro lado. Las reglas para mover un objeto son:

- Si hay una referencia en el conjunto recordado que provenga de otro tren (que tendrá una numeración más alta que la del tren del vagón que se está recolectando), entonces hay que mover el objeto a uno de esos trenes. Si hay espacio, el objeto puede ir en algún vagón existente del tren del cual proviene una referencia, o puede ir en el último vagón nuevo, si no hay espacio.
- Si no hay referencias provenientes de otros trenes, pero hay referencias que provienen del conjunto raíz o del primer tren, entonces hay que mover el objeto a cualquier otro vagón del mismo tren, creando un nuevo vagón al último si no hay espacio. Si es posible, hay que elegir un vagón desde el cual provenga una referencia, para ayudar a llevar las estructuras cíclicas a un solo vagón.

Después de mover todos los objetos alcanzables del primer vagón, eliminamos ese vagón.

Modo de pánico

Hay un problema con las reglas antes mencionadas. Para poder asegurarnos que se vaya a recolectar toda la basura en un momento dado, debemos estar seguros de que cada tren se convertirá en algún momento en el primer tren, y si este tren no es basura cíclica, entonces se eliminarán a la larga todos los vagones de ese tren y éste desaparecerá, un vagón a la vez. Sin embargo,

de acuerdo con la regla (2) anterior, al recolectar el primer vagón del primer tren se puede producir un nuevo vagón último. No se pueden producir dos o más vagones nuevos, ya que sin duda todos los objetos del primer vagón pueden acomodarse en el nuevo y último vagón. Sin embargo, ¿podríamos estar en una situación en la que cada paso de recolección para un tren provoque que se agregue un nuevo vagón, y nunca terminaríamos con este tren para avanzar a los demás trenes?

Por desgracia, la respuesta es que sí es posible una situación así. El problema surge si tenemos una estructura grande, cíclica, que no tenga basura, y el mutador se las arregla para modificar las referencias de tal forma que nunca veamos, al momento en que recolectamos un vagón, las referencias de los trenes superiores en el conjunto recordado. Si se elimina tan sólo un objeto del tren durante la recolección de un vagón, entonces todo está bien, ya que no se agregan nuevos objetos al primer tren, y por ende no hay duda de que el primer tren se quedará sin objetos en un momento dado. Sin embargo, tal vez no haya basura que podamos recolectar en cierta etapa, y corremos el riesgo de entrar en un ciclo en el que recolectaremos siempre la basura de sólo el primer tren actual.

Para evitar este problema, debemos comportarnos en forma distinta cada vez que nos encontramos con una recolección de basura *inútil*; es decir, un vagón del que ni siquiera pueda eliminarse un objeto como basura, o moverse a otro tren. En este “modo de pánico”, debemos hacer dos modificaciones:

1. Cuando se escriba una referencia a un objeto en el primer tren, debemos mantener la referencia como un nuevo miembro del conjunto raíz.
2. Al recolectar la basura, si un objeto en el primer vagón tiene una referencia proveniente del conjunto raíz, incluyendo las referencias inútiles establecidas por el punto (1), entonces movemos ese objeto a otro tren, aun cuando no tenga referencias provenientes de otros trenes. No importa a qué tren lo desplazemos, mientras que no sea el primero.

De esta forma, si hay referencias provenientes desde el exterior del primer tren a los objetos en el primer tren, estas referencias se consideran a la hora de recolectar cada vagón, y en algún momento dado se eliminará cierto objeto de ese tren. Entonces podemos salir del modo de pánico y continuar de manera normal, con la seguridad de que ahora el primer tren actual es más pequeño de lo que era antes.

7.7.6 Ejercicios para la sección 7.7

Ejercicio 7.7.1: Suponga que la red de objetos de la figura 7.20 se administra mediante un algoritmo incremental que utiliza las cuatro listas *SinAlcanzar*, *SinExplorar*, *Explorado* y *Libre*, como en el algoritmo de Baker. Para ser específicos, la lista *SinExplorar* se administra como una cola, y cuando se va a colocar más de un objeto en esta lista debido a la exploración de un objeto, lo hacemos en orden alfabético. Suponga también que usamos barreras de escritura para asegurarnos que ningún objeto alcanzable se convierta en basura. Empezando con *A* y *B* en la lista *SinExplorar*, suponga que ocurren los siguientes eventos:

- i. Se explora *A*.
- ii. El apuntador *A* → *D* se rescribe para convertirse en *A* → *H*.

- iii. Se explora B .
- iv. Se explora D .
- v. El apuntador $B \rightarrow C$ se rescribe para convertirse en $B \rightarrow I$.

Simule toda la recolección de basura incremental, suponiendo que no se rescriben más apuntadores. ¿Qué objetos son basura? ¿Qué objetos se colocan en la lista *Libre*?

Ejercicio 7.7.2: Repita el ejercicio 7.7.1, suponiendo que

- a) Se intercambia el orden de los eventos (ii) y (v).
- b) Los eventos (ii) y (v) ocurren antes que (i), (iii) y (iv).

Ejercicio 7.7.3: Suponga que el montículo consiste justo en los nueve vagones en los trenes de la figura 7.30 (es decir, ignore los tres puntos suspensivos). El objeto o en el vagón 11 tiene referencias de los vagones 12, 23 y 32. Al recolectar la basura en el vagón 11, ¿dónde podría terminar o ?

Ejercicio 7.7.4: Repita el ejercicio 7.7.3 para los casos en los que o tiene

- a) Sólo referencias de los vagones 22 y 31.
- b) Ninguna referencia más que la del vagón 11.

Ejercicio 7.7.5: Suponga que el montículo consiste exactamente en los nueve vagones de los tres trenes que se muestran en la figura 7.30 (es decir, ignore los tres puntos suspensivos). En este momento nos encontramos en el modo de pánico. El objeto o_1 en el vagón 11 sólo tiene una referencia, que proviene del objeto o_2 en el vagón 12. Esa referencia se rescribe. Al recolectar la basura en el vagón 11, ¿qué podría ocurrir a o_1 ?

7.8 Temas avanzados sobre la recolección de basura

Vamos a cerrar nuestra investigación sobre la recolección de basura con varios tratamientos breves sobre cuatro temas adicionales:

1. La recolección de basura en entornos en paralelo.
2. Reubicaciones parciales de objetos.
3. Recolección de basura para lenguajes que no cuentan con seguridad en los tipos.
4. La interacción entre la recolección de basura controlada por el programador y la recolección automática de basura.

7.8.1 Recolección de basura paralela y concurrente

La recolección de basura se vuelve aún más retadora cuando se usa en aplicaciones que se ejecutan en paralelo en una máquina con varios procesadores. Es común que las aplicaciones de servidor tengan miles de subprocesos que se ejecutan al mismo tiempo; cada uno de estos subprocesos es un mutador. Por lo general, el montículo consistirá en gigabytes de memoria.

Los algoritmos de recolección de basura escalables deben aprovechar la presencia de varios procesadores. Decimos que un recolector de basura es *paralelo* si utiliza varios subprocesos; decimos que es *concurrente* si se ejecuta en forma simultánea con el mutador.

Vamos a describir un recolector paralelo, y en su mayor parte concurrente, el cual utiliza una fase concurrente y paralela que realiza la mayor parte del trabajo de rastreo, y después una fase de detener el mundo que garantiza que se van a encontrar todos los objetos alcanzables y reclama el almacenamiento. Este algoritmo no introduce nuevos conceptos básicos relacionados con la recolección de basura; muestra cómo podemos combinar las ideas descritas hasta ahora para crear una solución completa al problema de la recolección paralela y concurrente. Sin embargo, hay ciertas cuestiones nuevas de implementación que surgen debido a la naturaleza de la ejecución en paralelo. Vamos a hablar acerca de cómo este algoritmo coordina varios subprocesos en un cálculo en paralelo, usando un modelo de cola de trabajo bastante común.

Para comprender el diseño del algoritmo, debemos tener en cuenta la escala del problema. Incluso hasta el conjunto raíz de una aplicación en paralelo es mucho más grande, ya que consiste en la pila, el conjunto de registros y las variables accesibles a nivel global de cada subproceso. La cantidad de almacenamiento en el montículo puede ser muy extensa, al igual que la cantidad de datos alcanzables. La velocidad en la que se llevan a cabo las mutaciones también es mucho mayor.

Para reducir el tiempo de pausa, podemos adaptar las ideas básicas desarrolladas para el análisis incremental, para traslapar la recolección de basura con la mutación. Recuerde que un análisis incremental, como vimos en la sección 7.7, realiza los siguientes tres pasos:

1. Buscar el conjunto raíz. Por lo general, este paso se realiza en forma automática, es decir, con el (los) mutador(es) detenido(s).
2. Intercalar el rastreo de los objetos alcanzables con la ejecución del (los) mutador(es). En este periodo, cada vez que un mutador escribe una referencia que apunta de un objeto *Explorado* a un objeto *SinAlcanzar*, recordamos esa referencia. Como vimos en la sección 7.7.2, tenemos opciones en relación con la granularidad con la que se recuerdan estas referencias. En esta sección vamos a asumir el esquema basado en tarjetas, en donde dividimos el montículo en secciones llamadas “tarjetas” y mantenemos un mapa de bits que indica qué tarjetas están *sucias* (que se hayan escrito una o más referencias dentro de ellas).
3. Detener el (los) mutador(es) de nuevo para volver a explorar todas las tarjetas que puedan tener referencias a objetos que no se hayan alcanzado.

Para una aplicación con subprocesamiento múltiple extensa, el conjunto de objetos alcanzados por el conjunto raíz puede ser bastante grande. No es factible tomarse el tiempo y el espacio para visitar todos esos objetos mientras cesan todas las mutaciones. Además, debido al gran tamaño del montículo y al extenso número de subprocesos de mutación, tal vez muchas tarjetas deban volver a explorarse después de que se hayan explorado todos los objetos una vez.

Por ende, es aconsejable explorar algunas de estas cartas en paralelo, mientras se permite a los mutadores continuar con su ejecución concurrente.

Para implementar el rastreo del paso (2) anterior, en paralelo, debemos usar varios subprocesos recolectores de basura en forma concurrente con los subprocesos mutadores, para rastrear *la mayoría* de los objetos alcanzables. Después, para implementar el paso (3) detenemos los mutadores y utilizamos subprocesos en paralelo para asegurar que se encuentren todos los objetos alcanzables.

Para llevar a cabo el rastreo del paso (2), hay que hacer que cada subproceso mutador realice parte de la recolección de basura, junto con su propio trabajo. Además, utilizamos subprocesos que se dedican exclusivamente a recolectar basura. Una vez que se ha iniciado la recolección de basura, cada que un subproceso mutador realiza cierta operación de asignación de memoria, también realiza cierto cálculo de rastreo. Los subprocesos que sólo recolectan basura se ponen en uso sólo cuando una máquina tiene ciclos inactivos. Al igual que en el análisis incremental, cada vez que un mutador escribe una referencia que apunta de un objeto *Explorado* a un objeto *SinAlcanzar*, la tarjeta que contiene esta referencia se marca como sucia y debe volver a explorarse.

He aquí una descripción del algoritmo de recolección de basura paralelo y concurrente.

1. Explorar el conjunto raíz para cada subproceso mutador, y colocar todos los objetos que son directamente alcanzables desde ese subproceso en el estado *SinExplorar*. El método incremental más simple para este paso es esperar hasta que un subproceso mutador llame al administrador de memoria y hacer que explore su propio conjunto raíz, si no lo ha hecho ya. Si algún proceso mutador no ha llamado a una función de asignación de memoria, pero el resto del rastreo está completo, entonces este subproceso debe interrumpirse para explorar su conjunto raíz.
2. Explorar los objetos que se encuentran en el estado *SinExplorar*. Para soportar la computación en paralelo, utilizamos una cola de trabajo con *paquetes de trabajo* de tamaño fijo, cada uno de los cuales contiene varios objetos *SinExplorar*. Éstos se colocan en paquetes de trabajo, a medida que se van descubriendo. Los subprocesos en busca de trabajo sacarán estos paquetes de trabajo de la cola y rastrearán los objetos *SinExplorar* que contienen. Esta estrategia permite distribuir el trabajo de manera uniforme entre los trabajadores en el proceso de rastreo. Si el sistema se queda sin espacio, y no podemos encontrar el espacio para crear estos paquetes de trabajo, simplemente marcamos las tarjetas que contienen a los objetos para obligarlas a que se vuelvan a explorar. Esto siempre es posible, ya que el arreglo de bits que contiene las marcas para las tarjetas ya se ha asignado.
3. Explorar los objetos en las tarjetas sucias. Cuando no hay más objetos *SinExplorar* en la cola de trabajo, y cuando se han explorado todos los conjuntos raíz de los hilos, las tarjetas se vuelven a explorar en busca de objetos alcanzables. Mientras que los mutadores continúen ejecutándose, seguirán produciéndose tarjetas sucias. Por ende, debemos detener el proceso de rastreo usando cierto criterio, como permitir que las tarjetas se vuelvan a explorar sólo una vez, o un número fijo de veces, o cuando el número de tarjetas pendientes se reduzca a cierto valor del umbral. Como resultado, este paso paralelo y concurrente termina, por lo general, antes de completar el rastreo, el cual se termina mediante el paso final, que se muestra a continuación.

4. El paso final garantiza que todos los objetos alcanzables se marquen como alcanzados. Con todos los mutadores detenidos, los conjuntos raíz para todos los subprocesos pueden ahora encontrarse con rapidez, usando todos los procesadores en el sistema. Debido a que se ha rastreado la capacidad de alcance de la mayoría de los objetos, sólo se tiene calculado colocar un pequeño número de objetos en el estado *SinExplorar*. Entonces, todos los subprocesos participan en el rastreo del resto de los objetos alcanzables, y en la reexploración de todas las tarjetas.

Es importante controlar la velocidad con la que se lleva a cabo el rastreo. La fase de rastreo es como una carrera. Los mutadores crean nuevos objetos y nuevas referencias que deben explorarse, y el rastreo trata de explorar todos los objetos alcanzables y volver a explorar todas las tarjetas sucias que se generaron en el proceso. No es conveniente empezar el rastreo mucho antes de necesitar una recolección de basura, ya que se incrementará la cantidad de basura flotante. Por otro lado, no podemos esperar hasta que se agote la memoria antes de que empiece el rastreo, ya que entonces los mutadores no podrán avanzar y la situación se degenerará hasta quedar como un recolector “marcar y limpiar”. Por ende, el algoritmo debe elegir el tiempo para comenzar la recolección y la velocidad del rastreo en forma apropiada. Podemos usar una estimación de la velocidad de mutación de ciclos de recolección anteriores para ayudarnos con la decisión. La velocidad de rastreo se ajusta de manera dinámica, para tener en cuenta el trabajo realizado por los hilos puros dedicados a la recolección de basura.

7.8.2 Reubicación parcial de objetos

Como vimos en la sección 7.6.4, los recolectores de copia o de compactación son ventajosos, ya que eliminan la fragmentación. Sin embargo, estos recolectores tienen sobrecargas que no son comunes. Un recolector de compactación requiere desplazar todos los objetos y actualizar todas las referencias al final de la recolección de basura. Un recolector de copia averigua a dónde van los objetos alcanzables, a medida que procede el rastreo; si éste se realiza en forma incremental, debemos traducir todas las referencias de un mutador, o mover todos los objetos y actualizar sus referencias al final. Ambas opciones son muy costosas, en especial para un montículo extenso.

En vez de ello, podemos usar un recolector de basura generacional de copiado. Es efectivo para recolectar objetos inmaduros y reducir la fragmentación, pero puede ser costoso al recolectar objetos maduros. Podemos usar el algoritmo del tren para limitar la cantidad de datos maduros que se analicen cada vez. Sin embargo, la sobrecarga del algoritmo del tren es sensible al tamaño del conjunto recordado para cada partición.

Hay un esquema de recolección híbrido que utiliza el rastreo concurrente para reclamar todos los objetos inalcanzables, y que al mismo tiempo desplaza sólo una parte de los objetos. Este método reduce la fragmentación sin incurrir en el costo total de la reubicación en cada ciclo de recolección.

1. Antes de empezar el rastreo, elija una parte del montículo que se debe evacuar.
2. A medida que se marcan los objetos alcanzables, recuerde también todas las referencias que apuntan a los objetos en el área designada.

3. Cuando termine el rastreo, limpie el almacenamiento en paralelo para reclamar el espacio ocupado por los objetos inalcanzables.
4. Por último, evague los objetos alcanzables que ocupan el área designada y corrija las referencias a los objetos evacuados.

7.8.3 Recolección conservadora para lenguajes inseguros

Como vimos en la sección 7.5.1, es imposible construir un recolector de basura cuyo funcionamiento se garantice para todos los programas en C y C++. Como siempre se da la oportunidad de calcular una dirección con operaciones aritméticas, no puede haber ubicaciones en C y C++ que sean inalcanzables. Sin embargo, muchos programas en C o C++ nunca fabrican las direcciones de esta forma. Se ha demostrado que puede construirse un recolector de basura conservador (uno que no necesariamente descarta toda la basura) para que trabaje bien en la práctica, para esta clase de programas.

Un recolector de basura conservador supone que no podemos fabricar una dirección, o derivar la dirección de un trozo asignado de memoria sin una dirección que apunte hacia alguna parte del mismo trozo. Podemos buscar toda la basura en los programas cumpliendo con dicha suposición, si tratamos cualquier patrón de bits que se encuentre en cualquier parte de la memoria alcanzable como una dirección válida, siempre y cuando ese patrón de bits pueda interpretarse como una ubicación de memoria. Este esquema puede clasificar ciertos datos en forma errónea como direcciones. Sin embargo, es correcto, ya que sólo provoca que el recolector sea conservador y que mantenga más datos de los necesarios.

La reubicación de objetos, que requiere que todas las referencias a las ubicaciones anteriores se actualicen para apuntar a las nuevas ubicaciones, es incompatible con la recolección automática de basura. Debido a que un recolector de basura conservador no sabe si cierto patrón de bits se refiere a una dirección actual, no puede modificar estos patrones para que apunten a nuevas direcciones.

He aquí la forma en que funciona un recolector de basura conservador. En primer lugar, el administrador de memoria se modifica para mantener un *mapa de datos* de todos los trozos asignados de memoria. Este mapa nos permite encontrar con facilidad el límite inicial y final del trozo de memoria que abarca a cierta dirección. El rastreo empieza con la exploración del conjunto raíz del programa, para encontrar cualquier patrón de bits que parezca una ubicación de memoria, sin preocuparse por su tipo. Al buscar estas direcciones potenciales en el mapa de datos, podemos encontrar la dirección inicial de aquellos trozos de memoria que podrían alcanzarse, y los colocamos en el estado *SinExplorar*. Después, exploramos todos los trozos sin explorar, buscamos más (supuestos) trozos alcanzables de memoria y los colocamos en la lista de trabajo, hasta que se vacíe. Después de terminar el rastreo, limpiamos el almacenamiento del montículo usando el mapa de datos para localizar y liberar todos los trozos inalcanzables de memoria.

7.8.4 Referencias débiles

Algunas veces, los programadores utilizan un lenguaje con recolección de basura, pero también desean administrar la memoria, o parte de ella, por sí mismos. Es decir, un programador puede

saber que ya nunca se va a tener acceso a ciertos objetos, aun cuando existan todavía referencias a los objetos. Un ejemplo del proceso de compilación sugerirá el problema.

Ejemplo 7.17: Hemos visto que, con frecuencia, el analizador léxico administra una tabla de símbolos creando un objeto para cada identificador que analiza. Estos objetos pueden aparecer como valores léxicos adjuntos a las hojas del árbol de análisis sintáctico que representan a esos identificadores, por ejemplo. Sin embargo, también es útil para crear una tabla hash, con la cadena del identificador como clave, para localizar estos objetos. Esa tabla facilita al analizador léxico la acción de buscar el objeto cuando encuentra un lexema que es un identificador.

Cuando el compilador pasa el alcance de un identificador I , el objeto de su tabla de símbolo ya no tiene referencias que provienen del árbol de análisis sintáctico, o probablemente de cualquier otra estructura intermedia utilizada por el compilador. Sin embargo, aún hay una referencia al objeto en la tabla hash. Como esta tabla es parte del conjunto raíz del compilador, el objeto no se puede recolectar como basura. Si encontramos otro identificador con el mismo lexema que I , entonces descubriremos que I se encuentra fuera de alcance, y se eliminará la referencia a su objeto. No obstante, si no encontramos otro identificador con este lexema, entonces el objeto de I puede permanecer como no recolectable pero sin uso, durante el proceso de compilación. \square

Si el problema que sugiere el ejemplo 7.17 es importante, entonces el escritor de compiladores podría arreglárselas para eliminar de la tabla hash todas las referencias a objetos, tan pronto como termine su alcance. Sin embargo, una técnica conocida como *referencias débiles* permite al programador depender de la recolección automática de basura, sin tener el montículo cargado con objetos alcanzables pero que en realidad no se utilizan. Dicho sistema permite declarar ciertas referencias como “débiles”. Un ejemplo serían todas las referencias en la tabla hash que hemos estado describiendo. Cuando el recolector de basura explora un objeto, no sigue las referencias débiles dentro de ese objeto, y no hace que los objetos a los que apuntan sean alcanzables. Desde luego que dicho objeto puede seguir siendo alcanzable, si hay otra referencia a él que no sea débil.

7.8.5 Ejercicios para la sección 7.8

! Ejercicio 7.8.1: En la sección 7.8.3 sugerimos que era posible recolectar basura para los programas en C que no fabrican expresiones que apuntan a un lugar dentro de un trozo, a menos que haya una dirección que apunte a cierta parte dentro de ese mismo trozo. Por ende, descartamos el código como:

```
p = 12345;  
x = *p;
```

debido a que, mientras p podría apuntar a cierto trozo de manera accidental, no podría haber otro apuntador a ese trozo. Por otro lado, con el código anterior es más probable que p no apunte hacia ninguna parte, y al ejecutar ese código se producirá un error de segmentación. Sin embargo, en C es posible escribir código de forma que se garantice que una variable como p apuntará a cierto trozo, y sin que haya un apuntador a ese trozo. Escriba un programa de este tipo.

7.9 Resumen del capítulo 7

- ◆ *Organización en tiempo de ejecución.* Para implementar las abstracciones expresadas en el lenguaje fuente, un compilador crea y administra un entorno en tiempo de ejecución, en armonía con el sistema operativo y la máquina de destino. El entorno en tiempo de ejecución tiene áreas estáticas de datos para el código objeto y los objetos de datos estáticos que se crean en tiempo de compilación. También tiene áreas de pila y montículo estáticas para administrar los objetos que se crean y se destruyen, a medida que se ejecuta el programa de destino.
- ◆ *Pila de control.* Por lo general, las llamadas a los procedimientos y sus retornos se administran mediante una pila en tiempo de ejecución, conocida como la *pila de control*. Podemos usar una pila, ya que las llamadas a los procedimientos, o *activaciones*, se anidan en el tiempo; es decir, si p llama a q , entonces esta activación de q se anida dentro de esta activación de p .
- ◆ *Asignación de la pila.* El almacenamiento para las variables locales puede asignarse en una pila en tiempo de ejecución, para los lenguajes que permiten o requieren que las variables locales se vuelvan inaccesibles cuando terminan sus procedimientos. Para tales lenguajes, cada activación en vivo tiene un *registro de activación* (o *marco*) en la pila de control, con la raíz del árbol de activación en la parte inferior, y la secuencia completa de registros de activación en la pila que corresponden a la ruta en el árbol de activación, que va hacia la activación en la que reside el control en ese momento. Esta última activación tiene su registro en la parte superior de la pila.
- ◆ *Acceso a los datos no locales en la pila.* Para los lenguajes como C que no permiten declaraciones de procedimientos anidados, la ubicación para una variable es global o se encuentra en el registro de activación, en la parte superior de la pila en tiempo de ejecución. Para los lenguajes con procedimientos anidados, podemos acceder a los datos no locales en la pila a través de *enlaces de acceso*, que son apuntadores que se agregan a cada registro de activación. Los datos no locales deseados se buscan siguiendo una cadena de enlaces de acceso al registro de activación apropiado. Un *display* es un arreglo auxiliar, que se utiliza en conjunto con los enlaces de acceso, el cual proporciona una alternativa eficiente de acceso directo a una cadena de enlaces de acceso.
- ◆ *Administración del montículo.* El *montículo* es la parte del almacén que se utiliza para los datos que pueden vivir de manera indefinida, o hasta que el programa los elimina en forma explícita. El *administrador de memoria* asigna y desasigna espacio dentro del montículo. La *recolección de basura* busca espacios dentro del montículo que ya no se utilicen y, por lo tanto, pueden reasignarse para alojar otros elementos de datos. Para los lenguajes que lo requieren, el recolector de basura es un subsistema importante del administrador de memoria.
- ◆ *Explotación de la localidad.* Al hacer un buen uso de la jerarquía de la memoria, los administradores de memoria pueden influenciar el tiempo de ejecución de un programa. El tiempo requerido para acceder a distintas partes de la memoria puede variar, desde unos cuantos nanosegundos hasta varios milisegundos. Por fortuna, la mayoría de los programas invierten la mayor parte de su tiempo ejecutando una fracción relativamente

pequeña del código y sólo utilizan una pequeña fracción de los datos. Un programa tiene *localidad temporal* si es probable que acceda de nuevo a las mismas ubicaciones de memoria pronto; tiene *localidad espacial* si es probable que acceda pronto a las ubicaciones de memoria cercanas.

- ♦ *Reducción de la fragmentación.* A medida que el programa asigna y desasigna memoria, el montículo puede *fragmentarse*, o dividirse en grandes cantidades de pequeños espacios libres no contiguos, u *huecos*. La estrategia del *mejor ajuste* (asignar el hueco más pequeño disponible que cumpla con una solicitud) ha resultado funcionar bien en el sentido práctico. Mientras que el mejor ajuste tiende a mejorar la utilización del espacio, tal vez no sea la mejor opción para la localidad espacial. La fragmentación puede reducirse mediante la combinación o *coalescencia* de los huecos adyacentes.
- ♦ *Desasignación manual.* La administración manual de la memoria tiene dos fallas comunes: la acción de no eliminar los datos que no se pueden referenciar es un error de *fuga de memoria*, y la acción de referenciar los datos eliminados es un error de *desreferencia de apuntador colgante*.
- ♦ *Capacidad de alcance.* La *basura* consiste en datos que no se pueden referenciar ni *alcanzar*. Hay dos formas básicas de encontrar objetos inalcanzables: se atrapa la transición al momento en que un objeto alcanzable se vuelve inalcanzable, o se localizan en forma periódica todos los objetos alcanzables y se infiere que todos los demás objetos son inalcanzables.
- ♦ Los *recolectores de conteo de referencias* mantienen un conteo de las referencias a un objeto; cuando el conteo llega a cero, el objeto se vuelve inalcanzable. Dichos recolectores introducen la sobrecarga de mantener las referencias y pueden fallar al tratar de encontrar basura “cíclica”, la cual consiste en objetos inalcanzables que se hacen referencia unos a otros, tal vez a través de una cadena de referencias.
- ♦ Los *recolectores de basura basados en el rastreo* examinan o rastrean en forma iterativa todas las referencias para buscar objetos alcanzables, empezando con el *conjunto raíz*, que consiste en objetos a los que se puede acceder de manera directa, sin tener que desreferenciar apuntadores.
- ♦ Los *recolectores marcar y limpiar* visitan y marcan todos los objetos alcanzables en un primer paso de rastreo, y después limpian el montículo para liberar los objetos inalcanzables.
- ♦ Los *recolectores marcar y compactar* mejoran el proceso de marcar y limpiar; *reubican* los objetos alcanzables en el montículo, para eliminar la fragmentación de memoria.
- ♦ Los *recolectores de copiado* interrumpen la dependencia entre el rastreo y la búsqueda de espacio libre. Particionan la memoria en dos *semiespacios*, *A* y *B*. Las solicitudes de asignación se satisfacen desde un semiespacio, por decir *A*, hasta que se llena, punto en el cual el recolector de basura se hace cargo, copia los objetos alcanzables al otro espacio, por decir *B*, e invierte las funciones de los semiespacios.
- ♦ *Recolectores incrementales.* Los recolectores simples basados en rastreo detienen el programa del usuario mientras se recolecta la basura. Los *recolectores incrementales* intercalan las acciones del recolector de basura y el *mutador* o programa de usuario. El mutador

puede interferir con el análisis de capacidad de alcance incremental, ya que puede modificar las referencias dentro de los objetos previamente explorados. Por lo tanto, los recolectores incrementales se van a lo seguro, al sobreestimar el conjunto de objetos alcanzables; cualquier “basura flotante” puede elegirse en la siguiente ronda de recolección.

- ◆ Los *recolectores parciales* también reducen las pausas; recolectan un subconjunto de la basura a la vez. El *recolector de basura generacional*, el más conocido de los algoritmos de recolección parcial, particiona los objetos según el tiempo transcurrido desde que fueron asignados, y recolecta los objetos recién creados con mucho más frecuencia, ya que suele tener tiempos de vida más cortos. Un algoritmo alternativo, el *algoritmo del tren*, utiliza particiones de longitud fija, conocidas como *vagones*, que se agrupan en *trenes*. Cada paso de la recolección se aplica al primer vagón restante del primer tren restante. Cuando se recolecta un vagón, los objetos alcanzables se mueven hacia otros vagones, por lo que este vagón se queda con basura y puede eliminarse del tren. Estos dos algoritmos pueden usarse en conjunto para crear un recolector parcial que aplique el algoritmo generacional a los objetos más jóvenes y el algoritmo del tren a los objetos más maduros.

7.10 Referencias para el capítulo 7

En la lógica matemática, las reglas de alcance y el paso de parámetros por sustitución se remontan hasta Frege [8]. El cálculo de lambda de Church [3] usa el alcance léxico; se ha utilizado como un modelo para estudiar lenguajes de programación. Algol 60 y sus sucesores, incluyendo C y Java, usan el alcance léxico. Una vez presentado por la implementación inicial de Lisp, el alcance dinámico se convirtió en una característica del lenguaje; McCarthy [14] brinda esta historia.

Muchos de los conceptos relacionados con la asignación de pilas fueron estimulados por bloques y por la recursividad en Algol 60. La idea de una visualización para acceder a los valores no locales en un lenguaje con alcance léxico se debe a Dijkstra [5]. En Randell y Russell [16] aparece una descripción detallada de la asignación de la pila, el uso de una visualización y la asignación dinámica de arreglos. Johnson y Ritchie [10] hablan sobre el diseño de una secuencia de llamadas, la cual permite que el número de argumentos de un procedimiento varíe de una llamada a otra.

La recolección de basura ha sido un área activa de investigación; vea Wilson [17]. El conteo de referencias se remonta hasta Collins [4]. La recolección basada en el rastreo se remonta a McCarthy [13], quien describe un algoritmo marcar y limpiar para las celdas de longitud fija. Knuth diseñó en 1962 la etiqueta delimitadora para administrar el espacio libre, y la publicó en [11].

El Algoritmo 7.14 se basa en Baker [1]. El Algoritmo 7.16 se basa en la versión no recursiva de Cheney [2], del recolector de copiado de Fenichel y Yochelson [7].

Dijkstra y sus colaboradores [6] exploran el análisis de capacidad de alcance incremental. Lieberman y Hewitt [12] presentan un recolector generacional como una extensión de la recolección de copiado. El algoritmo del tren empezó con Hudson y Moss [9].

1. Baker, H. G. Jr., “The treadmill: real-time garbage collection without motion sickness”, *ACM SIGPLAN Notices* 27:3 (Marzo, 1992), pp. 66-70.

2. Cheney, C. J., "A nonrecursive list compacting algorithm", *Comm. ACM* **13**:11 (Noviembre, 1970), pp. 677-678.
3. Church, A., *The Calculi of Lambda Conversion*, Annals of Math. Studies, Núm. 6, Princeton University Press, Princeton, N. J., 1941.
4. Collins, G. E., "A method for overlapping and erasure of lists", *Comm. ACM* **2**:12 (Diciembre, 1960), pp. 655-657.
5. Dijkstra, E. W., "Recursive programming", *Numerische Math.* **2** (1960), pp. 312-318.
6. Dijkstra, E. W., L. Lamport, A. J. Martin, C. S. Scholten y E. F. M. Steffens, "On-the-fly garbage collection: an exercise in cooperation", *Comm. ACM* **21**:11 (1978), pp. 966-975.
7. Fenichel, R. R. y J. C. Yochelson, "A Lisp garbage-collector for virtual memory computer systems", *Comm. ACM* **12**:11 (1969), pp. 611-612.
8. Frege, G., "Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought", (1879). En J. van Heijenoort, *From Frege to Gödel*, Harvard Univ. Press, Cambridge MA, 1967.
9. Hudson, R. L. y J. E. B. Moss, "Incremental Collection of Mature Objects", *Proc. Intl. Workshop on Memory Management*, Apuntes de la conferencia sobre Ciencias Computacionales **637** (1992), pp. 388-403.
10. Johnson, S. C. y D. M. Ritchie, "The C language calling sequence", Reporte técnico de Ciencias Computacionales 102, Bell Laboratories, Murray Hill NJ, 1981.
11. Knuth, D. E., *Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, Boston MA, 1968.
12. Lieberman, H. y C. Hewitt, "A real-time garbage collector based on the lifetimes of objects", *Comm. ACM* **26**:6 (Junio, 1983), pp. 419-429.
13. McCarthy, J., "Recursive functions of symbolic expressions and their computation by machine", *Comm. ACM* **3**:4 (Abril, 1960), pp. 184-195.
14. McCarthy, J., "History of Lisp.", vea las pp. 173-185 en R. L. Wexelblat (ed.), *History of Programming Languages*, Academic Press, Nueva York, 1981.
15. Minsky, M., "A LISP garbage collector algorithm using secondary storage", A. I. Memo 58, MIT Project MAC, Cambridge MA, 1963.
16. Randell, B. y L. J. Russell, *Algol 60 Implementation*, Academia Press, Nueva York, 1964.
17. Wilson, P. R., "Uniprocessor garbage collector techniques",
<ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps>

Capítulo 8

Generación de código

La fase final en nuestro modelo del compilador es el generador de código. Recibe como entrada la representación intermedia que produce el front-end del compilador, junto con la información relevante de la tabla de símbolos, y produce como salida un programa de destino equivalente en forma semántica, como se muestra en la figura 8.1.

Los requerimientos impuestos sobre un generador de código son estrictos. El programa destino debe preservar el significado semántico del programa fuente y ser de alta calidad; es decir, debe hacer un uso efectivo de los recursos disponibles de la máquina destino. Además, el generador de código en sí debe ejecutarse con eficiencia.

El reto es que, en sentido matemático, el problema de generar un programa destino óptimo para un programa fuente dado no se puede determinar; muchos de los subproblemas que surgen en la generación de código, como la repartición de registros, son indecidibles en sentido computacional. En la práctica, debemos conformarnos con las técnicas heurísticas que generan un buen código, aunque no necesariamente óptimo. Por fortuna, la heurística ha madurado tanto que un generador de código diseñado con cuidado puede producir código que es varias veces más rápido que el código producido por uno simple.

Los compiladores que requieren producir programas de destino eficientes incluyen una fase de optimización antes de la generación de código. El optimizador asigna la representación intermedia a otra representación intermedia desde la cual puede generarse código más eficiente. En general, las fases de optimización y generación de código de un compilador, a las que con frecuencia se les conoce como el *back-end*, pueden realizar varias pasadas sobre la representación intermedia antes de generar el programa destino. En el capítulo 9 se describe la optimización de código con detalle. Las técnicas que presentamos en este capítulo pueden usarse sin importar que haya o no una fase de optimización antes de la generación de código.

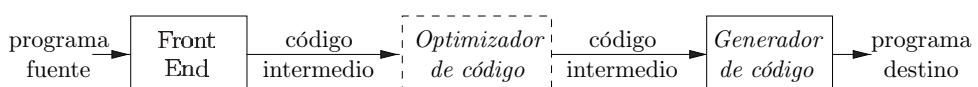


Figura 8.1: Posición del generador de código

Un generador de código tiene tres tareas principales: selección de instrucciones, repartición y asignación de registros, y ordenamiento de instrucciones. La importancia de estas tareas se describe en la sección 8.1. La selección de instrucciones implica la selección de instrucciones apropiadas de la máquina destino para implementar las instrucciones de la representación intermedia. La repartición y asignación de registros implica decidir qué valores mantener en qué registros. El ordenamiento de instrucciones implica decidir en qué orden se va a programar su ejecución.

Este capítulo presenta algoritmos que los generadores de código pueden usar para traducir la representación intermedia en una secuencia de instrucciones del lenguaje destino para las máquinas con registros simples. Ilustraremos estos algoritmos mediante el uso del modelo de la máquina de la sección 8.2. El capítulo 10 trata acerca del problema de la generación de código para las máquinas complejas modernas, que soportan mucho paralelismo dentro de una sola instrucción.

Después de hablar sobre las grandes cuestiones relacionadas con el diseño de un generador de código, mostraremos el tipo de código destino que debe generar un compilador para dar soporte a las abstracciones expresadas en un lenguaje fuente ordinario. En la sección 8.3 describiremos las implementaciones de la asignación y la asignación de pila de las áreas de datos, y mostraremos cómo los nombres en la representación intermedia pueden convertirse en direcciones en el código destino.

Muchos generadores de código partitionan las instrucciones de representación intermedia en “bloques básicos”, los cuales consisten en secuencias de instrucciones que siempre se ejecutan en conjunto. El partitionamiento de la representación intermedia en bloques básicos es el tema de la sección 8.4. La siguiente sección presenta las transformaciones locales simples que pueden usarse para convertir los bloques básicos en bloques básicos modificados, a partir de los cuales se puede generar un código más eficiente. Estas transformaciones son una forma rudimentaria de la optimización de código, aunque no trataremos la teoría más profunda de la optimización de código sino hasta el capítulo 9. Un ejemplo de una transformación local útil es el descubrimiento de las subexpresiones comunes al nivel del código intermedio, y la sustitución resultante de las operaciones aritméticas por operaciones de copia más simples.

La sección 8.6 presenta un algoritmo de generación de código simple, el cual genera código para cada instrucción en turno, manteniendo los operandos en registros el mayor tiempo posible. La salida de este tipo de generador de código puede mejorarse con facilidad mediante las técnicas de optimización de mirilla (peephole), como las que veremos en la sección 8.7.

Las secciones restantes exploran la selección de instrucciones y la repartición de registros.

8.1 Cuestiones sobre el diseño de un generador de código

Aunque los detalles dependen de las características específicas de la representación intermedia, el lenguaje destino y el sistema en tiempo de ejecución, las tareas como la selección de instrucciones, la repartición y asignación de registros, y el ordenamiento de instrucciones se encuentran en el diseño de casi todos los generadores de código.

El criterio más importante para un generador de código es que debe producir código correcto. La precisión de este código es muy relevante, debido al número de casos especiales que podría enfrentar un generador de código. Dada la importancia en la precisión del código, diseñar un generador de código de manera que pueda implementarse, probarse y mantenerse con facilidad es una meta primordial de diseño.

8.1.1 Entrada del generador de código

La entrada del generador de código es la representación intermedia del programa fuente producido por la interfaz de usuario, junto con la información en la tabla de símbolos que se utiliza para determinar las direcciones en tiempo de ejecución de los objetos de datos denotados por los nombres en la representación intermedia.

Entre las diversas opciones para la representación intermedia se encuentran las representaciones de tres direcciones, como los cuádruplos, tripletas, tripletas indirectos; las representaciones de máquinas virtuales como bytecodes y código máquina apilado; las representaciones lineales como la notación postfija; y las representaciones gráficas, como los árboles sintácticos y los GDAs. Muchos de los algoritmos en este capítulo se formulan en términos de las representaciones consideradas en el capítulo 6: código de tres direcciones, árboles y GDAs. Sin embargo, las técnicas que veremos pueden aplicarse también a las demás representaciones intermedias.

En este capítulo vamos a suponer que la interfaz de usuario ha explorado, realizado el análisis sintáctico y traducido el programa fuente en una representación intermedia de un nivel relativamente bajo, para que los valores de los nombres que aparecen en la representación intermedia puedan representarse mediante cantidades que la máquina destino pueda manipular en forma directa, como los números enteros y de punto flotante. También supondremos que se han detectado todos los errores sintácticos y semánticos estáticos, que se ha llevado a cabo la comprobación de tipos necesaria, y que se han insertado operadores de conversión de tipos en todos los lugares necesarios. Así, el generador de código puede proceder en base a la suposición de que su entrada está libre de estos tipos de errores.

8.1.2 El programa destino

La arquitectura del conjunto de instrucciones de la máquina destino tiene un impacto considerable en la dificultad de construir un buen generador de código que produzca código máquina de alta calidad. Las arquitecturas más comunes de las máquinas de destino son RISC (reduced instruction set computer), CISC (complex instruction set computer) y basadas en pilas.

Por lo general, una máquina RISC tiene muchos registros, instrucciones de tres direcciones, modos de direccionamiento simple y una arquitectura del conjunto de instrucciones relativamente sencilla. En contraste, una máquina CISC, por lo general, tiene menos registros, instrucciones de dos direcciones, una variedad de modos de direccionamiento, varias clases de registros, instrucciones de longitud variable e instrucciones con efectos adicionales.

En una máquina basada en pila, las operaciones se realizan metiendo operandos en una pila, y después llevando a cabo las operaciones con los operandos en la parte superior de la pila. Para lograr un alto rendimiento, la parte superior de la pila se mantiene, por lo general, en los registros. Estas máquinas desaparecieron casi por completo, ya que se creía que la organización de la pila era demasiado limitante y requería demasiadas operaciones de intercambio y copiado.

No obstante, las arquitecturas basadas en pila revivieron con la introducción de la Máquina virtual de Java (JVM). Ésta es un intérprete de software para bytecodes de Java, un lenguaje intermedio producido por los compiladores de Java. El intérprete proporciona compatibilidad de software a través de varias plataformas, un importante factor en el éxito de Java.

Para superar el castigo sobre el alto rendimiento de la interpretación, que puede estar en el orden de un factor de 10, se crearon los compiladores Java *just-in-time* (JIT). Estos compiladores JIT traducen los códigos de byte en tiempo de ejecución al conjunto de instrucciones de hardware nativo de la máquina de destino. Otro método para mejorar el rendimiento de Java es construir un compilador que compile directamente en las instrucciones de máquina de la máquina destino, pasando por alto los bytecodes de Java por completo.

Producir un programa en lenguaje máquina absoluto como salida tiene la ventaja de que puede colocarse en una ubicación fija en la memoria, y ejecutarse de inmediato. Los programas pueden compilarse y ejecutarse con rapidez.

Producir un programa en lenguaje máquina reubicable (que a menudo se le conoce como *módulo objeto*) como salida permite que los subprogramas se compilen por separado. Un conjunto de módulos objeto reubicables puede enlazarse y cargarse para que lo ejecute un cargador de enlace. Aunque debemos sufrir la sobrecarga adicional de enlazar y cargar si producimos módulos objeto reubicables, obtenemos mucha flexibilidad al poder compilar las subrutinas por separado y llamar a otros programas (compilados con anterioridad) desde un módulo objeto. Si la máquina destino no maneja la reubicación de manera automática, el compilador debe proporcionar información de reubicación explícita al cargador para enlazar los módulos del programa que se compilaron por separado.

Producir un programa en lenguaje ensamblador como salida facilita de alguna manera el proceso de la generación de código. Podemos generar instrucciones simbólicas y utilizar las facilidades de las macros del ensamblador para ayudar en la generación de código. El precio que se paga es el paso de ensamblado después de la generación de código.

En este capítulo utilizaremos una computadora tipo RISC muy simple como nuestra máquina destino. Le agregaremos algunos modos de direccionamiento tipo CISC, para poder hablar también sobre las técnicas de generación de código para las máquinas CISC. Por cuestión de legibilidad, usaremos código ensamblador como el lenguaje destino. Siempre y cuando puedan calcularse direcciones a partir de desplazamientos y que la demás información se almacene en la tabla de símbolos, el generador de código puede producir direcciones reubicables o absolutas para los nombres, con la misma facilidad que las direcciones simbólicas.

8.1.3 Selección de instrucciones

El generador de código debe asignar el programa de representación intermedia a una secuencia de código que la máquina de destino pueda ejecutar. La complejidad de realizar esta asignación se determina mediante factores como:

- El nivel de la representación intermedia.
- La naturaleza de la arquitectura del conjunto de instrucciones.
- La calidad deseada del código generado.

Si la representación intermedia es de alto nivel, el generador de código puede traducir cada instrucción de este tipo en una secuencia de instrucciones de máquina, usando plantillas de código. Sin embargo, tal generación de código instrucción por instrucción produce a menudo código de baja calidad, que requiere de más optimización. Si la representación intermedia refleja algunos

de los detalles de bajo nivel de la máquina subyacente, entonces el generador de código puede usar esta información para generar secuencias de código más eficientes.

La naturaleza del conjunto de instrucciones de la máquina destino tiene un fuerte efecto sobre la dificultad de la selección de instrucciones. Por ejemplo, la uniformidad y precisión del conjunto de instrucciones son factores importantes. Si la máquina destino no soporta cada tipo de datos de manera uniforme, entonces cada excepción a la regla general requiere de un manejo especial. Por ejemplo, en ciertas máquinas las operaciones de punto flotante se realizan mediante registros separados.

Las velocidades de las instrucciones y las características específicas de las máquinas son otros factores importantes. Si no nos preocupa la eficiencia del programa destino, la selección de instrucciones es un proceso simple. Para cada tipo de instrucción de tres direcciones, podemos diseñar un esqueleto de código que defina el código destino a generar para esa construcción. Por ejemplo, toda instrucción de tres direcciones de la forma $x = y + z$, en donde x , y y z se asignan en forma estática, puede traducirse en la siguiente secuencia de código:

```
LD  R0, y      // R0 = y      (carga y en el registro R0)
ADD R0, R0, z // R0 = R0 + z (suma z a R0)
ST  x, R0      // x = R0      (almacena R0 en x)
```

A menudo, esta estrategia produce operaciones de carga y almacenamiento redundantes. Por ejemplo, la siguiente secuencia de instrucciones de tres direcciones:

$$\begin{aligned} a &= b + c \\ d &= a + e \end{aligned}$$

se traduciría en el siguiente código:

```
LD  R0, b      // R0 = b
ADD R0, R0, c // R0 = R0 + c
ST  a, R0      // a = R0
LD  R0, a      // R0 = a
ADD R0, R0, e // R0 = R0 + e
ST  d, R0      // d = R0
```

Aquí, la cuarta instrucción es redundante, ya que carga un valor que acaba de almacenarse, y de igual forma la tercera, si no se utiliza la a más adelante.

Por lo general, la calidad del código generado se determina en base a su velocidad y tamaño. En la mayoría de las máquinas se puede implementar un programa representación intermedia mediante muchas secuencias de código distintas, con diferencias considerables en cuanto al rendimiento de las distintas implementaciones. Así, una traducción simple del código intermedio puede producir un código de destino correcto, pero demasiado ineficiente.

Por ejemplo, si la máquina de destino tiene una instrucción de “incremento” (INC), entonces la instrucción de tres direcciones $a = a + 1$ puede implementarse con más eficiencia mediante la instrucción individual INC a , en vez de hacerlo mediante una secuencia más obvia para cargar a en un registro, sumar uno al registro y después almacenar el resultado de vuelta en a :

```

LD  R0, a      // R0 = a
ADD R0, R0, #1 // R0 = R0 + 1
ST  a, R0      // a = R0

```

Debemos saber qué tan eficientes son las instrucciones para poder diseñar buenas secuencias de código pero, por desgracia, a menudo esta información es difícil de obtener. Para decidir qué secuencia de código máquina es mejor para una construcción de tres direcciones dada, tal vez también sea necesario tener un conocimiento acerca del contexto en el que aparece esa construcción.

En la sección 8.9 veremos que la selección de instrucciones puede modelarse como un proceso de búsqueda de coincidencias con un patrón tipo árbol, en donde representamos a la representación intermedia y a las instrucciones de máquina como árboles. Después tratamos de “revestir” el árbol de la representación intermedia con un conjunto de subárboles que correspondan a las instrucciones de máquina. Si asociamos un costo a cada subárbol de instrucciones de máquina, podemos usar la programación dinámica para generar secuencias de código óptimas. En la sección 8.11 hablaremos sobre la programación dinámica.

8.1.4 Asignación de registros

Un problema clave en la generación de código es decidir qué valores guardar en qué registros. Los registros son la unidad computacional más veloz en la máquina destino pero, por lo general, no tenemos suficientes como para contener todos los valores. Los valores que no se guardan en registros deben residir en la memoria. Las instrucciones que involucran operandos de registros son invariablemente más cortas y rápidas que las que involucran el uso de operandos en la memoria, por lo que la utilización eficiente de los registros es muy importante.

A menudo, el uso de los registros se divide en dos subproblemas:

1. *Repartición de registros*, durante la cual seleccionamos el conjunto de variables que residirán en los registros, en cada punto del programa.
2. *Asignación de registros*, durante la cual elegimos el registro específico en el que residirá una variable.

Es difícil encontrar una asignación óptima de registros a variables, incluso en las máquinas con un solo registro. En sentido matemático, el problema es NP-completo. El problema se complica aún más debido a que el hardware y el sistema operativo de la máquina destino pueden requerir que se sigan ciertas convenciones de uso de registros.

Ejemplo 8.1: Ciertas máquinas requieren *pares de registros* (un registro con numeración par y el siguiente con numeración impar) para ciertos operandos y resultados. Por ejemplo, en algunas máquinas la multiplicación y la división de enteros implican el uso de pares de registros. La instrucción de multiplicación es de la siguiente forma:

M x, y

en donde x, el multiplicando, es el registro par de un par de registros par/impar y y, el multiplicador, es el registro impar. El producto ocupa todo el par de registros par/impar completo. La instrucción de división es de la siguiente forma:

$D \ x, y$

en donde el dividendo ocupa un par de registros par/ímpar cuyo registro par es x ; el divisor es y . Después de la división, el registro par contiene el residuo y el registro ímpar el cociente.

Ahora, considere las dos secuencias de código de tres direcciones en la figura 8.2, en donde la única diferencia entre (a) y (b) es el operador en la segunda instrucción. Las secuencias de código ensamblador más cortas para (a) y (b) se muestran en la figura 8.3.

$t = a + b$ $t = t * c$ $t = t / d$	$t = a + b$ $t = t + c$ $t = t / d$
(a)	(b)

Figura 8.2: Dos secuencias de código de tres direcciones

L R1,a A R1,b M R0,c D R0,d ST R1,t	L R0, a A R0, b A R0, c SRDA R0, 32 D R0, d ST R1, t
(a)	(b)

Figura 8.3: Secuencias de código máquina óptimas

Ri representa al registro i . SRDA significa Aritmética doble de desplazamiento a la derecha y SRDA R0,32 desplaza el dividendo en R1 y borra R0, para que todos los bits sean iguales a su bit de signo. L, ST y A significan cargar, almacenar y sumar, respectivamente. Observe que la elección óptima para el registro en el que se va a cargar a depende de lo que pasará en última instancia a t . \square

En la sección 8.8 veremos estrategias para la repartición y asignación de registros. La sección 8.10 muestra que para ciertas clases de máquinas podemos construir secuencias de código que evalúen expresiones, usando la menor cantidad posible de registros.

8.1.5 Orden de evaluación

El orden en el que se realizan los cálculos puede afectar la eficiencia del código destino. Como veremos, ciertos órdenes de los cálculos requieren menos registros que otros para contener resultados inmediatos. Sin embargo, el proceso de elegir el mejor orden en el caso general es un problema NP-completo difícil. Al principio, debemos evitar el problema mediante la generación

de código para las instrucciones de tres direcciones, en el orden en el que el generador de código intermedio las produjo. En el capítulo 10 estudiaremos la programación del código para las máquinas con canalizaciones, que pueden ejecutar varias operaciones en un solo ciclo del reloj.

8.2 El lenguaje destino

Un requisito previo para diseñar un buen generador de código es estar familiarizado con la máquina destino y su conjunto de instrucciones. Por desgracia, en una explicación general sobre la generación de código no es posible describir una máquina destino con suficiente detalle como para generar un buen código para un lenguaje completo en esa máquina. En este capítulo vamos a usar código ensamblador como lenguaje destino para una computadora simple que representa a muchas máquinas con registros. Sin embargo, las técnicas de generación que presentaremos en este capítulo pueden usarse en muchas otras clases de máquinas también.

8.2.1 Un modelo simple de máquina destino

Nuestra computadora destino modela una máquina de tres direcciones con operaciones de carga y almacenamiento, operaciones de cálculo, operaciones de salto y saltos condicionales. La computadora subyacente es una máquina con direccionamiento por bytes y n registros de propósito general, $R0, R1, \dots, Rn - 1$. Un lenguaje ensamblador completo tendría muchas instrucciones. Para evitar ocultar los conceptos en una avalancha de detalles, vamos a usar un conjunto bastante limitado de instrucciones, y supondremos que todos los operandos son enteros. La mayoría de las instrucciones consisten en un operador, seguido de un destino, seguido de una lista de operandos de origen. Puede ir una etiqueta después de una instrucción. Vamos a suponer que existen los siguientes tipos de instrucciones:

- Operaciones de *carga*: La instrucción $LD\ dst, dir$ carga el valor que hay en la ubicación dir y lo coloca en la ubicación dst . Esta instrucción denota la asignación $dst = dir$. La forma más común de esta instrucción es $LD\ r, x$, que carga el valor que hay en la ubicación x y lo coloca en el registro r . Una instrucción de la forma $LD\ r_1, r_2$ es una *copia de registro a registro*, en la cual se copia el contenido del registro r_2 al registro r_1 .
- Operaciones de *almacenamiento*: La instrucción $ST\ x, r$ almacena el valor que hay en el registro r y lo coloca en la ubicación x . Esta instrucción denota la asignación $x = r$.
- Operaciones de *cálculo* de la forma $OP\ dst, orig_1, orig_2$, en donde OP es un operador como ADD o SUB, y $dst, orig_1$ y $orig_2$ son ubicaciones, no necesariamente distintas. El efecto de esta instrucción de máquina es aplicar la operación representada por OP a los valores en las ubicaciones $orig_1$ y $orig_2$, y colocar el resultado de esta operación en la ubicación dst . Por ejemplo, $SUB\ r_1, r_2, r_3$ calcula $r_1 = r_2 - r_3$. Cualquier valor que haya estado almacenado en r_1 se pierde, pero si r_1 es r_2 o r_3 , el valor anterior se lee primero. Los operadores unarios que sólo reciben un operando no tienen $orig_2$.

- *Saltos incondicionales*: La instrucción `BR L` provoca que el control se bifurque hacia la instrucción de máquina con la etiqueta `L`. (`BR` significa *bifurcación*.)
- *Saltos condicionales* de la forma `Bcond r, L`, en donde `r` es un registro, `L` una etiqueta y `cond` representa a cualquiera de las evaluaciones comunes sobre los valores del registro `r`. Por ejemplo, `BLTZ r, L` produce un salto a la etiqueta `L` si el valor en el registro `r` es menor que cero, y permite que el control pase a la siguiente instrucción de máquina si no es así.

Vamos a suponer que nuestra máquina destino tiene una variedad de modos de direccionamiento:

- En las instrucciones, una ubicación puede ser el nombre de una variable `x` que haga referencia a la ubicación de memoria que está reservada para `x` (es decir, el valor `l` de `x`).
- Una ubicación también puede ser una dirección indexada de la forma `a(r)`, en donde `a` es una variable y `r` un registro. La ubicación de memoria denotada por `a(r)` se calcula tomando el valor `l` de `a` y sumándolo al valor en el registro `r`. Por ejemplo, la instrucción `LD R1, a(R2)` tiene el efecto de establecer `R1 = contenido(a + contenido(R2))`, en donde `contenido(x)` denota el contenido del registro o la ubicación de memoria representada por `x`. Este modo de direccionamiento es útil para acceder a los arreglos, en donde `a` es la dirección base del arreglo (es decir, la dirección del primer elemento) y `r` contiene el número de bytes después de esa dirección que deseamos avanzar para llegar a uno de los elementos del arreglo `a`.
- Una ubicación de memoria puede ser un entero indexado por un registro. Por ejemplo, `LD R1, 100(R2)` tiene el efecto de establecer `R1 = contenido(100 + contenido(R2))`; es decir, carga en `R1` el valor que hay en la ubicación de memoria que se obtiene al sumar 100 al contenido del registro `R2`. Esta característica es útil para seguir apuntadores, como veremos en el ejemplo que viene a continuación.
- También permitimos dos modos de direccionamiento indirecto: `*r` significa que la ubicación de memoria que se encuentra en la ubicación representada por el contenido del registro `r`, y `*100(r)` indica la ubicación de memoria que se encuentra en la ubicación que se obtiene al sumar 100 al contenido de `r`. Por ejemplo, `LD R1, *100(R2)` tiene el efecto de establecer `R1 = contenido(contenido(100 + contenido(R2)))`; es decir, carga en `R1` el valor que hay en la ubicación de memoria almacenada en la ubicación de memoria que se obtiene al sumar 100 al contenido del registro `R2`.
- Por último, permitimos un modo de direccionamiento constante inmediato. La constante lleva el prefijo `#`. La instrucción `LD R1, #100` carga el entero 100 en el registro `R1`, y `ADD R1, R1, #100` carga el entero 100 en el registro `R1`.

Los comentarios al final de las instrucciones van precedidos por `//`.

Ejemplo 8.2: La instrucción de tres direcciones `x = y - z` puede implementarse mediante las siguientes instrucciones de máquina:

```

LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST  x, R1          // x = R1

```

Tal vez podamos hacerlo mejor. Una de las metas de un buen algoritmo de generación de código es evitar el uso de estas cuatro instrucciones, siempre que sea posible. Por ejemplo, y y z podrían calcularse en un registro, para así evitar el (los) paso(s) con LD. De igual forma, podríamos evitar almacenar x si su valor se utiliza dentro del conjunto de registros y no se necesita posteriormente.

Suponga que a es un arreglo cuyos elementos son valores de 8 bytes, tal vez números reales. Suponga además que los elementos de a se indexan empezando en 0. Podemos ejecutar la instrucción de tres direcciones $b = a[i]$ mediante las siguientes instrucciones de máquina:

```

LD  R1, i          // R1 = i
MUL R1, R1, 8      // R1 = R1 * 8
LD  R2, a(R1)      // R2 = contenido(a + contenido(R1))
ST  b, R2          // b = R2

```

Es decir, el segundo paso calcula $8i$, y el tercer paso coloca en el registro R2 el valor en el i -ésimo elemento de a; el que se encuentra en la ubicación que está $8i$ bytes después de la dirección base del arreglo a.

De manera similar, la asignación en el arreglo a, representada por la instrucción de tres direcciones $a[j] = c$, se implementa mediante:

```

LD  R1, c          // R1 = c
LD  R2, j          // R2 = j
MUL R2, R2, 8      // R2 = R2 * 8
ST  a(R2), R1      // contenido(a + contenido(R2)) = R1

```

Para implementar una indirección simple de un apuntador, como la instrucción de tres direcciones $x = *p$, podemos usar instrucciones de máquina como:

```

LD  R1, p          // R1 = p
LD  R2, 0(R1)      // R2 = contenido(0 + contenido(R1))
ST  x, R2          // x = R2

```

La asignación a través de un apuntador $*p = y$ se implementa de manera similar en código máquina, mediante:

```

LD  R1, p          // R1 = p
LD  R2, y          // R2 = y
ST  0(R1), R2      // contenido(0 + contenido(R1)) = R2

```

Por último, considere una instrucción de tres direcciones de salto condicional como:

```
if x < y goto L
```

El equivalente en código máquina sería algo como:

```
LD  R1, x      // R1 = x
LD  R2, y      // R2 = y
SUB R1, R1, R2 // R1 = R1 - R2
BLTZ R1, M     // if R1 < 0 salta a M
```

Aquí, M es la etiqueta que representa a la primera instrucción de máquina generada a partir de la instrucción de tres direcciones que tiene la etiqueta L. Al igual que para cualquier instrucción de tres direcciones, esperamos poder ahorrarnos algunas de estas instrucciones de máquina, debido a que las operaciones necesarias ya se encuentran en los registros, o porque el resultado no debe almacenarse. □

8.2.2 Costos del programa y las instrucciones

A menudo asociamos un costo con la compilación y la ejecución de un programa. Dependiendo del aspecto del programa que nos interese optimizar, algunas medidas de costos comunes son la longitud del tiempo de compilación y el tamaño, el tiempo de ejecución y el consumo de energía del programa de destino.

La determinación del costo actual de compilar y ejecutar un programa es un problema complejo. Buscar un programa destino óptimo para un programa fuente dado es un problema indecidible, y muchos de los subproblemas involucrados son NP-hard. Como hemos indicado antes, en la generación de código debemos a menudo contentarnos con las técnicas de heurística que producen buenos programas destino, pero que no necesariamente son óptimos.

Durante el resto de este capítulo, vamos a suponer que cada instrucción en lenguaje destino tiene un costo asociado. Por simplicidad, consideraremos que el costo de una instrucción será de uno más los costos asociados con los modos de direccionamiento de los operandos. Este costo corresponde a la longitud en palabras de la instrucción. Los modos de direccionamiento que implican el uso de registros tienen un costo adicional de cero, mientras que los que implican el uso de una ubicación de memoria o constante tienen un costo adicional de uno, ya que dichos operandos tienen que almacenarse en las palabras que van después de la instrucción. He aquí algunos ejemplos:

- La instrucción LD R0, R1 copia el contenido del registro R1 al registro R0. Esta instrucción tiene un costo de uno, ya que no se requieren palabras adicionales de memoria.
- La instrucción LD R0, M carga el contenido de la ubicación de memoria M al registro R0. El costo es de dos, ya que la dirección de la ubicación de memoria M está en la palabra que va después de la instrucción.
- La instrucción LD R1, *100(R2) carga al registro R1 el valor proporcionado por *contenido*(*contenido*(100 + *contenido*(R2))). El costo es de tres, debido a que la constante 100 se almacena en la palabra que va después de la instrucción.

En este capítulo vamos a suponer que el costo de un programa en lenguaje destino, dada cierta entrada, es la suma de los costos de las instrucciones individuales que se ejecutan cuando el programa se ejecuta sobre esa entrada. Los buenos algoritmos de generación de código buscan disminuir al mínimo la suma de los costos de las instrucciones ejecutadas por el programa destino que se genera sobre las entradas ordinarias. Más adelante veremos que en algunos casos, podemos realmente generar código óptimo para las expresiones en ciertas clases de máquinas de registro.

8.2.3 Ejercicios para la sección 8.2

Ejercicio 8.2.1: Genere el código para las siguientes instrucciones de tres direcciones, suponiendo que todas las variables se almacenan en ubicaciones de memoria.

- a) $x = 1$
- b) $x = a$
- c) $x = a + 1$
- d) $x = a + b$
- e) Las siguientes dos instrucciones:

```
x = b * c
y = a + x
```

Ejercicio 8.2.2: Genere el código para las siguientes instrucciones de tres direcciones, suponiendo que a y b son arreglos cuyos elementos son valores de 4 bytes.

- a) La siguiente secuencia de cuatro instrucciones:

```
x = a[i]
y = b[j]
a[i] = y
b[j] = x
```

- b) La siguiente secuencia de tres instrucciones:

```
x = a[i]
y = b[i]
z = x * y
```

- c) La siguiente secuencia de tres direcciones:

```
x = a[i]
y = b[x]
a[i] = y
```

Ejercicio 8.2.3: Genere el código para la siguiente secuencia de tres direcciones, suponiendo que p y q se encuentran en ubicaciones de memoria:

```
y = *q
q = q + 4
*p = y
p = p + 4
```

Ejercicio 8.2.4: Genere el código para la siguiente secuencia, suponiendo que x , y y z se encuentran en ubicaciones de memoria:

```
if x < y goto L1
z = 0
goto L2
L1: z = 1
```

Ejercicio 8.2.5: Genere el código para la siguiente secuencia, suponiendo que n está en una ubicación de memoria:

```
s = 0
i = 0
L1: if i > n goto L2
s = s + i
i = i + 1
goto L1
L2:
```

Ejercicio 8.2.6: Determine los costos de las siguientes secuencias de instrucciones:

- a) LD R0, y
LD R1, z
ADD R0, R0, R1
ST x, R0
- b) LD R0, i
MUL R0, R0, 8
LD R1, a(R0)
ST b, R1
- c) LD R0, c
LD R1, i
MUL R1, R1, 8
ST a(R1), R0
- d) LD R0, p
LD R1, 0(R0)
ST x, R1

- e) LD R0, p
 LD R1, x
 ST 0(R0), R1
- f) LD R0, x
 LD R1, y
 SUB R0, R0, R1
 BLTZ *R3, R0

8.3 Direcciones en el código destino

En esta sección le mostraremos cómo pueden convertirse los nombres en la representación intermedia en direcciones en el código destino, analizando la generación de código para las llamadas simples a procedimientos y sus retornos, mediante el uso de la asignación estática y de pila. En la sección 7.1 describimos la forma en que cada programa se ejecuta en su propio espacio de direcciones lógicas, que se particionaba en cuatro áreas de código y cuatro de datos:

1. Un área de *Código* determinada en forma estática, la cual contiene el código destino ejecutable. El tamaño del código destino se puede determinar en tiempo de compilación.
2. Un área de datos *Estática* determinada en forma estática, para contener constantes globales y demás datos que genera el compilador. El tamaño de las constantes globales y de los datos del compilador también puede determinarse en tiempo de compilación.
3. Un área *Montículo* administrada en forma dinámica, para contener objetos de datos que se asignan y se liberan durante la ejecución del programa. El tamaño del *Montículo* no puede determinarse en tiempo de compilación.
4. Un área *Pila* administrada en forma dinámica, para contener los registros de activación a medida que se crean y se destruyen, durante las llamadas a los procedimientos y sus retornos. Al igual que el *Montículo*, el tamaño de la *Pila* no puede determinarse en tiempo de compilación.

8.3.1 Asignación estática

Para ilustrar la generación de código para las llamadas y retornos simplificados de procedimientos, vamos a enfocarnos en las siguientes instrucciones de tres direcciones:

- **call receptor**
- **return**
- **halt**
- **action**, que es un receptáculo para otras instrucciones de tres direcciones.

El tamaño y la distribución de los registros de activación se determinan mediante el generador de código, a través de la información de los nombres almacenados en la tabla de símbolos. Primero vamos a ilustrar cómo almacenar la dirección de retorno en un registro de activación durante

la llamada a un procedimiento, y cómo devolverle el control después de la llamada. Por conveniencia, asumiremos que la primera ubicación en la activación contiene la dirección de retorno.

Primero vamos a considerar el código necesario para implementar el caso más simple, la asignación estática. Aquí, una instrucción `call receptor` en el código intermedio puede implementarse mediante una secuencia de dos instrucciones de máquina destino:

```
ST  receptor.areaEstatica, #aqui + 20
BR  receptor.areaCodigo
```

La instrucción `ST` guarda la dirección de retorno al principio del registro de activación para `receptor`, y la instrucción `BR` transfiere el control al código de destino para el procedimiento `receptor` al que se llamó. El atributo antes de `receptor.areaEstatica` es una constante que proporciona la dirección del inicio del registro de activación para `receptor`, y el atributo `receptor.areaCodigo` es una constante que se refiere a la dirección de la primera instrucción del procedimiento `receptor` al que se llamó en el área *Código* de la memoria en tiempo de ejecución.

El operando `#aqui + 20` en la instrucción `ST` es la dirección de retorno literal; es la dirección de la instrucción que sigue después de la instrucción `BR`. Suponemos que `#aqui` es la dirección de la instrucción actual, y que las tres constantes más las dos instrucciones en la secuencia de llamada tienen una longitud de 5 palabras, o 20 bytes.

El código para un procedimiento termina con un retorno al procedimiento que hizo la llamada, sólo que el primer procedimiento no tiene emisor, por lo que su instrucción final es `HALT`, la cual devuelve el control al sistema operativo. Una instrucción `return receptor` puede implementarse mediante una instrucción simple de salto:

```
BR  *receptor.areaEstatica
```

la cual transfiere el control a la dirección guardada al principio del registro de activación para `receptor`.

Ejemplo 8.3: Suponga que tenemos el siguiente código de tres direcciones:

```
// código para c
action1
call p
action2
halt
// código para p
action3
return
```

La figura 8.4 muestra el programa destino para este código de tres direcciones. Usamos la seudoinstrucción `ACTION` para representar la secuencia de instrucciones de máquina que van a ejecutar la instrucción `action`, que representa el código de tres direcciones que no es relevante para esta discusión. Empezamos de manera arbitraria el código para el procedimiento `c` en la dirección 100, y para el procedimiento `p` en la dirección 200. Suponemos que cada instrucción `ACTION` requiere 20 bytes. Además, suponemos que los registros de activación para estos procedimientos se asignación de manera estática en las ubicaciones 300 y 364, respectivamente.

Las instrucciones que empiezan en la dirección 100 implementan a las siguientes instrucciones:

```
action1; call p; action2; halt
```

del primer procedimiento *c*. Por lo tanto, la ejecución empieza con la instrucción ACTION₁ en la dirección 100. La instrucción ST en la dirección 120 guarda la dirección de retorno 140 en el campo de estado de la máquina, que es la primera palabra en el registro de activación de *p*. La instrucción BR en la dirección 132 transfiere el control a la primera instrucción en el código de destino del procedimiento *p* que se llamó.

```

          // código para c
100: ACTION1           // código para action1
120: ST 364, #140        // guarda la dirección de retorno 140 en la ubicación 364
132: BR 200              // llama a p
140: ACTION2
160: HALT                // regresa al sistema operativo
...
          // código para p
200: ACTION3
220: BR *364             // regresa a la dirección guardada en la ubicación 364
...
          // 300-363 contienen el registro de activación para c
300: 
304: 
...
          // 364-451 contienen el registro de activación para p
364: 
368: 
          // dirección de retorno
          // datos locales para c
...
          // dirección de retorno
          // datos locales para p

```

Figura 8.4: Código destino para la asignación estática

Después de ejecutar ACTION₃, se ejecuta la instrucción de salto en la ubicación 220. Como la ubicación 140 se guardó en la dirección 364 mediante la secuencia de llamada anterior, *364 representa 140 cuando se ejecuta la instrucción BR en la dirección 220. Por lo tanto, cuando termina el procedimiento *p*, el control regresa a la dirección 140 y se reanuda la ejecución del procedimiento *c*. □

8.3.2 Asignación de pila

La asignación estática se puede convertir en asignación de pila mediante el uso de direcciones relativas para el almacenamiento en los registros de activación. Sin embargo, en la asignación de pila la posición de un registro de activación para un procedimiento no se conoce sino hasta el tiempo de ejecución. Por lo general, esta posición se almacena en un registro, por lo que se puede acceder a las palabras en el registro de activación como desplazamientos a partir del valor en este registro. El modo de direccionamiento indexado para nuestra máquina destino es conveniente para este fin.

Las direcciones relativas en un registro de activación pueden tomarse como desplazamientos a partir de cualquier posición conocida en el registro de activación, como vimos en el capítulo 7.

Por conveniencia, vamos a usar desplazamientos positivos, manteniendo en un registro **SP** un apuntador al inicio del registro de activación en la parte superior de la pila. Cuando ocurre una llamada a un procedimiento, el procedimiento que llama incrementa a **SP** y transfiere el control al procedimiento al que llamó. Una vez que el control regresa al emisor, decrementamos **SP**, con lo cual se desasigna el registro de activación del procedimiento al que se llamó.

El código para el primer procedimiento inicializa la pila, estableciendo **SP** al inicio del área de la pila en la memoria:

```
LD  SP, #inicioPila          // inicializa la pila
código para el primer procedimiento
HALT                         // termina la ejecución
```

La secuencia de la llamada a un procedimiento incrementa a **SP**, guarda la dirección de retorno y transfiere el control al procedimiento al que se llamó:

```
ADD SP, SP, #emisor.tamRegistro // incrementa el apuntador de la pila
ST  *SP, #aqui + 16           // guarda la dirección de retorno
BR  receptor.areaCodigo      // regresa al emisor
```

El operando **#emisor.tamRegistro** representa el tamaño de un registro de activación, por lo que la instrucción ADD hace que **SP** apunte al siguiente registro de activación. El operando **#aqui + 16** en la instrucción ST es la dirección de la instrucción que va después de BR; se guarda en la dirección a la que apunta **SP**.

La secuencia de retorno consiste en dos partes. El procedimiento al que se llamó transfiere el control a la dirección de retorno, usando lo siguiente:

```
BR  *0(SP)                  // regresa al emisor
```

La razón de usar ***0(SP)** en la instrucción BR es que necesitamos dos niveles de indirección: **0(SP)** es la dirección de la primera palabra en el registro de activación, y ***0(SP)** es la dirección de retorno que está guardada ahí.

La segunda parte de la secuencia de retorno está en el emisor, el cual decrementa a **SP**, con lo cual **SP** se restaura a su valor anterior. Es decir, después de la resta **SP** apunta al inicio del registro de activación del emisor:

```
SUB SP, SP, #emisor.tamRegistro // decrementa el apuntador de la pila
```

El capítulo 7 contiene una explicación más amplia sobre las secuencias de las llamadas y las concesiones que se deben hacer en la división de la labor entre el procedimiento que llama y el procedimiento al que se llamó.

Ejemplo 8.4: El programa en la figura 8.5 es una abstracción del programa quicksort del capítulo anterior. El procedimiento *q* es recursivo, por lo que puede haber más de una activación de *q* viva al mismo tiempo.

Suponga que los tamaños de los registros de activación para los procedimientos *m*, *p* y *q* se han determinado como *mtam*, *ptam* y *qtam*, respectivamente. La primera palabra en cada registro de

```

// código para m
action1
call q
action2
halt
// código para p
action3
return
// código para q
action4
call p
action5
call q
action6
call q
return

```

Figura 8.5: Código para el ejemplo 8.4

activación guardará una dirección de retorno. Suponemos en forma arbitraria que el código para estos procedimientos empieza en las direcciones 100, 200 y 300, respectivamente, y que la pila empieza en la dirección 600. El programa destino se muestra en la figura 8.6.

Vamos a suponer que ACTION_4 contiene un salto condicional a la dirección 456 de la secuencia de retorno de q ; de no ser así, el procedimiento recursivo q se condene a llamarse a sí mismo por siempre.

Si $mtam$, $ptam$ y $qtam$ son 20, 40 y 60, respectivamente, la primera instrucción en la dirección 100 inicializa SP a 600, la dirección inicial de la pila. SP contiene 620 justo antes que el control se transfiera de m a q , ya que $mtam$ es 20. Más adelante, cuando q llama a p , la instrucción en la dirección 320 incrementa SP a 680, en donde empieza el registro de activación para p ; SP se regresa a 620 después de que el control regresa a q . Si las dos siguientes llamadas de q regresan de inmediato, el valor máximo de SP durante esta ejecución es de 680. Sin embargo, observe que la última ubicación de la pila que se utiliza es 739, ya que el registro de activación de q que empieza en la ubicación 680 se extiende 60 bytes. \square

8.3.3 Direcciones para los nombres en tiempo de ejecución

La estrategia de asignación de almacenamiento y la distribución de los datos locales en un registro de activación para un procedimiento son los que determinan cómo se accede al almacenamiento para los nombres. En el capítulo 6, asumimos que un nombre en una instrucción de tres direcciones es en realidad un apuntador a una entrada en la tabla de símbolos para ese nombre. Este método tiene una ventaja considerable; hace que el compilador sea más portable, ya que el front-end no tiene que modificarse, ni siquiera cuando el compilador se mueve a una máquina distinta, en donde se requiere una organización distinta en tiempo de ejecución. Por otro lado, la acción de generar la secuencia específica de pasos de acceso mientras se genera el código intermedio puede representar una ventaja considerable en un compilador optimizador,

```

100: LD SP, #600           // código para m
108: ACTION1             // inicializa la pila
128: ADD SP, SP, #mtam    // código para action1
136: ST *SP, #152          // empieza la secuencia de llamadas
144: BR 300                // mete la dirección de retorno
152: SUB SP, SP, #mtam    // llama a q
160: ACTION2             // restaura SP
180: HALT                 ...

200: ACTION3             ...
220: BR *0(SP)            // regresa
238: ...
300: ACTION4             ...
320: ADD SP, SP, #qtam    // código para q
328: ST *SP, #344          // contiene un salto condicional a 456
336: BR 200                // mete la dirección de retorno
344: SUB SP, SP, #qtam    // llama a p
352: ACTION5             ...
372: ADD SP, SP #qtam     ...
380: BR *SP, #396          // mete la dirección de retorno
388: BR 300                // llama a q
396: SUB SP, SP, #qtam    ...
404: ACTION6             ...
424: ADD SP, SP, #qtam    // mete la dirección de retorno
432: ST *SP, #440          // llama a q
440: BR 300                ...
448: SUB SP, SP, #qtam    ...
456: BR *0(SP)            // regresa
474: ...
600:                      // la pila empieza aquí

```

Figura 8.6: Código destino para la asignación de pila

ya que permite al optimizador aprovechar los detalles que no vería en la instrucción simple de tres direcciones.

En cualquier caso, los nombres deben sustituirse en algún momento dado por código para acceder a las ubicaciones de almacenamiento. Por ende, consideramos ciertas elaboraciones de la instrucción de copia simple de tres direcciones $x = 0$. Después de procesar las declaraciones en un procedimiento, suponga que la entrada en la tabla de símbolos para x contiene una dirección relativa 12 para x . Consideremos el caso en el que x se encuentra en un área asignación en forma estática, que empieza en la dirección *estatica*. Entonces, la verdadera dirección en tiempo de ejecución de x es *estatica* + 12. Aunque el compilador puede determinar en un momento dado el valor de *estatica* + 12 en tiempo de compilación, tal vez no se conozca la posición del área estática al generar el código intermedio para acceder al nombre. En ese caso, tiene sentido generar un código de tres direcciones para “calcular” *estatica* + 12, con el entendimiento de que este cálculo se llevará a cabo durante la fase de generación de código, o quizás por medio del cargador, antes de que se ejecute el programa. Así, la asignación $x = 0$ se traduce en:

```
estatica[12] = 0
```

Si el área estática empieza en la dirección 100, el código de destino para esta instrucción es:

```
LD 112, #0
```

8.3.4 Ejercicios para la sección 8.3

Ejercicio 8.3.1: Genere el código para las siguientes instrucciones de tres direcciones, suponiendo la asignación de pila en donde el registro SP apunta a la parte superior de la pila.

```
call p
call q
return
call r
return
return
```

Ejercicio 8.3.2: Genere el código para las siguientes instrucciones de tres direcciones, asumiendo la asignación de pila en donde el registro SP apunta a la parte superior de la pila.

- a) $x = 1$
- b) $x = a$
- c) $x = a + 1$
- d) $x = a + b$
- e) Las siguientes dos instrucciones:

```
x = b * c
y = a + x
```

Ejercicio 8.3.3: Genere el código para las siguientes instrucciones de tres direcciones, asumiendo de nuevo la asignación de pila y asumiendo que a y b son arreglos, cuyos elementos son valores de 4 bytes.

a) La siguiente secuencia de cuatro instrucciones:

```
x = a[i]
y = b[j]
a[i] = y
b[j] = x
```

b) La siguiente secuencia de tres instrucciones:

```
x = a[i]
y = b[i]
z = x * y
```

c) La siguiente secuencia de tres instrucciones:

```
x = a[i]
y = b[x]
a[i] = y
```

8.4 Bloques básicos y grafos de flujo

En esta sección presentaremos una representación gráfica del código intermedio, útil para hablar sobre la generación de código, incluso si el grafo no se construye en forma explícita mediante un algoritmo de generación de código. La generación de código se beneficia del contexto. Podemos realizar un mejor trabajo de asignar los registros si sabemos cómo se definen y utilizan los valores, como veremos en la sección 8.8. Podemos realizar un mejor trabajo de seleccionar instrucciones si analizamos las secuencias de las instrucciones de tres direcciones, como veremos en la sección 8.9.

La representación se construye de la siguiente forma:

1. Se partitiona el código intermedio en *bloques básicos*, los cuales son secuencias máximas de instrucciones consecutivas de tres direcciones, con las siguientes propiedades:
 - (a) El flujo de control sólo puede entrar en el bloque básico a través de la primera instrucción en el bloque. Es decir, no hay saltos hacia la parte media del bloque.
 - (b) El control saldrá del bloque sin detenerse o bifurcarse, excepto tal vez en la última instrucción en el bloque.
2. Los bloques básicos se convierten en los nodos de un *grafo de flujo*, cuyas flechas indican qué bloques pueden ir después de otros.

El efecto de las interrupciones

La noción de que el control, una vez que llega al inicio de un bloque básico, continuará sin duda hasta el final, requiere un poco de consideración. Existen muchas razones por las que una interrupción, que no se refleja en forma explícita en el código, podría hacer que el control saliera del bloque, tal vez para nunca regresar. Por ejemplo, una instrucción como $x = y/z$ parece no afectar el flujo de control, pero si z es 0, en realidad podría hacer que el programa abortara su ejecución.

No debemos preocuparnos por esas posibilidades. La razón se describe a continuación. El propósito de construir bloques básicos es para optimizar el código. En general, cuando ocurre una interrupción, ésta se maneja y el control regresa a la instrucción que produjo la interrupción, como si el control nunca se hubiera desviado, o el programa se detiene con un error. En este último caso, no importa cómo hayamos optimizado el código, aun cuando esto dependiera de que el control llegara al final del bloque básico, ya que de todas formas el programa no produjo su resultado esperado.

Desde el capítulo 9 empezaremos a hablar sobre las transformaciones en los grafos de flujo que convierten el código original intermedio en código intermedio “optimizado”, a partir del cual se puede generar un mejor código destino. El código intermedio “optimizado” se convierte en código de máquina usando las técnicas de generación código que se describen en este capítulo.

8.4.1 Bloques básicos

Nuestra primera tarea es particionar una secuencia de instrucciones de tres direcciones en bloques básicos. Empezamos un nuevo bloque básico con la primera instrucción y seguimos agregando instrucciones hasta llegar a un salto, un salto condicional o una etiqueta en la siguiente instrucción. En la ausencia de saltos y etiquetas, el control procede en forma secuencial, de una instrucción a la siguiente. Esta idea se formaliza en el siguiente algoritmo.

Algoritmo 8.5: Partitionar instrucciones de tres direcciones en bloques básicos.

ENTRADA: Una secuencia de instrucciones de tres direcciones.

SALIDA: Una lista de los bloques básicos para la secuencia en la que cada instrucción se asigna exactamente a un bloque básico.

MÉTODO: En primer lugar, determinamos las instrucciones en el código intermedio que son *líderes*; es decir, las primeras instrucciones en algún bloque básico. La instrucción que va justo después del programa intermedio no se incluye como líder. Las reglas para buscar líderes son:

1. La primera instrucción de tres direcciones en el código intermedio es líder.

2. Cualquier instrucción que sea el destino de un salto condicional o incondicional es líder.
3. Cualquier instrucción que siga justo después de un salto condicional o incondicional es líder.

Así, para cada instrucción líder, su bloque básico consiste en sí misma y en todas las instrucciones hasta, pero sin incluir a la siguiente instrucción líder o el final del programa intermedio. \square

```

1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)

```

Figura 8.7: Código intermedio para establecer una matriz de 10×10 a una matriz de identidad

Ejemplo 8.6: El código intermedio en la figura 8.7 convierte una matriz de 10×10 a en una matriz de identidad. Aunque no importa de dónde proviene este código, podría ser la traducción del seudocódigo de la figura 8.8. Para generar el código intermedio, hemos asumido que los elementos del arreglo con valores reales ocupan 8 bytes cada uno, y que la matriz a se almacena en forma de orden por filas.

```

for i de 1 a 10 do
    for j de 1 a 10 do
        a[i, j] = 0.0;
    for i de 1 a 10 do
        a[i, j] = 1.0;

```

Figura 8.8: Código fuente para la figura 8.7

En primer lugar, la instrucción 1 es líder según la regla (1) del Algoritmo 8.5. Para encontrar las otras instrucciones líderes, primero debemos encontrar los saltos. En este ejemplo hay tres saltos, todos condicionales, en las instrucciones 9, 11 y 17. Según la regla (2), los destinos de estos saltos son instrucciones líderes; éstas vienen siendo las instrucciones 3, 2 y 13, respectivamente. Ahora, según la regla (3), cada instrucción que va después de un salto es líder; éstas son las instrucciones 10 y 12. Observe que no hay ninguna instrucción después de la 17 en este código, pero si hubiera más código, la instrucción 18 sería también líder.

Concluimos que las instrucciones líderes son 1, 2, 3, 10, 12 y 13. El bloque básico de cada líder contiene todas las instrucciones a partir de esa instrucción líder, y hasta justo antes de la siguiente instrucción líder. Por ende, el bloque básico de 1 sólo contiene la instrucción 1, para la instrucción líder 2 el bloque sólo contiene la instrucción 2. Sin embargo, la instrucción líder 3 tiene un bloque básico que consiste en las instrucciones de la 3 a la 9, inclusive. El bloque de la instrucción 10 consiste en las instrucciones 10 y 11; el bloque de la instrucción 12 sólo contiene a la instrucción 12, y el bloque de la instrucción 13 contiene las instrucciones de la 13 a la 17. \square

8.4.2 Información de siguiente uso

Es esencial saber cuándo se usará el valor de una variable para generar buen código. Si el valor de una variable que se encuentra en un registro nunca se utilizará más adelante, entonces ese registro puede asignarse a otra variable.

El *uso* de un nombre en una instrucción de tres direcciones se define a continuación. Supongamos que la instrucción de tres direcciones i asigna un valor a x . Si la instrucción j tiene a x como operando, y el control puede fluir de la instrucción i hacia j a través de una ruta que no tiene instrucciones en las que intervenga x , entonces decimos que la instrucción j *utiliza* el valor de x calculado en la instrucción i . Además, decimos que x está *viva* en la instrucción i .

Deseamos determinar para la instrucción de tres direcciones $x = y + z$ cuáles van a ser los siguientes usos de x , y y z . Por el momento, no nos preocuparemos por los usos fuera del bloque básico que contiene esta instrucción de tres direcciones.

Nuestro algoritmo para determinar la información sobre la vida y el siguiente uso realiza una pasada invertida sobre cada bloque básico. Almacenamos la información en la tabla de símbolos. Podemos explorar con facilidad un flujo de instrucciones de tres direcciones para encontrar el final de los bloques básicos, como en el Algoritmo 8.5. Ya que los procedimientos pueden tener efectos adicionales, asumimos por conveniencia que cada llamada a un procedimiento empieza un nuevo bloque básico.

Algoritmo 8.7: Determinar la información sobre la vida y el uso siguiente para cada instrucción en un bloque básico.

ENTRADA: Un bloque básico B de instrucciones de tres direcciones. Suponemos que, al principio, la tabla de símbolos muestra que todas las variables no temporales en B están vivas al salir.

SALIDA: En cada instrucción i : $x = y + z$ en B , adjuntamos a i la información sobre la vida y el siguiente uso de x , y y z .

MÉTODO: Empezamos en la última instrucción en B , y exploramos en forma invertida hasta el inicio de B . En cada instrucción i : $x = y + z$ en B , hacemos lo siguiente:

1. Adjuntar a la instrucción i la información que se encuentra en ese momento en la tabla de símbolos, en relación con el siguiente uso y la vida de x , y y z .
2. En la tabla de símbolos, establecer x a “no viva” y “sin siguiente uso”.
3. En la tabla de símbolos, establecer y y z a “viva” y los siguientes usos de y y z a i .

Aquí hemos usado $a +$ como un símbolo que representa a cualquier operador. Si la instrucción de tres direcciones i es de la forma $x = + y$ o $x = y$, los pasos son los mismos, sólo que ignoramos z . Observe que el orden de los pasos (2) y (3) no puede intercambiarse, ya que x podría ser y o z . \square

8.4.3 Grafos de flujo

Una vez que un programa de código intermedio se partitiona en bloques básicos, representamos el flujo de control entre ellos mediante un grafo de flujo. Los nodos del grafo de flujo son los nodos básicos. Hay una flecha del bloque B al bloque C sí, y sólo si es posible que la primera instrucción en el bloque C vaya justo después de la última instrucción en el bloque B . Hay dos formas en las que podría justificarse dicha flecha:

- Hay un salto condicional o incondicional desde el final de B hasta el inicio de C .
- C sigue justo después de B en el orden original de las instrucciones de tres direcciones, y B no termina en un salto incondicional.

Decimos que B es un *predecesor* de C , y que C es un *sucesor* de B .

A menudo agregamos dos nodos, conocidos como *entrada* y *salida*, los cuales no corresponden a instrucciones intermedias ejecutables. Hay una flecha que va desde la entrada hasta el primer nodo ejecutable del grafo de flujo; es decir, al bloque básico que surge de la primera instrucción del código intermedio. Hay una flecha que va a la salida desde cualquier bloque básico que contenga una instrucción, que pudiera ser la última instrucción ejecutada del programa. Si la instrucción final del programa no es un salto incondicional, entonces el bloque que contiene la instrucción final del programa es un predecesor de la salida, pero también lo es cualquier bloque básico que tenga un salto hacia código que no forme parte del programa.

Ejemplo 8.8: El conjunto de bloques básicos construido en el ejemplo 8.6 produce el grafo de flujo de la figura 8.9. La entrada apunta al bloque básico B_1 , ya que éste contiene la primera instrucción del programa. El único sucesor de B_1 es B_2 , debido a que B_1 no termina en un salto incondicional y la instrucción líder de B_2 sigue justo después del final de B_1 .

El bloque B_3 tiene dos sucesores. Uno es el mismo B_3 , ya que la instrucción líder de B_3 (la instrucción 3) es el destino del salto condicional al final de B_3 , en la instrucción 9. El otro sucesor es B_4 , ya que el control puede pasar a través del salto condicional al final de B_3 y entrar a continuación a la instrucción líder de B_4 .

Sólo B_6 apunta a la salida del grafo de flujo, ya que la única manera de ir al código que sigue después del programa a partir del cual construimos el grafo de flujo es pasar a través del salto condicional que termina el bloque B_6 . \square

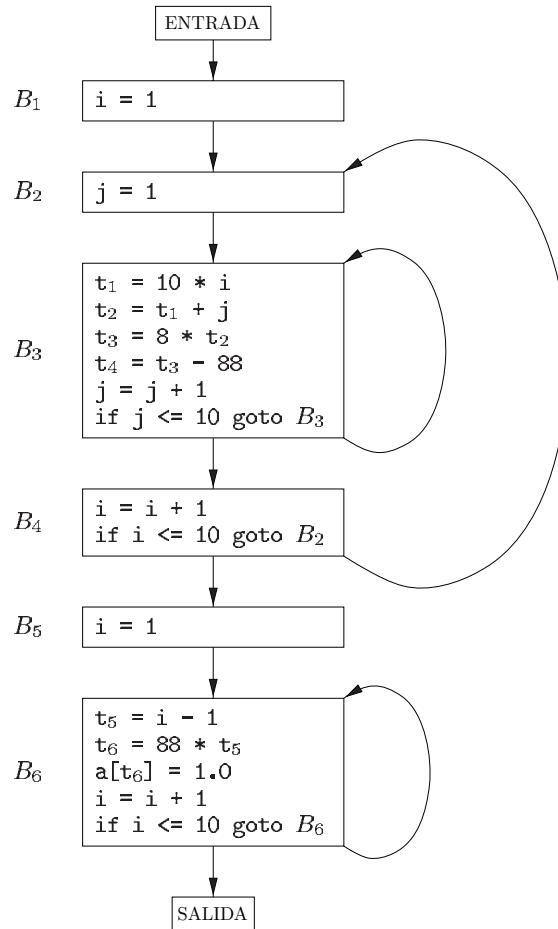


Figura 8.9: Grafo de flujo de la figura 8.7

8.4.4 Representación de los grafos de flujo

En primer lugar, observe en la figura 8.9 que en el grafo de flujo, es normal sustituir los saltos hacia números de instrucción o etiquetas, por saltos hacia bloques básicos. Recuerde que todo salto condicional o incondicional va hacia la instrucción líder de algún bloque básico, y que el salto ahora hará referencia a este bloque. La razón de este cambio es que después de construir el grafo de flujo, es común realizar cambios considerables a las instrucciones en los diversos bloques básicos. Si los saltos fueran a instrucciones, tendríamos que corregir los destinos de éstos cada vez que se modificara una de las instrucciones de destino.

Siendo grafos bastante ordinarios, los grafos de flujo pueden representarse mediante una de las estructuras de datos apropiadas para grafos. El contenido de los nodos (bloques básicos) necesita su propia representación. Podríamos representar el contenido de un nodo mediante un

apuntador a la instrucción líder en el arreglo de instrucciones de tres direcciones, junto con un conteo del número de instrucciones, o un segundo apuntador a la última instrucción. Sin embargo, como tal vez cambiemos el número de instrucciones en un bloque básico con frecuencia, es probable que sea más eficiente crear una lista enlazada de instrucciones para cada bloque básico.

8.4.5 Ciclos

Las construcciones de los lenguajes de programación, como las instrucciones while, do-while y for, ocasionan de manera natural el surgimiento de ciclos en los programas. Como casi cualquier programa invierte la mayor parte de su tiempo en ejecutar sus ciclos, es muy importante para un compilador generar un buen código para los ciclos. Muchas transformaciones de código dependen de la identificación de los “ciclos” en un grafo de flujo. Decimos que un conjunto de nodos L en un grafo de flujo es un *ciclo* si:

1. Hay un nodo en L llamado *entrada al ciclo*, con la propiedad de que ningún otro nodo en L tiene un predecesor fuera de L . Es decir, cada ruta desde la entrada de todo el grafo de flujo completo, hacia cualquier nodo en L , pasa a través de la entrada al ciclo.
2. Cada nodo en L tiene una ruta que no está vacía, completamente dentro de L , que va hacia la entrada de L .

Ejemplo 8.9: El grafo de flujo de la figura 8.9 tiene tres ciclos:

1. B_3 por sí solo.
2. B_6 por sí solo.
3. $\{B_2, B_3, B_4\}$.

Los primeros dos son nodos individuales con una flecha que va al mismo nodo. Por ejemplo, B_3 forma un ciclo con B_3 como su entrada. Observe que el segundo requerimiento para un ciclo es que haya una ruta que no esté vacía, desde B_3 hacia sí mismo. Por ende, un nodo individual como B_2 , que no tiene una flecha $B_2 \rightarrow B_2$, no es un ciclo, ya que no hay una ruta no vacía desde B_2 hacia sí mismo dentro de $\{B_2\}$.

El tercer ciclo, $L = \{B_2, B_3, B_4\}$, tiene a B_2 como su entrada al ciclo. Observe que entre estos tres nodos, sólo B_2 tiene un predecesor, B_1 , que no está en L . Además, cada uno de los tres nodos tiene una ruta no vacía hacia B_2 , que permanece dentro de L . Por ejemplo, B_2 tiene la ruta $B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2$. \square

8.4.6 Ejercicios para la sección 8.4

Ejercicio 8.4.1: La figura 8.10 es un programa simple de multiplicación de matrices.

- a) Traduzca el programa en instrucciones de tres direcciones, del tipo que hemos estado usando en esta sección. Suponga que las entradas de las matrices son números que requieren 8 bytes, y que las matrices se almacenan en orden por filas.

- b) Construya el grafo de flujo para su código a partir de (a).
- c) Identifique los ciclos en su grafo de flujo de (b).

```

for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        c[i][j] = 0.0;
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        for (k=0; k<n; k++)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];

```

Figura 8.10: Un algoritmo de multiplicación de matrices

Ejercicio 8.4.2: La figura 8.11 es código para contar los números primos desde 2 hasta n , usando el método de criba en un arreglo a con un tamaño adecuado. Es decir, $a[i]$ es TRUE al final sólo si no hay un número primo \sqrt{i} o menor, que se divide de manera uniforme entre i . Inicializamos todos los valores de $a[i]$ a TRUE y después establecemos $a[j]$ a FALSE si encontramos un divisor de j .

- a) Traduzca el programa en instrucciones de tres direcciones, del tipo que hemos estado utilizando en esta sección. Asuma que los enteros requieren 4 bytes.
- b) Construya el grafo de flujo para su código de (a).
- c) Identifique los ciclos en su grafo de flujo de (b).

```

for (i=2; i<=n; i++)
    a[i] = TRUE;
cuenta = 0;
s = sqrt(n);
for (i=2; i<=s; i++)
    if (a[i]) /* se encontró que i es un número primo */ {
        cuenta++;
        for (j=2*i; j<=n; j = j+i)
            a[j] = FALSE; /* ningún múltiplo de i es primo */
    }

```

Figura 8.11: Código para cribar en busca de números primos

8.5 Optimización de los bloques básicos

A menudo podemos obtener una considerable mejora sobre el tiempo de ejecución del código, con sólo realizar una optimización *local* dentro de cada bloque básico por sí mismo. En capítulos posteriores, empezando desde el capítulo 9, cubriremos la optimización *global* con más detalle, en la que se analiza la forma en que fluye la información a través de los bloques básicos de un programa. Es un tema complejo, en el cual hay que considerar muchas técnicas distintas.

8.5.1 La representación en GDA de los bloques básicos

Muchas técnicas importantes para la optimización local empiezan por transformar un bloque básico en un GDA (grafo dirigido acíclico; DAG, en sus siglas en inglés). En la sección 6.1.1 presentamos el GDA como una representación para las expresiones simples. La idea se extiende en forma natural hasta la colección de expresiones que se crean dentro de un bloque básico. Construimos un GDA para bloques básicos de la siguiente manera:

1. Hay un nodo en el GDA para cada uno de los valores iniciales de las variables que aparecen en el bloque básico.
2. Hay un nodo N asociado con cada instrucción s dentro del bloque. Los hijos de N son aquellos nodos que corresponden a las instrucciones que son las últimas definiciones, anteriores a s , de los operandos utilizados por s .
3. El nodo N se etiqueta mediante el operador que se aplica en s , y también a N se adjunta la lista de variables para las cuales es la última definición dentro del bloque.
4. Ciertos nodos se designan como *nodos de salida*. Éstos son los nodos cuyas variables están *vivas al salir* del bloque; es decir, sus valores pueden utilizarse después, en otro bloque del grafo de flujo. El cálculo de estas “variables vivas” corresponde al análisis de flujo global, que veremos en la sección 9.2.5.

La representación en GDA de un bloque básico nos permite realizar varias transformaciones para mejora de código, en el código representado por el bloque.

- a) Podemos eliminar las *subexpresiones locales comunes*; es decir, las instrucciones que calculan un valor que ya se ha calculado.
- b) Podemos eliminar el *código muerto*; es decir, las instrucciones que calculan un valor que nunca se utiliza.
- c) Podemos reordenar las instrucciones que no dependen unas de otras; dicho reordenamiento puede reducir el tiempo que requiere un valor temporal para preservarse en un registro.
- d) Podemos aplicar leyes algebraicas para reordenar los operandos de las instrucciones de tres direcciones, lo cual algunas veces simplifica los cálculos.

8.5.2 Búsqueda de subexpresiones locales comunes

Para detectar las subexpresiones comunes hay que observar, en el momento en el que un nuevo nodo M está a punto de agregarse, si hay un nodo N existente con los mismos hijos, en el mismo orden y con el mismo operador. De ser así, N calcula el mismo valor que M y puede usarse en su lugar. Esta técnica se introdujo como el método “número de valor” para detectar subexpresiones comunes en la sección 6.1.1.

Ejemplo 8.10: En la figura 8.12 se muestra un GDA para el siguiente bloque:

$$\begin{aligned} a &= b + c \\ b &= a - d \\ c &= b + c \\ d &= a - d \end{aligned}$$

Al construir el nodo para la tercera instrucción $c = b + c$, sabemos que el uso de b en $b + c$ se refiere al nodo de la figura 8.12 etiquetado como $-$, ya que es la definición más reciente de b . Por ende, no confundimos los valores calculados en las instrucciones uno y tres.

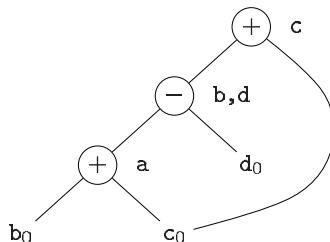


Figura 8.12: GDA para el bloque básico en el ejemplo 8.10

Sin embargo, el nodo que corresponde a la cuarta instrucción $d = a - d$ tiene el operador $-$ y los nodos con las variables adjuntas a y d_0 como hijos. Como el operador y los hijos son los mismos que para el nodo correspondiente a la instrucción dos, no creamos este nodo, sino que agregamos d a la lista de definiciones para el nodo etiquetado como $-$. \square

Podría parecer que, como sólo hay tres nodos que no son hojas en el GDA de la figura 8.12, el bloque básico en el ejemplo 8.10 puede sustituirse mediante un bloque con sólo tres instrucciones. De hecho, si b no está viva al salir del bloque, entonces no necesitamos calcular esa variable, y podemos usar d para recibir el valor representado por el nodo etiquetado como $-$ en la figura 8.12. Así, el bloque se convierte en:

$$\begin{aligned} a &= b + c \\ d &= a - d \\ c &= d + c \end{aligned}$$

No obstante, si tanto b como d están vivas al salir, entonces debemos usar una cuarta instrucción para copiar el valor de una a la otra.¹

Ejemplo 8.11: Cuando buscamos subexpresiones comunes, en realidad buscamos expresiones en las que se garantiza que calculen el mismo valor, sin importar cómo se calcule ese valor. Por ende, el método del GDA ignorará el hecho de que la expresión calculada por las instrucciones primera y cuarta en la secuencia

$$\begin{aligned} a &= b + c; \\ b &= b - d \\ c &= c + d \\ e &= b + c \end{aligned}$$

es la misma, a saber $b_0 + c_0$. Es decir, aun cuando b y c cambian entre la primera y la última instrucción, su suma permanece igual, ya que $b + c = (b - d) + (c + d)$. El GDA para esta secuencia se muestra en la figura 8.13, pero no exhibe subexpresiones comunes. Sin embargo, las identidades algebraicas que se aplican al GDA, como vimos en la sección 8.5.4, pueden exponer esta equivalencia. \square

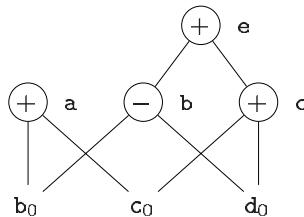


Figura 8.13: GDA para el bloque básico en el ejemplo 8.11

8.5.3 Eliminación del código muerto

La operación sobre GDAs que corresponde a la eliminación de código muerto se puede implementar de la siguiente forma. Eliminamos de un GDA cualquier raíz (nodo sin ancestros) que no tenga variables vivas adjuntas. La aplicación repetida de esta transformación eliminará todos los nodos del GDA que corresponden al código muerto.

Ejemplo 8.12: Si en la figura 8.13, a y b están vivas pero c y e no, podemos eliminar de inmediato la raíz etiquetada como e . Así, el nodo etiquetado como c se convierte en una raíz y puede eliminarse. Quedan las raíces etiquetadas como a y b , ya que cada una tiene variables vivas adjuntas. \square

¹En general, debemos tener cuidado, al reconstruir código de los GDAs, de la forma en que elegimos los nombres de las variables. Si una variable x se define dos veces, o si se asigna una vez y se utiliza también el valor inicial x_0 , entonces debemos asegurarnos de no modificar el valor de x sino hasta que hayamos efectuado todos los usos del nodo cuyo valor x contenía anteriormente.

8.5.4 El uso de identidades algebraicas

Las identidades algebraicas representan otra clase importante de optimizaciones en los bloques básicos. Por ejemplo, podemos aplicar identidades aritméticas, como:

$$\begin{array}{ll} x + 0 = 0 + x = x & x - 0 = x \\ x \times 1 = 1 \times x = x & x/1 = x \end{array}$$

para eliminar los cálculos de un bloque básico.

Otra clase de optimizaciones algebraicas incluye la *reducción por fuerza* local; es decir, sustituir un operador más costoso por uno más económico, como en:

COSTOSO		ECONÓMICO
x^2	=	$x \times x$
$2 \times x$	=	$x + x$
$x/2$	=	$x \times 0.5$

El *plegado de constantes* es una tercera clase de optimizaciones relacionadas. Aquí evaluamos las expresiones constantes en tiempo de compilación y las sustituimos por sus valores.² Por lo tanto, la expresión $2 * 3.14$ se sustituiría por 6.28 . Muchas expresiones constantes surgen en la práctica debido al uso frecuente de las constantes simbólicas en los programas.

El proceso de construcción de un GDA puede ayudarnos a aplicar estas y otras transformaciones algebraicas más generales, como la commutatividad y la asociatividad. Por ejemplo, suponga que el manual de referencia del lenguaje especifica que $*$ es commutativo; es decir, $x * y = y * x$. Antes de crear un nuevo nodo etiquetado como $*$, con un hijo izquierdo M y un hijo derecho N , siempre verificamos que ya exista un nodo así. No obstante, como $*$ es commutativo, entonces debemos verificar que haya un nodo con el operador $*$, un hijo izquierdo N y un hijo derecho M .

Algunas veces, los operadores relacionales como $< y$ generan subexpresiones comunes inesperadas. Por ejemplo, podemos probar también la condición $x > y$ si restamos los argumentos y realizamos una evaluación sobre el conjunto de códigos de condición mediante la resta.³ Por ende, tal vez sólo haya que generar un nodo del GDA para $x - y$ para $x > y$.

Las leyes asociativas también podrían aplicarse para exponer subexpresiones comunes. Por ejemplo, si el código fuente tiene las siguientes asignaciones:

$$\begin{array}{l} a = b + c; \\ e = c + d + b; \end{array}$$

podría generarse el siguiente código intermedio:

²Las expresiones aritméticas deberían evaluarse de la misma forma en tiempo de compilación que en tiempo de ejecución. K. Thompson sugirió una solución elegante al plegado de constantes: compilar la expresión constante, ejecutar el código de destino ahí mismo y sustituir la expresión con el resultado. Por ende, el compilador no necesita contener un intérprete.

³Sin embargo, la resta puede introducir desbordamientos y subdesbordamientos, mientras que una instrucción de comparación no lo haría.

```

a = b + c
t = c + d
e = t + b

```

Si t no se necesita fuera de este bloque, podemos modificar esta secuencia así:

```

a = b + c
e = a + d

```

utilizando tanto la ley asociativa como la conmutativa de $+$.

El desarrollador de compiladores debería examinar con cuidado el manual de referencia del lenguaje, para determinar qué reordenamientos de cálculos se permiten, ya que (debido a los posibles desbordamientos o subdesbordamientos) la aritmética de computadora no siempre obedece a las identidades algebráicas de las matemáticas. Por ejemplo, el estándar de Fortran establece que un compilador puede evaluar cualquier expresión de manera matemática equivalente, siempre y cuando no se viole la integridad de los paréntesis. Así, un compilador puede evaluar $x * y - x * z$ como $x * (y - z)$, pero no puede evaluar $a + (b - c)$ como $(a + b) - c$. Por lo tanto, un compilador de Fortran debe mantener el registro de la ubicación de los paréntesis en las expresiones del lenguaje fuente, si va a optimizar los programas de acuerdo con la definición del lenguaje.

8.5.5 Representación de referencias a arreglos

A primera vista, podría parecer que las instrucciones para indexar arreglos pueden tratarse como cualquier otro operador. Por ejemplo, considere la siguiente secuencia de instrucciones de tres direcciones:

```

x = a[i]
a[j] = y
z = a[i]

```

Si consideramos $a[i]$ como una operación en la que se involucran a e i , similar a $a + i$, entonces podría parecer que los dos usos de $a[i]$ son una subexpresión común. En este caso, podríamos vernos tentados a “optimizar” mediante la sustitución de la tercera instrucción $z = a[i]$ por una más simple, $z = x$. Sin embargo, como j podría ser igual a i , la instrucción de en medio puede de hecho modificar el valor de $a[i]$; por ende, no es legal realizar esta modificación.

La manera apropiada de representar los accesos a arreglos en un GDA es la siguiente:

1. Una asignación de un arreglo como $x = a[i]$ se representa mediante la creación de un nodo con el operador $=[]$ y dos hijos, los cuales representan el valor inicial del arreglo, en este caso a_0 , y el índice i . La variable x se convierte en una etiqueta de este nuevo nodo.
2. Una asignación a un arreglo, como $a[j] = y$, se representa mediante un nuevo nodo con el operador $[]=$ y tres hijos que representan a a_0 , j y y . No hay una variable para etiquetar

este nodo. La diferencia es que la creación de este nodo *elimina* a todos los nodos actuales construidos, cuyo valor depende de a_0 . Un nodo que se ha eliminado no puede recibir más etiquetas; es decir, no puede convertirse en una expresión común.

Ejemplo 8.13: El GDA para el siguiente bloque básico:

$$\begin{aligned} x &= a[i] \\ a[j] &= y \\ z &= a[i] \end{aligned}$$

se muestra en la figura 8.14. Primero se crea el nodo N para x , pero cuando se crea el nodo etiquetado como $[] =$, N se elimina. Así, cuando se crea el nodo para z , no puede identificarse con N y en su lugar debe crearse un nuevo nodo con los mismos operandos a_0 e i_0 . \square

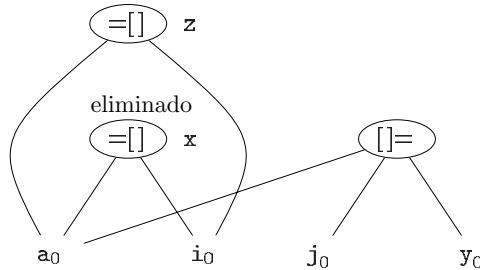


Figura 8.14: El GDA para una secuencia de asignaciones de arreglos

Ejemplo 8.14: Algunas veces, debe eliminarse un nodo aun cuando ninguno de sus hijos tenga un arreglo como a_0 en el ejemplo 8.13, como una variable adjunta. De igual forma, un nodo puede eliminar si tiene un descendiente que sea un arreglo, aun cuando ninguno de sus hijos sean nodos de arreglos. Por ejemplo, considere el siguiente código de tres direcciones:

$$\begin{aligned} b &= 12 + a \\ x &= b[i] \\ b[j] &= y \end{aligned}$$

Lo que está ocurriendo aquí es que, por razones de eficiencia, b se ha definido para ser una posición en un arreglo a . Por ejemplo, si los elementos de a son de cuatro bytes, entonces b representa el cuarto elemento de a . Si j e i representan el mismo valor, entonces $b[i]$ y $b[j]$ representan la misma ubicación. Por lo tanto, es importante hacer que la tercera instrucción, $b[j] = y$, elimine el nodo con x como su variable adjunta. Sin embargo y como veremos en la figura 8.15, tanto el nodo eliminado como el nodo que realiza la eliminación tienen a a_0 como nieto, no como hijo. \square

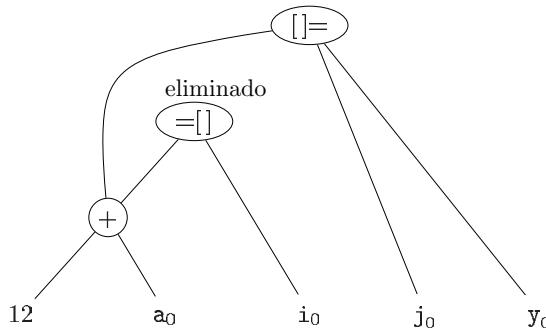


Figura 8.15: Un nodo que elimina el uso de un arreglo, no tiene que tener ese arreglo como hijo

8.5.6 Asignaciones de apuntadores y llamadas a procedimientos

Cuando realizamos una asignación indirecta a través de un apuntador, como en las siguientes asignaciones:

$$\begin{aligned} x &= *p \\ *q &= y \end{aligned}$$

no sabemos a dónde apuntan p o q . En realidad, $x = *p$ es un uso de cualquier variable, y $*q = y$ es una posible asignación a cualquier variable. Como consecuencia, el operador $=*$ debe aceptar todos los nodos que se encuentran asociados con identificadores como argumentos, lo cual es relevante para la eliminación de código muerto. Lo más importante es que el operador $==$ elimina a todos los demás nodos que se han construido hasta ahora en el GDA.

Existen análisis de apuntadores globales que podríamos realizar para limitar el conjunto de variables que un apuntador pudiera referenciar en una posición dada en el código. Inclusive hasta el análisis local podría restringir el alcance de un apuntador. Por ejemplo, en la siguiente secuencia:

$$\begin{aligned} p &= \&x \\ *p &= y \end{aligned}$$

sabemos que sólo la variable x recibe el valor de y , por lo que no necesitamos eliminar ningún nodo, sino el nodo al cual se adjunta x .

Las llamadas a procedimientos se comportan en forma muy parecida a las asignaciones a través de los apuntadores. En la ausencia de la información de flujo de datos global, debemos asumir que un procedimiento utiliza e intercambia los datos a los que tiene acceso. Por ende, si la variable x está en el alcance de un procedimiento P , una llamada a P utiliza el nodo con la variable adjunta x y elimina ese nodo.

8.5.7 Reensamblado de los bloques básicos a partir de los GDAs

Después de realizar todas las optimizaciones posibles mientras construimos el GDA, o mediante la manipulación del GDA una vez construido, podemos reconstituir el código de tres direcciones

para el bloque básico a partir del cual creamos el GDA. Para cada nodo que tenga una o más variables adjuntas, construimos una instrucción de tres direcciones que calcula el valor de una de estas variables. Preferimos calcular el resultado en una variable que esté viva al salir del bloque. No obstante, si no tenemos información sobre las variables globales vivas en la cual podemos trabajar, debemos suponer que todas las variables del programa (pero no las temporales que genera el compilador para procesar expresiones) están vivas al salir del bloque.

Si el nodo tiene más de una variable viva adjunta, entonces tenemos que introducir instrucciones de copia para dar el valor correcto a cada una de esas variables. Algunas veces la eliminación global puede eliminar esas copias, si podemos arreglárnoslas para usar una de dos variables en vez de la otra.

Ejemplo 8.15: Recuerde el GDA de la figura 8.12. En la explicación que sigue del ejemplo 8.10, decidimos que si b no está viva al salir del bloque, entonces las siguientes tres instrucciones:

$$\begin{aligned} a &= b + c \\ d &= a - d \\ c &= d + c \end{aligned}$$

bastan para reconstruir el bloque básico. La tercera instrucción, $c = d + c$, debe usar d como un operando en vez de b , ya que el bloque optimizado nunca calcula b .

Si tanto b como d están vivas al salir, o si no estamos seguros de si están vivas o no al salir, entonces debemos calcular b al igual que d . Para ello, utilizamos la siguiente secuencia:

$$\begin{aligned} a &= b + c \\ d &= a - d \\ b &= d \\ c &= d + c \end{aligned}$$

Este bloque básico es aún más eficiente que el original. Aunque el número de instrucciones es igual, hemos sustituido una resta por una copia, la cual tiende a ser menos costosa en la mayoría de las máquinas. Además, si realizamos un análisis global, tal vez podamos eliminar el uso de este cálculo de b fuera del bloque, sustituyéndolo por usos de d . En ese caso, podemos regresar a este bloque básico y eliminar $b = d$ después. Por intuición, podemos eliminar esta copia si en cualquier lugar en el que se utilice este valor de b , d todavía contiene el mismo valor. Esta situación puede ser verdadera o no, dependiendo de la forma en que el programa vuelva a calcular a d . \square

Al reconstruir el bloque básico a partir de un GDA, no sólo tenemos que preocuparnos acerca de qué variables se utilizan para contener los valores de los nodos del GDA; sino también acerca del orden en el que listamos las instrucciones que calculan los valores de los diversos nodos. Las reglas a recordar son:

1. El orden de las instrucciones debe respetar el orden de los nodos en el GDA. Es decir, no podemos calcular el valor de un nodo, sino hasta que hayamos calculado un valor para cada uno de sus hijos.

2. Las asignaciones a un arreglo deben ir después de todas las asignaciones previas a, o evaluaciones de, el mismo arreglo, de acuerdo con el orden de estas instrucciones en el bloque básico original.
3. Las evaluaciones de los elementos de los arreglos deben ir después de cualquier asignación previa (de acuerdo con el bloque original) al mismo arreglo. La única permutación permitida es que dos evaluaciones del mismo arreglo pueden realizarse en cualquier orden, siempre y cuando ninguna se cruce sobre una asignación a ese arreglo.
4. Cualquier uso de una variable debe ir después de todas las llamadas a procedimientos previas (de acuerdo con el bloque original) o asignaciones indirectas a través de un apuntador.
5. Cualquier llamada a un procedimiento o asignación indirecta a través de un apuntador debe ir después de todas las evaluaciones previas (de acuerdo con el bloque original) de cualquier variable.

Es decir, al reordenar el código, ninguna instrucción puede cruzar una llamada a un procedimiento o asignación a través de un apuntador, y los usos del mismo arreglo pueden cruzarse entre sí, sólo si ambos son accesos a arreglos, pero no asignaciones a elementos del arreglo.

8.5.8 Ejercicios para la sección 8.5

Ejercicio 8.5.1: Construya el GDA para el siguiente bloque básico:

```

d = b * c
e = a + b
b = b * c
a = e - d

```

Ejercicio 8.5.2: Simplifique el código de tres direcciones del ejercicio 8.5.1, suponiendo lo siguiente:

- a) Sólo *a* está viva al salir del bloque.
- b) *a*, *b* y *c* están vivas al salir del bloque.

Ejercicio 8.5.3: Construya el bloque básico para el código en el bloque B_6 de la figura 8.9. No olvide incluir la comparación $i \leq 10$.

Ejercicio 8.5.4: Construya el bloque básico para el código en el bloque B_3 de la figura 8.9.

Ejercicio 8.5.5: Extienda el Algoritmo 8.7 para procesar tres instrucciones de la forma:

- a) $a[i] = b$
- b) $a = b[i]$
- c) $a = *b$
- d) $*a = b$

Ejercicio 8.5.6: Construya el GDA para el siguiente bloque básico:

```

a[i] = b
*p = c
d = a[j]
e = *p
*p = a[i]

```

suponiendo que:

- p puede apuntar a cualquier parte.
- p sólo puede apuntar a b o a d.

! Ejercicio 8.5.7: Si un apuntador o expresión de arreglo, como $a[i]$ o $*p$, se asigna y después se utiliza, sin la posibilidad de modificarse en el intervalo de instrucciones, podemos aprovechar la situación para simplificar el GDA. Por ejemplo, en el código del ejercicio 8.5.6, como p no se asigna entre la segunda y la cuarta instrucción, la instrucción $e = *p$ puede sustituirse por $e = c$, sin importar hacia dónde apunta p. Modifique el algoritmo de construcción de GDAs para aprovechar dichas situaciones, y aplique su algoritmo al código del ejemplo 8.5.6.

Ejercicio 8.5.8: Suponga que se forma un bloque básico a partir de las siguientes instrucciones en C:

```

y = a + b + c + d + e + f;
y = a + c + e;

```

- Proporcione las instrucciones de tres direcciones (sólo una suma por instrucción) para este bloque.
- Use las leyes asociativa y conmutativa para modificar el bloque y utilizar el menor número posible de instrucciones, suponiendo que tanto x como y están vivas al salir del bloque.

8.6 Un generador de código simple

En esta sección vamos a considerar un algoritmo que genera código para un solo bloque básico. Considera una instrucción de tres direcciones a la vez, y lleva la cuenta de qué valores se encuentran en qué registros, para poder evitar la generación de operaciones de carga y almacenamiento innecesarias.

Una de las cuestiones principales durante la generación de código es decidir cómo usar los registros para sacarles el máximo provecho. Hay cuatro usos principales de los registros:

- En la mayoría de las arquitecturas de máquinas, algunos o todos los operandos de una operación deben estar en registros, para poder llevarla a cabo.
- Los registros son excelentes variables temporales: lugares para guardar el resultado de una subexpresión mientras se evalúa una expresión más grande o, dicho en forma más general, un lugar para guardar una variable que se utiliza sólo dentro de un bloque básico individual.

- Los registros se utilizan para guardar valores (*globales*) que se calculan en un bloque básico y se utilizan en otros bloques; por ejemplo, un índice de ciclo que se incrementa al pasar por el ciclo y se utiliza varias veces dentro del mismo.
- A menudo, los registros se utilizan para ayudar con la administración del almacenamiento en tiempo de ejecución, por ejemplo, para administrar la pila en tiempo de ejecución, incluyendo el mantenimiento de los apuntadores de la pila y posiblemente los elementos superiores de ésta.

Éstas son necesidades en competencia, ya que el número de registros disponibles es limitado.

El algoritmo en esta sección supone que hay cierto conjunto de registros disponible para guardar los valores que se utilizan dentro del bloque. Por lo general, este conjunto de registros no incluye a todos los registros de la máquina, ya que algunos registros se reservan para las variables globales y para administrar la pila. Suponemos que el bloque básico ya se ha transformado en una secuencia preferida de instrucciones de tres direcciones, mediante transformaciones como la combinación de subexpresiones comunes. Suponemos también que para cada operador, hay exactamente una instrucción de máquina que recibe los operandos necesarios en los registros y que realiza esa operación, dejando el resultado en un registro. Las instrucciones de máquina son de la siguiente forma:

- LD *reg, mem*
- ST *mem, reg*
- *OP reg, reg, reg*

8.6.1 Descriptores de registros y direcciones

Nuestro algoritmo de generación de código analiza una instrucción de tres direcciones a la vez y decide qué operaciones de carga son necesarias para meter los operandos requeridos en los registros. Después de generar las cargas, genera la operación en sí. Más tarde, si hay necesidad de almacenar el resultado en una ubicación de memoria, también genera esa operación de almacenamiento.

Para poder realizar las decisiones necesarias, requerimos una estructura de datos que nos indique qué variables del programa tienen actualmente su valor en un registro, y en qué registro o registros están, si es así. También debemos saber si la ubicación de memoria para una variable dada tiene actualmente el valor apropiado para esa variable, ya que tal vez se haya calculado un nuevo valor para esa variable en un registro y todavía no se almacena. La estructura de datos deseada tiene los siguientes descriptores:

1. Para cada registro disponible, un *descriptor de registro* lleva la cuenta de los nombres de las variables cuyo valor actual se encuentra en ese registro. Como sólo vamos a utilizar los registros que estén disponibles para uso local dentro de un bloque básico, suponemos que al principio todos los descriptores de registro están vacíos. A medida que progrese la generación de código, cada registro contendrá el valor de cero o más nombres.
2. Para cada variable del programa, un *descriptor de acceso* lleva la cuenta de la ubicación o ubicaciones en las que puede encontrarse el valor actual de esa variable. La ubicación podría ser un registro, una dirección de memoria, una ubicación en la pila o algún conjunto de más de uno de éstos. La información puede almacenarse en la entrada en la tabla de símbolos para ese nombre de variable.

8.6.2 El algoritmo de generación de código

Una parte esencial del algoritmo es una función llamada $obtenReg(I)$, la cual selecciona los registros para cada ubicación de memoria asociada con la instrucción de tres direcciones I . La función $obtenReg$ tiene acceso a los descriptores de registro y de acceso para todas las variables del bloque básico, y también puede tener acceso a cierta información útil del flujo de datos, como las variables que están vivas al salir del bloque. Después de presentar el algoritmo básico, hablaremos sobre $obtenReg$. Aunque no conocemos el número total de registros disponibles para los datos locales pertenecientes a un bloque básico, suponemos que hay suficientes registros de forma que, después de liberar a todos los registros disponibles almacenando sus valores en la memoria, hay suficientes registros como para realizar cualquier operación de tres direcciones.

En una instrucción de tres direcciones como $x = y + z$, vamos a tratar a $+$ como un operador genérico y a ADD como la instrucción de máquina equivalente. Por lo tanto, no aprovechamos la ley conmutativa de $+$. Así, al implementar la operación, el valor de y debe estar en el segundo registro mencionado en la instrucción ADD, nunca en el tercero. Una posible mejora para el algoritmo es generar código para $x = y + z$ y para $x = z + y$ siempre que $+$ sea un operador conmutativo, y elegir la mejor secuencia de código.

Instrucciones de máquina para las operaciones

Para una instrucción de tres direcciones como $x = y + z$, haga lo siguiente:

1. Usar $obtenReg(x = y + z)$ para seleccionar registros para x , y y z . Llamar a estos registros R_x , R_y y R_z .
2. Si y no está en R_y (de acuerdo con el descriptor de registro para R_y), entonces generar una instrucción LD R_y, y' , en donde y' es una de las ubicaciones de memoria para y (de acuerdo con el descriptor de acceso para y).
3. De manera similar, si z no está en R_z , generar una instrucción LD R_z, z' , en donde z' es una ubicación para z .
4. Generar la instrucción ADD R_x, R_y, R_z .

Instrucciones de máquina para las instrucciones de copia

Hay un caso especial importante: una instrucción de copia de tres direcciones, de la forma $x = y$. Suponemos que $obtenReg$ siempre elegirá el mismo registro tanto para x como para y . Si y no se encuentra ya en el registro R_y , entonces se genera la instrucción de máquina LD R_y, y . Si y ya se encontraba en R_y , no hacemos nada. Sólo es necesario ajustar la descripción de registro para R_y , de manera que incluya a x como una de las variables que se encuentran ahí.

Terminación del bloque básico

Como hemos descrito el algoritmo, las variables utilizadas por el bloque pueden terminar en el punto en el que su única ubicación sea un registro. Si la variable es temporal y se utiliza sólo dentro del bloque, está bien; cuando termina el bloque, podemos olvidarnos del valor de la variable temporal y suponer que su registro está vacío. No obstante, si la variable está viva al salir del bloque, o si no sabemos qué variables están vivas al salir, entonces tenemos que suponer que el valor de la variable se necesita más adelante. En ese caso, para cada variable x cuyo descriptor de ubicación no indique que su valor se encuentra en la ubicación de memoria para x , debemos generar la instrucción $ST\ x, R$, en donde R es un registro en el cual el valor de x existe al final del bloque.

Administración de los descriptores de registro y de dirección

A medida que el algoritmo de generación de código genera instrucciones de carga, almacenamiento y demás instrucciones de máquina, debe actualizar los descriptores de registro y de dirección. Las reglas para ello son:

1. Para la instrucción $LD\ R, x$
 - (a) Modificar el descriptor de registro para el registro R , de manera que sólo contenga a x .
 - (b) Modificar el descriptor de dirección para x , agregando el registro R como una ubicación adicional.
2. Para la instrucción $ST\ x, R$, modificar el descriptor de dirección para x , de manera que incluya su propia ubicación de memoria.
3. Para una operación como $ADD\ R_x, R_y, R_z$, implementar una instrucción de tres direcciones $x = y + z$
 - (a) Modificar el descriptor de registro para R_x , de manera que sólo contenga a x .
 - (b) Modificar el descriptor de dirección para x , de manera que su única ubicación sea R_x . Observe que la ubicación de memoria para x *no* se encuentra ahora en el descriptor de dirección para x .
 - (c) Eliminar R_x del descriptor de dirección de cualquier variable distinta de x .
4. Al procesar una instrucción de copia $x = y$, después de generar la carga de y en el registro R_y , si es necesario, y después de administrar los descriptores para todas las instrucciones de carga (de acuerdo con la regla 1):
 - (a) Agregar x al descriptor de registro para R_y .
 - (b) Modificar el descriptor de dirección para x , de manera que su única ubicación sea R_y .

Ejemplo 8.16: Vamos a traducir el bloque básico que consiste en las siguientes instrucciones de tres direcciones:

```

t = a - b
u = a - c
v = t + u
a = d
d = v + u

```

Aquí suponemos que **t**, **u** y **v** son variables temporales, locales para el bloque, mientras que **a**, **b**, **c** y **d** son variables que están vivas al salir del bloque. Como no hemos hablado todavía sobre la forma en que podría trabajar la función *obtenReg*, sólo asumiremos que hay tantos registros como necesitemos, pero que cuando ya no se necesite el valor de un registro (por ejemplo, que sólo contenga un valor temporal, del cual hayan pasado ya todos los usos), entonces volvemos a utilizar su registro.

En la figura 8.16 se muestra un resumen de todas las instrucciones de código máquina generadas. Esta figura también muestra los descriptores de registro y de dirección, antes y después de la traducción de cada instrucción de tres direcciones.

	R1	R2	R3	a	b	c	d	t	u	v
t = a - b				a	b	c	d			
LD R1, a										
LD R2, b										
SUB R2, R1, R2										
u = a - c	a	t		a, R1	b	c	d	R2		
LD R3, c										
SUB R1, R1, R3										
v = t + u	u	t	c	a	b	c, R3	d	R2	R1	
ADD R3, R2, R1										
a = d	u	t	v	a	b	c	d	R2	R1	R3
LD R2, d										
d = v + u	u	a, d	v	R2	b	c	d, R2		R1	R3
ADD R1, R3, R1										
salida	d	a	v	R2	b	c	R1			R3
ST a, R2										
ST d, R1										
	d	a	v	a, R2	b	c	d, R1			R3

Figura 8.16: Instrucciones generadas y los cambios en los descriptores de registro y de dirección

Para la primera instrucción de tres direcciones, **t = a - b**, debemos generar tres instrucciones, ya que al principio no hay nada en un registro. Por ende, vemos que **a** y **b** se cargan en los

registros $R1$ y $R2$, y se produce el valor t en el registro $R2$. Observe que podemos usar $R2$ para t , ya que el valor b que estaba antes en $R2$ no se necesita dentro del bloque. Como se supone que b está viva al salir del bloque, si no se hubiera encontrado en su propia ubicación de memoria (como lo indica su descriptor de dirección), hubiéramos tenido que almacenar $R2$ en b primero. La decisión de hacer esto, si hubiéramos necesitado a $R2$, sería responsabilidad de $obtenReg$.

La segunda instrucción, $u = a - c$, no requiere una carga de a debido a que ya se encuentra en el registro $R1$. Además, podemos usar $R1$ para el resultado u , ya que el valor de a , que se encontraba antes en ese registro, ya no se necesita dentro del bloque, y su valor se encuentra en su propia ubicación de memoria, en caso de que a se necesite fuera del bloque. Observe que modificamos el descriptor de dirección para a , para indicar que ya no se encuentra en $R1$, sino en la ubicación de memoria llamada a .

La tercera instrucción, $v = t + u$, sólo requiere la suma. Además, podemos usar $R3$ para el resultado v , ya que el valor de c en ese registro no se necesita más dentro del bloque, y c tiene su valor en su propia ubicación de memoria.

La instrucción de copia, $a = d$, requiere una carga de d , ya que no se encuentra en la memoria. Mostramos al descriptor de registro de $R2$, el cual contiene a a y d . La adición de a al descriptor de registro es el resultado que obtenemos al procesar la instrucción de copia, y no el resultado de cualquier instrucción de máquina.

La quinta instrucción, $d = v + u$, usa dos valores que se encuentran en registros. Como u es un valor temporal cuyo valor ya no se necesita, hemos optado por reutilizar su registro $R1$ para el nuevo valor de d . Observe que d ahora se encuentra sólo en $R1$, y no en su propia ubicación de memoria. Lo mismo se aplica para a , que se encuentra en $R2$ y no en la ubicación de memoria llamada a . Como resultado, necesitamos un “coda” al código de máquina para el bloque básico que almacena las variables a y d , que están vivas al salir, en sus ubicaciones de memoria. Mostramos éstas como las últimas dos instrucciones. \square

8.6.3 Diseño de la función $obtenReg$

Por último, vamos a considerar cómo implementar $obtenReg(I)$, para una instrucción I de tres direcciones. Existen muchas opciones, aunque también hay ciertas prohibiciones absolutas contra las opciones que conducen a un código incorrecto, debido a la pérdida del valor de una o más variables vivas. Empezamos nuestro examen con el caso del paso de una operación, para el cual utilizamos de nuevo a $x = y + z$ como ejemplo genérico. Primero, debemos elegir un registro para y y un registro para z . Las cuestiones son las mismas, por lo que nos concentraremos en elegir el registro R_y para y . Las reglas son las siguientes:

1. Si y se encuentra actualmente en un registro, elija un registro que ya contenga a y como R_y . No genere una instrucción de máquina para cargar este registro, ya que no se necesita ninguna.
2. Si y no está en un registro, pero hay un registro vacío en ese momento, elija un registro como R_y .
3. El caso difícil ocurre cuando y no se encuentra en un registro, y no hay un registro vacío en ese momento. Debemos elegir uno de los registros permitidos de todas formas, y debemos asegurarnos que sea seguro volver a utilizarlo. Suponga que R es un registro candidato y que v es una de las variables que el descriptor de registro para R dice que

está en R . Tenemos que estar seguros de que el valor de v no se necesite realmente, o que haya alguna otra parte a la que podamos ir para obtener el valor de R . Las posibilidades son:

- Si el descriptor de dirección para v dice que v está en alguna otra parte además de R , entonces estamos bien.
- Si v es x , el valor calculado por la instrucción I , y x no es tampoco uno de los otros operandos de la instrucción I (z en este ejemplo), entonces estamos bien. La razón es que en este caso, sabemos que este valor de x nunca se va a volver a utilizar, por lo que tenemos la libertad de ignorarlo.
- Por el contrario, si v no se utiliza más adelante (es decir, después de la instrucción I no hay más usos de v , y si v está viva al salir del bloque, entonces v se recalcula dentro del bloque), entonces estamos bien.
- Si no estamos bien debido a uno de los dos primeros casos, entonces debemos generar la instrucción de almacenamiento $ST v, R$ para colocar una copia de v en su propia ubicación de memoria. A esta operación se le conoce como un *derrame*.

Como R puede contener diversas variables en un momento dado, repetimos los pasos anteriores para cada una de esas variables v . Al final, la “puntuación” de R es el numero de instrucciones de almacenamiento que tuvimos que generar. Elija uno de los registros con la puntuación más baja.

Ahora, considere la selección del registro R_x . Las cuestiones y opciones son casi iguales que para y , por lo que sólo mencionaremos las diferencias.

- Como se va a calcular un nuevo valor de x , un registro que sólo contiene a x es siempre una elección aceptable para R_x . Esta instrucción se aplica incluso si x es y o z , ya que nuestras instrucciones de máquina permiten que dos registros sean iguales en una instrucción.
- Si y no se utiliza después de la instrucción I , en el sentido descrito para la variable v en el punto (3c), y R_y contiene sólo a y después de cargarse, si es necesario, entonces R_y también puede utilizarse como R_x . Se aplica una opción similar en z y R_z .

La última cuestión a considerar de manera especial es cuando I es una instrucción de copia $x = y$. Elegimos el registro R_y como vimos antes. Después, siempre elegimos $R_x = R_y$.

8.6.4 Ejercicios para la sección 8.6

Ejercicio 8.6.1: Para cada una de las siguientes instrucciones de asignación en C:

- $x = a + b*c;$
- $x = a/(b+c) - d*(e+f);$
- $x = a[i] + 1;$

- d) $a[i] = b[c[i]];$
- e) $a[i][j] = b[i][k] + c[k][j];$
- f) $*p++ = *q++;$

genere código de tres direcciones, suponiendo que todos los elementos de los arreglos sean enteros que ocupan cuatro bytes cada uno. En las partes (d) y (e), asuma que a , b y c son constantes que proporcionan la ubicación de los primeros (0vo.) elementos de los arreglos con esos nombres, como en todos los ejemplos anteriores de los accesos a arreglos en este capítulo.

Ejercicio 8.6.2: Repita los incisos (d) y (e) del ejercicio 8.6.1, suponiendo que los arreglos a , b y c se ubican mediante los apuntadores pa , pb y pc , respectivamente, apuntando a las ubicaciones de sus primeros elementos respectivamente.

Ejercicio 8.6.3: Convierta su código de tres direcciones del ejercicio 8.6.1 en código máquina para el modelo de la máquina de esta sección. Puede usar todos los registros que necesite.

Ejercicio 8.6.4: Convierta su código de tres direcciones del ejercicio 8.6.1 en código máquina, usando el algoritmo de generación de código simple de esta sección; suponga que hay tres registros disponibles. Muestre los descriptores de registro y de dirección después de cada paso.

Ejercicio 8.6.5: Repita el ejercicio 8.6.4, pero suponga que sólo hay dos registros disponibles.

8.7 Optimización de mirilla (peephole)

Aunque la mayoría de los compiladores de producción producen buen código a través de un proceso cuidadoso de selección de instrucciones y asignación de registros, unos cuantos utilizan una estrategia alternativa: generan código simple y mejoran la calidad del código de destino, aplicando transformaciones de “optimización” al programa destino. El término “optimización” es un poco confuso, ya que no hay garantía de que el código resultante sea óptimo bajo ninguna medida matemática. Sin embargo, muchas transformaciones simples pueden mejorar en forma considerable el tiempo de ejecución o requerimiento de espacio del programa destino.

Una técnica simple pero efectiva para mejorar el código destino en forma local es la *optimización de mirilla (peephole)*, la cual se lleva a cabo mediante el análisis de una ventana deslizable de instrucciones de destino (conocida como la *mirilla*), y sustituyendo las secuencias de instrucciones dentro de la mirilla por una secuencia más corta o rápida, cada vez que sea posible. La optimización de mirilla también puede aplicarse de manera directa después de la generación de código, para mejorar la representación intermedia.

La mirilla es una pequeña ventana deslizable en un programa. El código en la mirilla no tiene que ser contiguo, aunque algunas implementaciones lo requieren. Una característica de la optimización de mirilla es que cada mejora puede engendrar oportunidades para mejoras adicionales. En general, es necesario realizar varias pasadas a través del código fuente para

obtener el beneficio máximo. En esta sección vamos a proporcionar los siguientes ejemplos de transformaciones de programas, que son característicos de las optimizaciones de mirilla:

- Eliminación de instrucciones redundantes.
- Optimizaciones del flujo de control.
- Simplificaciones algebraicas.
- Uso de características específicas de máquinas.

8.7.1 Eliminación de instrucciones redundantes de carga y almacenamiento

Si vemos la siguiente secuencia de instrucciones:

```
LD  a, R0
ST  R0, a
```

en un programa destino, podemos eliminar la instrucción de almacenamiento, ya que cada vez que se ejecute, la primera instrucción se asegurará de que el valor de `a` se haya cargado en el registro `R0`. Tenga en cuenta que si la instrucción de almacenamiento tuviera una etiqueta, no podríamos estar seguros de que la primera instrucción se ejecutara siempre antes de la segunda, por lo que no podríamos eliminar la instrucción de almacenamiento. Dicho de otra forma, las dos instrucciones tienen que estar en el mismo bloque básico para que esta transformación sea segura.

Las instrucciones redundantes de carga y almacenamiento de este tipo no se generan mediante el algoritmo de generación de código simple de la sección anterior. Sin embargo, un algoritmo de generación de código simple como el de la sección 8.1.3 puede generar secuencias redundantes como éstas.

8.7.2 Eliminación de código inalcanzable

Otra oportunidad para la optimización de mirilla es la eliminación de instrucciones inalcanzables. Una instrucción sin etiqueta que va justo después de un salto incondicional puede eliminarse. Esta operación se puede repetir para eliminar una secuencia de instrucciones. Por ejemplo, para fines de depuración, un programa extenso puede tener en su interior ciertos fragmentos de código que se ejecuten sólo si una variable `debug` es igual a 1. En la representación intermedia, este código podría ser así:

```
if debug == 1 goto L1
goto L2
L1: imprimir información de depuración
L2:
```

Una optimización de mirilla obvia es la eliminación de saltos sobre saltos. Por ende, sin importar cuál sea el valor de `debug`, la secuencia de código anterior puede sustituirse por:

```

if debug != 1 goto L2
imprimir información de depuración
L2:

```

Si `debug` se establece en 0 al principio del programa, la propagación de constantes transformaría esta secuencia en:

```

if 0 != 1 goto L2
imprimir información de depuración
L2:

```

Ahora el argumento de la primera instrucción siempre se evalúa como *verdadero*, por lo que la instrucción puede sustituirse por `goto L2`. Así, todas las instrucciones que impriman información de depuración serán inalcanzables y podrán eliminarse una a la vez.

8.7.3 Optimizaciones del flujo de control

Con frecuencia, los algoritmos de generación de código intermedio simples producen saltos hacia saltos, saltos hacia saltos condicionales, o saltos condicionales hacia saltos. Estos saltos innecesarios se pueden eliminar, ya sea en el código intermedio o en el código destino, mediante los siguientes tipos de optimizaciones de mirilla. Podemos sustituir la siguiente secuencia:

```

goto L1
...
L1: goto L2

```

por la secuencia:

```

goto L2
...
L1: goto L2

```

Si ahora no hay saltos a L1, entonces tal vez sea posible eliminar la instrucción L1: `goto L2`, siempre y cuando vaya precedida de un salto incondicional.

De manera similar, la secuencia:

```

if a < b goto L1
...
L1: goto L2

```

puede sustituirse por la secuencia:

```

if a < b goto L2
...
L1: goto L2

```

Por último, suponga que sólo hay un salto hacia L1, y que L1 va precedida de un `goto` incondicional. Entonces, la secuencia:

```

goto L1
...
L1: if a < b goto L2
L3:

```

puede sustituirse por la secuencia:

```

if a < b goto L2
goto L3
...
L3:

```

Mientras que el número de instrucciones en las dos secuencias sea el mismo, algunas veces omitimos el salto incondicional en la segunda secuencia, pero nunca en la primera. Por ende, la segunda secuencia es superior a la primera en tiempo de ejecución.

8.7.4 Simplificación algebraica y reducción por fuerza

En la sección 8.5 hablamos sobre las identidades algebraicas que pueden utilizarse para simplificar los GDAs. Un optimizador de mirilla puede utilizar estas identidades algebraicas para eliminar las instrucciones de tres direcciones, como:

```

x = x + 0
o
x = x * 1

```

en la mirilla.

De manera similar, pueden aplicarse transformaciones de reducción por fuerza a la mirilla para sustituir las operaciones costosas por expresiones equivalentes más económicas en la máquina destino. Ciertas instrucciones de máquina son considerablemente más económicas que otras, y a menudo pueden usarse como casos especiales de operadores más costosos. Por ejemplo, x^2 es sin duda más económica de implementarse como $x * x$ que una llamada a una rutina de exponentiación. La multiplicación de punto flotante o división entre un exponente de dos es más económica de implementar que un desplazamiento. La división de punto flotante entre una constante puede aproximarse como una multiplicación por una constante, que puede ser más económica.

8.7.5 Uso de características específicas de máquina

La máquina destino puede tener instrucciones de hardware para implementar ciertas operaciones específicas con eficiencia. La acción de detectar situaciones que permitan el uso de estas instrucciones puede reducir el tiempo de ejecución en forma considerable. Por ejemplo, algunas máquinas tienen modos de direccionamiento de autoincremento y autodecremento. Estos modos suman o restan uno a un operando, antes o después de usar su valor. El uso de estos modos mejora en forma considerable la calidad del código al meter o sacar datos de una pila, como en el paso de parámetros. Estos modos pueden usarse también en el código para instrucciones como $x = x + 1$.

8.7.6 Ejercicios para la sección 8.7

Ejercicio 8.7.1: Construya un algoritmo que realice la eliminación de instrucciones redundantes en una mirilla deslizable, en el código de la máquina de destino.

Ejercicio 8.7.2: Construya un algoritmo que realice optimizaciones de control de flujo en una mirilla deslizable, en el código de la máquina de destino.

Ejercicio 8.7.3: Construya un algoritmo que realice simplificaciones algebraicas simples y reducciones en fuerza en una mirilla deslizable, en el código de la máquina de destino.

8.8 Repartición y asignación de registros

Las instrucciones que sólo utilizan operandos tipo registro son más rápidas que las que utilizan operandos de memoria. En las máquinas modernas, las velocidades de los procesadores son a menudo un orden o más de magnitud más rápidas que las velocidades de las memorias. Por lo tanto, la utilización eficiente de los registros es de extrema importancia para generar un buen código. Esta sección presenta diversas estrategias para decidir en cada punto de un programa qué valores deben residir en registros (repartición de registros) y en qué registro debe residir cada valor (asignación de registros).

Un método para la repartición y asignación de registros es asignar valores específicos en el programa de destino a ciertos registros. Por ejemplo, podríamos decidir asignar direcciones base a un grupo de registros, cálculos aritméticos a otro, la parte superior de la pila a un registro fijo, y así en lo sucesivo.

Este método tiene la ventaja de que simplifica el diseño de un generador de código. Su desventaja es que, si se aplica con demasiado rigor, utiliza los registros en forma inefficiente; tal vez no se utilicen ciertos registros en partes considerables del código, mientras que se generan instrucciones de carga y almacenamiento innecesarias en los demás registros. No obstante, es razonable en la mayoría de los entornos computacionales reservar unos cuantos registros para los registros base, apuntadores de pila y demás, y permitir que el generador de código utilice el resto de los registros según sea apropiado.

8.8.1 Repartición global de registros

El algoritmo de generación de código de la sección 8.6 utiliza registros para guardar valores durante el tiempo en que se ejecute un bloque básico individual. Sin embargo, todas las variables vivas se almacenan al final de cada bloque. Para ahorrar varias de estas instrucciones de almacenamiento y sus instrucciones correspondientes de carga, podríamos hacer que se asignen registros a las variables que se utilizan con frecuencia y mantener esos registros consistentes entre los límites de los bloques (en forma *global*). Debido a que los programas invierten la mayoría de su tiempo en ciclos internos, un método natural para la repartición global de registros es tratar de mantener un valor utilizado con frecuencia en un registro fijo, durante todo un ciclo. Mientras tanto, suponga que conocemos la estructura del ciclo de un grafo de flujo, y sabemos qué valores calculados en un bloque básico se utilizan fuera de ese bloque. El siguiente capítulo trata sobre las técnicas para calcular esta información.

Una estrategia para la repartición global de registros es asignar cierto número fijo de registros para guardar los valores más activos en cada ciclo interno. Los valores seleccionados pueden ser distintos en varios ciclos. Los registros que no se hayan repartido ya pueden usarse para contener valores que sean locales para un bloque, como en la sección 8.6. Este método tiene la desventaja de que el número fijo de registros no es siempre el número correcto de registros que debe haber disponibles para la repartición global de los mismos. Aun así, es sencillo implementar el método y se utilizó en Fortran H, el compilador optimizador de Fortran desarrollado por IBM, para las máquinas de la serie 360 a finales de la década de 1960.

Con los primeros compiladores de C, un programador podía realizar cierta repartición de registros en forma explícita, mediante el uso de declaraciones de registros para mantener ciertos valores en registros durante la ejecución de un procedimiento. El uso juicioso de las declaraciones de registros agilizó muchos programas, pero se alentó a los programadores a perfilar primero sus programas, para determinar los puntos clave del programa antes de realizar su propia repartición de registros.

8.8.2 Conteo de uso

En esta sección vamos a suponer que el ahorro que se obtiene al mantener una variable x en un registro durante la ejecución de un ciclo L es de una unidad de costo por cada referencia a x , si es que x ya se encuentra en un registro. No obstante, si utilizamos el método de la sección 8.6 para generar código para un bloque, hay una buena probabilidad de que una vez que se haya calculado x en el bloque, permanecerá en un registro si hay usos subsiguientes de x en ese bloque. Por ende, contamos un ahorro de una unidad para cada uso de x en el ciclo L que no vaya precedido por una asignación a x en el mismo bloque. Por ende, si a x se le asigna un registro, contamos un ahorro de dos unidades por cada bloque en el ciclo L , para el cual x esté viva al salir y en el que a x se le asigne un valor.

En el lado deudor, si x está viva al entrar al encabezado del ciclo, debemos cargar x en su registro justo antes de entrar al ciclo L . Esta carga cuesta dos unidades. De manera similar, para cada bloque B de salida del ciclo L , en el cual x esté viva al entrar a algún sucesor de B fuera de L , debemos almacenar x a un costo de dos. No obstante, si suponemos que el ciclo se itera muchas veces, podemos ignorar estas deudas ya que sólo ocurren una vez cada que entramos al ciclo. Así, una fórmula aproximada para poder obtener el beneficio de asignar un registro x dentro del ciclo L es:

$$\sum_{\text{bloques } B \text{ en } L} uso(x, B) + 2 * viva(x, B) \quad (8.1)$$

en donde $uso(x, B)$ es el número de veces que se utiliza x en B antes de cualquier definición de x ; $viva(x, B)$ es 1 si x está viva al salir de B y se le asigna un valor en B , y $viva(x, B)$ es 0 en cualquier otro caso. Observe que la ecuación (8.1) es aproximada, ya que no todos los bloques en un ciclo se ejecutan con igual frecuencia, y también porque la ecuación (8.1) se basa en la suposición de que un ciclo se itera muchas veces. En máquinas específicas, habría que desarrollar una fórmula análoga a (8.1), pero tal vez muy distinta de ella.

Ejemplo 8.17: Considere los bloques básicos en el ciclo interno descrito en la figura 8.17, en donde se han omitido las instrucciones de salto y de salto condicional. Suponga que los registros R0, R1 y R2 se reparten para guardar valores a lo largo del ciclo. Las variables vivas al entrar y al salir de cada bloque se muestran en la figura 8.17 por conveniencia, justo encima y debajo de cada bloque. Hay algunos puntos delicados acerca de las variables vivas que trataremos en el siguiente capítulo. Por ejemplo, observe que tanto e como f están vivas al final de B_1 , pero de éstas, sólo e está viva al entrar a B_2 y sólo f al entrar a B_3 . En general, las variables vivas al final de un bloque son la unión de las que están vivas al principio de cada uno de sus bloques sucesores.

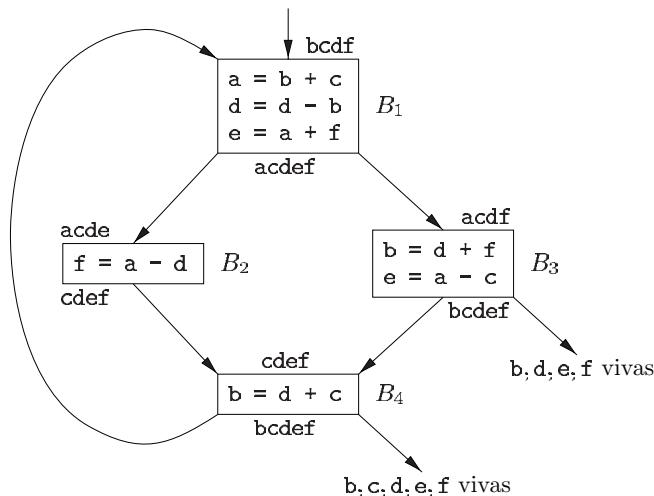


Figura 8.17: Grafo de flujo de un ciclo interno

Para evaluar la ecuación (8.1) para $x = a$, observamos que a está viva al salir de B_1 y se le asigna un valor ahí, pero no está viva al salir de B_2 , B_3 o B_4 . Por ende, $\sum_{B \text{ en } L} uso(a, B) = 2$. Así, el valor de (8.1) para $x = a$ es 4. Es decir, podemos ahorrar cuatro unidades de costo si seleccionamos a para uno de los registros globales. Los valores de (8.1) para b , c , d , e y f son 5, 3, 6, 4 y 4, respectivamente. De este modo, podemos seleccionar a , b y d para los registros R0, R1 y R2, respectivamente. Usar R0 para e o f en vez de a sería otra elección con el mismo beneficio aparente. La figura 8.18 muestra el código ensamblador generado a partir de la figura 8.17, suponiendo que se utiliza la estrategia de la sección 8.6 para generar código para cada bloque. No mostramos el código generado para los saltos condicionales o incondicionales omitidos que terminan cada bloque en la figura 8.17 y, por lo tanto, no mostraremos el código generado como un solo flujo, como aparecería en la práctica. \square

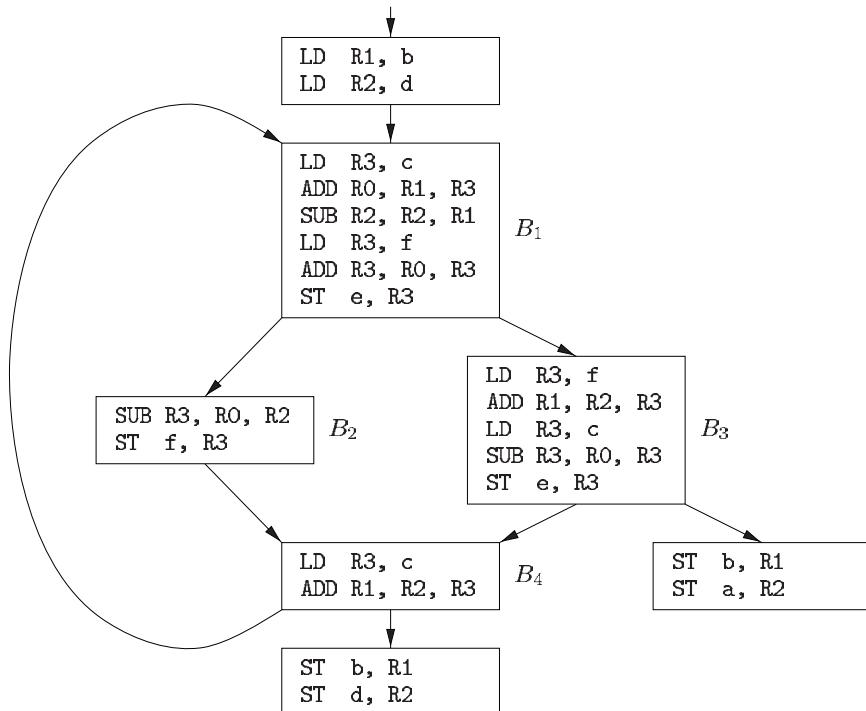


Figura 8.18: Secuencia de código que utiliza la repartición global de registros

8.8.3 Asignación de registros para ciclos externos

Ya que hemos asignado registros y generado código para los ciclos internos, podemos aplicar la misma idea a ciclos circundantes cada vez más grandes. Si un ciclo externo L_1 contiene un ciclo interno L_2 , a los nombres que se les asignaron registros en L_2 no se les necesita asignar registros en $L_1 - L_2$. De manera similar, si elegimos asignar a x un registro en L_2 pero no en L_1 , debemos cargar x al entrar a L_2 y almacenar x al salir de L_2 . Dejamos como ejercicio para el lector la derivación de un criterio para seleccionar los nombres que se van a asignar a los registros en un ciclo externo L , dado que ya se han hecho elecciones para todos los ciclos anidados dentro de L .

8.8.4 Asignación de registros mediante la coloración de grafos

Cuando se requiere un registro para un cálculo pero todos los registros disponibles están en uso, el contenido de uno de los registros debe almacenarse (*derramarse*) en una ubicación de memoria, para poder liberar un registro. La coloración de grafos es una técnica simple y sistemática para asignar registros y administrar los derrames de los mismos.

En este método se utilizan dos pasadas. En la primera, se seleccionan instrucciones de la máquina destino como si hubiera un número infinito de registros simbólicos; en efecto, los nombres

que se utilizan en el código intermedio se convierten en los nombres de los registros, y las instrucciones de tres direcciones se convierten en instrucciones en lenguaje máquina. Si el acceso a las variables requiere instrucciones que utilicen apuntadores de pila, apuntadores de visualización, registros base o demás cantidades que ayuden al acceso, entonces asumimos que estas cantidades se guardan en los registros reservados para cada fin. Por lo general, su uso puede traducirse en forma directa a un modo de acceso para una dirección mencionada en una instrucción de máquina. Si el acceso es más complejo, debe descomponerse en varias instrucciones de máquina, y tal vez haya que crear un registro simbólico temporal (o varios).

Una vez seleccionadas las instrucciones, una segunda pasada asigna los registros físicos a los simbólicos. El objetivo es encontrar una asignación que disminuya al mínimo el costo de los derrames.

En la segunda pasada, para cada procedimiento se construye un *grafo de interferencia de registros*, en el cual los nodos son registros simbólicos y una flecha conecta a dos nodos, si uno está vivo en un punto en el que el otro está definido. Por ejemplo, un gráfico de interferencia de registros para la figura 8.17 tendría nodos para los nombres *a* y *d*. En el bloque *B*₁, *a* está viva en la segunda instrucción, que define *a* *d*; por lo tanto, en el gráfico habría un flanco entre los nodos para *a* y *d*.

Se hace un intento por colorear el grafo de interferencia de registros usando *k* colores, en donde *k* es el número de registros asignables. Se dice que un grafo está *coloreado* si a cada nodo se le asigna un color de tal forma que no haya dos nodos adyacentes con el mismo color. Un color representa a un registro, y este color se asegura de que a dos registros simbólicos, que puedan interferir uno con el otro, no se les asigne el mismo registro físico.

Aunque el problema de determinar si un grafo puede colorearse con *k* colores es NP-completo en general, por lo regular se puede utilizar la siguiente técnica heurística para realizar la coloración con rapidez. Suponga que un nodo *n* en un grafo *G* tiene menos de *k* vecinos (nodos conectados a *n* por una flecha). Elimine a *n* y sus flechas de *G* para obtener un grafo *G'*. Una coloración con *k* colores de *G'* puede extenderse a una coloración con *k* colores de *G*, si se asigna a *n* un color que no se haya asignado a ninguno de sus vecinos.

Al eliminar en forma repetida los nodos que tengan menos de *k* flancos del grafo de interferencia de registros, podemos obtener el grafo vacío, en cuyo caso podemos producir una coloración con *k* colores para el grafo original, coloreando los nodos en el orden inverso en el que se eliminaron, o podemos obtener un gráfico en el cual cada nodo tenga *k* o más nodos adyacentes. En este último caso, ya no es posible realizar una coloración con *k* colores. En este punto, se derrama un nodo introduciendo código para almacenar y recargar el registro. Chaitin ha ideado varias heurísticas para elegir el nodo a derramar. Una regla general es evitar introducir código de derrame en los ciclos internos.

8.8.5 Ejercicios para la sección 8.8

Ejercicio 8.8.1: Construya el grafo de interferencia de registros para el programa en la figura 8.17.

Ejercicio 8.8.2: Idee una estrategia de asignación de registros, suponiendo que almacenamos de manera automática todos los registros en la pila antes de cada llamada a un procedimiento, y que los restauramos después de su retorno.

8.9 Selección de instrucciones mediante la rescritura de árboles

La selección de instrucciones puede ser una extensa tarea combinatoria, en especial en las máquinas que tienen muchos modos de direccionamiento, como las máquinas CISC, o en las máquinas con instrucciones de propósito especial, digamos, para el procesamiento de señales. Aun cuando asumimos que se proporciona el orden de evaluación y que los registros se asignan mediante un mecanismo separado, la selección de instrucciones (el problema de seleccionar instrucciones en el lenguaje destino para implementar los operadores en la representación intermedia) sigue siendo una extensa tarea combinatoria.

En esta sección, manejamos la selección de instrucciones como un problema de rescritura de árboles. Las representaciones tipo árbol de las instrucciones destino se han utilizado de manera efectiva en los generadores de generadores de código, que en forma automática construyen la fase de selección de instrucciones de un generador de código, a partir de una especificación de alto nivel de la máquina destino. Podría obtenerse mejor código para algunas máquinas mediante el uso de GDAs en lugar de árboles, pero la coincidencia de GDAs es más compleja que la coincidencia de árboles.

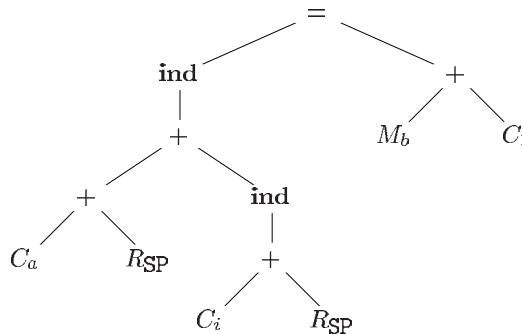
8.9.1 Esquemas de traducción de árboles

A lo largo de esta sección, la entrada al proceso de generación de código será una secuencia de árboles en el nivel semántico de la máquina de destino. Los árboles son lo que podríamos obtener después de insertar instrucciones en tiempo de ejecución en la representación intermedia, como se describe en la sección 8.3. Además, las hojas de los árboles contienen información acerca de los tipos de almacenamiento de sus etiquetas.

Ejemplo 8.18: La figura 8.19 contiene un árbol para la instrucción de asignación $a[i] = b + 1$, en donde el arreglo a se almacena en la pila en tiempo de ejecución, y la variable b es global en la ubicación de memoria M_b . Las direcciones en tiempo de ejecución de las variables locales a e i se proporcionan como los desplazamientos constantes C_a y C_i de SP, el registro que contiene el apuntador al principio del registro de activación actual.

La asignación para $a[i]$ es una asignación indirecta, en la cual el valor r de la ubicación para $a[i]$ se establece al valor r de la expresión $b + 1$. Las direcciones del arreglo a y la variable i se proporcionan agregando los valores de las constantes C_a y C_i , respectivamente, al contenido del registro SP. Para simplificar los cálculos de las direcciones del arreglo, asumimos que todos los valores son caracteres de un byte. Algunos conjuntos de instrucciones hacen provisiones especiales para las multiplicaciones por constantes, como 2, 4 y 8, durante los cálculos de las direcciones.

En el árbol, el operador **ind** trata a su argumento como una dirección de memoria. Como el hijo izquierdo de un operador de asignación, el nodo **ind** proporciona la ubicación en la que se almacenará el valor r del lado derecho del operador de asignación. Si un argumento de $a + o$ del operador **ind** es una ubicación de memoria o un registro, entonces se toma el contenido de esa ubicación de memoria o registro como valor. Las hojas en el árbol se etiquetan con atributos; un subíndice indica el valor del atributo. \square

Figura 8.19: Árbol de código intermedio para $a[i] = b + 1$

El código destino se genera aplicando una secuencia de reglas de rescritura de árboles, para reducir el árbol de entrada a un solo nodo. Cada regla de rescritura de árboles tiene la siguiente forma:

$$\text{sustituto} \leftarrow \text{plantilla} \{ \text{acción} \}$$

en donde *sustituto* es un solo nodo, *plantilla* es un árbol y *acción* es un fragmento de código, como en un esquema de traducción orientado por la sintaxis.

A un conjunto de reglas de rescritura de árboles se le conoce como *esquema de traducción de árboles*.

Cada regla de rescritura de árboles representa la traducción de una parte del árbol que proporciona la plantilla. La traducción consiste en una secuencia posiblemente vacía de instrucciones de máquina, emitida por la acción asociada con la plantilla. Las hojas de la plantilla son atributos con subíndices, como en el árbol de entrada. Algunas veces se aplican ciertas restricciones a los valores de los subíndices en las plantillas; estas restricciones se especifican como predicados semánticos que deben cumplirse para poder decir que la plantilla tiene una coincidencia. Por ejemplo, un predicado podría especificar que el valor de una constante caiga en un cierto rango.

Un esquema de traducción de árboles es una forma conveniente de representar la fase de selección de instrucciones de un generador de código. Como ejemplo de una regla de rescritura de árboles, considere la regla para la instrucción de suma, de registro a registro:

$$R_i \leftarrow \begin{array}{c} + \\ R_i \quad R_j \end{array} \{ \text{ADD } R_i, R_i, R_j \}$$

Esta regla se utiliza de la siguiente manera. Si el árbol de entrada contiene un subárbol que coincide con esta plantilla de árbol, es decir, un subárbol cuya raíz esté etiquetada por el operador $+$ y cuyos hijos izquierdo y derecho sean cantidades en los registros i y j , entonces podemos sustituir ese subárbol por un solo nodo etiquetado como R_i y generar la instrucción $\text{ADD } R_i, R_i, R_j$ como salida. A esta sustitución le llamamos *revestimiento (tiling)* del subárbol. Más de una plantilla puede coincidir con un subárbol en un momento dado; en breve describiremos los mecanismos para decidir qué regla aplicar, en casos de conflicto.

Ejemplo 8.19: La figura 8.20 contiene reglas de rescritura de árboles para algunas instrucciones de nuestra máquina de destino. A lo largo de esta sección utilizaremos estas reglas en un

ejemplo abierto. Las primeras dos reglas corresponden a instrucciones de carga, las siguientes dos para las instrucciones de almacenamiento, y el resto para las cargas y sumas indexadas. Observe que la regla (8) requiere que el valor de la constante sea 1. Esta condición se especifica mediante un predicado semántico. \square

8.9.2 Generación de código mediante el revestimiento de un árbol de entrada

Un esquema de traducción de árboles funciona de la siguiente manera. Dado un árbol de entrada, se aplican las plantillas en las reglas de rescritura para revestir sus subárboles. Si una plantilla coincide, el subárbol coincidente en el árbol de entrada se sustituye con el nodo de reemplazo de la regla y termina la acción asociada con la regla. Si la acción contiene una secuencia de instrucciones de máquina, se generan las instrucciones. Este proceso se repite hasta que el árbol se reduce a un solo nodo, o hasta que no haya coincidencia entre las plantillas. La secuencia de instrucciones de máquina generadas a medida que el árbol de entrada se reduce a un solo nodo constituye la salida del esquema de traducción de árboles en el árbol de entrada dado.

El proceso de especificar un generador de código es muy similar al de usar un esquema de traducción orientado por la sintaxis para especificar a un traductor. Escribimos un esquema de traducción de árboles para describir el conjunto de instrucciones de una máquina destino. En la práctica es conveniente encontrar un esquema que genere una secuencia de instrucciones con un costo mínimo para cada árbol de entrada. Existen varias herramientas para ayudar a construir un generador de código de manera automática, a partir de un esquema de traducción de árboles.

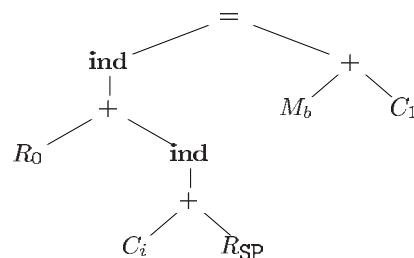
Ejemplo 8.20: Vamos a usar el esquema de traducción de árboles de la figura 8.20 para generar código en el árbol de entrada de la figura 8.19. Suponga que se aplica la primera regla para cargar la constante C_a en el registro R_0 :

$$1) \quad R_0 \leftarrow C_a \quad \{ \text{LD } R_0, \#a \}$$

La etiqueta de la hoja de más a la izquierda cambia entonces de C_a a R_0 y se genera la instrucción $\text{LD } R_0, \#a$. Ahora la séptima regla coincide con el subárbol de más a la izquierda, con la raíz etiquetada como $+$:

$$7) \quad R_0 \leftarrow \begin{array}{c} + \\ R_0 \quad R_{SP} \end{array} \quad \{ \text{ADD } R_0, R_0, SP \}$$

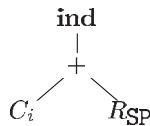
Mediante esta regla, rescribimos este subárbol como un solo nodo etiquetado como R_0 y generamos la instrucción $\text{ADD } R_0, R_0, SP$. Ahora el árbol tiene la siguiente apariencia:



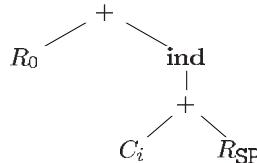
1)	$R_i \leftarrow C_a$	$\{ \text{LD } R_i, \#a \}$
2)	$R_i \leftarrow M_x$	$\{ \text{LD } R_i, x \}$
3)	$M \leftarrow \begin{array}{c} = \\ M_x \quad R_i \end{array}$	$\{ \text{ST } x, R_i \}$
4)	$M \leftarrow \begin{array}{c} = \\ \text{ind} \quad R_j \\ \\ R_i \end{array}$	$\{ \text{ST } *R_i, R_j \}$
5)	$R_i \leftarrow \begin{array}{c} \text{ind} \\ \\ + \\ C_a \quad R_j \end{array}$	$\{ \text{LD } R_i, a(R_j) \}$
6)	$R_i \leftarrow \begin{array}{c} + \\ R_i \quad \text{ind} \\ \\ + \\ C_a \quad R_j \end{array}$	$\{ \text{ADD } R_i, R_i, a(R_j) \}$
7)	$R_i \leftarrow \begin{array}{c} + \\ R_i \quad R_j \end{array}$	$\{ \text{ADD } R_i, R_i, R_j \}$
8)	$R_i \leftarrow \begin{array}{c} + \\ R_i \quad C_1 \end{array}$	$\{ \text{INC } R_i \}$

Figura 8.20: Reglas de rescritura de árboles para algunas instrucciones de la máquina destino

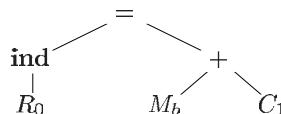
En este punto, podríamos aplicar la regla (5) para reducir el subárbol:



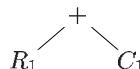
a un solo nodo etiquetado como R_1 , por ejemplo. También podríamos usar la regla (6) para reducir el subárbol más grande:



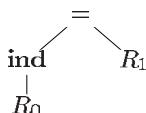
a un solo nodo etiquetado como R_0 y generar la instrucción ADD R0, R0, i(SP). Suponiendo que es más eficiente usar una sola instrucción para calcular el subárbol más grande, en vez del más pequeño, elegimos la regla (6) para obtener lo siguiente:



En el subárbol derecho, la regla (2) se aplica a la hoja M_b . Genera una instrucción para cargar b en el registro R1, por ejemplo. Ahora, usando la regla (8) podemos igualar el subárbol:



y generar la instrucción de incremento INC R1. En este punto, el árbol de entrada se ha reducido a:



Este árbol restante se iguala mediante la regla (4), la cual reduce el árbol a un solo nodo y genera la instrucción ST *R0, R1. Generamos la siguiente secuencia de código:

```

LD R0, #a
ADD R0, R0, SP
ADD R0, R0, i(SP)
LD R1, b
INC R1
ST *R0, R1
  
```

en el proceso de reducir el árbol a un solo nodo. \square

Para poder implementar el proceso de reducción de árboles en el ejemplo 8.18, debemos considerar ciertas cuestiones relacionadas con la coincidencia de patrones de árboles:

- ¿Cómo deben coincidir los patrones de árboles? La eficiencia del proceso de generación de código (en tiempo de compilación) depende de la eficiencia del algoritmo para igualar árboles.
- ¿Qué hacemos si más de una plantilla coincide en un momento dado? La eficiencia del código generado (en tiempo de ejecución) puede depender del orden en el que se igualan las plantillas, ya que en general, distintas secuencias de coincidencias conducirán a distintas secuencias de código de la máquina destino, algunas más eficientes que otras.

Si ninguna plantilla coincide, entonces el proceso de generación de código se bloquea. En el otro extremo, debemos protegernos contra la posibilidad de que se escriba un solo nodo en forma indefinida, generando una secuencia infinita de instrucciones de movimiento de registros, o una secuencia infinita de instrucciones de carga y almacenamiento.

Para evitar el bloqueo, suponemos que cada operador en el código intermedio puede implementarse mediante una o más instrucciones de la máquina destino. Además, suponemos que hay suficientes registros para calcular cada uno de los nodos del árbol por sí solos. Así, no importa cómo se dé el proceso de igualar el árbol, ya que el árbol restante siempre podrá traducirse en instrucciones de la máquina destino.

8.9.3 Coincidencias de los patrones mediante el análisis sintáctico

Antes de considerar el proceso general de igualar árboles, consideraremos un método especializado que utiliza un analizador sintáctico LR para buscar las coincidencias de los patrones. El árbol de entrada puede tratarse como una cadena, usando su representación en prefijo. Por ejemplo, la representación prefijo para el árbol de la figura 8.19 es:

$$= \mathbf{ind} + + C_a R_{\mathbf{SP}} \mathbf{ind} + C_i R_{\mathbf{SP}} + M_b C_1$$

El esquema de traducción de árboles puede convertirse en un esquema de traducción orientado por la sintaxis, para lo cual se sustituyen las reglas de rescritura de árboles con las producciones de una gramática libre de contexto, en la que los lados derechos son representaciones prefijas de las plantillas de instrucciones.

Ejemplo 8.21: El esquema de traducción orientado por la sintaxis en la figura 8.21 se basa en el esquema de traducción de árboles de la figura 8.20.

Los no terminales de la gramática subyacente son R y M . El terminal \mathbf{m} representa una ubicación en memoria específica, como la ubicación para la variable global \mathbf{b} en el ejemplo 8.18. La producción $M \rightarrow \mathbf{m}$ en la regla (10) puede considerarse como el proceso de igualar M con \mathbf{m} , antes de usar una de las plantillas que involucran a M . De manera similar, introducimos el terminal \mathbf{sp} para el registro \mathbf{SP} y agregamos la producción $R \rightarrow \mathbf{SP}$. Por último, la terminal c representa a las constantes.

1) $R_i \rightarrow \mathbf{c}_a$	$\{ \text{LD } R_i, \#a \}$
2) $R_i \rightarrow M_x$	$\{ \text{LD } R_i, x \}$
3) $M \rightarrow = M_x R_i$	$\{ \text{ST } x, R_i \}$
4) $M \rightarrow = \mathbf{ind} R_i R_j$	$\{ \text{ST } *R_i, R_j \}$
5) $R_i \rightarrow \mathbf{ind} + \mathbf{c}_a R_j$	$\{ \text{LD } R_i, a(R_j) \}$
6) $R_i \rightarrow + R_i \mathbf{ind} + \mathbf{c}_a R_j$	$\{ \text{ADD } R_i, R_i, a(R_j) \}$
7) $R_i \rightarrow + R_i R_j$	$\{ \text{ADD } R_i, R_i, R_j \}$
8) $R_i \rightarrow + R_i \mathbf{c}_1$	$\{ \text{INC } R_i \}$
9) $R \rightarrow \mathbf{sp}$	
10) $M \rightarrow \mathbf{m}$	

Figura 8.21: Esquema de traducción orientado por la sintaxis, construido a partir de la figura 8.20

Usando estos terminales, la cadena para el árbol de entrada en la figura 8.19 es:

$$= \mathbf{ind} + + \mathbf{c}_a \mathbf{sp} \mathbf{ind} + \mathbf{c}_i \mathbf{sp} + \mathbf{m}_b \mathbf{c}_1$$

□

De las producciones para el esquema de traducción, construimos un analizador sintáctico LR usando una de las técnicas de construcción de analizadores sintácticos LR que vimos en el capítulo 4. Para generar el código destino, se genera la instrucción de máquina correspondiente a cada reducción.

Por lo general, una gramática de generación de código es muy ambigua, y hay que tener cuidado en la forma en que se resuelven los conflictos de acción de análisis sintáctico cuando se construye el analizador sintáctico. En la ausencia de información sobre los costos, una regla general es dar preferencia a las reducciones más grandes sobre las más pequeñas. Esto significa que en un conflicto de reducción-reducción, se favorece a la reducción más larga; en un conflicto de desplazamiento-reducción, se elige el movimiento por desplazamiento. Este método del “máximo procesado” hace que se realice un gran número de operaciones con una sola instrucción de máquina.

Hay algunos beneficios en el uso del análisis sintáctico LR en la generación de código. En primer lugar, el método del análisis sintáctico es eficiente y bien comprendido, por lo que pueden producirse generadores de código confiables y eficientes mediante el uso de los algoritmos descritos en el capítulo 4. En segundo lugar, es muy sencillo redirigir el generador de código resultante; puede construirse un selector de código para una nueva máquina mediante la escritura de una gramática para describir las instrucciones de la nueva máquina. En tercer lugar, la calidad del código generado puede volverse eficiente al agregar producciones de casos especiales para aprovechar características específicas de las máquinas.

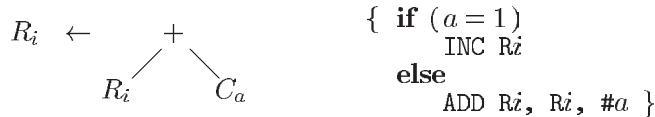
Sin embargo, existen también algunos retos. El método de análisis sintáctico fija un orden de evaluación de izquierda a derecha. Además, para algunas máquinas con grandes números de modos de direccionamiento, la gramática de descripción de la máquina y el analizador sintáctico resultante pueden aumentar de tamaño en forma desordenada. Como consecuencia, son necesarias técnicas especializadas para codificar y procesar las gramáticas de descripción de la máquina. También debemos tener cuidado en que el analizador sintáctico resultante no se bloquee (que no pueda hacer más movimientos) mientras analiza un árbol de expresiones, ya sea debido a que la gramática no maneja algunos patrones de operadores, o porque el analizador

sintáctico ha tomado la resolución incorrecta de algún conflicto de acción del análisis sintáctico. También debemos asegurarnos de que el analizador sintáctico no entre en un ciclo infinito de reducciones de producciones con símbolos individuales en el lado derecho. El problema de los ciclos puede resolverse mediante el uso de una técnica de división de estados, al mismo tiempo en el que se generan las tablas del analizador sintáctico.

8.9.4 Rutinas para la comprobación semántica

En un esquema de traducción para la generación de código, aparecen los mismos atributos que en un árbol de entrada, pero a menudo con restricciones en los valores que pueden tener los subíndices. Por ejemplo, una instrucción de máquina puede requerir que el valor de un atributo se encuentre dentro de cierto rango, o que los valores de dos atributos estén relacionados.

Estas restricciones sobre los valores de los atributos pueden especificarse como predicados que se invocan antes de realizar una reducción. De hecho, el uso general de acciones semánticas y predicados puede proporcionar una mayor flexibilidad y facilidad de descripción que una especificación sólo gramatical de un generador de código. Pueden utilizarse plantillas genéricas para representar clases de instrucciones y después pueden usarse las acciones semánticas para elegir instrucciones en los casos específicos. Por ejemplo, dos formas de la instrucción de suma pueden representarse con una plantilla:



Los conflictos de acción del análisis sintáctico pueden resolverse mediante predicados para eliminar ambigüedades, los cuales pueden permitir el uso de distintas estrategias de selección en varios contextos. Es posible una descripción más pequeña de una máquina destino, ya que ciertos aspectos de la arquitectura de la máquina, como los modos de direccionamiento, pueden factorizarse en los atributos. La complicación en este método es que puede ser difícil verificar la precisión del esquema de traducción como una descripción fiel de la máquina destino, aunque este problema se comparte hasta cierto grado por todos los generadores de código.

8.9.5 Proceso general para igualar árboles

El método de análisis sintáctico LR para la coincidencia de patrones basada en las representaciones prefijas favorece al operando izquierdo de un operador binario. En una representación prefija **op** $E_1 E_2$, las decisiones del análisis sintáctico LR de lectura anticipada limitada deben hacerse en base a cierto prefijo de E_1 , ya que E_1 puede tener una longitud arbitraria. Por ende, la coincidencia de patrones puede omitir matices del conjunto de instrucciones destino que se deben a los operandos derechos.

En vez de la representación prefija, podríamos usar una representación postfija. Pero, entonces un método de análisis sintáctico LR para la coincidencia de patrones favorecería al operando derecho.

Para un generador de código escrito a mano podemos usar plantillas de árboles, como en la figura 8.20, como una guía y escribir un igualador ad hoc. Por ejemplo, si la raíz del árbol de entrada se etiqueta como **ind**, entonces el único patrón que podría coincidir es para la regla (5);

en cualquier otro caso, si la raíz está etiquetada como $+$, entonces los patrones que podría coincidir son para las reglas (6-8).

Para un generador de generadores de código, necesitamos un algoritmo general de coincidencia de árboles. Podemos desarrollar un algoritmo descendente eficiente, extendiendo las técnicas de coincidencia de patrones de cadenas del capítulo 3. La idea es representar cada plantilla como un conjunto de cadenas, en donde una cadena corresponde a una ruta de la raíz hacia una hoja en la plantilla. Tratamos a todos los operandos de igual forma, incluyendo el número de posición de un hijo, de izquierda a derecha, en las cadenas.

Ejemplo 8.22: Para construir el conjunto de cadenas para un conjunto de instrucciones, vamos a omitir los subíndices, ya que la coincidencia de patrones se basa sólo en los atributos, no en sus valores.

Las plantillas de la figura 8.22 tienen el siguiente conjunto de cadenas, desde la raíz hasta una hoja:

$$\begin{aligned} C \\ + 1 \ R \\ + 2 \ \mathbf{ind} \ 1 + 1 \ C \\ + 2 \ \mathbf{ind} \ 1 + 2 \ R \\ + 2 \ R \end{aligned}$$

La cadena C representa a la plantilla que tiene a C en la raíz. La cadena $+ 1 \ R$ representa al $+$ y su operando izquierdo R en las dos plantillas que tienen a $+$ en la raíz. \square

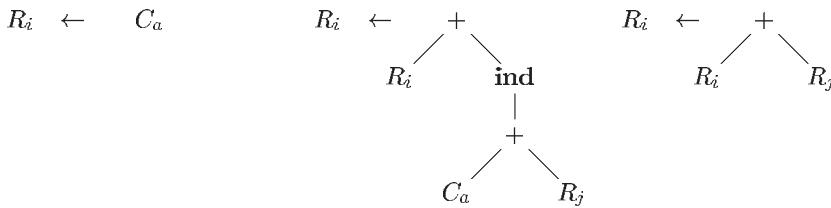


Figura 8.22: Un conjunto de instrucciones para la coincidencia de árboles

Si utilizamos conjuntos de cadenas como en el ejemplo 8.22, podemos construir un igualador de patrones de árboles mediante el uso de las técnicas para igualar con eficiencia varias cadenas en paralelo.

En la práctica, el proceso de rescritura de árboles puede implementarse mediante la ejecución del igualador de patrones de árboles durante un recorrido en profundidad (depth-first) del árbol de entrada, y realizando las reducciones a medida que se visitan los nodos por última vez.

Podemos tomar en cuenta los costos de las instrucciones, asociando con cada regla de rescritura de árboles el costo de la secuencia de instrucciones de máquina que se generan, si se aplica esa regla. En la sección 8.11 veremos un algoritmo de programación dinámica que puede usarse en conjunto con la coincidencia de patrones de árboles.

Al ejecutar el algoritmo de programación dinámica en forma concurrente, podemos seleccionar una secuencia óptima de coincidencias, usando la información de costos asociada con

cada regla. Tal vez haya que diferir la decisión sobre una coincidencia hasta que se conozca el costo de todas las alternativas. Mediante el uso de este método, podemos construir con rapidez un generador de código eficiente y pequeño, a partir de un esquema de rescritura de árboles. Además, el algoritmo de programación dinámica libera al diseñador del generador de código de tener que resolver las coincidencias conflictivas, o decidir sobre un orden para la evaluación.

8.9.6 Ejercicios para la sección 8.9

Ejercicio 8.9.1: Construya árboles sintácticos para cada una de las siguientes instrucciones, suponiendo que todos los operandos no constantes se encuentran en ubicaciones de memoria:

- a) $x = a * b + c * d;$
- b) $x[i] = y[j] * z[k];$
- c) $x = x + 1;$

Use el esquema de rescritura de árboles de la figura 8.20 para generar el código de cada instrucción.

Ejercicio 8.9.2: Repita el ejercicio 8.9.1 anterior, mediante el esquema de traducción orientado por la sintaxis de la figura 8.21 en vez del esquema de rescritura de árboles.

! Ejercicio 8.9.3: Extienda el esquema de rescritura de árboles de la figura 8.20 para que se aplique a las instrucciones while.

! Ejercicio 8.9.4: ¿Cómo extendería la rescritura de árboles para aplicarla a los GDAs?

8.10 Generación de código óptimo para las expresiones

Podemos elegir registros de manera óptima cuando un bloque básico consiste en una sola evaluación de una expresión, o si aceptamos que basta con generar código para un bloque, una expresión a la vez. En el siguiente algoritmo, presentamos un esquema de numeración para los nodos de un árbol de expresión (un árbol sintáctico para una expresión) que nos permite generar el código óptimo para un árbol de expresión, cuando hay un número fijo de registros con los que se puede evaluar la expresión.

8.10.1 Números de Ershov

Empezaremos por asignar a los nodos de un árbol de expresión un número que indique cuántos registros se necesitan para evaluar ese nodo, sin almacenar valores temporales. A estos números se les llama algunas veces *números de Ershov*, por A. Ershov, quien utilizó un esquema similar para las máquinas con un solo registro aritmético. Para nuestro modelo de máquina, las reglas son:

1. Etiquetar cualquier hoja con 1.
2. La etiqueta de un nodo interior con un hijo es la etiqueta de su hijo.

3. La etiqueta de un nodo interior con dos hijos es:

- (a) La más grande de las etiquetas de sus hijos, si esas etiquetas son distintas.
- (b) Uno más la etiqueta de sus hijos, si las etiquetas son iguales.

Ejemplo 8.23: En la figura 8.23 vemos un árbol de expresión (se omitieron los operadores) que podría ser el árbol para la expresión $(a - b) + e \times (c + d)$ o el siguiente código de tres direcciones:

```
t1 = a - b
t2 = c + d
t3 = e * t2
t4 = t1 + t3
```

Cada una de las cinco hojas se etiqueta como 1 mediante la regla (1). Así, podemos etiquetar el nodo interior para $t1 = a - b$, ya que ambos de sus hijos están etiquetados. La regla (3b) se aplica, por lo que obtiene una etiqueta más que las etiquetas de sus hijos, es decir, 2. Lo mismo se aplica para el nodo interior para $t2 = c + d$.

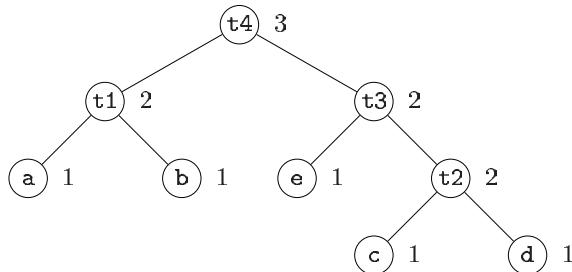


Figura 8.23: Un árbol etiquetado con números de Ershov

Ahora podemos trabajar en el nodo para $t3 = e * t2$. Sus hijos tienen las etiquetas 1 y 2, por lo que la etiqueta del nodo para $t3$ es la máxima, 2, según la regla (3a). Por último, la raíz, el nodo para $t4 = t1 + t3$, tiene dos hijos con la etiqueta 2 y, por lo tanto, obtiene la etiqueta 3. \square

8.10.2 Generación de código a partir de árboles de expresión etiquetados

Puede demostrarse que, en nuestro modelo de máquina, en donde todos los operandos deben estar en registros, y los registros pueden utilizarse tanto por un operando como por el resultado de una operación, la etiqueta de un nodo es la menor cantidad de registros con los que se puede evaluar la expresión, sin utilizar almacenamientos de resultados temporales. Como en este modelo estamos obligados a cargar cada operando y a calcular el resultado correspondiente a cada nodo interior, lo único que puede hacer que el código generado sea inferior al código óptimo es si hay almacenamientos innecesarios de valores temporales. El argumento para esta declaración está incrustado en el siguiente algoritmo para generar código sin almacenamientos de valores temporales, utilizando un número de registros igual a la etiqueta de la raíz.

Algoritmo 8.24: Generación de código a partir de un árbol de expresión etiquetado.

ENTRADA: Un árbol etiquetado con cada operando que aparece una vez (es decir, sin subexpresiones comunes).

SALIDA: Una secuencia óptima de instrucciones de máquina para evaluar la raíz y colocarla en un registro.

MÉTODO: A continuación se muestra un algoritmo recursivo para generar el código máquina. Los siguientes pasos se aplican, empezando en la raíz del árbol. Si el algoritmo se aplica a un nodo con la etiqueta k , entonces sólo se utilizarán k registros. No obstante, hay una “base” $b \geq 1$ para los registros utilizados, de manera que los registros actuales utilizados son $R_b, R_{b+1}, \dots, R_{b+k-1}$. El resultado siempre aparece en R_{b+k-1} .

1. Para generar código máquina para un nodo interior con la etiqueta k y dos hijos con etiquetas iguales (que deben ser $k-1$), haga lo siguiente:
 - (a) Genere código en forma recursiva para el hijo derecho, usando la base $b+1$. El resultado del hijo derecho aparece en el registro R_{b+k} .
 - (b) Genere código en forma recursiva para el hijo izquierdo, usando la base b ; el resultado aparece en R_{b+k-1} .
 - (c) Genere la instrucción $\text{OP } R_{b+k}, R_{b+k-1}, R_{b+k}$, en donde OP es la operación apropiada para el nodo interior en cuestión.
2. Suponga que tenemos un nodo interior con la etiqueta k e hijos con etiquetas desiguales. Entonces uno de los hijos, al que llamaremos el hijo “grande”, tiene la etiqueta k , y el otro hijo, el hijo “pequeño”, tiene cierta etiqueta $m < k$. Haga lo siguiente para generar código para este nodo interior, usando la base b :
 - (a) Genere código en forma recursiva para el hijo grande, usando la base b ; el resultado aparece en el registro R_{b+k-1} .
 - (b) Genere código en forma recursiva para el hijo pequeño, usando la base b ; el resultado aparece en el registro R_{b+m-1} . Observe que como $m < k$, no se utiliza R_{b+k-1} ni cualquier otro registro con numeración más alta.
 - (c) Genere la instrucción $\text{OP } R_{b+k-1}, R_{b+m-1}, R_{b+k-1}$ o la instrucción $\text{OP } R_{b+k-1}, R_{b+k-1}, R_{b+m-1}$, dependiendo de si el hijo grande es el derecho o el izquierdo, respectivamente.
3. Para una hoja que represente al operando x , si la base es b genere la instrucción $\text{LD } R_b, x$.

□

Ejemplo 8.25: Vamos a aplicar el Algoritmo 8.24 al árbol de la figura 8.23. Como la etiqueta de la raíz es 3, el resultado aparecerá en R_3 y sólo se utilizarán R_1, R_2 y R_3 . La base para la raíz es $b = 1$. Como la raíz tiene hijos con etiquetas iguales, generamos código para el hijo derecho primero, con la base 2.

Al generar código para el hijo derecho de la raíz, etiquetado como t_3 , encontramos que el hijo grande es el hijo derecho y que el hijo pequeño es el hijo izquierdo. Por ende, generamos código para el hijo derecho primero, con $b = 2$. Si aplicamos las reglas para hijos y hojas con etiquetas iguales, generamos el siguiente código para el nodo etiquetado como t_2 :

```
LD R3, d
LD R2, c
ADD R3, R2, R3
```

A continuación, generamos código para el hijo izquierdo del hijo derecho de la raíz; este nodo es la hoja etiquetada como e . Debido a que $b = 2$, la instrucción apropiada es:

```
LD R2, e
```

Ahora podemos completar el código para el hijo derecho de la raíz, agregando la siguiente instrucción:

```
MUL R3, R2, R3
```

El algoritmo procede a generar código para el hijo izquierdo de la raíz, dejando el resultado en R_2 y con la base 1. La secuencia completa de instrucciones se muestra en la figura 8.24. \square

```
LD R3, d
LD R2, c
ADD R3, R2, R3
LD R2, e
MUL R3, R2, R3
LD R2, b
LD R1, a
SUB R2, R1, R2
ADD R3, R2, R3
```

Figura 8.24: Código óptimo de tres registros para el árbol de la figura 8.23

8.10.3 Evaluación de expresiones con un suministro insuficiente de registros

Cuando hay un número de registros disponibles menor a la etiqueta de la raíz del árbol, no podemos aplicar el Algoritmo 8.24 en forma directa. Debemos introducir ciertas instrucciones de almacenamiento que derramen los valores de los subárboles en la memoria, y después tenemos que cargar esos valores de vuelta en los registros, según se vayan requiriendo. He aquí el algoritmo modificado que toma en cuenta una limitación sobre el número de registros.

Algoritmo 8.26: Generación de código a partir de un árbol de expresión etiquetado.

ENTRADA: Un árbol etiquetado, en el que cada operando aparece una vez (es decir, no hay subexpresiones comunes) y un número de registros $r \geq 2$.

SALIDA: Una secuencia óptima de instrucciones de máquina para evaluar la raíz y colocarla en un registro, usando no más de r registros, que suponemos son R_1, R_2, \dots, R_r .

MÉTODO: Aplique el siguiente algoritmo recursivo, empezando en la raíz del árbol, con la base $b = 1$. Para un nodo N con la etiqueta r o menor, el algoritmo es exactamente el mismo que el Algoritmo 8.24, por lo que no repetiremos aquí esos pasos. Sin embargo, para los nodos interiores con una etiqueta $k > r$, debemos trabajar en cada lado del árbol por separado y almacenar el resultado del subárbol más grande. Ese resultado se regresa a la memoria justo antes de evaluar el nodo N , y el paso final se llevará a cabo en los registros R_{r-1} y R_r . Las modificaciones al algoritmo básico son las siguientes:

1. El nodo N tiene por lo menos un hijo con la etiqueta r o mayor. Elija el hijo más grande (o cualquiera si sus etiquetas son iguales) para que sea el hijo “grande” y que el otro hijo sea el “pequeño”.
2. Genere código en forma recursiva para el hijo grande, usando la base $b = 1$. El resultado de esta evaluación aparecerá en el registro R_r .
3. Genere la instrucción de máquina **ST** t_k, R_r , en donde t_k es una variable temporal que se utiliza para los resultados temporales que ayudan a evaluar los nodos con la etiqueta k .
4. Genere código para el hijo pequeño de la siguiente forma. Si el hijo pequeño tiene la etiqueta r o mayor, elija la base $b = 1$. Si la etiqueta del hijo pequeño es $j < r$, entonces elija $b = r - j$. Despues aplique este algoritmo en forma recursiva al hijo pequeño; el resultado aparece en R_r .
5. Genere la instrucción **LD** R_{r-1}, t_k .
6. Si el hijo grande es el hijo derecho de N , entonces genere la instrucción **OP** R_r, R_r, R_{r-1} . Si el hijo grande es el izquierdo, genere **OP** R_r, R_{r-1}, R_r .

□

Ejemplo 8.27: Vamos a repasar la expresión representada por la figura 8.23, pero ahora suponga que $r = 2$; es decir, sólo están disponibles los registros **R1** y **R2** para guardar los temporales usados en la evaluación de las expresiones. Al aplicar el Algoritmo 8.26 a la figura 8.23, podemos ver que la raíz, con la etiqueta 3, tiene una etiqueta más grande que $r = 2$. Por ende, tenemos que identificar a uno de los hijos como el hijo “grande”. Como tienen etiquetas iguales, cualquiera puede serlo. Suponga que elegimos el hijo derecho como el hijo grande.

Como la etiqueta del hijo grande de la raíz es 2, hay suficientes registros. Por ende, aplicamos el Algoritmo 8.24 a este subárbol, con $b = 1$ y dos registros. El resultado es muy parecido al código que generamos en la figura 8.24, pero con los registros **R1** y **R2** en vez de **R2** y **R3**. Este código es:

```

LD  R2, d
LD  R1, c
ADD R2, R1, R2
LD  R1, e
MUL R2, R1, R2

```

Ahora, como necesitamos ambos registros para el hijo izquierdo de la raíz, debemos generar la siguiente instrucción:

```
ST t3, R2
```

A continuación, se maneja el hijo izquierdo de la raíz. De nuevo, el número de registros es suficiente para este hijo, y el código es:

```
LD R2, b
LD R1, a
SUB R2, R1, R2
```

Por último, recargamos la variable temporal que contiene el hijo derecho de la raíz con la siguiente instrucción:

```
LD R1, t3
```

y ejecutamos la operación en la raíz del árbol con la siguiente instrucción:

```
ADD R2, R2, R1
```

La secuencia completa de instrucciones se muestra en la figura 8.25. \square

```
LD R2, d
LD R1, c
ADD R2, R1, R2
LD R1, e
MUL R2, R1, R2
ST t3, R2
LD R2, b
LD R1, a
SUB R2, R1, R2
LD R1, t3
ADD R2, R2, R1
```

Figura 8.25: Código óptimo de tres registros para el árbol de la figura 8.23, usando sólo dos registros

8.10.4 Ejercicios para la sección 8.10

Ejercicio 8.10.1: Calcule los números de Ershov para las siguientes expresiones:

- $a/(b + c) - d * (e + f)$.
- $a + b * (c * (d + e))$.

c) $(-a + *p) * ((b - *q)/(-c + *r)).$

Ejercicio 8.10.2: Genere código óptimo, usando dos registros para cada una de las expresiones del ejercicio 8.10.1.

Ejercicio 8.10.3: Genere código óptimo, usando tres registros para cada una de las expresiones del ejercicio 8.10.1.

! Ejercicio 8.10.4: Generalice el cálculo de los números de Ershov para los árboles de expresión con nodos interiores que tengan tres o más hijos.

! Ejercicio 8.10.5: Una asignación a un elemento de un arreglo, como $a[i] = x$, parece ser un operador con tres operandos: a , i y x . ¿Cómo modificaría el esquema de etiquetado de árboles para generar código óptimo para este modelo de máquina?

! Ejercicio 8.10.6: Los números de Ershov originales se utilizaron para una máquina que permitía que el operando derecho de una expresión estuviera en la memoria, en vez de estar en un registro. ¿Cómo modificaría el esquema de etiquetado de árboles para generar código óptimo para este modelo de máquina?

! Ejercicio 8.10.7: Algunas máquinas requieren dos registros para ciertos valores de precisión simple. Suponga que el resultado de una multiplicación de cantidades de un solo registro requiere dos registros consecutivos, y que cuando dividimos a/b , el valor de a debe guardarse en dos registros consecutivos. ¿Cómo modificaría el esquema de etiquetado de árboles para generar código óptimo en este modelo de máquina?

8.11 Generación de código de programación dinámica

El Algoritmo 8.26 en la sección 8.10 produce código óptimo a partir de un árbol de expresión, utilizando una cantidad de tiempo que es una función lineal del tamaño del árbol. Este procedimiento funciona para las máquinas en las que todos los cálculos se realizan en registros, y en las que las instrucciones consisten en un operador que se aplica a dos registros, o a un registro y a una ubicación de memoria.

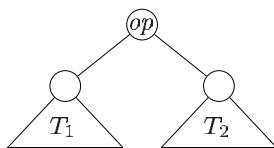
Podemos usar un algoritmo basado en el principio de la programación dinámica con el fin de extender la clase de máquinas para las cuales se puede generar un código óptimo, a partir de árboles de expresión en tiempo lineal. El algoritmo de programación dinámica se aplica a una amplia clase de máquinas de registro con conjuntos complejos de instrucciones.

El algoritmo de programación dinámica se puede utilizar en la generación de código para cualquier máquina con r registros intercambiables $R0, R1, \dots, Rr - 1$ e instrucciones de carga, almacenamiento y suma. Por cuestión de simplicidad, suponemos que cada instrucción tiene un costo de una unidad, aunque el algoritmo de programación dinámica puede modificarse con facilidad para trabajar aun cuando cada instrucción tiene su propio costo.

8.11.1 Evaluación contigua

El algoritmo de programación dinámica partitiona el problema de generar código óptimo para una expresión en los subproblemas de generar código óptimo para las subexpresiones de la expresión dada. Como un ejemplo simple, considere una expresión E de la forma $E_1 + E_2$. Un programa óptimo para E se forma mediante la combinación de programas óptimos para E_1 y E_2 , en un orden o en el otro, seguidos de código para evaluar el operador $+$. Los subproblemas de generar código óptimo para E_1 y E_2 se resuelven de manera similar.

Un programa óptimo producido por el algoritmo de programación dinámica tiene una importante propiedad. Evalúa una expresión $E = E_1 \text{ op } E_2$ en forma “contigua”. Podemos apreciar lo que esto significa si analizamos el árbol sintáctico T para E :



Aquí, T_1 y T_2 son árboles para E_1 y E_2 , respectivamente.

Decimos que un programa P evalúa a un árbol T en forma *contigua* si primero evalúa los subárboles de T que deben calcularse en la memoria. Después, evalúa el resto de T , ya sea en el orden T_1 , T_2 , y después la raíz, o en el orden T_2 , T_1 , y después la raíz, en cualquier caso usando los valores antes calculados de la memoria, siempre que sea necesario. Como ejemplo de evaluación no contigua, P primero podría evaluar parte de T_1 , dejando el valor en un registro (en vez de dejarlo en la memoria), después evaluar T_2 y luego regresar a evaluar el resto de T_1 .

Para la máquina de registro en esta sección, podemos demostrar que dado cualquier programa P en lenguaje máquina para evaluar un árbol de expresión T , podemos encontrar un programa equivalente P' de forma que:

1. P' no tenga un mayor costo que P .
2. P' no utilice más registros que P .
3. P' evalúe el árbol en forma contigua.

Este resultado implica que cada árbol de expresión se puede evaluar de manera óptima mediante un programa contiguo.

En contraste, las máquinas con pares de registros impar-par no siempre tienen evaluaciones contiguas óptimas; la arquitectura x86 utiliza pares de registros para la multiplicación y la división. Para tales máquinas, podemos dar ejemplos de árboles de expresión en los que un programa óptimo en lenguaje máquina primero debe evaluar en un registro una porción del subárbol izquierdo de la raíz, después una porción del subárbol derecho, sucesivamente otra parte del subárbol izquierdo, después otra parte del subárbol derecho, y así en lo sucesivo. Este tipo de oscilación es innecesaria para una evaluación óptima de cualquier árbol de expresión, si usamos la máquina de esta sección.

La propiedad de evaluación contigua antes definida asegura que, para cualquier árbol de expresión T , siempre existe un programa óptimo que consista de programas óptimos para los subárboles de la raíz, seguidos de una instrucción para evaluar la raíz. Esta propiedad nos permite usar un algoritmo de programación dinámica en la generación de un programa óptimo para T .

8.11.2 El algoritmo de programación dinámica

El algoritmo de programación dinámica procede en tres fases (suponga que la máquina destino tiene r registros):

1. Calcula, de abajo hacia arriba, para cada nodo n del árbol de expresión T , un arreglo C de costos, en el cual el i -ésimo componente $C[i]$ es el costo óptimo de calcular el subárbol S con raíz en n y colocarlo en un registro, suponiendo que hay i registros disponibles para el cálculo, para $1 \leq i \leq r$.
2. Recorre T , usando los vectores de costo para determinar cuáles subárboles de T deben calcularse y colocarse en la memoria.
3. Recorre cada árbol usando los vectores de costo y las instrucciones asociadas para generar el código de destino final. El código para los árboles que se calculan en las ubicaciones de memoria se genera primero.

Cada una de estas fases puede implementarse para ejecutarse en el tiempo, en forma linealmente proporcional al tamaño del árbol de expresión.

El costo de calcular un nodo n incluye todas las instrucciones de carga y almacenamiento necesarias para evaluar S en el número dado de registros. También incluye el costo de calcular el operador en la raíz de S . El componente cero del vector de costos es el costo óptimo de calcular el subárbol S y colocar el resultado en la memoria. La propiedad de evaluación contigua asegura que pueda generarse un programa óptimo para S , al considerar las combinaciones de los programas óptimos sólo para los subárboles de la raíz de S . Esta restricción reduce el número de casos que hay que considerar.

Para poder calcular los costos $C[i]$ en el nodo n , vemos las instrucciones como reglas de rescritura de árboles, como en la sección 8.9. Considere cada plantilla E que coincide con el árbol de entrada en el nodo n . Al examinar los vectores de costo en los descendientes correspondientes de n , determine los costos de evaluar los operandos en las hojas de E . Para aquellos operandos de E que sean registros, considere todos los órdenes posibles en los que pueden evaluarse los correspondientes subárboles de T y colocarse en registros. En cada ordenamiento, el primer subárbol correspondiente a un operando de registro puede evaluarse mediante el uso de i registros disponibles, el segundo usando $i - 1$ registros, y así en lo sucesivo. Para tomar en cuenta el nodo n , agregue el costo de la instrucción asociada con la plantilla E . El valor $C[i]$ es entonces el costo mínimo sobre todos los posibles órdenes.

Los vectores de costos para todo el árbol T pueden calcularse de abajo hacia arriba en el tiempo, de una forma linealmente proporcional al número de nodos en T . Es conveniente almacenar en cada nodo la instrucción utilizada para lograr el mejor costo para $C[i]$, por cada valor de i . El menor costo en el vector para la raíz de T proporciona el mínimo costo de evaluar a T .

Ejemplo 8.28: Considere una máquina que tiene dos registros R0 y R1, y las siguientes instrucciones, cada una de un costo unitario:

LD Ri, Mj	// $Ri = Mj$
op Ri, Ri, Rj	// $Ri = Ri \ op \ Rj$
op Ri, Rj, Mj	// $Ri = Ri \ op \ Mj$
LD Ri, Rj	// $Ri = Rj$
ST Mi, Rj	// $Mi = Rj$

En estas instrucciones, Ri es R0 o R1, y Mj es una ubicación de memoria. El operador op corresponde a los operadores aritméticos.

Vamos a aplicar el algoritmo de programación dinámica para generar código óptimo para el árbol sintáctico de la figura 8.26. En la primera fase, calculamos los vectores de costos que se muestran en cada nodo. Para ilustrar este cálculo de costos, considere el vector de costos en la hoja **a**. $C[0]$, el costo de calcular **a** en la memoria, es 0 debido a que ya se encuentra ahí. $C[1]$, el costo de calcular **a** en un registro es 1, ya que podemos cargarla en un registro con la instrucción LD R0, **a**. $C[2]$, el costo de cargar **a** en un registro con dos registros disponibles es el mismo que con un registro disponible. Por lo tanto, el vector de costos en la hoja **a** es (0, 1, 1).

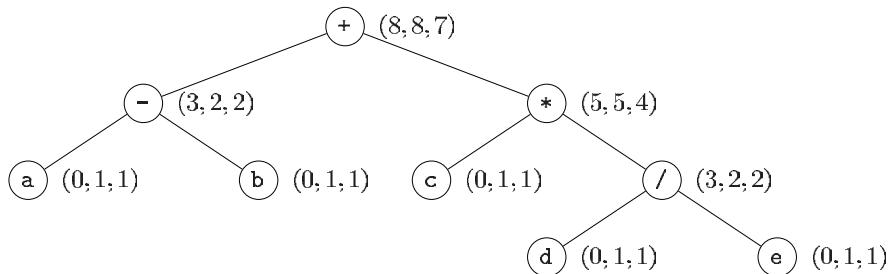


Figura 8.26: Árbol sintáctico para $(a-b)+c*(d/e)$, con un vector de costos en cada nodo

Considere el vector de costos en la raíz. Primero determinamos el costo mínimo de calcular la raíz con uno y dos registros disponibles. La instrucción de máquina ADD R0, R0, M coincide con la raíz, ya que está etiquetada con el operador $+$. Utilizando esta instrucción, el costo mínimo de evaluar la raíz con un registro disponible es el costo mínimo de calcular su subárbol derecho en la memoria, más el costo mínimo de calcular su subárbol izquierdo en el registro, más 1 por la instrucción; no existe otra manera. Los vectores de costos en los hijos derecho e izquierdo de la raíz muestran que el costo mínimo de calcular la raíz con un registro disponible es $5 + 2 + 1 = 8$.

Ahora considere el costo mínimo de evaluar la raíz con dos registros disponibles. Surgen tres casos, dependiendo de qué instrucción se utilice para calcular la raíz y en qué orden se evalúen los subárboles izquierdo y derecho de la raíz.

1. Calcule el subárbol izquierdo con dos registros disponibles en el registro R0, calcule el subárbol derecho con un registro disponible en el registro R1 y utilice la instrucción ADD R0, R0, R1 para calcular la raíz. Esta secuencia tiene un costo de $2 + 5 + 1 = 8$.
2. Calcule el subárbol derecho con dos registros disponibles en R1, calcule el subárbol izquierdo con un registro disponible en R0, y utilice la instrucción ADD R0, R0, R1. Esta secuencia tiene un costo de $4 + 2 + 1 = 7$.
3. Calcule el subárbol derecho en la ubicación de memoria M, calcule el subárbol izquierdo con dos registros disponibles en el registro R0, y utilice la instrucción ADD R0, R0, M. Esta secuencia tiene un costo de $5 + 2 + 1 = 8$.

La segunda elección produce el costo mínimo de 7.

El costo mínimo de calcular la raíz en la memoria se determina sumando uno al costo mínimo de calcular la raíz con todos los registros disponibles; es decir, calculamos la raíz en un registro y después almacenamos el resultado. Por lo tanto, el vector de costos en la raíz es (8, 8, 7).

A partir de los vectores de costos podemos construir con facilidad la secuencia de código, mediante un recorrido del árbol. Del árbol en la figura 8.26, suponiendo que hay dos registros disponibles, una secuencia de código óptimo sería:

```

LD  R0, c          // R0 = c
LD  R1, d          // R1 = d
DIV R1, R1, e      // R1 = R1 / e
MUL R0, R0, R1    // R0 = R0 * R1
LD  R1, a          // R1 = a
SUB R1, R1, b      // R1 = R1 - b
ADD R1, R1, R0     // R1 = R1 + R0

```

□

Las técnicas de programación dinámica se han utilizado en una variedad de compiladores, incluyendo la segunda versión del compilador C portable, PCC2. La técnica facilita la redestinación, debido a la capacidad de aplicación de la técnica de programación dinámica a una amplia clase de máquinas.

8.11.3 Ejercicios para la sección 8.11

Ejercicio 8.11.1: Aumente el esquema de rescritura de árboles en la figura 8.20 con costos, y utilice la programación dinámica y la coincidencia de árboles en la generación de código para las instrucciones del ejercicio 8.9.1.

!! Ejercicio 8.11.2: ¿Cómo extendería la programación dinámica para realizar la generación de código óptimo en GDAs?

8.12 Resumen del capítulo 8

- ◆ La *generación de código* es la fase final de un compilador. El generador de código asigna la representación intermedia producida por el front-end, o si hay una fase de optimización de código correspondiente al optimizador de código, al programa de destino.
- ◆ La *selección de instrucciones* es el proceso de elegir instrucciones del lenguaje de destino para cada instrucción representación intermedia.
- ◆ La *repartición de registros* es el proceso de decidir qué valores de representación intermedia mantener en los registros. La coloración de grafos es una técnica efectiva para realizar la repartición de registros en los compiladores.
- ◆ La *asignación de registros* es el proceso de decidir qué registro debe contener un valor de representación intermedia dado.
- ◆ Un *compilador redestinable* es uno que puede generar código para varios conjuntos de instrucciones.
- ◆ Una *máquina virtual* es un intérprete para un lenguaje intermedio de bytecodes, producido por lenguajes como Java y C#.
- ◆ Una *máquina CISC* es, por lo general, una máquina de dos direcciones, con pocos registros, varias clases de registros e instrucciones de longitud variable con modos de direccionamiento complejos.
- ◆ Una *máquina RISC* es, por lo general, una máquina de tres direcciones con muchos registros, en donde las operaciones se realizan en los registros.
- ◆ Un *bloque básico* es una secuencia máxima de instrucciones consecutivas de tres direcciones, en donde el flujo de control sólo puede entrar en la primera instrucción del bloque y salir en la última instrucción, sin detenerse o bifurcarse, tal vez excepto en la última instrucción del bloque básico.
- ◆ Un *grafo de flujo* es una representación gráfica de un programa, en el cual los nodos del gráfico son bloques básicos y las flechas del grafo muestran cómo puede fluir el control entre los bloques.
- ◆ Un *ciclo* en un grafo de flujo es una región con conexión sólida, con un solo punto de entrada conocido como el encabezado del ciclo.
- ◆ Una representación *GDA* de un bloque básico es un grafo dirigido acíclico, en el cual los nodos del GDA representan a las instrucciones dentro del bloque y cada hijo de un nodo corresponde a la instrucción, que es la última definición de un operando utilizado en la instrucción.
- ◆ Las *optimizaciones de mirilla (peephole)* son transformaciones para mejorar el código local que pueden aplicarse a un programa, por lo general, a través de una ventana deslizable.

- ◆ La *selección de instrucciones* puede realizarse mediante un proceso de rescritura de árboles, en el cual los patrones tipo árbol que corresponden a las instrucciones de máquina se utilizan para revestir un árbol sintáctico. Podemos asociar los costos con las reglas de rescritura de árboles y aplicar la programación dinámica para obtener un revestimiento óptimo para clases útiles de máquinas y expresiones.
- ◆ Un *número de Ershov* indica cuántos registros se necesitan para evaluar una expresión sin almacenar valores temporales.
- ◆ El *código de derrame* es una secuencia de instrucciones que almacena un valor en un registro en la memoria, con el fin de hacer espacio para guardar otro valor en ese registro.

8.13 Referencias para el capítulo 8

Muchas de las técnicas que se vieron en este capítulo tienen sus orígenes en los primeros compiladores. El algoritmo de etiquetado de Ershov apareció en 1958 [7]. Sethi y Ullman [16] utilizaron este etiquetado en un algoritmo que demostraron que generaba código óptimo para las expresiones aritméticas. Aho y Jonson [1] utilizaron la programación dinámica en la generación de código óptimo para los árboles de expresiones en máquinas CISC. Hennessy y Patterson [12] tiene una buena exposición acerca de la evolución de las arquitecturas de las máquinas CISC y RISC, y de las concesiones implicadas en el diseño de un buen conjunto de instrucciones.

Las arquitecturas RISC se hicieron populares después de 1990, aunque sus orígenes se remontan a las computadoras como la CDC 6600, que se produjo por primera vez en 1964. Muchas de las computadoras diseñadas antes de 1990 eran máquinas CISC, pero la mayoría de las computadoras de propósito general instaladas después de 1990 siguen siendo máquinas CISC, debido a que se basan en la arquitectura Intel 80x86 y sus descendientes, como el Pentium. La computadora Burroughs B5000 producida en 1963 fue una de las primeras máquinas basadas en pila.

Muchas de las heurísticas para la generación de código propuestas en este capítulo se han utilizado en varios compiladores. Nuestra estrategia de repartir un número fijo de registros para guardar variables durante la ejecución de un ciclo se utilizó en la implementación de Fortran H, por Lowry y Medlock [13].

También se han estudiado las técnicas de asignación y repartición eficiente de registros desde la aparición de las primeras computadoras. Cocke, Ershov [8] y Schwartz [15] propusieron la coloración de grafos como técnica de repartición de registros. Muchas variantes de los algoritmos de la coloración de grafos se han propuesto para la repartición de registros. Nuestro tratamiento de la coloración de gráficos se basa en Chaitin [3] [4]. Chow y Hennessy describen su algoritmo de la coloración basado en prioridades para la repartición de registros en [5]. En [6] podrá ver una explicación de las técnicas más recientes de la división de grafos y reescritura, para la asignación de registros.

Los generadores de analizadores léxicos y sintácticos incitaron el desarrollo de la selección de instrucciones dirigida por patrones. Glanville y Graham [11] utilizaron las técnicas de generación de analizadores sintácticos LR para la selección de instrucciones automatizada. Los generadores de código controlados por tablas evolucionaron en una variedad de herramientas de generación de código para la coincidencia de patrones tipo árbol [14]. Aho, Ganapathi y Tjiang [2]

combinaron técnicas eficientes para la coincidencia de patrones de árboles con la programación dinámica en la herramienta de generación de código *twig*. Fraser, Hanson y Proebsting [10] refinaron aún más estas ideas en su generador de generadores de código simple y eficiente.

1. Aho, A. V. y S. C. Johnson, “Optimal code generation for expression trees”, *J. ACM* **23**:3, pp. 488-501.
2. Aho, A. V., M. Ganapathi y S. W. K. Tjiang, “Code generation using tree matching and dynamic programming”, *ACM Trans. Programming Languages and Systems* **11**:4 (1989), pp. 491-516.
3. Chaitin, G. J., M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins y P. W. Markstein, “Register allocation via coloring”, *Computer Languages* **6**:1 (1981), pp. 47-57.
4. Chaitin, G. J., “Register allocation and spilling via graph coloring”, *ACM SIGPLAN Notices* **17**:6 (1982), pp. 201-207.
5. Chow, F. y J. L. Hennessy, “The priority-based coloring approach to register allocation”, *ACM Trans. Programming Languages and Systems* **12**:4 (1990), pp. 501-536.
6. Cooper, K. D. y L. Torczon, *Engineering a Compiler*, Morgan Kaufmann, San Francisco CA, 2004.
7. Ershov, A. P., “On programming of arithmetic operations”, *Comm. ACM* **1**:8 (1958), pp. 3-6. Además, *Comm. ACM* **1**:9 (1958), p. 16.
8. Ershov, A. P., *The Alpha Automatic Programming System*, Academic Press, Nueva York, 1971.
9. Fischer, C. N. y R. J. LeBlanc, *Crafting a Compiler with C*, Benjamin-Cummings, Redwood City, CA, 1991.
10. Fraser, C. W., D. R. Hanson y T. A. Proebsting, “Engineering a simple, efficient code generator generator”, *ACM Letters on Programming Languages and Systems* **1**:3 (1992), pp. 213-226.
11. Glanville, R. S. y S. L. Graham, “A new method for compiler code generation”, *Conf. Rec. Fifth ACM Symposium on Principles of Programming Languages* (1978), pp. 231-240.
12. Hennessy, J. L. y D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Tercera Edición, Morgan Kaufman, San Francisco, 2003.
13. Lowry, E. S. y C. W. Medlock, “Object code optimization”, *Comm. ACM* **12**:1 (1969), pp. 13-22.

14. Pelegri-Llopert, E. y S. L. Graham, “Optimal code generation for expressions trees: an application of BURS theory”, *Conf. Rec. Fifteenth Annual ACM Symposium on Principles of Programming Languages* (1988), pp. 294-308.
15. Schwartz, J. T., *On Programming: An Interim Report on the SETL Project*, Informe técnico, Courant Institute of Mathematical Sciences, Nueva York, 1973.
16. Sethi, R. y J. D. Ullman, “The generation of optimal code for arithmetic expressions”, *J. ACM* **17**:4 (1970), pp. 715-728.

Capítulo 9

Optimizaciones independientes de la máquina

Las construcciones de lenguajes de alto nivel pueden introducir una sobrecarga considerable en el tiempo de ejecución si traducimos directamente cada construcción de manera independiente en código máquina. Este capítulo habla de cómo eliminar muchas de estas ineficiencias. A la eliminación de instrucciones innecesarias en el código objeto, o la sustitución de una secuencia de instrucciones por una secuencia más rápida de instrucciones que haga lo mismo, se le conoce como “mejora de código” u “optimización de código”.

La optimización de código *local* (mejora de código dentro de un bloque básico) se presentó en la sección 8.5. Este capítulo maneja la optimización de código *global*, en donde las mejoras toman en cuenta lo que ocurre a través de los bloques básicos. En la sección 9.1 empezamos con una explicación sobre las principales oportunidades para la mejora de código.

La mayoría de las optimizaciones globales se basan en el *análisis del flujo de datos*, que son algoritmos para recopilar información acerca de un programa. Todos los resultados de los análisis del flujo de datos tienen la misma forma: para cada instrucción en el mismo, especifican cierta propiedad que debe aplicarse cada vez que se ejecute esa instrucción. Los análisis difieren en las propiedades que calculan. Por ejemplo, un análisis de propagación de constantes calcula, para cada punto en el programa, y para cada variable utilizada por el programa, si esa variable tiene un valor constante único en ese punto. Por ejemplo, esta información puede utilizarse para sustituir las referencias a las variables por valores constantes. Como otro ejemplo, un análisis del estado de vida determina, para cada punto en el programa, si es seguro que el valor que contiene una variable específica en ese punto se sobreescriba antes de leerlo. De ser así, no necesitamos preservar ese valor, ya sea en un registro o en una ubicación en memoria.

En la sección 9.2 presentamos el análisis del flujo de datos, incluyendo varios ejemplos importantes del tipo de información que recopilamos en forma global y después utilizamos para mejorar el código. La sección 9.3 muestra la idea general de un marco de trabajo del flujo de datos, del cual los análisis del flujo de datos en la sección 9.2 son casos especiales. En esencia podemos usar los mismos algoritmos para todas estas instancias del análisis del flujo de datos,

y podemos medir el rendimiento de estos algoritmos, incluso demostrar que sean correctos en todas las instancias. La sección 9.4 es un ejemplo del framework general que realiza un análisis más poderoso que los primeros ejemplos. Después, en la sección 9.5 consideraremos una técnica poderosa, conocida como “eliminación parcial de redundancia”, para optimizar la colocación de cada evaluación de una expresión en el programa. La solución a este problema requiere la solución de una variedad de distintos problemas del flujo de datos.

En la sección 9.6 nos encargaremos del descubrimiento y análisis de los ciclos en los programas. La identificación de ciclos conduce a otra familia de algoritmos para resolver los problemas del flujo de datos, los cuales se basan en la estructura jerárquica de los ciclos de un programa bien formado (“reducible”). Este método para el análisis del flujo de datos se cubre en la sección 9.7. Por último, la sección 9.8 utiliza el análisis jerárquico para eliminar las variables de inducción (en esencia, variables que cuentan el número de iteraciones alrededor de un ciclo). Esta mejora en el código es una de las más importantes que podemos hacer para los programas escritos en lenguajes de programación de uso común.

9.1 Las fuentes principales de optimización

La optimización de un compilador debe preservar la semántica del programa original. Excepto en circunstancias muy especiales, una vez que un programador elige e implementa un algoritmo específico, el compilador no puede comprender lo suficiente acerca del programa como para sustituirlo con un algoritmo considerablemente diferente y más eficiente. Un compilador sólo sabe cómo aplicar transformaciones semánticas de un nivel relativamente bajo, mediante hechos generales como identidades algebraicas del tipo $i + 0 = i$, o semántica de programas como el hecho de que al realizar la misma operación sobre los mismos valores se produce el mismo resultado.

9.1.1 Causas de redundancia

Existen muchas operaciones redundantes en un programa típico. Algunas veces, la redundancia está disponible en el nivel de origen. Por ejemplo, un programador puede encontrar que es más directo y conveniente volver a calcular cierto resultado, dejando al compilador la opción de reconocer que sólo es necesario un cálculo de ese tipo. Pero con más frecuencia, la redundancia es un efecto adicional de haber escrito un programa en lenguajes de alto nivel. En la mayoría de los lenguajes (distintos de C o C++, en donde se permite la aritmética de apuntadores), los programadores no tienen otra opción de referirse a los elementos de un arreglo o a los campos en un estructura a través de accesos como $A[i][j]$ o $X \rightarrow f1$.

A medida que se compila un programa, cada uno de estos accesos a estructuras de datos de alto nivel se expande en un número de operaciones aritméticas de bajo nivel, como el cálculo de la ubicación del (i, j) -ésimo elemento de una matriz A . A menudo, los accesos a la misma estructura de datos comparten muchas operaciones comunes de bajo nivel. Los programadores no están conscientes de estas operaciones y no pueden eliminar las redundancias por sí mismos. De hecho, es preferible desde una perspectiva de ingeniería de software que los programadores sólo accedan a los elementos de datos mediante sus nombres de alto nivel; los programas son más fáciles de escribir y, lo que es más importante, más fáciles de comprender y de desarrollar.

Al hacer que un compilador elimine las redundancias, obtenemos lo mejor de ambos mundos: los programas son tanto eficientes como fáciles de mantener.

9.1.2 Un ejemplo abierto: Quicksort

A continuación, vamos a usar el fragmento de un programa para ordenar datos, llamado *quicksort*, para ilustrar varias transformaciones importantes relacionadas con la mejora de código. El programa en C de la figura 9.1 se deriva de Sedgewick,¹ quien describió la optimización manual de un programa así. No vamos a hablar sobre todos los aspectos algorítmicos sutiles de este programa aquí y, por ejemplo, el hecho de que $a[0]$ debe contener el más pequeño de los elementos ordenados, y $a[max]$ el más grande.

```
void quicksort(int m, int n)
    /* ordena desde a[m] hasta a[n] en forma recursiva */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* aquí empieza el fragmento */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i]=a[j]; a[j] = x; /* intercambia a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* intercambia a[i], a[n] */
    /* aquí termina el fragmento */
    quicksort(m,j); quicksort(i+1,n);
}
```

Figura 9.1: Código en C para el algoritmo Quick Sort

Antes de poder optimizar las redundancias en los cálculos de direcciones, primero debemos descomponer las operaciones con direcciones en un programa en operaciones aritméticas de bajo nivel, para exponer las redundancias. Durante el resto de este capítulo, vamos a suponer que la representación intermedia consiste en instrucciones de tres direcciones, en donde se utilizan variables temporales para guardar todos los resultados de las expresiones intermedias. El código intermedio para el fragmento marcado del programa de la figura 9.1 se muestra en la figura 9.2.

En este ejemplo suponemos que los enteros ocupan cuatro bytes. La asignación $x = a[i]$ se traduce como en la sección 6.4.4 en las siguientes dos instrucciones de tres direcciones:

¹R. Sedgewick, “Implementing Quicksort Programs”, *Comm. ACM*, **21**, 1978, pp. 847-857.

(1) $i = m-1$	(16) $t7 = 4*i$
(2) $j = n$	(17) $t8 = 4*j$
(3) $t1 = 4*n$	(18) $t9 = a[t8]$
(4) $v = a[t1]$	(19) $a[t7] = t9$
(5) $i = i+1$	(20) $t10 = 4*j$
(6) $t2 = 4*i$	(21) $a[t10] = x$
(7) $t3 = a[t2]$	(22) $goto (5)$
(8) $if \ t3 < v \ goto \ (5)$	(23) $t11 = 4*i$
(9) $j = j-1$	(24) $x = a[t11]$
(10) $t4 = 4*j$	(25) $t12 = 4*i$
(11) $t5 = a[t4]$	(26) $t13 = 4*n$
(12) $if \ t5 > v \ goto \ (9)$	(27) $t14 = a[t13]$
(13) $if \ i >= j \ goto \ (23)$	(28) $a[t12] = t14$
(14) $t6 = 4*i$	(29) $t15 = 4*n$
(15) $x = a[t6]$	(30) $a[t15] = x$

Figura 9.2: Código de tres direcciones para el fragmento de código de la figura 9.1

```
t6 = 4*i
x = a[t6]
```

como se muestra en los pasos (14) y (15) de la figura 9.2. De manera similar, $a[j] = x$ se convierte en:

```
t10 = 4*j
a[t10] = x
```

en los pasos (20) y (21). Tenga en cuenta que todo acceso a un arreglo en el programa original se traduce en un par de pasos, que consisten en una multiplicación y una operación con subíndices de arreglos. Como resultado, este fragmento de programa corto se traduce en una secuencia bastante extensa de operaciones de tres direcciones.

La figura 9.3 es el grafo de flujo para el programa de la figura 9.2. El bloque B_1 es el nodo de entrada. Todos los saltos condicionales e incondicionales a las instrucciones en la figura 9.2 se han sustituido en la figura 9.3 por saltos hacia el bloque del cual las instrucciones son líderes, como en la sección 8.4. En la figura 9.3, hay tres ciclos. Los bloques B_2 y B_3 son bloques por sí mismos. Los bloques B_2 , B_3 , B_4 y B_5 forman en conjunto un ciclo, en donde B_2 es el único punto de entrada.

9.1.3 Transformaciones que preservan la semántica

Hay varias formas en las que un compilador puede mejorar un programa, sin cambiar la función que calcula. La eliminación de subexpresiones comunes, la propagación de copia, la eliminación de código muerto y el cálculo previo de constantes son ejemplos comunes de dichas transformaciones que preservan las funciones (o que *preservan la semántica*); consideraremos cada uno de estos ejemplos, uno a la vez.

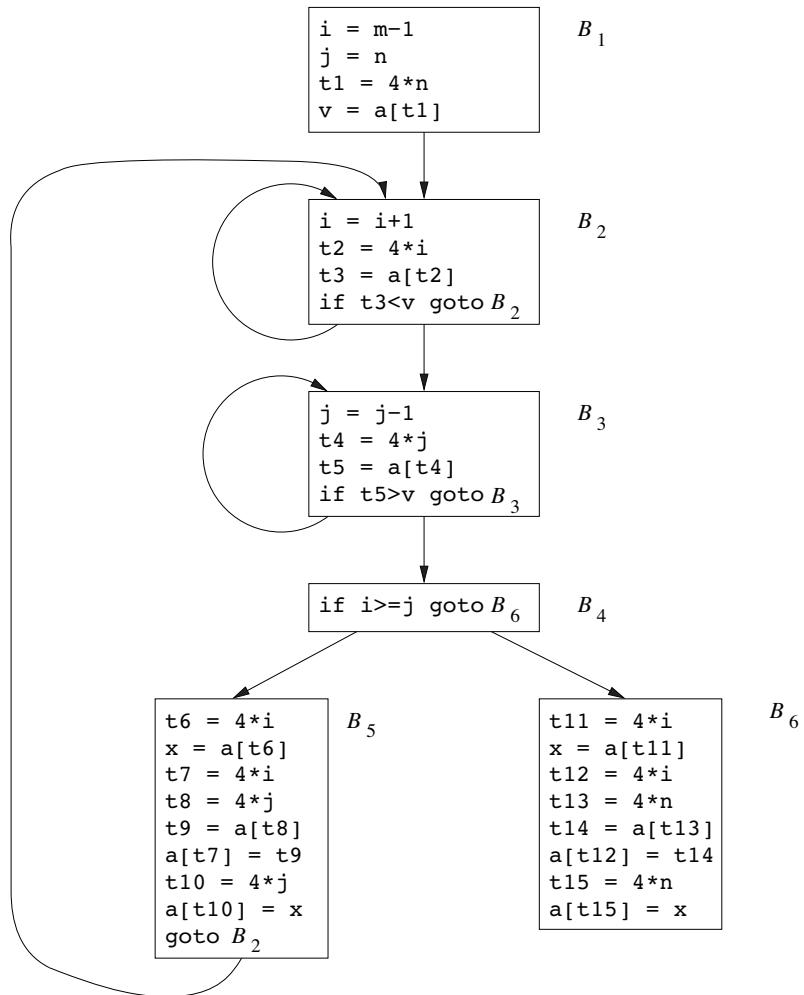


Figura 9.3: Grafo de flujo para el fragmento quicksort

Es común que un programa incluya varios cálculos del mismo valor, como un desplazamiento en un arreglo. Como dijimos en la sección 9.1.2, el programador no puede evitar algunos de estos cálculos duplicados, ya que se encuentran por debajo del nivel de detalle accesible dentro del lenguaje fuente. Por ejemplo, el bloque B_5 que se muestra la figura 9.4(a) vuelve a calcular $4 * i$ y $4 * j$, aunque el programador no solicitará de manera explícita ninguno de estos cálculos.

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

(a) Antes.

 B_5

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

 B_5

(b) Despues.

Figura 9.4: Eliminación de subexpresiones comunes locales

9.1.4 Subexpresiones comunes globales

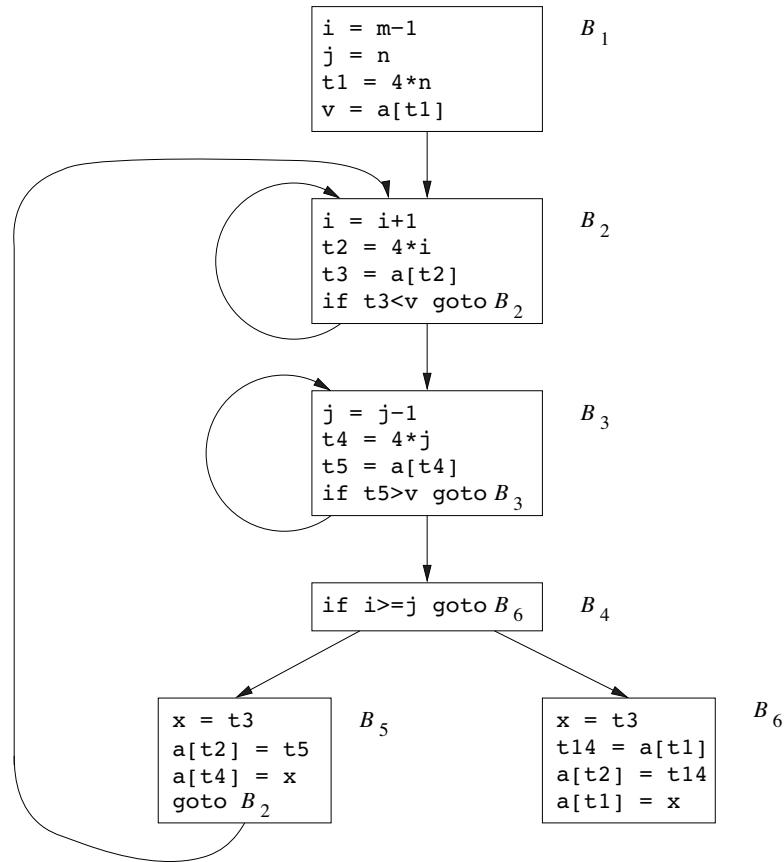
A una ocurrencia de una expresión E se le conoce como *subexpresión común* si E se calculó previamente y los valores de las variables en E no han cambiado desde ese cálculo previo. Evitamos volver a calcular E si podemos usar su valor calculado previamente; es decir, la variable x a la cual se asignó el cálculo anterior de E no ha cambiado en ese lapso.²

Ejemplo 9.1: Las asignaciones a $t7$ y $t10$ en la figura 9.4(a) calculan las subexpresiones comunes $4 * i$ y $4 * j$, respectivamente. Estos pasos se han eliminado en la figura 9.4(b), la cual utiliza a $t6$ en vez de $t7$, y $t8$ en vez de $t10$. \square

Ejemplo 9.2: La figura 9.5 muestra el resultado de la eliminación de subexpresiones comunes tanto globales como locales de los bloques B_5 y B_6 en el grafo de flujo de la figura 9.3. Primero hablaremos sobre la transformación de B_5 y después mencionaremos algunas sutilezas relacionadas con los arreglos.

Después de eliminar las subexpresiones comunes locales, B_5 sigue evaluando $4 * i$ y $4 * j$, como se muestra la figura 9.4(b). Ambas son subexpresiones comunes; en especial, las tres instrucciones

²Si x ha cambiado, aún puede ser posible reutilizar el cálculo de E si asignamos su valor a una nueva variable y , así como a x , y utilizamos el valor de y en lugar de volver a calcular a E .

Figura 9.5: B_5 y B_6 después de la eliminación de subexpresiones comunes

```

t8 = 4*j
t9 = a[t8]
a[t8] = x

```

en B_5 pueden sustituirse por

```

t9 = a[t4]
a[t4] = x

```

utilizando el valor de $t4$ que se calculó en el bloque B_3 . En la figura 9.5, observe que a medida que el control pasa de la evaluación de $4 * j$ en B_3 a B_5 , no se modifican j ni $t4$, por lo que se puede usar $t4$ si se necesita $4 * j$.

Otra subexpresión común surge en B_5 , cuando $t4$ sustituye a $t8$. La nueva expresión $a[t4]$ corresponde al valor de $a[j]$ en el nivel de origen. No sólo j retiene su valor cuando el control sale de B_3 y después entra a B_5 , sino que $a[j]$, un valor que se calcula en una variable temporal $t5$,

también retiene su valor, ya que no hay asignaciones a los elementos del arreglo a en ese lapso. Por lo tanto, las instrucciones

```
t9 = a[t4]
a[t6] = t9
```

en B_5 pueden sustituirse por

```
a[t6] = t5
```

De manera análoga, el valor asignado a x en el bloque B_5 de la figura 9.4(b) se ve como el mismo valor asignado a $t3$ en el bloque B_2 . El bloque B_5 en la figura 9.5 es el resultado de eliminar subexpresiones comunes que corresponden a los valores de las expresiones del nivel de origen $a[i]$ y $a[j]$ de B_5 en la figura 9.4(b). En la figura 9.5 se ha realizado una serie similar de transformaciones a B_6 .

La expresión $a[t1]$ en los bloques B_1 y B_6 de la figura 9.5 no se consideran una subexpresión común, aunque $t1$ puede usarse en ambos lugares. Una vez que el control sale de B_1 y antes de que llegue a B_6 , puede pasar a través de B_5 , en donde hay asignaciones para a . Por ende, $a[t1]$ tal vez no tenga el mismo valor al llegar a B_6 que tenía al salir de B_1 , y no es seguro tratar a $a[t1]$ como una subexpresión común. \square

9.1.5 Propagación de copias

El bloque B_5 en la figura 9.5 puede mejorarse aún más mediante la eliminación de x , mediante dos nuevas transformaciones. Una se relaciona con las asignaciones de la forma $u = v$, conocidas como *instrucciones de copia*, o simplemente *copias*. Si hubiéramos entrado en más detalles en el ejemplo 9.2, las copias hubieran surgido mucho antes, debido a que el algoritmo normal para eliminar subexpresiones comunes las introduce, al igual que varios otros algoritmos.

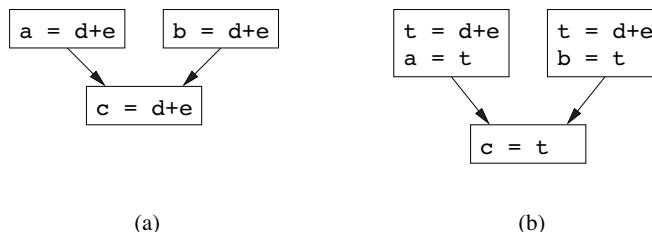


Figura 9.6: Copias introducidas durante la eliminación de subexpresiones comunes

Ejemplo 9.3: Para poder eliminar la subexpresión común de la instrucción $c = d+e$ en la figura 9.6(a), debemos utilizar una nueva variable t para guardar el valor de $d + e$. En la figura 9.6(b) se asigna a c el valor de la variable t , en vez del valor de la expresión $d + e$. Como el control puede llegar a $c = d+e$ ya sea después de la asignación a a , o después de la asignación a b , sería incorrecto sustituir $c = d+e$ por $c = a$ o por $c = b$. \square

La idea de la transformación por propagación de copias es utilizar v para u , siempre que sea posible después de la instrucción de copia $u = v$. Por ejemplo, la asignación $x = t3$ en el bloque B_5 de la figura 9.5 es una copia. La propagación de copia que se aplica a B_5 produce el código de la figura 9.7. Este cambio tal vez no parezca ser una mejora pero, como veremos en la sección 9.1.6, nos da la oportunidad de eliminar la asignación a x .

```

x = t3
a[t2] = t5
a[t4] = t3
goto B2

```

Figura 9.7: Bloque básico B_5 después de la propagación de copias

9.1.6 Eliminación de código muerto

Una variable está *viva* en un punto en el programa, si su valor puede utilizarse más adelante; en caso contrario, está *muerta* en ese punto. Una idea relacionada es el *código muerto* (o *inútil*): instrucciones que calculan valores que nunca se utilizan. Aunque es poco probable que el programador introduzca código muerto de manera intencional, puede aparecer como resultado de las transformaciones anteriores.

Ejemplo 9.4: Suponga que `debug` se establece a `TRUE` o `FALSE` en varios puntos en el programa, y que se utiliza en instrucciones como:

```
if (debug) print ...
```

Puede ser posible para el compilador deducir que cada vez que el programa llega a esta instrucción, el valor de `debug` es `FALSE`. Por lo general, se debe a que hay una instrucción en particular:

```
debug = FALSE
```

que debe ser la última asignación a `debug`, antes de cualquier evaluación del valor de `debug`, sin importar que secuencia de bifurcaciones tome realmente el programa. Si la propagación de copia sustituye a `debug` por `FALSE`, entonces la instrucción de impresión está muerta, ya que no se puede llegar a ella. Podemos eliminar del código objeto tanto la operación de evaluación como la de impresión. En forma más general, al proceso de deducir en tiempo de compilación que el valor de una expresión es una constante, y utilizar mejor esa constante se le conoce como *cálculo previo de constantes*. \square

Una ventaja de la propagación de copias es que a menudo convierte la instrucción de copia en código muerto. Por ejemplo, la propagación de copias seguida de la eliminación de código muerto elimina la asignación a x y transforma el código de la figura 9.7 en:

```

a[t2] = t5
a[t4] = t3
goto B2

```

Este código es una mejora más al bloque B_5 de la figura 9.5.

9.1.7 Movimiento de código

Los ciclos son un lugar muy importante para las optimizaciones, en especial los ciclos internos en donde los programas tienden a invertir la mayor parte su tiempo. El tiempo de ejecución de un programa puede mejorarse si reducimos el número de instrucciones en un ciclo interno, incluso si incrementamos la cantidad de código fuera de ese ciclo.

Una modificación importante que reduce la cantidad de código en un ciclo es el *movimiento de código*. Esta transformación toma una expresión que produce el mismo resultado sin importar el número de veces que se ejecute un ciclo (el *cálculo de una invariante de ciclo*) y evalúa la expresión antes del ciclo. Observe que la noción “antes del ciclo” asume la existencia de una entrada para el ciclo, es decir, un bloque básico al cual se dirigen todos los saltos desde el exterior del ciclo (de la sección 8.4.5).

Ejemplo 9.5: La evaluación de $limite - 2$ es un cálculo de una invariante de ciclo en la siguiente instrucción while:

```
while (i <= limite-2) /* instrucción que no cambia el límite */
```

El movimiento de código producirá el siguiente código equivalente:

```

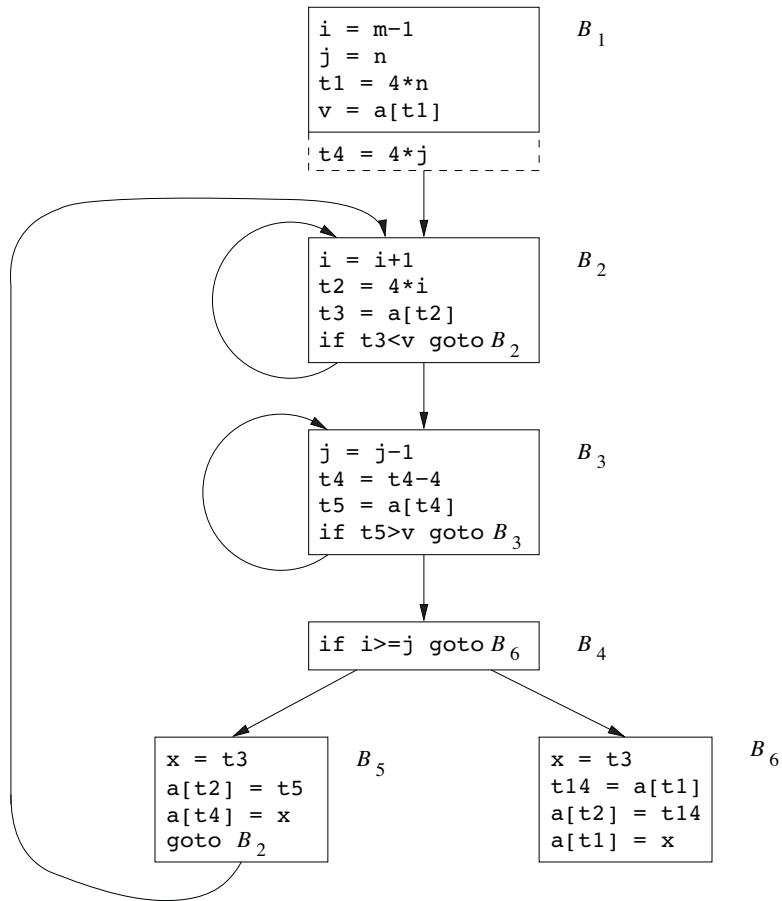
t = limite-2
while (i <= t) /* instrucción que no cambia límite ni t */

```

Ahora, el cálculo de $limite - 2$ se realizó una vez, antes de entrar al ciclo. Anteriormente, habría $n + 1$ cálculos de $limite - 2$ si iteráramos el cuerpo del ciclo n veces. \square

9.1.8 Variables de inducción y reducción en fuerza

Otra optimización importante es la de buscar variables de inducción en ciclos y optimizar su cálculo. Se dice que una variable x es una “variable de inducción” si hay una constante c positiva o negativa tal que cada vez que se asigne x , su valor aumente en base a c . Por ejemplo, i y $t2$ son las variables de inducción en el ciclo que contiene a B_2 de la figura 9.5. Las variables de inducción pueden calcularse con un solo incremento (suma o resta) por cada iteración de un ciclo. La transformación de sustituir una operación costosa, como la multiplicación, por una más económica, como la suma, se conoce como *reducción de fuerza*. Pero las variables de inducción no sólo nos permiten algunas veces realizar una reducción de fuerza; a menudo es posible eliminar todos excepto un grupo de variables de inducción, cuyos valores permanezcan en un paso bloqueado a medida que recorremos el ciclo.

Figura 9.8: Reducción de fuerza aplicada a $4 * j$ en el bloque B_3

Al procesar los ciclos, es útil trabajar de “adentro hacia fuera”; es decir, empezamos con los ciclos internos y pasamos a los ciclos circundantes, que cada vez se hacen más grandes. Por ende, vamos a ver cómo se aplica esta optimización a nuestro ejemplo de ordenamiento rápido (quicksort), empezando con uno de los ciclos más internos: B_3 por sí solo. Observe que los valores de j y $t4$ permanecen en paso bloqueado; cada vez que el valor de j se decrementa en 1, el valor de $t4$ se decrementa en 4, ya que $4 * j$ se asigna a $t4$. Estas variables, j y $t4$, forman, por lo tanto, un buen ejemplo de un par de variables de inducción.

Cuando hay dos o más variables de inducción en un ciclo, puede ser posible deshacerse de todas menos de una. Para el ciclo interno de B_3 en la figura 9.5, no podemos deshacernos de j o $t4$ por completo; $t4$ se utiliza en B_3 y j se utiliza en B_4 . Sin embargo, podemos ilustrar la reducción en fuerza y una parte del proceso de eliminación de variables de inducción. En un momento dado, j se eliminará cuando lleguemos al ciclo externo que consiste en los bloques B_2 , B_3 , B_4 y B_5 .

Ejemplo 9.6: Como es seguro que la relación $t4 = 4 * j$ sea válida después de la asignación a $t4$ en la figura 9.5, y que $t4$ no se modifica en ningún otro lado del ciclo interno alrededor de B_3 , resulta ser que justo después de la instrucción $j = j-1$, la relación $t4 = 4 * j + 4$ debe ser válida. Por lo tanto, podemos sustituir la asignación $t4 = 4*j$ por $t4 = t4-4$. El único problema es que $t4$ no tiene un valor cuando entramos al bloque B_3 por primera vez.

Como debemos mantener la relación $t4 = 4 * j$ al entrar al bloque B_3 , colocamos una inicialización de $t4$ al final del bloque en donde se inicializa la misma j , lo cual se muestra mediante la parte adicional en líneas punteadas en el bloque B_1 de la figura 9.8. Aunque hemos agregado una instrucción más, la cual se ejecuta una vez en el bloque B_1 , la sustitución de una multiplicación por una resta aumentará la velocidad del código objeto, si la multiplicación ocupa más tiempo que la suma o una resta, como es el caso en muchas máquinas. \square

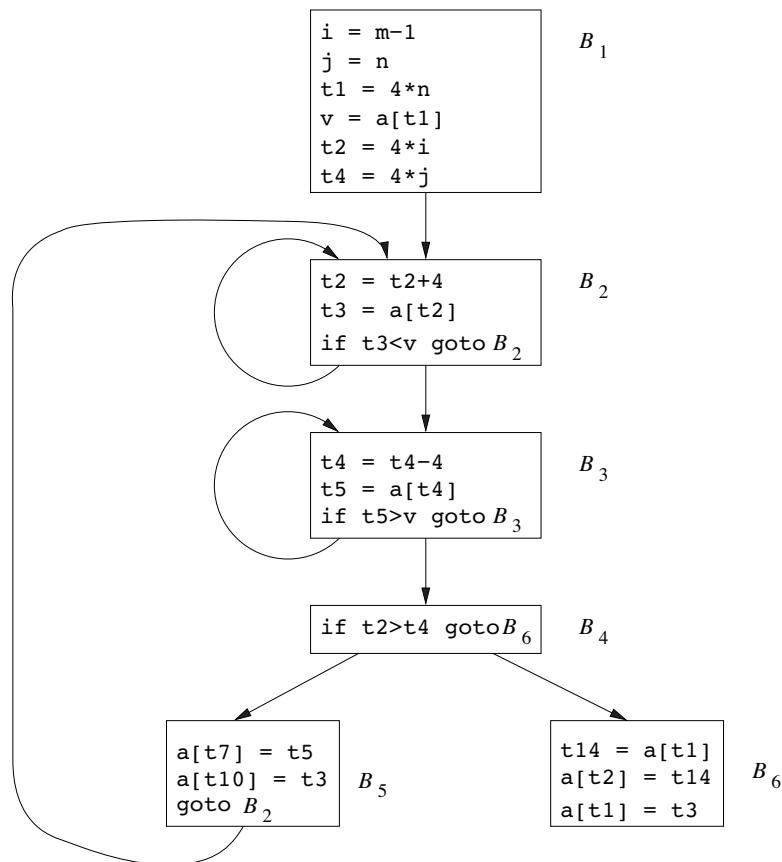


Figura 9.9: Grafo de flujo después de eliminar las variables de inducción

Concluimos esta sección con una instancia más de eliminación de variables de inducción. Este ejemplo trata a i y a j en el contexto del ciclo externo que contiene a B_2 , B_3 , B_4 y B_5 .

Ejemplo 9.7: Una vez que se aplica la reducción en fuerza a los ciclos internos alrededor de B_2 y B_3 , el único uso de i y j es determinar el resultado de la evaluación en el bloque B_4 . Sabemos que los valores de i y $t2$ satisfacen la relación $t2 = 4 * i$, mientras que los valores de j y $t4$ satisfacen la relación $t4 = 4 * j$. Por ende, la prueba $t2 \geq t4$ puede sustituir a $i \geq j$. Una vez que se realiza esta sustitución, las variables i en el bloque B_2 y j en el bloque B_3 se convierten en variables muertas, y las asignaciones a ellas en estos bloques se convierten en código muerto que puede eliminarse. El grafo de flujo resultante se muestra en la figura 9.9. \square

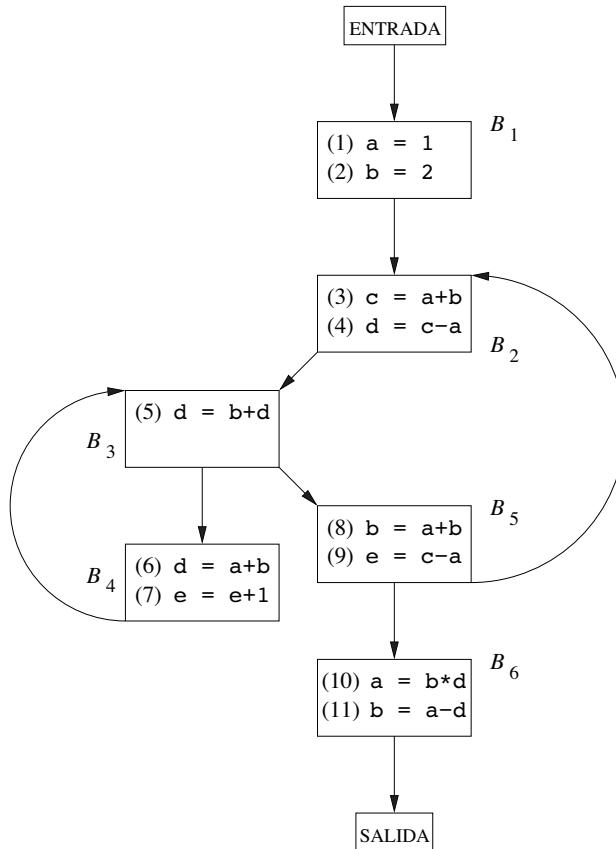


Figura 9.10: Grafo de flujo para el ejercicio 9.1.1

Las transformaciones para mejorar el código que hemos visto han sido efectivas. En la figura 9.9, los números de las instrucciones en los bloques B_2 y B_3 se han reducido de 4 a 3, en comparación con el grafo de flujo original la figura 9.3. En B_5 , el número se ha reducido de 9 a 3, y en B_6 de 8 a 3.

Es cierto que B_1 ha aumentado de cuatro instrucciones a seis, pero B_1 se ejecuta sólo una vez en el fragmento, por lo que el tiempo total de ejecución casi no se ve afectado por el tamaño de B_1 .

9.1.9 Ejercicios para la sección 9.1

Ejercicio 9.1.1: Para el grafo de flujo de la figura 9.10:

- Identifique los ciclos del grafo de flujo.
- Las instrucciones (1) y (2) en B_1 son instrucciones de copia, en las cuales a y b son valores constantes dados. ¿Para qué usos de a y b podemos realizar la propagación de copias y sustituir estos usos de variables por usos de constantes? Haga esto cada vez que sea posible.
- Identifique cualquier subexpresión común global para cada ciclo.
- Identifique cualquier variable de inducción para cada ciclo. Asegúrese de tomar en cuenta todas las constantes introducidas en (b).
- Identifique cualquier cálculo de una invariante de ciclo para cada uno de ellos.

Ejercicio 9.1.2: Aplique las transformaciones de esta sección al grafo de flujo de la figura 8.9.

Ejercicio 9.1.3: Aplique las transformaciones de esta sección a sus grafos de flujo del (a) ejercicio 8.4.1; (b) ejercicio 8.4.2.

Ejercicio 9.1.4: En la figura 9.11 hay código intermedio para calcular el producto punto de dos vectores A y B . Optimice este código, eliminando las subexpresiones comunes, realizando la reducción en fuerza sobre las variables de inducción, y eliminando todas las variables de inducción que pueda.

```

dp = 0.
i = 0
L: t1 = i*8
  t2 = A[t1]
  t3 = i*8
  t4 = B[t3]
  t5 = t2*t4
  dp = dp+t5
  i = i+1
  if i<n goto L
  
```

Figura 9.11: Código intermedio para calcular el producto punto

9.2 Introducción al análisis del flujo de datos

Todas las optimizaciones que se presentan en la sección 9.1 dependen del *análisis del flujo de datos*. El “análisis del flujo de datos” se refiere a un cuerpo de técnicas que derivan información acerca del flujo de datos a lo largo de los caminos de ejecución de los programas. Por ejemplo, una manera de implementar la eliminación de subexpresiones comunes globales requiere que determinemos si dos expresiones textualmente idénticas se evalúan con el mismo valor, a lo largo de cualquier camino de ejecución posible del programa. Como otro ejemplo, si el resultado de una asignación no se utiliza a lo largo de cualquier camino de ejecución subsiguiente, entonces podemos eliminar esa asignación como si fuera código muerto. El análisis del flujo de datos puede responder a ésta y muchas otras preguntas importantes.

9.2.1 La abstracción del flujo de datos

De acuerdo con la sección 1.6.2, la ejecución de un programa puede verse como una serie de transformaciones del estado del programa, qué consiste en los valores de todas las variables en el programa, incluyendo aquellas que están asociadas con los marcos de pila debajo del tope de la pila en tiempo de ejecución. Cada ejecución de una instrucción de código intermedio transforma un estado de entrada en un nuevo estado de salida. El estado de entrada se asocia con el *punto del programa antes* de la instrucción y el de salida se asocia con el *punto del programa después* de la instrucción.

Al analizar el comportamiento de un programa, debemos considerar todas las posibles secuencias de punto del programa (“caminos”) a través de un grafo de flujo que la ejecución del programa puede tomar. Después extraemos, de los posibles estados del programa en cada punto, la información que necesitamos para el problema específico de análisis del flujo de datos que deseamos resolver. En análisis más complejos, debemos considerar caminos que saltan entre los grafos de flujo para varios procedimientos, a medida que se ejecutan las llamadas y los retornos. Sin embargo, para comenzar nuestro estudio, vamos a concentrados en los caminos a través de un solo grafo de flujo para un solo procedimiento.

Vamos a ver lo que nos indica el grafo de flujo acerca de los posibles caminos de ejecución.

- Dentro de un bloque básico, el punto del programa después de la instrucción es el mismo que el punto del programa antes de la siguiente instrucción.
- Si hay una flecha del bloque B_1 al bloque B_2 , entonces el punto del programa después de la última instrucción de B_1 puede ir seguido inmediatamente del punto del programa antes de la primera instrucción de B_2 .

Por ende, podemos definir un *camino de ejecución* (o sólo *camino*) del punto p_1 al punto p_n como una secuencia de puntos p_1, p_2, \dots, p_n tal que para cada $i = 1, 2, \dots, n - 1$, se cumpla uno de los siguientes puntos:

1. p_i es el punto que va justo antes de una instrucción y p_{i+1} es el punto que va justo después de esa misma instrucción.
2. p_i es el final de un bloque y p_{i+1} es el inicio de un bloque sucesor.

En general, hay un número infinito de caminos posibles de ejecución a través de un programa, y no hay un límite superior finito en cuanto a la longitud de un camino de ejecución. Los análisis de los programas sintetizan todos los estados posibles de un programa que pueden ocurrir en un punto en el programa con un conjunto finito de hechos. Los distintos análisis pueden elegir abstraer información distinta y, en general, ningún análisis es necesariamente una representación perfecta del estado.

Ejemplo 9.8: Incluso hasta el programa simple de la figura 9.12 describe un número ilimitado de caminos de ejecución. Sin entrar al ciclo en absoluto, el camino de ejecución completo más corto consiste en los puntos (1, 2, 3, 4, 9) del programa. El siguiente camino más corto ejecutó una iteración del ciclo y consiste en los puntos (1, 2, 3, 4, 5, 6, 7, 8, 3, 4, 9). Por ejemplo, sabemos que la primera vez que se ejecuta el punto (5) del programa, el valor de a se debe a la definición d_1 . Decimos que d_1 *llega* al punto (5) en la primera iteración. En las iteraciones siguientes, d_3 llega al punto (5) y el valor de a es 243.

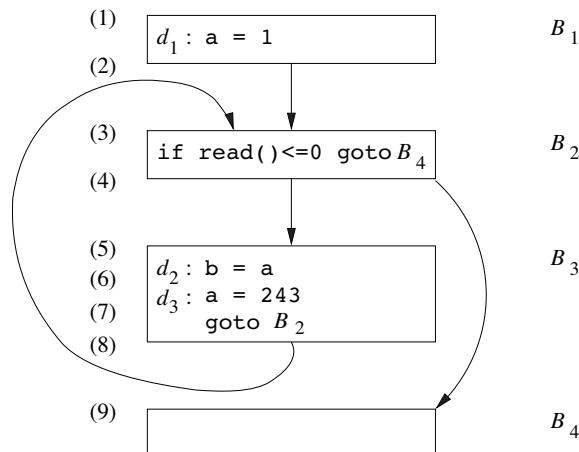


Figura 9.12: Programa de ejemplo que ilustra la abstracción del flujo de datos

En general, no es posible llevar el registro de todos los estados del programa para todas los caminos posibles. En el análisis del flujo de datos, no diferenciamos entre los caminos que se toman para llegar a un punto del programa. Tampoco llevamos el registro de los estados completos; en vez de ello, abstraemos ciertos detalles, y mantenemos sólo los datos que necesitamos para el propósito del análisis. Dos ejemplos ilustrarán cómo los estados del mismo programa pueden llevar a que se abstraiga información distinta en un punto.

1. Para ayudar a los usuarios a depurar sus programas, tal vez sea conveniente averiguar cuáles son todos los valores que una variable puede tener en un punto del programa, en donde pueden definirse esos valores. Por ejemplo, podemos sintetizar todos los estados del programa en el punto (5) al decir que el valor de a es uno de $\{1, 243\}$, y que puede definirse mediante uno de $\{d_1, d_3\}$. Las definiciones que *pueden* llegar a un punto del programa a lo largo de cierto camino se conocen como *definiciones de alcance*.

2. Suponga, por el contrario, que estamos interesados en implementar el cálculo previo de constantes. Si podemos llegar a un uso de la variable x sólo mediante una definición, y esa definición asigna una constante a x , entonces podemos sólo sustituir x por esa constante. Si, por otra parte, varias definiciones de x pueden llegar a un solo punto del programa, entonces no podemos realizar el cálculo previo de constantes sobre x . En consecuencia, para el cálculo previo de constantes queremos encontrar las definiciones que sean la única definición de su variable para llegar a un punto dado del programa, sin importar qué camino de ejecución se tome. Para el punto (5) de la figura 9.12, no hay una definición que *deba* ser la definición de a en ese punto, por lo que este conjunto está vacío para a en el punto (5). Incluso si una variable tiene una definición única en un punto, esa definición debe asignar una constante a la variable. Por ende, sólo podemos describir ciertas variables como “no constantes”, en vez de recolectar todos sus valores o definiciones posibles.

Así, podemos ver que la misma información puede resumirse en forma distinta, dependiendo del propósito del análisis. \square

9.2.2 El esquema del análisis del flujo de datos

En cada aplicación del análisis del flujo de datos, asociamos con cada punto del programa un *valor del flujo de datos* que representa una abstracción del conjunto de todos los posibles estados del programa que pueden observarse para ese punto. El conjunto de posibles valores del flujo de datos es el *dominio* para esta aplicación. Por ejemplo, el dominio de valores del flujo de datos para llegar a las definiciones es el conjunto de todos los subconjuntos de definiciones en el programa. Un valor de flujo de datos específico es un conjunto de definiciones, y deseamos asociar con cada punto del programa el conjunto exacto de definiciones que pueden llegar a ese punto. Como dijimos antes, la elección de la abstracción depende del objetivo del análisis; para que sea eficiente, sólo llevamos el registro de la información relevante.

Denotamos los valores del flujo de datos antes y después de cada instrucción s mediante $\text{ENT}[s]$ y $\text{SAL}[s]$, respectivamente. El *problema del flujo de datos* es encontrar una solución para un conjunto de restricciones sobre los valores $\text{ENT}[s]$ y $\text{SAL}[s]$, para todas las instrucciones s . Existen dos conjuntos de restricciones: las que están basadas en la semántica de las instrucciones (“funciones de transferencia”) y las que están basadas en el flujo de control.

Funciones de transferencia

Los valores del flujo de datos antes y después de una instrucción se restringen mediante la semántica de la instrucción. Por ejemplo, suponga que nuestro análisis del flujo de datos implica el determinar el valor constante de las variables en los puntos. Si la variable a tiene el valor v antes de ejecutar la instrucción $b = a$, entonces tanto a como b tendrán el valor v después de la instrucción. Esta relación entre los valores del flujo de datos antes y después de la instrucción de asignación se conoce como una *función de transferencia*.

Las funciones de transferencia son de dos tipos: la información puede prepararse hacia delante, a lo largo de los caminos de ejecución, o puede fluir hacia atrás, hacia arriba de los caminos de ejecución. En un problema de flujo hacia delante, la función de transferencia de una

instrucción s que, por lo general, denotamos como f_s , recibe el valor del flujo de datos antes de la instrucción y produce un nuevo valor del flujo de datos después de la instrucción. Es decir,

$$\text{SAL}[s] = f_s(\text{ENT}[s]).$$

Por el contrario, en un problema de flujo hacia atrás, la función de transferencia f_s para la instrucción s convierte un valor de flujo de datos después de la instrucción a un nuevo valor del flujo de datos antes de la instrucción. Es decir,

$$\text{ENT}[s] = f_s(\text{SAL}[s]).$$

Restricciones del flujo de control

El segundo conjunto de restricciones en los valores del flujo de datos se deriva del flujo de control. Dentro de un bloque básico, el flujo de control es simple. Si un bloque B consiste en las instrucciones s_1, s_2, \dots, s_n en ese orden, entonces el valor del flujo de control fuera de s_i es el mismo que el valor de flujo de control que entra a s_{i+1} . Es decir,

$$\text{ENT}[s_{i+1}] = \text{SAL}[s_i] \text{ para todas las } i = 1, 2, \dots, n-1.$$

Sin embargo, las aristas del flujo de control entre los bloques básicos crean restricciones más complejas entre la última instrucción de un bloque básico y la primera instrucción del siguiente bloque. Por ejemplo, si nos interesa recolectar todas las definiciones que pueden llegar a un punto del programa, entonces el conjunto de definiciones que llega a la instrucción líder de un bloque básico es la unión de las definiciones después de las últimas instrucciones de cada uno de los bloques predecesores. La siguiente sección proporciona los detalles de cómo fluyen los datos entre los bloques.

9.2.3 Esquemas del flujo de datos en bloques básicos

Mientras que, técnicamente, un esquema de flujo de datos involucra a los valores del flujo de datos en cada punto del programa, podemos ahorrar tiempo y espacio al reconocer que lo que ocurre dentro de un bloque es, por lo general, muy simple. El control fluye desde el inicio hasta el final del bloque, sin interrupción ni bifurcación. Por ende, podemos reformular el esquema en términos de los valores del flujo de datos que entran y salen de los bloques. Denotamos los valores del flujo de datos justo antes y justo después de cada bloque básico B mediante $\text{ENT}[B]$ y $\text{SAL}[B]$, respectivamente. Las restricciones que involucran a $\text{ENT}[B]$ y $\text{SAL}[B]$ pueden derivarse de aquellas que involucran a $\text{ENT}[s]$ y $\text{SAL}[s]$ para las diversas instrucciones s en B , como se muestra a continuación.

Suponga que el bloque B consiste en las instrucciones s_1, \dots, s_n , en ese orden. Si s_1 es la primera instrucción del bloque básico B , entonces $\text{ENT}[B] = \text{ENT}[s_1]$. De manera similar, si s_n es la última instrucción del bloque básico B , entonces $\text{SAL}[B] = \text{SAL}[s_n]$. La función de transferencia de un bloque básico B , que denotamos como f_B , puede derivarse mediante la composición de las funciones de transferencia de las instrucciones del bloque. Es decir, dejamos que f_{s_i} sea la función de transferencia de la instrucción s_i . Entonces $f_B = f_{s_n} \circ \dots \circ f_{s_2} \circ f_{s_1}$. La relación entre el inicio y el final del bloque es:

$$\text{SAL}[B] = f_B(\text{ENT}[B]).$$

Las restricciones debidas al flujo de control entre los bloques básicos pueden rescribirse con facilidad mediante la sustitución de $\text{ENT}[B]$ y $\text{SAL}[B]$ por $\text{ENT}[s_1]$ y $\text{SAL}[s_n]$, respectivamente. Por ejemplo, si los valores del flujo de datos son información acerca de los conjuntos de constantes que *pueden* asignarse una variable, entonces tenemos un problema de flujo hacia delante, en el cual:

$$\text{ENT}[B] = \bigcup_{P \text{ un predecesor de } B} \text{SAL}[P].$$

Cuando el flujo de datos es hacia atrás, como pronto veremos en el análisis de variables vivas, las ecuaciones son similares, pero con los papeles de los valores ENT y SAL invertidos. Es decir,

$$\begin{aligned}\text{ENT}[B] &= f_B(\text{SAL}[B]) \\ \text{SAL}[B] &= \bigcup_{S \text{ un sucesor de } B} \text{ENT}[S].\end{aligned}$$

A diferencia de las ecuaciones aritméticas lineales, las ecuaciones del flujo de datos, por lo general, no tienen una solución única. Nuestro objetivo es encontrar la solución más “precisa” que satisfaga los dos conjuntos de restricciones: del flujo de control y de transferencia. Es decir, necesitamos una solución que respalde las mejoras de código válido, pero que no justifique las transformaciones inseguras: aquellas que cambian lo que el programa calcula. En breve hablaremos sobre esta cuestión, en el recuadro titulado “Estimación conservadora” y con más detalles en la sección 9.3.4. En las siguientes subsecciones, hablaremos sobre algunos de los ejemplos más importantes de los problemas que pueden resolverse mediante el análisis del flujo de datos.

9.2.4 Definiciones de alcance

“Las definiciones de alcance” es uno de los esquemas del flujo de datos más útiles y comunes. Al conocer en qué parte de un programa puede haberse definido cada variable x cuando el control llega al punto p , podemos determinar muchas cosas acerca de x . Para sólo dos ejemplos, un compilador sabe entonces si x es una constante en el punto p , y un depurador puede saber si es posible que x sea una variable indefinida, en caso de que x se utilice en p .

Decimos que una definición d *llega* a un punto p si hay un camino desde el punto que va justo después de d hacia p , de tal forma que d no se “elimine” a lo largo de ese camino. *Eliminamos* una definición de una variable x si hay cualquier otra definición de x en cualquier parte a lo largo del camino.³ De manera intuitiva, si una definición d de cierta variable x llega al punto p , entonces d podría ser el lugar en el que se definió por última vez el valor de x utilizado en p .

Una definición de una variable x es una instrucción que asigna, o puede asignar, un valor a x . Los parámetros de los procedimientos, los accesos a los arreglos y las referencias indirectas pueden tener alias, y no es fácil saber si una instrucción está haciendo referencia a una variable x en especial. El análisis del programa debe ser conservador; si no sabemos si una instrucción s está asignando un valor a x , debemos asumir que *puede* asignarle un valor; es decir, la variable x después

³Observe que el camino puede tener ciclos, por lo que podríamos llegar a otra ocurrencia de d a lo largo del camino, lo cual no “elimina” a d .

Detección de los posibles usos antes de la definición

He aquí cómo utilizamos una solución al problema de las definiciones de alcance para detectar los usos antes de la definición. El truco es introducir una definición falsa para cada variable x en la entrada al grafo de flujo. Si la definición falsa de x llega a un punto p en donde podría usarse x , entonces podría presentarse la oportunidad de usar x antes de la definición. Tenga en cuenta que nunca podremos estar completamente seguros de que el programa tiene un error, ya que puede haber alguna razón, que posiblemente involucre el uso de un argumento lógico complejo, por lo que nunca se podrá tomar el camino a lo largo de la cual se llega a p sin una definición real de x .

de la instrucción s puede tener su valor original antes de s , o el nuevo valor creado por s . Por cuestión de simplicidad, el resto del capítulo supone que estamos tratando sólo con variables que no tienen alias. Esta clase de variables incluye a todas las variables escalares locales en la mayoría de los lenguajes; en el caso de C y C++, se excluyen las variables locales cuyas direcciones se hayan calculado en algún punto.

Ejemplo 9.9: En la figura 9.13 se muestra un grafo de flujo con siete definiciones. Vamos a enfocarnos en las definiciones que llegan al bloque B_2 . Todas las definiciones en el bloque B_1 llegan al inicio del bloque B_2 . La definición $d_5: j = j-1$ en el bloque B_2 también llega al inicio del bloque B_2 , ya que no pueden encontrarse otras definiciones de j en el ciclo que conduce de vuelta a B_2 . Sin embargo, esta definición elimina a la definición $d_2: j = n$, la cual evita que llegue a B_3 o B_4 . Pero la instrucción $d_4: i = i+1$ en B_2 no llega al inicio de B_2 , ya que la variable i siempre se redefine mediante $d_7: i = u3$. Por último, la definición $d_6: a = u2$ también llega al inicio del bloque B_2 . \square

Al especificar las definiciones de alcance como lo hemos hecho, en ocasiones les permitimos imprecisiones. Sin embargo, todas se encuentran en la dirección “segura” o “conservadora”. Por ejemplo, observe nuestra suposición de que pueden recorrerse todos las flechas de un grafo de flujo. Esta suposición tal vez no se aplique en la práctica. Por ejemplo, para ningún valor de a y de b el flujo de control puede llegar realmente a la *instrucción 2* en el siguiente fragmento de programa:

```
if (a == b) instrucción 1; else if (a == b) instrucción 2;
```

El proceso de decidir en general si se puede tomar cada camino en un grafo de flujo es un problema indecidible. Por ende, tan sólo suponemos que cada camino en el grafo de flujo puede seguirse en cierta ejecución del programa. En la mayoría de las aplicaciones de las definiciones de alcance, es conservador asumir que una definición puede llegar a un punto aun cuando no sea así. En consecuencia, podemos permitir caminos que nunca se recorrerán en ninguna ejecución del programa, y podemos permitir que las definiciones pasen a través de definiciones ambiguas de la misma variable en forma segura.

Estimación conservadora en el análisis del flujo de datos

Como todos los esquemas del flujo de datos calculan aproximaciones a la verdad fundamental (según se define mediante todos los posibles caminos de ejecución del programa), estamos obligados a asegurar que cualquier error se encuentre en la dirección “segura”. La decisión de una política es *segura* (o *estimación conservadora*) si nunca nos permite modificar lo que el programa calcula. Las políticas seguras pueden, por desgracia, hacer que pasemos por alto ciertas mejoras de código que podrían retener el significado del programa, pero en casi todas las optimizaciones de código no hay una política segura que garantice no omitir nada. Por lo general, sería inaceptable utilizar una política insegura (una que aumente la velocidad del código a expensas de modificar lo que el programa calcula).

Por ende, al diseñar un esquema de flujo de datos, debemos estar conscientes de la forma en que se utilizará la información, y asegurarse de que cualquier aproximación que hagamos se encuentre en la dirección “conservadora” o “segura”. Cada esquema y aplicación deben considerarse en forma independiente. Por ejemplo, si utilizamos definiciones de alcance para el cálculo previo de constantes, es seguro pensar que una definición llega cuando no lo hace (podríamos pensar que x no es una constante, cuando de hecho lo es y podría haberse calculado antes), pero no es seguro pensar que una definición no llega cuando si lo hace (podríamos sustituir x por una constante, cuando el programa algunas veces tuviera un valor para x distinto de una constante).

Ecuaciones de transferencia para las definiciones de alcance

Ahora vamos a establecer las restricciones para el problema de las definiciones de alcance. Empezaremos por examinar los detalles de una sola instrucción. Considere la siguiente definición:

$$d: u = v + w$$

Aquí, y con frecuencia en el resto de la explicación, $+$ se utiliza como un operador binario genérico.

Esta instrucción “genera” una definición d de la variable u y “elimina” a todas las demás definiciones en el programa que definen a la variable u , al mismo tiempo que no afecta al resto de las definiciones entrantes. Por lo tanto, la función de transferencia de la definición d puede expresarse como:

$$f_d(x) = \text{gen}_d \cup (x - \text{eliminar}_d) \quad (9.1)$$

en donde $\text{gen}_d = \{d\}$, el conjunto definiciones generadas por la instrucción, y eliminar_d es el conjunto de todas las demás definiciones de u en el programa.

Como vimos en la sección 9.2.2, la función de transferencia de un bloque básico puede encontrarse mediante la composición de las funciones de transferencia de las instrucciones ahí contenidas. La composición de funciones de la forma (9.1), a la cual nos referiremos como “forma

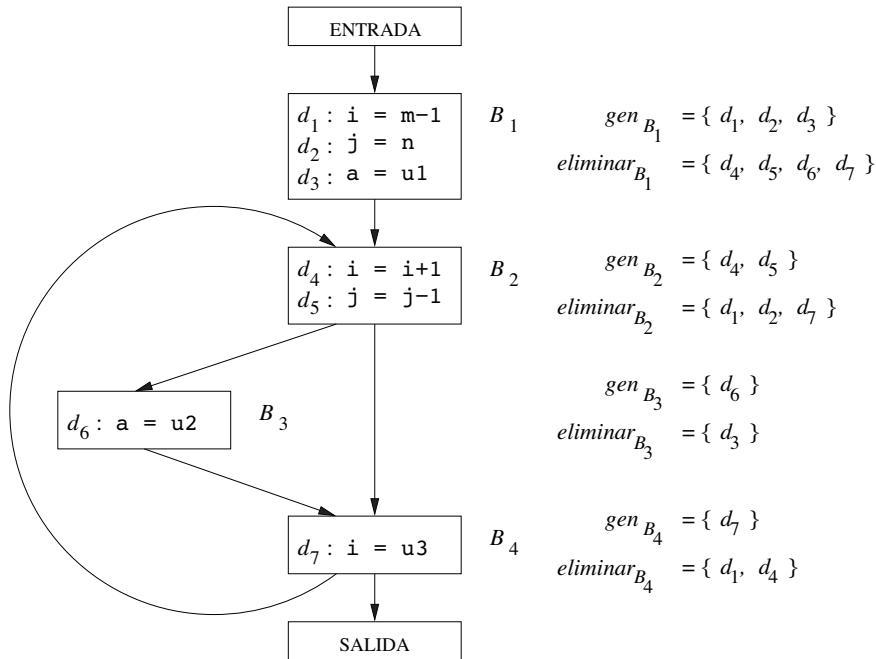


Figura 9.13: Grafo de flujo para ilustrar las definiciones de alcance

“gen-eliminar”, es también de esa forma, como podemos ver a continuación. Suponga que hay dos funciones $f_1(x) = gen_1 \cup (x - eliminar_1)$ y $f_2(x) = gen_2 \cup (x - eliminar_2)$. Entonces:

$$\begin{aligned} f_2(f_1(x)) &= gen_2 \cup (gen_1 \cup (x - eliminar_1) - eliminar_2) \\ &= (gen_2 \cup (gen_1 - eliminar_2)) \cup (x - (eliminar_1 \cup eliminar_2)) \end{aligned}$$

Esta regla se extiende a un bloque que consiste en cualquier número de instrucciones. Suponga que el bloque B tiene n instrucciones, con funciones de transferencia $f_i(x) = gen_i \cup (x - eliminar_i)$ para $i = 1, 2, \dots, n$. Entonces, la función de transferencia para el bloque B puede rescribirse como:

$$f_B(x) = gen_B \cup (x - eliminar_B),$$

en donde

$$eliminar_B = eliminar_1 \cup eliminar_2 \cup \dots \cup eliminar_n$$

y

$$\begin{aligned} gen_B &= gen_n \cup (gen_{n-1} - eliminar_n) \cup (gen_{n-2} - eliminar_{n-1} - eliminar_n) \cup \\ &\dots \cup (gen_1 - eliminar_2 - eliminar_3 - \dots - eliminar_n) \end{aligned}$$

Por ende, al igual que una instrucción, un bloque básico también genera un conjunto de definiciones y elimina a un conjunto de definiciones. El conjunto *gen* contiene todas las definiciones dentro del bloque que son “visibles” justo después del bloque; nos referimos a ellas como *expuestas hacia abajo*. Una definición está expuesta hacia abajo en un bloque básico sólo si no la “elimina” una definición siguiente a la misma variable dentro del mismo bloque básico. El conjunto *eliminar* de un bloque básico es tan sólo la unión de todas las definiciones que eliminan las instrucciones individuales. Observe que una definición puede aparecer tanto en el conjunto *gen* como en el conjunto *eliminar* de un bloque básico. De ser así, el hecho de que se encuentra en *gen* tiene precedencia, ya que en la forma *gen-eliminar*, el conjunto *eliminar* se aplica antes del conjunto *gen*.

Ejemplo 9.10: El conjunto *gen* para el siguiente bloque básico:

$$\begin{aligned} d_1: \quad a &= 3 \\ d_2: \quad a &= 4 \end{aligned}$$

es $\{d_2\}$, ya que d_1 no está expuesta hacia abajo. El conjunto *eliminar* contiene tanto a d_1 como a d_2 , ya que d_1 elimina a d_2 y viceversa. Sin embargo, como la resta del conjunto *eliminar* precede a la operación de unión con el conjunto *gen*, el resultado de la función de transferencia para este bloque siempre incluye la definición d_2 . \square

Ecuaciones de flujo de control

A continuación vamos a considerar el conjunto de restricciones que se derivan del flujo de control entre los bloques básicos. Como una definición llega a un punto de programa siempre y cuando exista por lo menos un camino a lo largo del cual pueda llegar la definición, $\text{SAL}[P] \subseteq \text{ENT}[B]$ cada vez que hay una flecha de flujo de control de P a B . No obstante, como una definición no puede llegar a un punto, a menos que haya un camino a lo largo del cual pueda llegar, $\text{ENT}[B]$ debe ser no mayor que la unión de las definiciones a las que se llega de todos los bloques predecesores. Es decir, es seguro suponer que:

$$\text{ENT}[B] = \bigcup_{P \text{ un predecesor de } B} \text{SAL}[P]$$

Nos referimos a la unión como el *operador de reunión* para las definiciones de alcance. En cualquier esquema de flujo de datos, el operador de reunión es el que utilizamos para crear un resumen de las contribuciones de distintos caminos, en la confluencia de ellos.

Algoritmo iterativo para las definiciones de alcance

Asumimos que todo grafo del flujo de control tiene dos bloques básicos vacíos, un nodo ENTRADA, el cual representa el punto inicial del grafo, y un nodo SALIDA, al cual van todas las salidas del grafo. Como no hay definiciones que lleguen al inicio del grafo, la función de transferencia para el bloque ENTRADA es una función constante simple que devuelve \emptyset como respuesta. Es decir, $\text{SAL}[\text{ENTRADA}] = \emptyset$.

El problema de las definiciones de alcance se define mediante las siguientes ecuaciones:

$$\text{SAL}[\text{ENTRADA}] = \emptyset$$

y para todos los bloques básicos B distintos de ENTRADA,

$$\text{SAL}[B] = \text{gen}_B \cup (\text{ENT}[B] - \text{eliminar}_B)$$

$$\text{ENT}[B] = \bigcup_{P \text{ un predecesor de } B} \text{SAL}[P].$$

Estas ecuaciones pueden resolverse mediante el siguiente algoritmo. El resultado del algoritmo es el *mínimo punto fijo* de las ecuaciones; es decir, la solución cuyos valores asignados a los valores de ENT y SAL estén contenidos en los valores correspondientes para cualquier otra solución a las ecuaciones. El resultado del algoritmo que se muestra a continuación es aceptable, ya que sin duda, cualquier definición en uno de los conjuntos ENT o SAL debe llegar al punto descrito. Es una solución deseable, ya que no incluye las definiciones que podemos estar seguros que no llegarán.

Algoritmo 9.11: Definiciones de alcance.

ENTRADA: Un grafo de flujo para el cual se han calculado eliminar_B y gen_B para cada bloque B .

SALIDA: $\text{ENT}[B]$ y $\text{SAL}[B]$, el conjunto de definiciones que llegan a la entrada y la salida de cada bloque B del grafo de flujo.

MÉTODO: Utilizamos un método iterativo, en el cual empezamos con el “estimado” $\text{SAL}[B] = \emptyset$ para todos los valores de B y convergemos a los valores deseados de ENT y SAL. Como debemos iterar hasta que converjan los valores de ENT (y por ende los de SAL), podríamos usar una variable booleana llamada *cambio* para registrar, en cada pasada a través de los bloques, si ha cambiado cualquier valor de SAL. No obstante, en este y en algoritmos similares que describiremos más adelante, vamos a suponer que el mecanismo exacto para llevar el registro de los cambios es comprensible, por lo que eludiremos esos detalles.

El algoritmo se bosqueja en la figura 9.14. Las primeras dos líneas inicializan ciertos valores del flujo de datos.⁴ La línea (3) inicia el ciclo en el cual iteraremos hasta la convergencia, y el ciclo interno de las líneas (4) a la (6) aplica las ecuaciones del flujo de datos a todos los bloques excepto el de entrada. \square

De manera intuitiva, el Algoritmo 9.11 propaga las definiciones hasta donde lleguen sin que las eliminen, simulando así todas las ejecuciones posibles del programa. En un momento dado, el Algoritmo 9.11 se detendrá, ya que para cada B , $\text{SAL}[B]$ nunca se reduce; una vez que se agrega una definición, permanece ahí para siempre (vea el ejercicio 9.2.6). Como el conjunto de todas las definiciones es finito, en un momento dado debe haber una pasada del ciclo while durante el cual no se agrega nada a ningún valor de SAL, y entonces el algoritmo termina. Es seguro terminar entonces, ya que si los valores de SAL no han cambiado, los

⁴El lector observador descubrirá que podríamos combinar con facilidad las líneas (1) y (2). No obstante, en algoritmos de flujo de datos similares, puede ser necesario inicializar al nodo de entrada o de salida en forma distinta a la que usamos para inicializar los demás nodos. Por ende, seguimos un patrón en todos los algoritmos iterativos de aplicar una “condición del límite” como la línea (1) en forma separada de la inicialización de la línea (2).

- 1) $\text{SAL}[\text{ENTRADA}] = \emptyset;$
- 2) **for** (cada bloque básico B que no sea ENTRADA) $\text{SAL}[B] = \emptyset;$
- 3) **while** (ocurran cambios a cualquier SAL)
 - 4) **for** (cada bloque básico B que no sea ENTRADA) {
 - 5) $\text{ENT}[B] = \bigcup_{P \text{ un predecesor de } B} \text{SAL}[P];$
 - 6) $\text{SAL}[B] = \text{gen}_B \cup (\text{ENT}[B] - \text{eliminar}_B);$

Figura 9.14: Algoritmo iterativo para calcular las definiciones de alcance

valores de ENT no cambiarán la siguiente pasada. Y, si los valores de ENT no cambian, los valores de SAL tampoco, por lo que en todas las pasadas siguientes no puede haber cambios.

El número de nodos en el grafo de flujo es un límite superior sobre el número de veces que se recorre el ciclo while. La razón es que si una definición llega a un punto, puede hacerlo a lo largo de un camino sin ciclos, y número de nodos en un grafo de flujo es un límite superior sobre el número de nodos en un camino sin ciclo. Cada vez que se recorre el ciclo while, cada definición progresa por lo menos un nodo a lo largo del camino en cuestión, y a menudo progresa más de un nodo, dependiendo del orden en el que se visiten los nodos.

De hecho, si ordenamos en forma apropiada los bloques en el ciclo for de la línea (5), existe una evidencia empírica de que el número promedio de iteraciones del ciclo while es menor de 5 (vea la sección 9.6.7). Como los conjuntos de definiciones se pueden representar mediante vectores de bits, y las operaciones sobre estos conjuntos se pueden implementar mediante operaciones lógicas en los vectores de bits, el Algoritmo 9.11 es muy eficiente en la práctica.

Ejemplo 9.12: Vamos a representar las siete definiciones d_1, d_2, \dots, d_7 en el grafo de flujo de la figura 9.13 mediante vectores de bits, en donde el bit i de la izquierda representa a la definición d_i . La unión de conjuntos se calcula tomando la operación OR lógica de los correspondientes vectores de bits. La diferencia de dos conjuntos $S - T$ se calcula mediante el complemento del vector de bits de T , y después se toma la operación AND lógica de ese complemento, con el vector de bits para S .

En la tabla de la figura 9.15 se encuentran los valores tomados por los conjuntos ENT y SAL en el Algoritmo 9.11. Los valores iniciales, que se indican mediante un superíndice 0, como en $\text{SAL}[B]^0$, se asignan mediante el ciclo de la línea (2) de la figura 9.14. Cada uno de ellos es el conjunto vacío, representado por el vector de bits 000 0000. Los valores de las siguientes pasadas del algoritmo también se indican mediante superíndices, y se etiquetan como $\text{ENT}[B]^1$ y $\text{SAL}[B]^1$ para la primera pasada, y $\text{ENT}[B]^2$ y $\text{SAL}[B]^2$ para la segunda.

Suponga que el ciclo for de las líneas (4) a la (6) se ejecuta cuando B toma los siguientes valores:

$$B_1, B_2, B_3, B_4, \text{SALIDA}$$

en ese orden. Con $B = B_1$, como $\text{SAL}[\text{ENTRADA}] = \emptyset$, $\text{ENT}[B]^1$ es el conjunto vacío, y $\text{SAL}[B]^1$ es gen_{B_1} . Este valor difiere del valor anterior $\text{SAL}[B_1]^0$, por lo que ahora sabemos que hay un cambio en la primera ronda (y procederemos a una segunda ronda).

Bloque B	$\text{SAL}[B]^0$	$\text{ENT}[B]^1$	$\text{SAL}[B]^1$	$\text{ENT}[B]^2$	$\text{SAL}[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
SALIDA	000 0000	001 0111	001 0111	001 0111	001 0111

Figura 9.15: Cálculo de ENT y SAL

Después consideramos $B = B_2$ y calculamos

$$\begin{aligned}\text{ENT}[B_2]^1 &= \text{SAL}[B_1]^1 \cup \text{SAL}[B_4]^0 \\ &= 111 0000 + 000 0000 = 111 0000 \\ \text{SAL}[B_2]^1 &= \text{gen}[B_2] \cup (\text{ENT}[B_2]^1 - \text{eliminar}[B_2]) \\ &= 000 1100 + (111 0000 - 110 0001) = 001 1100\end{aligned}$$

Este cálculo se resume en la figura 9.15. Por ejemplo, al final de la primera pasada, $\text{SAL}[B_2]^1 = 001 1100$, reflejando el hecho de que d_4 y d_5 se generan en B_2 , mientras que d_3 llega al inicio de B_2 y no se elimina en B_2 .

Observe que después de la segunda ronda, $\text{SAL}[B_2]$ ha cambiado para reflejar el hecho de que d_6 también llega al inicio de B_2 y éste no la elimina. No aprendimos ese hecho en la primera pasada, ya que el camino de d_6 hasta el final de B_2 , que es $B_3 \rightarrow B_4 \rightarrow B_2$, no se recorre en ese orden mediante una sola pasada. Es decir, para cuando sabemos que d_6 llega al final de B_4 , ya hemos calculado $\text{ENT}[B_2]$ y $\text{SAL}[B_2]$ en la primera pasada.

No hay cambios en ninguno de los conjuntos SAL después de la segunda pasada. Por ende, después de una tercera pasada, el algoritmo termina, con los valores de ENT y SAL como en las últimas dos columnas de la figura 9.15. \square

9.2.5 Análisis de variables vivas

Algunas transformaciones para mejorar el código dependen de la información que se calcula en la dirección opuesta al flujo de control de un programa; vamos a examinar uno de esos ejemplos ahora. En el *análisis de variables vivas* deseamos conocer para la variable x y el punto p si el valor de x en p podría utilizarse a lo largo de algún camino en el grafo de flujo, empezando en p . Si es así, decimos que x está *viva* en p ; en cualquier otro caso, x está *muerta* en p .

Un uso importante para la información de las variables vivas es la repartición de registros para los bloques básicos. En las secciones 8.6 y 8.8 se presentaron los aspectos relacionados con esta cuestión. Después de calcular un valor en un registro, y supuestamente utilizarlo dentro de un bloque, no es necesario almacenar ese valor si está muerto al final del bloque. Además, si todos los registros están llenos y necesitamos otro registro, es preferible usar un registro con un valor muerto, ya que ese valor tiene que almacenarse.

Aquí, definimos las ecuaciones del flujo de datos directamente en términos de $\text{ENT}[B]$ y $\text{SAL}[B]$, que representan el conjunto de variables vivas en los puntos justo antes y después del bloque B , respectivamente. Estas ecuaciones también pueden derivarse si primero definimos las funciones de transferencia de las instrucciones individuales y las componemos para crear la función de transferencia de un bloque básico. Se definen:

1. def_B como el conjunto de variables *definidas* (es decir, los valores asignados en forma definitiva) en B , antes de cualquier uso de esa variable en B .
2. uso_B como el conjunto de variables cuyos valores pueden usarse en B , antes de cualquier definición de la variable.

Ejemplo 9.13: Por ejemplo, el bloque B_2 en la figura 9.13 utiliza a i en forma definitiva. También utiliza a j antes de cualquier redefinición de j , a menos que sea posible que i y j sean alias una de la otra. Suponiendo que no hay alias entre las variables de la figura 9.13, entonces $\text{uso}_{B_2} = \{i, j\}$. Además, B_2 define claramente a i y j . Suponiendo que no hay alias, $\text{def}_{B_2} = \{i, j\}$, también. \square

Como consecuencia de las definiciones, cualquier variable en uso_B debe considerarse viva al entrar al bloque B , mientras que las definiciones de las variables en def_B definitivamente están muertas al inicio de B . En efecto, la membresía en def_B “elimina” cualquier oportunidad de que una variable esté viva, debido a los caminos que empiezan en B .

Por ende, las ecuaciones que relacionan a def y uso con los valores desconocidos de ENT y SAL se definen de la siguiente manera:

$$\text{ENT}[\text{SALIDA}] = \emptyset$$

y para todos los bloques básicos B distintos de SALIDA,

$$\begin{aligned}\text{ENT}[B] &= \text{uso}_B \cup (\text{SAL}[B] - \text{def}_B) \\ \text{SAL}[B] &= \bigcup_{S \text{ un sucesor de } B} \text{ENT}[S]\end{aligned}$$

La primera ecuación especifica la condición delimitadora, la cual establece que no hay variables vivas al salir del programa. La segunda ecuación dice que una variable está viva al entrar a un bloque si se utiliza antes de su redefinición en el bloque, o si está viva al salir del bloque y no se redefine en el mismo. La tercera ecuación dice que una variable está viva al salir de un bloque si, y sólo si está viva al entrar a uno de sus sucesores.

Hay que tener en cuenta la relación entre las ecuaciones para el estado de vida y las ecuaciones de las definiciones de alcance:

- Ambos conjuntos de ecuaciones tienen la unión como operador de reunión. La razón es que en cada esquema de flujo de datos propagamos información a lo largo de los caminos, y sólo nos preocupa saber si existe *algún* camino con las propiedades deseadas, en vez de saber si algo es verdadero a lo largo de *todos* los caminos.
- Sin embargo, el flujo de información para el estado de vida viaja “a la inversa”, en sentido opuesto a la dirección del flujo de control, ya que en este problema deseamos asegurarnos de que el uso de una variable x en un punto p se transmita a todos los puntos anteriores a p en algún camino de ejecución, de manera que sepamos en el punto anterior que se utilizará el valor de x .

Para resolver un problema inverso, en vez de inicializar $\text{SAL}[\text{ENTRADA}]$, inicializamos $\text{ENT}[\text{SALIDA}]$. Se intercambian los papeles de los conjuntos ENT y SAL , además *uso* y *def* sustituyen a *gen* y *eliminar*, respectivamente. En cuanto a las definiciones de alcance, la solución a las ecuaciones del estado de vida no es necesariamente única, y deseamos la solución con los conjuntos más pequeños de variables vivas. El algoritmo utilizado es en esencia una versión inversa del Algoritmo 9.11.

Algoritmo 9.14: Análisis de variables vivas.

ENTRADA: Un grafo de flujo en el que se calculan *def* y *uso* para cada bloque.

SALIDA: $\text{ENT}[B]$ y $\text{SAL}[B]$, el conjunto de variables vivas al entrar y salir de cada bloque B del grafo de flujo.

MÉTODO: Ejecute el programa de la figura 9.16. \square

```

 $\text{ENT}[\text{SALIDA}] = \emptyset;$ 
for (cada bloque básico  $B$  que no sea SALIDA)  $\text{ENT}[B] = \emptyset$ ;
while (se realicen modificaciones a cualquier ENT)
  for (cada bloque básico  $B$  que no sea SALIDA) {
     $\text{SAL}[B] = \bigcup_{S \text{ un sucesor de } B} \text{ENT}[S];$ 
     $\text{ENT}[B] = \text{uso}_B \cup (\text{SAL}[B] - \text{def}_B);$ 
  }

```

Figura 9.16: Algoritmo iterativo para calcular las variables vivas

9.2.6 Expresiones disponibles

Una expresión $x + y$ está *disponible* en el punto p si todas las rutas desde el nodo de entrada hacia p se evalúan como $x + y$, y si después de la última evaluación de este tipo, antes de llegar a p , no hay asignaciones siguientes para x o y .⁵ Para el esquema de flujo de datos de expresiones disponibles decimos que un bloque *elimina* a la expresión $x + y$ si asigna (o puede asignar) a x o a y ,

⁵Observe que, como es usual en este capítulo, utilizamos el operador $+$ como un operador genérico, que no necesariamente representa a la suma.

y no vuelve a calcular posteriormente la expresión $x + y$. Un bloque *genera* la expresión $x + y$ si en definitiva evalúa $x + y$ y no define posteriormente a x o a y .

Observe que la noción de “eliminar” o “generar” una expresión disponible no es exactamente igual que para las definiciones de alcance. Sin embargo, estas nociones de “eliminar” y “generar” se comportan en esencia de la misma forma que para las definiciones de alcance.

El uso principal de la información sobre expresiones disponibles es para detectar las subexpresiones comunes globales. Por ejemplo, en la figura 9.17(a), la expresión $4 * i$ en el bloque B_3 será una subexpresión común si $4 * i$ está disponible en el punto de entrada del bloque B_3 . Estará disponible si no se asigna a i un nuevo valor en el bloque B_2 , o si, como en la figura 9.17(b), $4 * i$ se vuelve a calcular después de que i se asigna en B_2 .

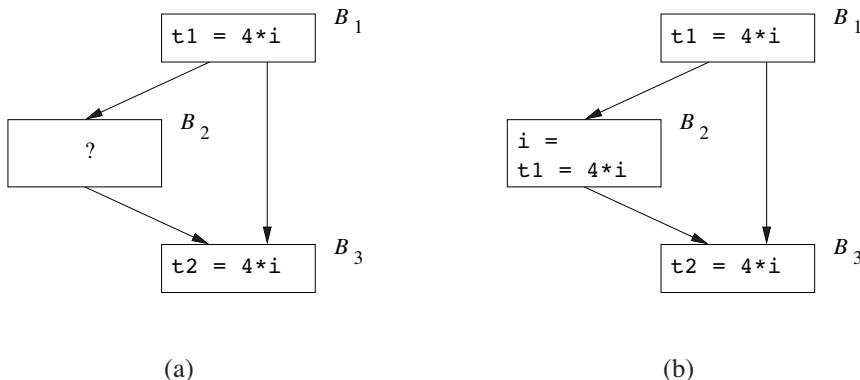


Figura 9.17: Subexpresiones comunes potenciales entre los bloques

Podemos calcular el conjunto de expresiones generadas para cada punto en un bloque, trabajando desde el principio hasta el final del bloque. En el punto antes del bloque, no se generan expresiones. Si en el punto p está disponible el conjunto S de expresiones, y q es el punto después de p , con la instrucción $x = y+z$ entre ellos, entonces formamos el conjunto de expresiones disponible en q mediante los siguientes dos pasos.

1. Agregar a S la expresión $y + z$.
2. Eliminar de S cualquier expresión que involucre a la variable x .

Hay que tener en cuenta que los pasos deben realizarse en el orden correcto, ya que x podría ser igual que y o z . Una vez que llegamos al final del bloque, S es el conjunto de expresiones generadas para el bloque. El conjunto de expresiones eliminadas consiste en todas las expresiones, por decir $y + z$, de tal forma que se defina y o z en el bloque, y que el bloque no genere a $y + z$.

Ejemplo 9.15: Considere las cuatro instrucciones de la figura 9.18. Después de la primera, $b + c$ está disponible. Después de la segunda instrucción, $a - d$ se vuelve disponible, pero $b + c$ ya no está disponible, debido a que b se ha definido de nuevo. La tercera instrucción no hace que $b + c$ esté disponible otra vez, ya que el valor de c se modifica de inmediato.

Después de la última instrucción, $a - d$ ya no está disponible, debido a que d ha cambiado. Por ende no se generan expresiones, y se eliminan todas las expresiones en las que se involucren a , b , c o d . \square

Instrucción	Expresiones disponibles
	\emptyset
$a = b + c$	$\{b + c\}$
$b = a - d$	$\{a - d\}$
$c = b + c$	$\{a - d\}$
$d = a - d$	\emptyset

Figura 9.18: Cálculo de las expresiones disponibles

Podemos encontrar las expresiones disponibles de una forma parecida a la manera en la que se calculan las definiciones de alcance. Suponga que U es el conjunto “universal” de todas las expresiones que aparecen a la derecha de una o más instrucciones del programa. Para cada bloque B , hagamos que $\text{ENT}[B]$ sea el conjunto de expresiones en U que están disponibles en el punto justo antes del inicio de B . Hagamos que $\text{SAL}[B]$ sea igual que para el punto que va después del final de B . Definamos a e_gen_B para que represente a las expresiones que se generan en B , y a $e_eliminar_B$ para ser el conjunto de expresiones en U eliminadas en B . Observe que ENT , SAL , e_gen , y $e_eliminar$ pueden representarse mediante vectores de bits. Las siguientes ecuaciones relacionan los valores desconocidos ENT y SAL entre sí, y las cantidades conocidas e_gen y $e_eliminar$:

$$\text{SAL}[\text{ENTRADA}] = \emptyset$$

y para todos los bloques básicos B que no sean ENTRADA,

$$\text{SAL}[B] = e_gen_B \cup (\text{ENT}[B] - e_eliminar_B)$$

$$\text{ENT}[B] = \bigcap_{P \text{ un predecesor de } B} \text{SAL}[P].$$

Las ecuaciones anteriores se ven casi idénticas a las ecuaciones para las definiciones de alcance. Al igual que las definiciones de alcance, la condición delimitadora es $\text{SAL}[\text{ENTRADA}] = \emptyset$, ya que en la salida del nodo ENTRADA no hay expresiones disponibles. La diferencia más importante es que el operador de reunión es la intersección, en vez de la unión. Este operador es el apropiado, ya que una expresión está disponible al inicio de un bloque sólo si está disponible al final de *todos* sus predecesores. En contraste, una definición llega al final de un bloque cada vez que llega al final de uno o más de sus predecesores.

El uso de \cap en vez de \cup hace que las ecuaciones de las expresiones disponibles se comporten en forma distinta a las de las definiciones de alcance. Aunque ningún conjunto tiene una solución única, para las definiciones de alcance, la solución con los conjuntos más pequeños es la que corresponde a la definición de “alcance”, y obtuvimos esa solución empezando con la suposición de que nada llegaba a ninguna parte, y progresamos hasta llegar a la solución. En esa forma, nunca se supuso que una definición d podría llegar a un punto p , a menos de que pudiera encontrarse un camino actual que se propagara de d hasta p . En contraste, para las ecuaciones de las expresiones disponibles queremos la solución con los conjuntos más grandes de expresiones disponibles, por lo que empezamos con una aproximación demasiado grande y la vamos reduciendo.

Tal vez no sea evidente que al empezar con la suposición “todo (es decir, el conjunto U) está disponible en cualquier parte, excepto al final del bloque de entrada” y eliminar sólo aquellas expresiones para las cuales podamos descubrir un camino a lo largo de la cual no esté disponible, podemos llegar a un conjunto de expresiones verdaderamente disponibles. En el caso de las expresiones disponibles, es conservador producir un subconjunto del conjunto exacto de expresiones disponibles. El argumento para que los subconjuntos sean conservadores es que el uso que pretendemos de la información es sustituir el cálculo de una expresión disponible por un valor calculado con anterioridad. Al no saber que una expresión está disponible no podemos mejorar el código, mientras que al creer que una expresión está disponible cuando en realidad no es así, podría modificar lo que el programa calcula.

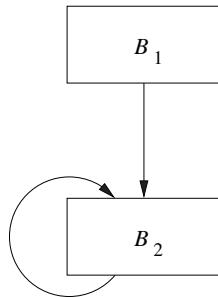


Figura 9.19: La inicialización de los conjuntos SAL a \emptyset es demasiado restrictiva

Ejemplo 9.16: Vamos a concentrarlos en un solo bloque, B_2 en la figura 9.19, para ilustrar el efecto de la aproximación inicial de $\text{SAL}[B_2]$ sobre $\text{ENT}[B_2]$. Hagamos que G y K sean abreviaciones de $e\text{-gen}_{B_2}$ y $e\text{-eliminar}_{B_2}$, respectivamente. Las ecuaciones del flujo de datos para el bloque B_2 son:

$$\text{ENT}[B_2] = \text{SAL}[B_1] \cap \text{SAL}[B_2]$$

$$\text{SAL}[B_2] = G \cup (\text{ENT}[B_2] - K)$$

Estas ecuaciones pueden rescribirse como recurrencias, en donde I^j y O^j son las j -ésimas aproximaciones de $\text{ENT}[B_2]$ y $\text{SAL}[B_2]$, respectivamente:

$$I^{j+1} = \text{SAL}[B_1] \cap O^j$$

$$O^{j+1} = G \cup (I^{j+1} - K)$$

Empezando con $O^0 = \emptyset$, obtenemos $I^1 = \text{SAL}[B_1] \cap O^0 = \emptyset$. Sin embargo, si empezamos con $O^0 = U$, entonces obtenemos $I^1 = \text{SAL}[B_1] \cap O^0 = \text{SAL}[B_1]$, como deberíamos. Por intuición, la solución que se obtiene empezando con $O^0 = U$ es una mejor opción, ya que refleja de manera correcta el hecho de que las expresiones en $\text{SAL}[B_1]$ que B_2 no elimina están disponibles al final de B_2 . \square

Algoritmo 9.17: Expresiones disponibles.

ENTRADA: Un grafo del flujo en el cual se calculan $e_eliminar_B$ y e_gen_B para cada bloque B . El bloque inicial es B_1 .

SALIDA: $\text{ENT}[B]$ y $\text{SAL}[B]$, el conjunto de expresiones disponibles en la entrada y salida de cada bloque B del grafo de flujos.

MÉTODO: Ejecute el algoritmo de la figura 9.20. La explicación de los pasos es similar a la de la figura 9.14. \square

```

 $\text{SAL}[\text{ENTRADA}] = \emptyset;$ 
for (cada bloque básico  $B$  que no sea ENTRADA)  $\text{SAL}[B] = U$ ;
while (ocurra algún cambio en SAL)
  for (cada bloque básico  $B$  que no sea ENTRADA) {
     $\text{ENT}[B] = \bigcap_{P \text{ un predecesor de } B} \text{SAL}[P];$ 
     $\text{SAL}[B] = e\_gen_B \cup (\text{ENT}[B] - e\_eliminar_B);$ 
  }
}

```

Figura 9.20: Algoritmo iterativo para calcular las expresiones disponibles

9.2.7 Resumen

En esta sección, hablamos sobre tres instancias de los problemas del flujo de datos: las definiciones de alcance, las variables vivas y las expresiones disponibles. Como se resume en la figura 9.21, la definición de cada problema se proporciona mediante el dominio de los valores del flujo de datos, la dirección del flujo de datos, la familia de funciones de transferencia, la condición delimitadora y el operador de reunión. Denotamos el operador de reunión en forma genérica como \wedge .

La última fila muestran los valores iniciales utilizados en el algoritmo iterativo. Estos valores se eligen de tal forma que el algoritmo iterativo encuentre la solución más precisa a las ecuaciones. Esta elección no forma estrictamente una parte de la definición del problema del

flujo de datos, ya que es un dispositivo necesario para el algoritmo iterativo. Existen otras formas de resolver el problema. Por ejemplo, vimos cómo la función de transferencia de un bloque básico puede derivarse mediante la composición de las funciones de transferencia de las instrucciones individuales en el bloque; puede utilizarse un método de composición similar al calcular una función de transferencia para todo el procedimiento; o funciones de transferencia desde la entrada del procedimiento hasta cualquier punto del programa. En la sección 9.7 hablaremos sobre dicho método.

	Definiciones de alcance	Variables vivas	Expresiones disponibles
Dominio	Conjuntos de definiciones	Conjuntos de variables	Conjuntos de expresiones
Dirección	Hacia delante	Hacia atrás	Hacia delante
Función de transferencia	$gen_B \cup (x - \text{eliminar}_B)$	$uso_B \cup (x - \text{def}_B)$	$e_gen_B \cup (x - e_eliminar_B)$
Límite	$\text{SAL}[\text{ENTRADA}] = \emptyset$	$\text{ENT}[\text{SALIDA}] = \emptyset$	$\text{SAL}[\text{ENTRADA}] = \emptyset$
Reunión (\wedge)	\cup	\cup	\cap
Ecuaciones	$\text{SAL}[B] = f_B(\text{ENT}[B])$ $\text{ENT}[B] = \bigwedge_{P, \text{pred}(B)} \text{SAL}[P]$	$\text{ENT}[B] = f_B(\text{SAL}[B])$ $\text{SAL}[B] = \bigwedge_{S, \text{suc}(B)} \text{ENT}[S]$	$\text{SAL}[B] = f_B(\text{ENT}[B])$ $\text{ENT}[B] = \bigwedge_{P, \text{pred}(B)} \text{SAL}[P]$
Inicializar	$\text{SAL}[B] = \emptyset$	$\text{ENT}[B] = \emptyset$	$\text{SAL}[B] = U$

Figura 9.21: Resumen de tres problemas de flujo de datos

9.2.8 Ejercicios para la sección 9.2

Ejercicio 9.2.1: Para el grafo de flujo de la figura 9.10 (vea los ejercicios de la sección 9.21), calcule:

- Los conjuntos gen y $eliminar$ para cada bloque.
- Los conjuntos ENT y SAL para cada bloque.

Ejercicio 9.2.2: Para el grafo de flujo de la figura 9.10, calcule los conjuntos e_gen , $e_eliminar$, ENT y SAL para las expresiones disponibles.

! Ejercicio 9.2.3: Para el grafo de flujo de la figura 9.10, calcule los conjuntos def , uso , ENT y SAL para el análisis de variables vivas.

Ejercicio 9.2.4: Suponga que V es el conjunto de números complejos. ¿Cuál de las siguientes operaciones puede servir como la operación de reunión para un semi-lattice en V ?

- Suma: $(a + ib) \wedge (c + id) = (a + c) + i(b + d)$.
- Multiplicación: $(a + ib) \wedge (c + id) = (ac - bd) + i(ad + bc)$.

Por qué funciona el algoritmo de expresiones disponibles

Debemos explicar por qué al iniciar todos los valores de SAL excepto los bloques de entrada con U , el conjunto de todas las expresiones, nos lleva a una solución conservadora para las ecuaciones de flujo de datos; es decir, todas las expresiones que se encuentran como disponibles en realidad lo *están*. En primer lugar, debido a que la intersección es el operador de reunión en este esquema de flujo de datos, cualquier motivo por el que se encuentre que una expresión $x + y$ no está disponible en un punto se propagará hacia delante en el grafo de flujo, junto con todas las caminos posibles, hasta que $x + y$ se vuelva a calcular y sea disponible otra vez. En segundo lugar, sólo hay dos razones por las que $x + y$ no podría estar disponible:

1. $x + y$ se elimina en el bloque B debido a que x o y están definidas sin un cálculo siguiente de $x + y$. En este caso, la primera vez que apliquemos la función de transferencia f_B , $x + y$ se eliminará de $\text{SAL}[B]$.
2. $x + y$ nunca se calcula a lo largo de algún camino. Como $x + y$ nunca está en $\text{SAL}[\text{ENTRADA}]$, y nunca se genera a lo largo del camino en cuestión, podemos mostrar mediante la inducción sobre la longitud del camino que $x + y$ se elimina en un momento dado de los valores de ENT y SAL a lo largo del camino.

Así, una vez que terminen los cambios, la solución proporcionada por el algoritmo iterativo de la figura 9.20 sólo incluirá expresiones que en verdad estén disponibles.

- c) Componente mínimo: $(a + ib) \wedge (c + id) = \min(a, c) + i \min(b, d)$.
- d) Componente máximo: $(a + ib) \wedge (c + id) = \max(a, c) + i \max(b, d)$.

! Ejercicio 9.2.5: Afirmando que si un bloque B consiste en n instrucciones, y la i -ésima instrucción tiene los conjuntos gen y eliminar gen_i y $eliminar_i$, entonces la función de transferencia para el bloque B tiene los conjuntos gen y eliminar gen_B y $eliminar_B$ dados por:

$$eliminar_B = eliminar_1 \cup eliminar_2 \cup \dots \cup eliminar_n$$

$$gen_B = gen_n \cup (gen_{n-1} - eliminar_n) \cup (gen_{n-2} - eliminar_{n-1} - eliminar_n) \cup \dots \cup (gen_1 - eliminar_2 - eliminar_3 - \dots - eliminar_n).$$

Demuestre esta afirmación mediante la inducción sobre n .

! Ejercicio 9.2.6: Demuestre mediante la inducción sobre el número de iteraciones del ciclo for de las líneas (4) a la (6) del Algoritmo 9.11, que ninguno de los valores de ENT o SAL se reduce. Es decir, una vez que se coloca una definición en uno de sus conjuntos en cierta ronda, nunca desaparece en una ronda siguiente.

! Ejercicio 9.2.7: Muestre que el algoritmo 9.11 es correcto. Es decir, muestre que:

- Si la definición d se coloca en $\text{ENT}[B]$ o $\text{SAL}[B]$, entonces hay un camino que va desde d hasta el inicio o final del bloque B , respectivamente, a lo largo de la cual la variable definida por d podría no redefinirse.
- Si la definición d se coloca en $\text{ENT}[B]$ o $\text{SAL}[B]$, entonces no hay un camino que vaya desde d hasta el inicio o final del bloque B , respectivamente, a lo largo de la cual la variable definida por d podría no redefinirse.

! Ejercicio 9.2.8: Demuestre lo siguiente acerca del algoritmo 9.14:

- Los valores de ENT y SAL nunca se reducen.
- Si la variable x se coloca en $\text{ENT}[B]$ o $\text{SAL}[B]$, entonces hay un camino que va desde el inicio o final del bloque B , respectivamente, a lo largo de la cual x podría usarse.
- Si la variable x no se coloca en $\text{ENT}[B]$ o $\text{SAL}[B]$, entonces no hay un camino que vaya desde el inicio o final del bloque B , respectivamente, a lo largo de la cual x podría usarse.

! Ejercicio 9.2.9: Demuestre lo siguiente acerca del Algoritmo 9.17:

- Los valores de ENT y SAL nunca aumentan; es decir, los valores sucesivos de estos conjuntos son subconjuntos (no necesariamente propios) de sus valores anteriores.
- Si la expresión e se elimina de $\text{ENT}[B]$ o $\text{SAL}[B]$, entonces hay un camino que va desde la entrada del grafo de flujo hasta el inicio o final del bloque B , respectivamente, a lo largo de la cual e nunca se calcula, o después de su último cálculo, uno de sus argumentos podría redefinirse.
- Si la expresión e permanece en $\text{ENT}[B]$ o $\text{SAL}[B]$, entonces a lo largo de cada camino que va desde la entrada del grafo de flujo hasta el inicio o final del bloque B , respectivamente, e se calcula y, después del último cálculo, no podría redefinirse ningún argumento de e .

! Ejercicio 9.2.10: El lector astuto observará que en el Algoritmo 9.11 podríamos haber ahorrado algo de tiempo, inicializando $\text{SAL}[B]$ con gen_B para todos los bloques B . De igual forma, en el Algoritmo 9.14 podríamos haber inicializado $\text{ENT}[B]$ con gen_B . No lo hicimos por cuestión de uniformidad en el tratamiento del tema, como veremos en el Algoritmo 9.25. Sin embargo, ¿es posible inicializar $\text{SAL}[B]$ con $e\text{-}\text{gen}_B$ en el Algoritmo 9.17? ¿Por qué sí o por qué no?

! Ejercicio 9.2.11: Hasta ahora, nuestros análisis del flujo de datos no aprovechan la semántica de las instrucciones condicionales. Suponga que al final de un bloque básico encontramos una prueba como:

```
if (x < 10) goto ...
```

¿Cómo podríamos usar nuestra comprensión de lo que significa la prueba $x < 10$ para mejorar nuestro conocimiento de las definiciones de alcance? Recuerde que aquí “mejorar” significa que eliminamos ciertas definiciones de alcance que en realidad nunca podrán llegar a un cierto punto del programa.

9.3 Fundamentos del análisis del flujo de datos

Después de mostrar varios ejemplos útiles en la abstracción del flujo de datos, ahora estudiaremos la familia de los esquemas del flujo de datos como un todo, en forma abstracta. Vamos a responder de manera formal varias preguntas básicas acerca de los algoritmos de flujo de datos:

1. ¿Bajo qué circunstancias es correcto el algoritmo iterativo que se utiliza en el análisis del flujo de datos?
2. ¿Qué tan precisa es la solución obtenida por el algoritmo iterativo?
3. ¿Convergerá el algoritmo iterativo?
4. ¿Cuál es el significado de la solución a las ecuaciones?

En la sección 9.2, tratamos cada una de las preguntas anteriores de manera informal, al describir el problema de las definiciones de alcance. En vez de responder a las mismas preguntas para cada problema siguiente partiendo desde cero, nos basamos en las analogías con los problemas que ya habíamos expuesto para explicar los problemas nuevos. Aquí presentamos un método general que responde a todas esas preguntas, de una vez por todas, con rigor y para una larga familia de problemas de flujo de datos. Primero identificamos las propiedades deseadas de los esquemas del flujo de datos y demostramos las implicaciones de estas propiedades en la exactitud, precisión y convergencia del algoritmo de flujo de datos, así como el significado de la solución. Así, para comprender algoritmos viejos o formular nuevos, sólo mostramos que las definiciones propuestas al problema del flujo de datos tienen ciertas propiedades, y las respuestas a todas las preguntas difíciles anteriores están disponibles de inmediato.

El concepto de tener un marco de trabajo común teórico para una clase de esquemas también tiene implicaciones prácticas. El marco de trabajo nos ayuda a identificar los componentes reutilizables del algoritmo en nuestro diseño de software. No sólo se reduce el esfuerzo de codificación, sino que los errores de programación también lo hacen al no tener que volver a codificar los detalles similares varias veces.

Un *marco de trabajo de análisis del flujo de datos* (D, V, \wedge, F) consiste en:

1. Una dirección del flujo de datos D , que puede ser HACIADELANTE o HACIAATRAS.
2. Un semi-lattice (vea la sección 9.3.1 para la definición), que incluye un *dominio* de valores V y un *operador de reunión* \wedge .
3. Una familia F de funciones de transferencia de V a V . Esta familia debe incluir funciones apropiadas para las condiciones delimitadoras, que son funciones de transferencia de constantes para los nodos especiales ENTRADA y SALIDA en cualquier grafo de flujo.

9.3.1 Semi-lattices

Un *semi-lattice* consiste en un conjunto V y un operador de reunión binario \wedge tal que para todas las x, y y z en V :

1. $x \wedge x = x$ (el operador de reunión es *idempotente*).
2. $x \wedge y = y \wedge x$ (el operador de reunión es *comutativo*).
3. $x \wedge (y \wedge z) = (x \wedge y) \wedge z$ (el operador de reunión es *asociativo*).

Un semi-lattice tiene un elemento *superior*, denotado como \top , de tal forma que

$$\text{para todas las } x \text{ en } V, \top \wedge x = x.$$

De manera opcional, un semi-lattice puede tener un elemento *inferior*, denotado como \perp , de tal forma que

$$\text{para todas las } x \text{ en } V, \perp \wedge x = \perp.$$

Órdenes parciales

Como veremos más adelante, el operador de reunión de un semilattice define un orden parcial sobre los valores del dominio. Una relación \leq es un *orden parcial* sobre un conjunto V si para todas las x, y y z en V :

1. $x \leq x$ (el orden parcial es *reflexivo*).
2. Si $x \leq y$ y $y \leq x$, entonces $x = y$ (el orden parcial es *antisimétrico*).
3. Si $x \leq y$ y $y \leq z$, entonces $x \leq z$ (el orden parcial es *transitivo*).

El par (V, \leq) se llama *poset*, o *conjunto parcialmente ordenado*. También es conveniente tener una relación $<$ para un poset, la cual se define como:

$$x < y \text{ si, y sólo si } (x \leq y) \text{ y } (x \neq y).$$

El orden parcial para un semi-lattice

Es útil definir un orden parcial \leq para un semi-lattice (V, \wedge) . Para todas las x y y en V , definimos

$$x \leq y \text{ si, y sólo si } x \wedge y = x.$$

Como el operador de reunión \wedge es idempotente, comutativo y asociativo, el orden \leq se define como reflexivo, antisimétrico y transitivo. Para ver por qué, tome en cuenta que:

- **Reflexividad:** para todas las x , $x \leq x$. La prueba es que $x \wedge x = x$, ya que el operador de reunión es idempotente.
- **Antisimetría:** si $x \leq y$ y $y \leq x$, entonces $x = y$. Para demostrar esto, $x \leq y$ significa que $x \wedge y = x$ y $y \leq x$ significa que $y \wedge x = y$. Por la condición comutativa de \wedge , $x = (x \wedge y) = (y \wedge x) = y$.

- Transitividad: si $x \leq y$ y $y \leq z$, entonces $x \leq z$. Para demostrar esto, $x \leq y$ y $y \leq z$ significa que $x \wedge y = x$ y $y \wedge z = y$. Entonces $(x \wedge z) = ((x \wedge y) \wedge z) = (x \wedge (y \wedge z)) = (x \wedge y) = x$, usando la asociatividad del operador de reunión. Como $x \wedge z = x$ se ha demostrado, tenemos $x \leq z$, lo cual demuestra la condición transitiva.

Ejemplo 9.18: Los operadores de reunión utilizados en los ejemplos de la sección 9.2 son la unión y la intersección de conjuntos. Ambos son idempotentes, conmutativos y asociativos. Para la unión de conjuntos, el elemento superior es \emptyset y el elemento inferior es U , el conjunto universal, ya que para cualquier subconjunto x de U , $\emptyset \cup x = x$ y $U \cup x = U$. Para la intersección de conjuntos, \top es U y \perp es \emptyset . V , el dominio de valores del semi-lattice, es el conjunto de todos los subconjuntos de U , que algunas veces se le llama el *conjunto potencia* de U y se denota como 2^U .

Para todas las x y y en V , $x \cup y = x$ implica $x \supseteq y$; por lo tanto, el orden parcial impuesto por una unión de conjuntos es \supseteq , el conjunto inclusión. De manera correspondiente, el orden parcial impuesto por la intersección de conjuntos es \subseteq , la contención de conjuntos. Es decir, para la intersección de conjuntos, los conjuntos con menos elementos se consideran más pequeños en el orden parcial. No obstante, para la unión de conjuntos, los conjuntos con *más* elementos se consideran más pequeños en el orden parcial. Es contra intuitivo decir que los conjuntos de mayor tamaño son más pequeños en el orden parcial; sin embargo, esta situación es una consecuencia inevitable de las definiciones.⁶

Como vimos en la sección 9.2, por lo general, hay muchas soluciones para un conjunto de ecuaciones de un flujo de datos, en donde la solución más grande (en el sentido del orden parcial \leq) es la más precisa. Por ejemplo, en las definiciones de alcance, la más precisa de todas las soluciones a las ecuaciones de un flujo de datos es la que tiene el menor número de definiciones, que corresponde al mayor elemento en el orden parcial definido por la operación de reunión, la unión. En las expresiones disponibles, la solución más precisa es la que tiene el mayor número de expresiones. De nuevo, es la solución más grande en el orden parcial definido por la intersección como operación de reunión. \square

Límites inferiores más grandes

Hay otra relación útil entre la operación de reunión y el orden parcial que impone. Suponga que (V, \wedge) es un semi-lattice. Un *límite menor más grande* (o *glb*) de los elementos del dominio x y y es un elemento g tal que:

1. $g \leq x$,
2. $g \leq y$, y
3. Si z es un elemento tal que $z \leq x$ y $z \leq y$, entonces $z \leq g$.

Resulta que el operador de reunión de x y y es su único límite inferior más grande. Para ver por qué, hagamos que $g = x \wedge y$. Observe que:

⁶Y si definiéramos el orden parcial como \geq en vez de \leq , entonces el problema surgiría cuando el operador de reunión fuera la intersección, aunque no para la unión.

Uniones, límites superiores más pequeños (lub) y lattices

En simetría con la operación glb sobre los elementos de un poset, podemos definir el *límite superior más pequeño* (o *lub*) de los elementos x y y como aquél elemento b tal que $x \leq b$, $y \leq b$, y si z es cualquier elemento tal que $x \leq z$ y $y \leq z$, entonces $b \leq z$. Podemos mostrar que hay por lo menos un elemento b así, si es que existe.

En un verdadero *lattice*, hay dos operaciones sobre elementos del dominio, la operación de reunión \wedge , que hemos visto, y el operador *combinación*, denotado como \vee , el cual proporciona el lub de dos elementos (que, por lo tanto, siempre deben existir en el lattice). Sólo hemos hablado sobre los “semij” lattices, en donde sólo existe uno de los operadores de reunión y de combinación. Es decir, nuestros semi-lattices son *semi-laticces de reunión*. También podríamos hablar de los *semi-laticces de combinación*, en donde sólo existe el operador de combinación, y de hecho cierta literatura sobre el análisis de los programas utiliza la notación de semi-lattices de combinación. Como la literatura tradicional sobre flujos de datos habla de los semi-lattices de reunión, también lo haremos en este libro.

- $g \leq x$, ya que $(x \wedge y) \wedge x = x \wedge y$. La prueba implica usos simples de condiciones asocia-
tividad, conmutatividad e idempotencia. Es decir,

$$\begin{aligned} g \wedge x &= ((x \wedge y) \wedge x) = (x \wedge (y \wedge x)) = \\ &= (x \wedge (x \wedge y)) = ((x \wedge x) \wedge y) = \\ &= (x \wedge y) = g \end{aligned}$$

- $g \leq y$ por un argumento similar.
- Suponga que z es un elemento tal que $z \leq x$ y $z \leq y$. Afirmamos que $z \leq g$ y, por lo tanto, z no puede ser un glb de x y y , a menos que también sea g . Como demostración: $(z \wedge g) = (z \wedge (x \wedge y)) = ((z \wedge x) \wedge y)$. Como $z \leq x$, sabemos que $(z \wedge x) = z$, por lo que $(z \wedge g) = (z \wedge y)$. Como $z \leq y$, sabemos que $z \wedge y = z$ y, por lo tanto, $z \wedge g = z$. Hemos demostrado que $z \leq g$ y concluimos que $g = x \wedge y$ es el único glb de x y y .

Diagramas de lattices

A menudo es útil dibujar el dominio V como un diagrama de lattices, el cual es un grafo cuyos nodos son los elementos de V , y cuyas flechas se dirigen hacia abajo, desde x hasta y si $y \leq x$. Por ejemplo, la figura 9.22 muestra el conjunto V para un esquema de flujo de datos con definiciones de alcance, en donde hay tres definiciones: d_1 , d_2 y d_3 . Como \leq es \supseteq , una flecha se dirige hacia abajo desde cualquier subconjunto de estas tres definiciones hasta cada uno de sus superconjuntos. Como \leq es transitivo, por convención omitimos la flecha que va desde x hasta y , siempre y

cuando haya otro camino de x a y a la izquierda en el diagrama. Por ende, aunque $\{d_1, d_2, d_3\} \leq \{d_1\}$, no dibujamos esta flecha ya que se representa mediante el camino a través de $\{d_1, d_2\}$, por ejemplo.

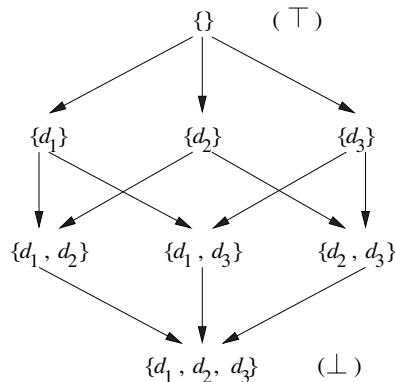


Figura 9.22: Lattices de subconjuntos de definiciones

También es útil observar que podemos leer el operador de reunión de dichos diagramas. Como $x \wedge y$ es el glb, siempre es la z más alta para la cual hay caminos hacia abajo, desde x o y . Por ejemplo, si x es $\{d_1\}$ y y es $\{d_2\}$, entonces z en la figura 9.22 es $\{d_1, d_2\}$, lo cual tiene sentido, ya que el operador de reunión es la unión. El elemento superior aparecerá en la parte superior del diagrama del lattice; es decir, hay un camino hacia abajo, desde \top hacia cada elemento. De igual forma, el elemento inferior aparecerá en la parte inferior, con un camino hacia abajo desde cada elemento hacia \perp .

Laticces de productos

Mientras que la figura 9.22 sólo involucra tres definiciones, el diagrama de laticces de un programa común puede ser bastante extenso. El conjunto de valores del flujo de datos es el conjunto de potencia de las definiciones que, por lo tanto, contiene 2^n elementos si hay n definiciones en el programa. No obstante, el que una definición llegue a un programa o no es independiente de la capacidad de alcance de las demás definiciones. Así, podemos expresar el lattice⁷ de definiciones en términos de un “lattice de productos”, creado a partir de un simple lattice para cada definición. Es decir, si sólo hubiera una definición de d en el programa, entonces el lattice tendría dos elementos: $\{\}$, el conjunto vacío, que es el elemento superior, y $\{d\}$, que es el elemento inferior.

De manera formal, podemos construir laticces de productos de la siguiente manera. Suponga que (A, \wedge_A) y (B, \wedge_B) son (semi) laticces. El *lattice de productos* para estos dos lattice se define de la siguiente manera:

1. El domino del lattice de productos es $A \times B$.

⁷En esta explicación y de aquí en adelante, con frecuencia omitiremos el “semi”, ya que los laticces como el de esta explicación tienen un operador de combinación o lub, aun cuando no lo utilizamos.

2. El operador de reunión \wedge para el lattice de productos se define de la siguiente forma. Si (a, b) y (a', b') son elementos del dominio del lattice de productos, entonces

$$(a, b) \wedge (a', b') = (a \wedge a', b \wedge b'). \quad (9.19)$$

Es simple expresar el orden parcial \leq para el lattice de productos en términos de los órdenes parciales \leq_A y \leq_B para A y B

$$(a, b) \leq (a', b') \text{ si y sólo si } a \leq_A a' \text{ y } b \leq_B b'. \quad (9.20)$$

Para ver por qué (9.20) sigue a partir de (9.19) debemos tener en cuenta que:

$$(a, b) \wedge (a', b') = (a \wedge_A a', b \wedge_B b').$$

Por lo tanto, podríamos preguntar ¿bajo qué circunstancias $(a \wedge_A a', b \wedge_B b') = (a, b)$? Esto ocurre exactamente cuando $a \wedge_A a' = a$ y $b \wedge_B b' = b$. Pero estas dos condiciones son las mismas que $a \leq_A a'$ y $b \leq_B b'$.

El producto de los lattices es una operación asociativa, por lo que podemos demostrar que las reglas (9.19) y (9.20) se extienden a cualquier número de lattices. Es decir, si recibimos los lattices (A_i, \wedge_i) para $i = 1, 2, \dots, k$, entonces el producto de todos los k lattices, en este orden, tiene el dominio $A_1 \times A_2 \times \dots \times A_k$, un operador de reunión definido por

$$(a_1, a_2, \dots, a_k) \wedge (b_1, b_2, \dots, b_k) = (a_1 \wedge_1 b_1, a_2 \wedge_2 b_2, \dots, a_k \wedge_k b_k).$$

y un orden parcial definido por

$$(a_1, a_2, \dots, a_k) \leq (b_1, b_2, \dots, b_k) \text{ si y sólo si } a_i \leq b_i \text{ para todas las } i.$$

Altura de un semi-lattice

Podemos aprender algo acerca de la velocidad de convergencia de un algoritmo de análisis de flujo de datos, si estudiamos la “altura” del semi-lattice asociado. Una *cadena ascendente* en un poset (V, \leq) es una secuencia en donde $x_1 < x_2 < \dots < x_n$. La *altura* de un semi-lattice es el número más grande de relaciones $<$ en cualquier cadena ascendente; es decir, la altura es uno menos que el número de elementos en la cadena. Por ejemplo, la altura del semi-lattice de definiciones de alcance para un programa con n definiciones es n .

Es mucho más sencillo mostrar la convergencia de un algoritmo de flujo de datos iterativo si el semi-lattice tiene una altura finita. Es evidente que un lattice que consiste en un conjunto finito de valores tendrá una altura finita; también es posible que un lattice con un número infinito de valores tenga una altura finita. El lattice que se utiliza en el algoritmo de propagación de constantes es uno de estos ejemplos que examinaremos más de cerca en la sección 9.4.

9.3.2 Funciones de transferencia

La familia de funciones de transferencia $F: V \rightarrow V$ en un marco de trabajo de un flujo de datos tiene las siguientes propiedades:

1. F tiene una función de identidad I , de tal forma que $I(x) = x$ para todas las x en V .
2. F se cierra bajo la composición; es decir, para dos funciones f y g cualesquiera en F , la función h definida por $h(x) = g(f(x))$ está en F .

Ejemplo 9.21: En las definiciones de alcance, F tiene la identidad, la función en donde *gen* y *eliminar* son el conjunto vacío. El cierre bajo la composición se mostró de hecho en la sección 9.2.4; aquí repetiremos el argumento brevemente. Suponga que tenemos dos funciones:

$$f_1(x) = G_1 \cup (x - K_1) \text{ y } f_2(x) = G_2 \cup (x - K_2).$$

Entonces:

$$f_2(f_1(x)) = G_2 \cup ((G_1 \cup (x - K_1)) - K_2).$$

El lado derecho de la ecuación anterior es algebraicamente equivalente a:

$$(G_2 \cup (G_1 - K_2)) \cup (x - (K_1 \cup K_2)).$$

Si hacemos que $K = K_1 \cup K_2$ y $G = G_2 \cup (G_1 - K_2)$, entonces hemos demostrado que la composición de f_1 y f_2 , que es $f(x) = G \cup (x - K)$, es de la forma que la convierte en miembro de F . Si consideramos las expresiones disponibles, los mismos argumentos que se utilizan para las definiciones de alcance también demuestran que F tiene una identidad y se cierra bajo la composición. \square

Marcos de trabajo monótonos

Para crear un algoritmo iterativo para el trabajo del análisis de un flujo de datos, necesitamos que el marco de trabajo del flujo de datos cumpla con una condición más. Decimos que un marco de trabajo es *monótono* si cuando aplicamos una función de transferencia f en F a dos miembros de V , siendo el primero no mayor que el segundo, el primer resultado no es mayor que el segundo.

De manera formal, el marco de datos de un flujo de datos (D, F, V, \wedge) es *monótono* si:

$$\text{Para todas las } x \text{ y } y \text{ en } V \text{ y } f \text{ en } F, x \leq y \text{ implica que } f(x) \leq f(y). \quad (9.22)$$

De manera equivalente, la monotonicidad puede definirse como:

$$\text{Para todas las } x \text{ y } y \text{ en } V \text{ y } f \text{ en } F, f(x \wedge y) \leq f(x) \wedge f(y). \quad (9.23)$$

La ecuación (9.23) dice que si tomamos la reunión de dos valores y después aplicamos f , el resultado nunca será mayor de lo que se obtiene al aplicar f a los valores, primero de manera individual y después “reuniendo” los resultados. Como las dos definiciones de monotonicidad parecen tan distintas, ambas son útiles. Descubriremos que una o la otra es más útil bajo distintas circunstancias. Más adelante realizaremos el bosquejo de una prueba para demostrar que, sin duda, son equivalentes.

Primero vamos a suponer que se cumple la (9.22) y demostraremos que la (9.23) es válida. Como $x \wedge y$ es el límite inferior más grande de x y y , sabemos que:

$$x \wedge y \leq x \text{ y } x \wedge y \leq y.$$

En consecuencia, según (9.22),

$$f(x \wedge y) \leq f(x) \text{ y } f(x \wedge y) \leq f(y).$$

Como $f(x) \wedge f(y)$ es el límite inferior más grande de $f(x)$ y $f(y)$, tenemos a (9.23).

Por el contrario, vamos a suponer que se cumple la (9.23) y demostraremos la (9.22). Suponemos que $x \leq y$ y usamos (9.23) para concluir que $f(x) \leq f(y)$, con lo cual se demuestra la (9.22). La ecuación (9.23) nos dice que:

$$f(x \wedge y) \leq f(x) \wedge f(y).$$

Pero como se supone que $x \leq y$, $x \wedge y = x$, por definición. Así, la ecuación (9.23) dice que:

$$f(x) \leq f(x) \wedge f(y).$$

Como $f(x) \wedge f(y)$ es el glb de $f(x)$ y $f(y)$, sabemos que $f(x) \wedge f(y) \leq f(y)$. Por ende,

$$f(x) \leq f(x) \wedge f(y) \leq f(y)$$

y la ecuación (9.23) implica a la (9.22).

Marcos de trabajo distribuidos

A menudo, un marco de trabajo obedece a una condición más fuerte que la (9.23), a lo cual le llamamos la *condición distributiva*,

$$f(x \wedge y) = f(x) \wedge f(y)$$

para todas las x y y en V , y f en F . Sin duda, si $a = b$, entonces $a \wedge b = a$ por la idempotencia, así que $a \leq b$. Por ende, la distributividad implica a la condición de monotonicidad, aunque esto no es válido al revés.

Ejemplo 9.24: Hagamos que y y z sean conjuntos de definiciones en el marco de trabajo de las definiciones de alcance. Hagamos que f sea una función definida por $f(x) = G \cup (x - K)$ para algunos conjuntos de definiciones G y K . Podemos verificar que el marco de trabajo de las definiciones de alcance cumpla con la distributividad, si comprobamos que:

$$G \cup ((y \cup z) - K) = (G \cup (y - K)) \cup (G \cup (z - K))$$

Aunque la ecuación anterior puede parecer formidable, considere primero aquellas definiciones en G . Estas definiciones sin duda se encuentran en los conjuntos definidos tanto por el lado izquierdo como por el derecho. Así, sólo tenemos que considerar las definiciones que no están en G . En ese caso, podemos eliminar a G de todas partes, y verificar la siguiente igualdad:

$$(y \cup z) - K = (y - K) \cup (z - K).$$

Esta última igualdad puede comprobarse con facilidad, mediante el uso de un diagrama de Venn. \square

9.3.3 El algoritmo iterativo para los marcos de trabajo generales

Podemos generalizar el Algoritmo 9.11 para hacer que funcione en una gran variedad de problemas de flujo de datos.

Algoritmo 9.25: Solución iterativa a los marcos de trabajo de flujo de datos generales.

ENTRADA: Un marco de trabajo del flujo de datos con los siguientes componentes:

1. Un grafo de flujo de datos, con nodos ENTRADA y SALIDA etiquetados en forma especial.
2. Una dirección del flujo de datos D .
3. Un conjunto de valores V .
4. Un operador de reunión \wedge .
5. Un conjunto de funciones F , en donde f_B en F es la función de transferencia para el bloque B .
6. Un valor constante v_{ENTRADA} o v_{SALIDA} en V , que representa la condición delimitadora para los marcos de trabajo de avance e inversos, respectivamente.

SALIDA: Valores en V para $\text{ENT}[B]$ y $\text{SAL}[B]$ para cada bloque B en el grafo de flujo de datos.

MÉTODO: Los algoritmos para resolver problemas de flujos de datos hacia delante y hacia atrás se muestran en las figuras 9.23(a) y 9.23(b), respectivamente. Al igual que con los algoritmos de flujo de datos iterativos familiares de la sección 9.2, calculamos ENT y SAL para cada bloque mediante la aproximación sucesiva. \square

Es posible escribir las versiones hacia delante y hacia atrás del algoritmo 9.25, de tal forma que una función que implemente el operador de reunión sea un parámetro, así como una función que implemente la función de transferencia para cada bloque. El mismo grafo de flujos y el valor delimitador son también parámetros. De esta forma, el implementador del compilador puede evitar tener que volver a codificar el algoritmo iterativo básico para cada marco de trabajo del flujo de datos utilizado por la fase de optimización del compilador.

Podemos usar el marco de trabajo abstracto que hemos visto hasta ahora para probar varias propiedades útiles del algoritmo iterativo:

1. Si el algoritmo 9.25 converge, el resultado es una solución a las ecuaciones del flujo de datos.
2. Si el marco de trabajo es monótono, entonces la solución encontrada es el punto fijo máximo (MFP) de las ecuaciones del flujo de datos. Un *punto fijo máximo* es una solución con la propiedad de que en cualquier otra solución, los valores de $\text{ENT}[B]$ y $\text{SAL}[B]$ son \leq los valores correspondientes del MFP.
3. Si el semi-lattice del marco de trabajo es monótono y de altura finita, entonces garantiza que el algoritmo va a converger.

```

1)  $\text{SAL}[\text{ENTRADA}] = v_{\text{ENTRADA}};$ 
2) for (cada bloque básico  $B$  que no sea ENTRADA)  $\text{SAL}[B] = \top$ ;
3) while (ocurran cambios a cualquier SAL)
4)   for (cada bloque básico  $B$  que no sea ENTRADA) {
5)      $\text{ENT}[B] = \bigwedge_{P \text{ un predecesor de } B} \text{SAL}[P];$ 
6)      $\text{SAL}[B] = f_B(\text{ENT}[B]);$ 
}

```

(a) algoritmo iterativo para un problema de flujo de datos hacia delante.

```

1)  $\text{ENT}[\text{SALIDA}] = v_{\text{SALIDA}};$ 
2) for (cada bloque básico  $B$  que no sea SALIDA)  $\text{ENT}[B] = \top$ ;
3) while (ocurran cambios a cualquier ENT)
4)   for (cada bloque básico  $B$  que no sea SALIDA) {
5)      $\text{SAL}[B] = \bigwedge_{S \text{ un sucesor de } B} \text{ENT}[S];$ 
6)      $\text{ENT}[B] = f_B(\text{SAL}[B]);$ 
}

```

(b) Algoritmo iterativo para un problema de flujo de datos hacia atrás.

Figura 9.23: Versiones hacia delante y hacia atrás del algoritmo iterativo

Vamos a argumentar estos puntos, asumiendo que el marco de trabajo es de avance. El caso de los marcos de trabajo inversos es en esencia el mismo. La primera propiedad es fácil de demostrar. Si las ecuaciones no se satisfacen para cuando termina el ciclo while, entonces habrá por lo menos un cambio a un valor de SAL (en el caso de avance) o ENT (en el caso inverso), y debemos recorrer el ciclo otra vez.

Para demostrar la segunda propiedad, primero mostramos que los valores que reciben $\text{ENT}[B]$ y $\text{SAL}[B]$ para cualquier B sólo pueden disminuir (en el sentido de la relación \leq para los lattices) a medida que el algoritmo itera. Esta afirmación puede demostrarse mediante la inducción.

BASE: El caso base es para mostrar que el valor de $\text{ENT}[B]$ y $\text{SAL}[B]$ después de la primera iteración no es mayor que el valor inicializado. Esta instrucción es trivial, ya que $\text{ENT}[B]$ y $\text{SAL}[B]$ para todos los bloques $B \neq \text{ENTRADA}$ se inicializan con \top .

INDUCCIÓN: Suponga que después de la k -ésima iteración, ninguno de los valores es mayor que los que se obtienen después de la $(k-1)$ -ésima iteración, y muestran lo mismo para la iteración $k+1$, comparada con la iteración k . La línea (5) de la figura 9.23(a) tiene:

$$\text{ENT}[B] = \bigwedge_{P \text{ un predecesor de } B} \text{SAL}[P].$$

Vamos a usar la notación $\text{ENT}[B]^i$ y $\text{SAL}[B]^i$ para denotar los valores de $\text{ENT}[B]$ y $\text{SAL}[B]$ después de la iteración i . Suponiendo que $\text{SAL}[P]^k \leq \text{SAL}[P]^{k-1}$, sabemos que $\text{ENT}[B]^{k+1} \leq$

$\text{ENT}[B]^k$, debido a las propiedades del operador de reunión. A continuación, la línea (6) dice que:

$$\text{SAL}[B] = f_B(\text{ENT}[B]).$$

Como $\text{ENT}[B]^{k+1} \leq \text{ENT}[B]^k$, tenemos que $\text{SAL}[B]^{k+1} \leq \text{SAL}[B]^k$ por la monotonicidad.

Hay que tener en cuenta que cualquier cambio que se observe para los valores de $\text{ENT}[B]$ y $\text{SAL}[B]$ es necesario para satisfacer la ecuación. Los operadores de reunión devuelven el límite inferior más grande de sus entradas, y las funciones de transferencia devuelven la única solución consistente con el bloque en sí y su entrada dada. Por ende, si el algoritmo iterativo termina, el resultado debe tener valores que sean por lo menos tan grandes como los valores correspondientes en cualquier otra solución; es decir, el resultado del Algoritmo 9.25 es el MFP de la ecuación.

Por último, considere el tercer punto, en donde el marco de trabajo del flujo de datos tiene una altura finita. Como todos los valores de los conjuntos $\text{ENT}[B]$ y $\text{SAL}[B]$ disminuyen con cada cambio, y el algoritmo se detiene si en cierta ronda no cambia nada, se garantiza que el algoritmo convergerá después de una cantidad de rondas no mayor que el producto de la altura del marco de trabajo y el número de nodos del grafo de flujo.

9.3.4 Significado de una solución de un flujo de datos

Ahora sabemos que la solución que se encuentra usando el algoritmo iterativo es el punto fijo máximo, pero ¿qué representa el resultado, desde un punto de vista de la semántica del programa? Para comprender la solución del marco de trabajo de un flujo de datos (D, F, V, \wedge) , primero vamos a describir lo que sería una solución ideal para el marco de trabajo. Demostraremos que, en general, no puede obtenerse la solución ideal, pero que el Algoritmo 9.25 se aproxima al ideal de una manera conservadora.

La solución ideal

Sin perder la generalidad, vamos a suponer por ahora que el marco de trabajo del flujo de datos de interés es un problema que fluye hacia adelante. Considere el punto de entrada de un bloque básico B . La solución ideal empieza por encontrar todos los caminos de ejecución *posibles*, que conducen desde la entrada del programa hasta el inicio de B . Un camino es “posible” sólo si hay algún cálculo del programa que siga exactamente ese camino. Entonces, la solución ideal sería calcular el valor del flujo de datos al final de cada posible camino y aplicar el operador de reunión a estos valores para encontrar su límite inferior más grande. Así, ninguna ejecución del programa puede producir un valor más pequeño para ese punto del programa. Además, el límite es estrecho; no hay un valor mayor del flujo de datos que un glb para el valor que se calcula a lo largo de cada posible camino hacia B en el grafo de flujo.

Ahora trataremos de definir la solución ideal con más formalidad. Para cada bloque B en un grafo de flujo, hagamos que f_B sea la función de transferencia para B . Considere cualquier camino

$$P = \text{ENTRADA} \rightarrow B_1 \rightarrow B_2 \rightarrow \cdots \rightarrow B_{k-1} \rightarrow B_k$$

desde el nodo inicial ENTRADA hasta cierto bloque B_k . El camino del programa puede tener círculos, por lo que un bloque básico puede aparecer varias veces en el camino P . Defina la función

de transferencia para P , f_P , como la composición de $f_{B_1}, f_{B_2} \dots, f_{B_{k-1}}$. Observe que f_{B_k} no forma parte de la composición, lo cual refleja el hecho de que este camino se toma para llegar al inicio del bloque B_k , no a su final. El valor del flujo de datos creado al ejecutar este camino es, por lo tanto, $f_P(v_{\text{ENTRADA}})$, en donde v_{ENTRADA} es el resultado de la función de transferencia constante que representa al nodo inicial ENTRADA. El resultado ideal para el bloque B es por ende:

$$\text{IDEAL}[B] = \bigwedge_{P, \text{ un posible camino desde ENTRADA hasta } B} f_P(v_{\text{ENTRADA}}).$$

Afirmamos que, en términos del orden parcial \leq teórico del lattice para el marco de trabajo en cuestión,

- Cualquier respuesta que sea mayor que IDEAL es incorrecta.
- Cualquier valor más pequeño o igual que el ideal es conservador; es decir, seguro.

Por intuición, entre más cercano sea el valor al ideal, será más preciso.⁸ Para ver por qué las soluciones deben ser \leq que la solución ideal, observe que cualquier solución más grande que IDEAL para cualquier bloque podría obtenerse ignorando cierto camino de ejecución que el programa pudiera tomar, y no podemos estar seguros que no haya cierto efecto a lo largo de ese camino para invalidar cualquier mejora al programa que podríamos realizar, con base en la solución más grande. Por el contrario, cualquier solución menor que IDEAL puede verse como la inclusión de ciertos caminos que, o no existen en el grafo de flujo, o que sí existen pero que el programa nunca podrá seguir. La solución menor sólo permitiría transformaciones que sean correctas para todas las posibles ejecuciones del programa, pero puede prohibir ciertas transformaciones que IDEAL permitiría.

La solución de reunión sobre los caminos (Meet-Over-Paths, MOP)

Sin embargo, como vimos en la sección 9.1, el proceso de buscar todos los posibles caminos de ejecución es indecidible. Por lo tanto, debemos aproximar. En la abstracción del flujo de datos, suponemos que puede tomarse cada camino en el grafo de flujo. Por ende, podemos definir la solución de reunión sobre los caminos para B de la siguiente forma:

$$\text{MOP}[B] = \bigwedge_{P, \text{ un camino desde ENTRADA hasta } B} f_P(v_{\text{ENTRADA}}).$$

Observe que, en lo que respecta a IDEAL, la solución MOP[B] proporciona valores para ENT[B] en los marcos de trabajo con flujo hacia delante. Si consideramos los marcos de trabajo con flujo hacia atrás, entonces tenemos que considerar a MOP[B] como un valor para SAL[B].

Los caminos que se consideran en la solución MOP son un superconjunto de todos los caminos que es posible ejecutar. Por ende, la solución MOP reúne no sólo los valores del flujo de datos de todos los caminos ejecutables, sino también los valores adicionales asociados con los caminos que no es posible ejecutar. Si tomamos la reunión de la solución ideal más los términos

⁸Observe que en los problemas hacia delante, el valor IDEAL[B] es lo que quisiéramos que fuera ENT[B]. En los problemas hacia atrás, que no veremos aquí, definiríamos a IDEAL[B] como el valor ideal de SAL[B].

adicionales, no podemos crear una solución más grande que la ideal. Así, para todos los valores de B tenemos que $MOP[B] \leq IDEAL[B]$, y simplemente diremos que $MOP \leq IDEAL$.

Comparación entre el punto fijo máximo y la solución MOP

Observe que en la solución MOP, el número de caminos considerados sigue siendo ilimitado si el grafo de flujo contiene ciclos. Por ende, la definición MOP no se presta para un algoritmo directo. Sin duda, el algoritmo iterativo no busca primero todos los caminos que conducen a un bloque básico antes de aplicar el operador de reunión.

En vez de ello,

1. El algoritmo iterativo visita los bloques básicos, no necesariamente en el orden de ejecución.
2. En cada punto de confluencia, el algoritmo aplica el operador de reunión a los valores del flujo de datos que se han obtenido hasta ese momento. Algunos de estos valores utilizados se introdujeron de manera artificial en el proceso de inicialización, lo cual no representa el resultado de ninguna ejecución desde el inicio del programa.

Entonces, ¿cuál es la relación entre la solución MOP y la solución del punto fijo máximo producida por el Algoritmo 9.25?

Primero hablaremos sobre el orden en el cual se visitan los nodos. En una iteración, podemos visitar un bloque básico antes de haber visitado a sus predecesores. Si el predecesor es el nodo ENTRADA, SAL[ENTRADA] ya se habría inicializado con el valor constante apropiado. De no ser así, significa que ya se ha inicializado con \top , un valor que no es más pequeño que la respuesta final. Por la condición de monotonía, el resultado que se obtiene al usar \top como entrada no es más pequeño que la solución deseada. En un sentido, podemos considerar que \top no representa ninguna información.

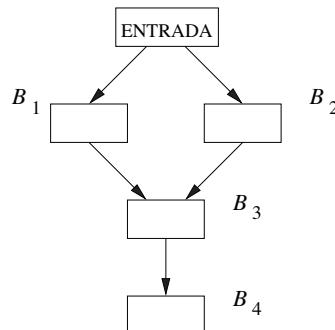


Figura 9.24: Grafo de flujo que ilustra el efecto de la reunión anticipada sobre los caminos

¿Cuál es el efecto de aplicar el operador de reunión de manera anticipada? Considere el ejemplo simple de la figura 9.24, y suponga que nos interesa el valor de $ENT[B_4]$. Por la definición de MOP,

$$\text{MOP}[B_4] = ((f_{B_3} \circ f_{B_1}) \wedge (f_{B_3} \circ f_{B_2}))(v_{\text{ENTRADA}})$$

En el algoritmo iterativo, si visitamos los nodos en el orden B_1, B_2, B_3, B_4 , entonces

$$\text{ENT}[B_4] = f_{B_3}((f_{B_1}(v_{\text{ENTRADA}}) \wedge f_{B_2}(v_{\text{ENTRADA}})))$$

Mientras que el operador de reunión se aplica al final en la definición del MOP, el algoritmo iterativo lo aplica antes. La respuesta es la misma, sólo si el marco de trabajo del flujo de datos es distributivo. Si el marco de trabajo del flujo de datos es monótono pero no distributivo, aún tenemos que $\text{ENT}[B_4] \leq \text{MOP}[B_4]$. Recuerde que, en general, una solución $\text{ENT}[B]$ es segura (conservadora) si $\text{ENT}[B] \leq \text{IDEAL}[B]$ para todos los bloques B . Con seguridad, $\text{MOP}[B] \leq \text{IDEAL}[B]$.

Ahora vamos a proporcionar un breve bosquejo de por qué en general la solución de punto fijo máximo que proporciona el algoritmo iterativo siempre es segura. Una inducción sencilla sobre i muestra que los valores que se obtienen después de i iteraciones son más pequeños o iguales que la reunión sobre todos los caminos de longitud i o menor. Pero el algoritmo iterativo sólo termina si llega a la misma respuesta que se obtendría al iterar un número ilimitado de veces. Por ende, el resultado no es mayor que la solución MOP. Como $\text{MOP} \leq \text{IDEAL}$ y $\text{MFP} \leq \text{MOP}$, sabemos que $\text{MFP} \leq \text{IDEAL}$ y, por lo tanto, la solución de punto fijo máximo proporcionada por el algoritmo iterativo es segura.

9.3.5 Ejercicios para la sección 9.3

Ejercicio 9.3.1: Construya un diagrama de lattices para el producto de tres lattices, cada uno de ellos basado en una sola definición d_i , para $i = 1, 2, 3$. ¿Cómo se relaciona su diagrama de lattices con el de la figura 9.22?

! Ejercicio 9.3.2: En la sección 9.3.3 argumentamos que si el marco de trabajo tiene una altura finita, entonces el algoritmo iterativo converge. He aquí un ejemplo en donde el marco de trabajo no tiene una altura finita, y el algoritmo iterativo no converge. Hagamos que el conjunto de valores V sea los números reales no negativos, y que el operador de reunión sea el mínimo. Hay tres funciones de transferencia:

- i. La identidad, $f_I(x) = x$.
- ii. “mitad”; es decir, la función $f_M(x) = x/2$.
- iii. “uno”; es decir, la función $f_U(x) = 1$.

El conjunto de funciones de transferencia F es estas tres funciones, más las funciones formadas al componerlas en todas las formas posibles.

- a) Describa el conjunto F .
- b) ¿Cuál es la relación \leq para este marco de trabajo?

- c) Proporcione un ejemplo de un grafo de flujo con funciones de transferencia asignadas, de manera que el Algoritmo 9.25 no converja.
- d) ¿Este marco de trabajo es monótono? ¿Es distributivo?

! Ejercicio 9.3.3: Argumentamos que el Algoritmo 9.25 converge si el marco de trabajo es monótono y de altura finita. He aquí un ejemplo de un marco de trabajo que muestra que la monotonía es esencial; no basta con una altura finita. El dominio V es $\{1, 2\}$, el operador de reunión es \min , y el conjunto de funciones F es sólo la función identidad (f_I) y la función “intercambio” ($f_S(x) = 3 - x$), que intercambia 1 y 2.

- a) Muestre que este marco de trabajo es de altura finita, pero no monótono.
- b) Proporcione un ejemplo del grafo de flujo y la asignación de funciones de transferencia, de manera que el Algoritmo 9.25 no converja.

! Ejercicio 9.3.4: Hagamos que $MOP_i[B]$ sea la reunión sobre todos los caminos de longitud i o menores, desde la entrada hasta el bloque B . Demuestre que después de i iteraciones del Algoritmo 9.25, $ENT[B] \leq MOP_i[B]$. Además, demuestre que como consecuencia, si el Algoritmo 9.25 converge, entonces convergerá en algo que sea \leq que la solución MOP.

! Ejercicio 9.3.5: Suponga que el conjunto F de funciones para un marco de trabajo consiste sólo en funciones de la forma gen-eliminar. Es decir, el dominio V es el conjunto de potencia de cierto conjunto, y $f(x) = G \cup (x - K)$ para ciertos conjuntos G y K . Demuestre que si el operador de reunión es (a) la unión o (b) la intersección, entonces el marco de trabajo es distributivo.

9.4 Propagación de constantes

Todos los esquemas del flujo de datos que vimos en la sección 9.2 son en realidad ejemplos simples de marcos de trabajo distributivos con una altura finita. Por ende, el Algoritmo 9.25 iterativo se aplica a ellos, ya sea en su versión hacia delante o hacia atrás, y produce la solución MOP en cada caso. En esta sección, examinaremos con detalle un útil marco de trabajo del flujo de datos con propiedades más interesantes.

Recuerde que la propagación de constantes, o “cálculo previo de constantes”, sustituye a las expresiones que se evalúan con la misma constante cada vez que se ejecutan, por esa constante. El marco de trabajo de propagación de constantes que describiremos a continuación es distinto de todos los problemas de flujos de datos que hemos visto hasta ahora, en cuanto a que:

- a) Tiene un conjunto ilimitado de posibles valores del flujo de datos, incluso para un grafo de flujo fijo.
- b) No es distributivo.

La propagación de constantes es un problema del flujo de datos hacia delante. El semi-lattice que representa los valores del flujo de datos y la familia de funciones de transferencia se presentan a continuación.

9.4.1 Valores del flujo de datos para el marco de trabajo de propagación de constantes

El conjunto de valores del flujo de datos es un lattice de productos, con un componente para cada variable en un programa. El lattice para una sola variable consiste en lo siguiente:

1. Todas las constantes apropiadas para el tipo de la variable.
2. El valor NAC, que representa “no es una constante”. Una variable se asigna a este valor si se determina que no tiene un valor constante. La variable puede haber recibido un valor de entrada, o se puede derivar de una variable que no sea una constante, o se le pueden haber asignado distintas constantes a lo largo de diferentes caminos que conducen hacia el mismo punto en el programa.
3. El valor UNDEF, que representa “indefinido”. Una variable recibe este valor si todavía no hay nada que pueda afirmarse; presuntamente, no se ha descubierto una definición de la variable para llegar al punto en cuestión.

Observe que NAC y UNDEF no son iguales; en esencia son valores opuestos. NAC indica que hemos visto muchas formas en las que podría definirse una variable, por lo cual sabemos que no es constante; UNDEF indica que hemos visto muy poco sobre la variable, por lo cual no podemos saber nada certero.

El semi-lattice para una variable típica con valor entero se muestra en la figura 9.25. Aquí el elemento superior es UNDEF, y el elemento inferior es NAC. Es decir, el valor más grande en el orden parcial es UNDEF y el menor es NAC. Los valores constantes están desordenados, pero son menores que UNDEF y mayores que NAC. Como vimos en la sección 9.3.1, la reunión de dos valores es su límite menor más grande. Así, para todos los valores v ,

$$\text{UNDEF} \wedge v = v \text{ y } \text{NAC} \wedge v = \text{NAC}.$$

Para cualquier constante c ,

$$c \wedge c = c$$

y dadas dos constantes distintas c_1 y c_2 ,

$$c_1 \wedge c_2 = \text{NAC}.$$

Un valor del flujo de datos para este marco de trabajo es un mapa que va desde cada variable en el programa, hacia uno de los valores en el semi-lattice de constantes. El valor de una variable v en un mapa m se denota mediante $m(v)$.

9.4.2 La reunión para el marco de trabajo de propagación de constantes

El semi-lattice de valores del flujo de datos es sólo el producto de los semi-lattices como la figura 9.25, uno para cada variable. Por ende, $m \leq m'$ si y sólo si para todas las variables v , tenemos que $m(v) \leq m'(v)$. Dicho de otra forma, $m \wedge m' = m''$ si $m''(v) = m(v) \wedge m'(v)$ para todas las variables v .

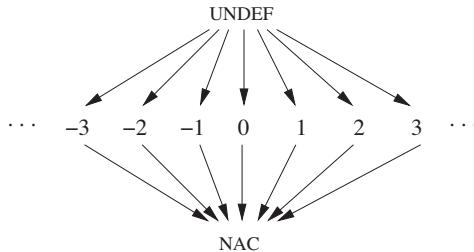


Figura 9.25: Semi-lattice que representa a los posibles “valores” de una sola variable entera

9.4.3 Funciones de transferencia para el marco de trabajo de propagación de constantes

A continuación vamos a suponer que un bloque básico contiene sólo una instrucción. Las funciones de transferencia para los bloques básicos que contienen varias instrucciones pueden construirse mediante la composición de las funciones correspondientes a las instrucciones individuales. El conjunto F consiste en ciertas funciones de transferencia que aceptan un mapa de variables asignadas a valores en el semi-lattice de constantes, y devuelven otro mapa de ese tipo.

F contiene la función de identidad, la cual recibe un mapa como entrada y devuelve ese mismo mapa como salida. F también contiene la función de transferencia de constantes para el nodo ENTRADA. Esta función de transferencia, que recibe un mapa de entrada, devuelve un mapa m_0 , en donde $m_0(v) = \text{UNDEF}$ para todas las variables v . Esta condición delimitadora tiene sentido, ya que antes de ejecutar cualquier instrucción del programa no hay definiciones para ninguna variable.

En general, hagamos que f_s sea la función de transferencia de la instrucción s , y hagamos que m y m' representen valores del flujo de datos de tal forma que $m' = f(m)$. Describiremos a f_s en términos de la relación entre m y m' .

1. Si s no es una instrucción de asignación, entonces f_s es simplemente la función de identidad.
2. Si s es una asignación para la variable x , entonces $m'(v) = m(v)$ para todas las variables $v \neq x$, siempre y cuando sea válida una de las siguientes condiciones:
 - (a) Si el lado derecho (Right-Hand-Side, RHS) de la instrucción s es una constante c , entonces $m'(x) = c$.
 - (b) Si el RHS es de la forma $y + z$, entonces⁹

$$m'(x) = \begin{cases} m(y) + m(z) & \text{si } m(y) \text{ y } m(z) \text{ son valores constantes} \\ \text{NAC} & \text{si } m(y) \text{ o } m(z) \text{ es NAC} \\ \text{UNDEF} & \text{en cualquier otro caso} \end{cases}$$

- (c) Si el RHS es cualquier otra expresión (por ejemplo, una llamada a una función o una asignación a través de un apuntador), entonces $m'(x) = \text{NAC}$.

⁹Como siempre, $+$ representa a un operador genérico, no necesariamente la suma.

9.4.4 Monotonía en el marco de trabajo de propagación de constantes

Vamos a demostrar que el marco de trabajo de propagación constante es monótono. Primero, podemos considerar el efecto de una función f_s en una sola variable. En todos los casos excepto el 2(b), f_s no modifica el valor de $m(x)$, o cambia el mapa para regresar una constante o NAC. En estos casos, f_s debe sin duda ser monótona.

Para el caso 2(b), el efecto de f_s se tabula en la figura 9.26. Las columnas primera y segunda representan los posibles valores de entrada de y y z ; la última representa el valor de salida de x . Los valores se ordenan del más grande al más pequeño en cada columna o subcolumna. Para mostrar que la función es monótona, comprobamos que para cada posible valor de entrada de y , el valor de x no se haga más grande a medida que el valor de z se hace más pequeño. Por ejemplo, en el caso en donde y tiene un valor constante c_1 , a medida que varía el valor de z desde UNDEF hasta c_2 , y luego hasta NAC, el valor de x varía desde UNDEF, hasta $c_1 + c_2$, y después hasta NAC, respectivamente. Podemos repetir este procedimiento para todos los valores posibles de y . Debido a la simetría, ni siquiera tenemos que repetir el procedimiento para el segundo operando, para poder concluir que el valor de salida no puede hacerse más grande a medida que la entrada se hace más pequeña.

$m(y)$	$m(z)$	$m'(x)$
UNDEF	UNDEF	UNDEF
	c_2	UNDEF
	NAC	NAC
c_1	UNDEF	UNDEF
	c_2	$c_1 + c_2$
	NAC	NAC
NAC	UNDEF	NAC
	c_2	NAC
	NAC	NAC

Figura 9.26: La función de transferencia de propagación de constantes para $x = y+z$

9.4.5 La distributividad del marco de trabajo de propagación de constantes

El marco de trabajo de propagación de constantes es monótono según su definición, pero no distributivo. Es decir, la solución iterativa de punto fijo máximo es segura, pero puede ser más pequeña que la solución MOP. Un ejemplo demostrará que el marco de trabajo no es distributivo.

Ejemplo 9.26: En el programa de la figura 9.27, x y y se establecieron a 2 y 3 en el bloque B_1 , y a 3 y 2, respectivamente, en el bloque B_2 . Sabemos que sin importar qué camino se tome, el valor de z al final del bloque B_3 es 5. Sin embargo, el algoritmo iterativo no descubre este hecho. En lugar de eso, aplica el operador de reunión a la entrada de B_3 , y obtiene NACs como

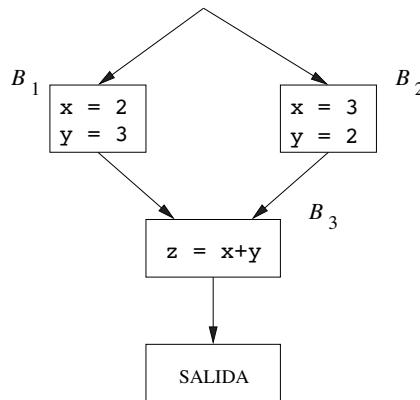


Figura 9.27: Un ejemplo que demuestra que el marco de trabajo de propagación de constantes no es distributivo

valores para x y y . Como la suma de dos NACs produce un NAC, la salida producida por el Algoritmo 9.25 es que $z = \text{NAC}$ al final del programa. El resultado es seguro, pero impreciso. El Algoritmo 9.25 es impreciso, ya que no lleva el registro de la correlación de cada vez que x es 2, y es 3, y viceversa. Es posible, pero considerablemente más costoso, usar un marco de trabajo más complejo que rastree todas las igualdades posibles que sean válidas entre pares de expresiones que involucren a las variables en el programa; en el ejercicio 9.4.2 hablaremos sobre este método.

En teoría, podemos atribuir esta pérdida de precisión a la no distributividad del marco de trabajo de propagación de constantes. Hagamos que f_1 , f_2 y f_3 sean las funciones de transferencia que representan a los bloques B_1 , B_2 y B_3 , respectivamente. Como se muestra en la figura 9.28,

$$f_3(f_1(m_0) \wedge f_2(m_0)) < f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$$

con lo cual el marco de trabajo se convierte en no distributivo. \square

m	$m(x)$	$m(y)$	$m(z)$
m_0	UNDEF	UNDEF	UNDEF
$f_1(m_0)$	2	3	UNDEF
$f_2(m_0)$	3	2	UNDEF
$f_1(m_0) \wedge f_2(m_0)$	NAC	NAC	UNDEF
$f_3(f_1(m_0) \wedge f_2(m_0))$	NAC	NAC	NAC
$f_3(f_1(m_0))$	2	3	5
$f_3(f_2(m_0))$	3	2	5
$f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$	NAC	NAC	5

Figura 9.28: Ejemplo de funciones de transferencia no distributivas

9.4.6 Interpretación de los resultados

El valor UNDEF se utiliza en el algoritmo iterativo para dos fines: inicializar el nodo ENTRADA e inicializar los puntos interiores del programa antes de las iteraciones. El significado es un poco distinto en los dos casos. El primero dice que las variables están indefinidas al inicio de la ejecución del programa; el segundo dice que por falta de información al inicio del proceso iterativo, aproximamos la solución con el elemento superior UNDEF. Al final del proceso iterativo, las variables a la salida del nodo ENTRADA seguirán conteniendo el valor UNDEF, ya que SAL[ENTRADA] nunca cambia.

Es posible que puedan aparecer valores UNDEF en algunos otros puntos del programa. Cuando lo hacen, significa que no se han observado definiciones para esa variable a lo largo de cualquiera de los caminos que conducen hasta ese punto del programa. Observe que con la forma en que definimos el operador de reunión, siempre y cuando exista un camino que defina una variable que llegue a un punto del programa, la variable no tendrá un valor UNDEF. Si todas las definiciones que llegan a un punto del programa tienen el mismo valor constante, la variable se considera una constante, aun cuando tal vez no esté definida a lo largo de cierto camino del programa.

Al suponer que el programa es correcto, el algoritmo puede encontrar más constantes que de otro modo no lo haría. Es decir, el algoritmo elige de manera conveniente ciertos valores para aquellas variables que posiblemente están indefinidas, para que el programa pueda ser más eficiente. El cambio es legal en la mayoría de los lenguajes de programación, ya que se permite que las variables indefinidas tomen cualquier valor. Si la semántica del lenguaje requiere que todas las variables indefinidas reciban cierto valor específico, entonces debemos cambiar nuestra formulación del problema de manera acorde. Y si en vez de ello nos interesa buscar las variables que quizás estén indefinidas en un programa, podemos formular un análisis de flujo de datos distinto para proveer ese resultado (vea el ejercicio 9.4.1).

Ejemplo 9.27: En la figura 9.29, los valores de x son 10 y UNDEF en la salida de los bloques básicos B_2 y B_3 , respectivamente. Como $\text{UNDEF} \wedge 10 = 10$, el valor de x es 10 al entrar al bloque B_4 . Por ende, el bloque B_5 , en donde se usa x , puede optimizarse al sustituir x por 10. Si el camino ejecutado hubiera sido $B_1 \rightarrow B_3 \rightarrow B_4 \rightarrow B_5$, el valor de x al llegar al bloque básico B_5 hubiera sido indefinido. Por lo tanto, parece incorrecto sustituir el uso de x por 10.

No obstante, si es imposible que el predicado Q sea falso mientras que Q' es verdadero, entonces este camino de ejecución nunca ocurrirá. Aunque el programador puede estar consciente de este hecho, el poder determinarlo puede estar más allá de la capacidad de cualquier análisis de flujo de datos. Por ende, si suponemos que el programa es correcto y que todas las variables están definidas antes de usarlas, sin duda es correcto que el valor de x al inicio del bloque básico B_5 sólo puede ser 10. Y si, para empezar, el programa es incorrecto, entonces elegir 10 como el valor de x no puede ser peor que permitir que x asuma algún valor aleatorio. \square

9.4.7 Ejercicios para la sección 9.4

! **Ejercicio 9.4.1:** Suponga que deseamos detectar todas las posibilidades de que una variable esté sin inicializar a lo largo de cualquier camino, hasta un punto en el que se utilice. ¿Cómo modificaría el marco de trabajo de esta sección para detectar dichas situaciones?

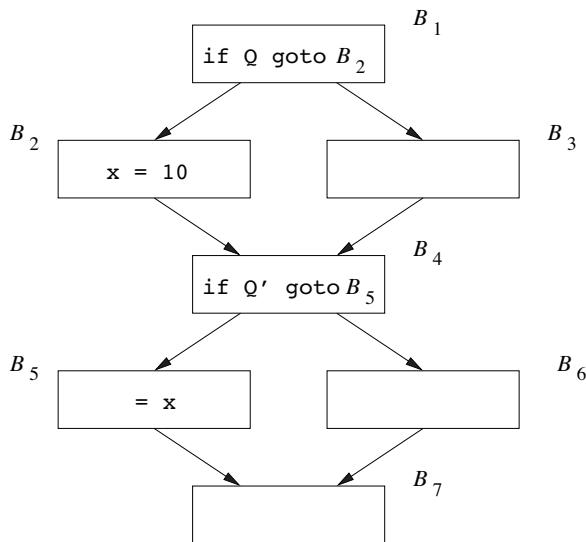


Figura 9.29: Reunión de UNDEF y una constante

!! **Ejercicio 9.4.2:** Un marco de trabajo de análisis del flujo de datos interesante y poderoso se obtiene al imaginar que el dominio V consiste en todas las posibles particiones de expresiones, de tal forma que dos expresiones se encuentren en la misma clase si, y sólo si es seguro que tendrán el mismo valor a lo largo de cualquier camino hacia el punto en cuestión. Para evitar tener que presentar una infinidad de expresiones, podemos representar a V mencionando sólo los pares mínimos de expresiones equivalentes. Por ejemplo, si ejecutamos las siguientes instrucciones:

$a = b$
 $c = a + d$

entonces el conjunto mínimo de equivalencias es $\{a \equiv b, c \equiv a + d\}$. A éstas les siguen otras equivalencias, como $c \equiv b + d$ y $a + e \equiv b + e$, pero no hay necesidad de presentarlas en forma explícita.

- ¿Cuál es el operador de reunión apropiado para este marco de trabajo?
- Proporcione una estructura de datos para representar valores de dominio y un algoritmo para implementar el operador de reunión.
- ¿Cuáles son las funciones apropiadas para asociar con instrucciones? Explique el efecto que una instrucción como $a = b+c$ debería tener sobre una partición de expresiones (es decir, sobre un valor en V).
- ¿Este marco de trabajo es monótono? ¿distributivo?

9.5 Eliminación de redundancia parcial

En esta sección, consideraremos con detalle cómo minimizar el número de evaluaciones de expresiones. Es decir, queremos considerar todas las secuencias posibles de ejecución en un grafo de flujo, y analizar el número de veces que se evalúa una expresión como $x + y$. Al desplazarnos alrededor de los lugares en los que $x + y$ se evalúa, manteniendo el resultado en una variable temporal siempre que sea necesario, a menudo podemos reducir el número de evaluaciones de esta expresión a lo largo de muchas de los caminos de ejecución, sin incrementar al mismo tiempo ese número a lo largo de cualquier otro camino. Observe que puede aumentar el número de lugares distintos en el grafo de flujo en donde $x + y$ se evalúa, pero eso casi no tiene importancia, siempre y cuando se reduzca el número de *evaluaciones* de la expresión $x + y$.

Al aplicar la transformación de código aquí desarrollada, se mejora el rendimiento del código resultante ya que, como veremos, nunca se aplica una operación a menos que sea absolutamente necesario. Cada compilador optimizador implementa algo parecido a la transformación aquí descrita, incluso si utiliza un algoritmo menos “agresivo” que el de esta sección. No obstante, existe otra motivación para hablar sobre el problema. Para encontrar el lugar o lugares adecuados en el grafo de flujo, en donde se debe evaluar cada expresión, se requieren cuatro tipos distintos de análisis del flujo de datos. Por ende, el estudio de la “eliminación de redundancia parcial”, como se le llama al proceso de minimizar el número de expresiones de evaluación, mejorará nuestra comprensión del papel que juega el análisis del flujo de datos en un compilador.

La redundancia en los programas existe en varias formas. Como vimos en la sección 9.1.4, puede existir en la forma de subexpresiones comunes, en donde varias evaluaciones de la expresión producen el mismo valor. También puede existir en la forma de una expresión invariante de ciclo, que se evalúa con el mismo valor en cada iteración del ciclo. La redundancia también puede ser parcial, si se encuentra a lo largo de algunas de los caminos, aunque no necesariamente a lo largo de *todas* ellas. Podemos ver a las subexpresiones comunes y las expresiones invariantes de ciclo como casos especiales de redundancia parcial; así, podemos idear un solo algoritmo de eliminación de redundancia parcial para eliminar todas las diversas formas de redundancia.

A continuación, primero hablaremos sobre las distintas formas de redundancia, para poder desarrollar nuestra intuición acerca del problema. Después describiremos el problema de eliminación de redundancia generalizado, y por último presentaremos el algoritmo. Este algoritmo es en especial interesante, ya que implica resolver varios problemas de flujo de datos, tanto en dirección hacia delante como hacia atrás.

9.5.1 Los orígenes de la redundancia

La figura 9.30 ilustra las tres formas de redundancia: subexpresiones comunes, expresiones invariantes de ciclo y expresiones con redundancia parcial. La figura muestra el código, antes y después de cada optimización.

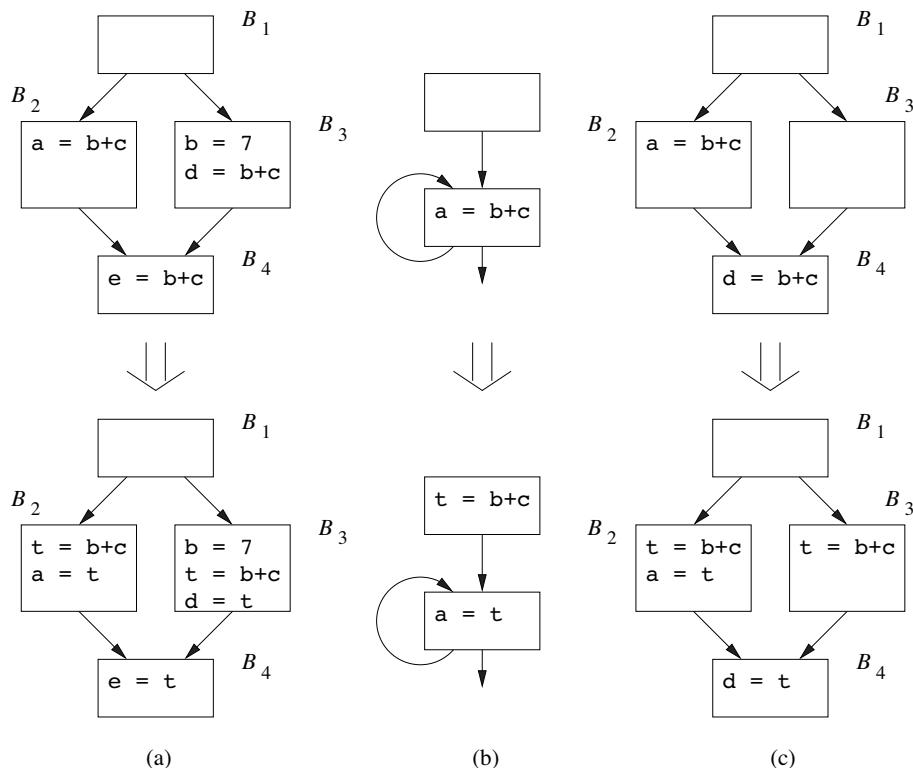


Figura 9.30: Ejemplos de (a) subexpresión común global, (b) movimiento de código invariante de ciclo, (c) eliminación de redundancia parcial

Subexpresiones comunes globales

En la figura 9.30(a), la expresión $b + c$ calculada en el bloque B_4 es redundante; ya se ha evaluado para cuando el flujo de control llega a B_4 , sin importar el camino que se tome para llegar ahí. Como podemos observar en este ejemplo, el valor de la expresión puede ser diferente en los distintos caminos. Podemos optimizar el código, almacenando el resultado de los cálculos de $b + c$ en los bloques B_2 y B_3 en la misma variable temporal, por decir t , y después asignando el valor de t a la variable e en el bloque B_4 , en vez de evaluar de nuevo la expresión. Si hubiera una asignación a b o c después del último cálculo de $b + c$ pero antes del bloque B_4 , la expresión en el bloque B_4 no sería redundante.

De manera formal, decimos que una expresión $b + c$ es (completamente) *redundante* en el punto p , si es una expresión disponible, en el sentido de la sección 9.2.6, en ese punto. Es decir, la expresión $b + c$ se ha calculado a lo largo de todos los caminos que llegan a p , y las variables b y c no se redefinieron después de evaluar la última expresión. Esta última condición es necesaria, ya que aun cuando la expresión $b + c$ se ejecuta textualmente antes de llegar al punto p , el valor de $b + c$ calculado en el punto p hubiera sido distinto, debido a que los operandos podrían haber cambiado.

Búsqueda de subexpresiones comunes “profundas”

El uso del análisis de las expresiones disponibles para identificar expresiones redundantes sólo funciona para las expresiones que son textualmente idénticas. Por ejemplo, una aplicación de la eliminación de subexpresiones comunes reconocerá que $t1$ en el siguiente fragmento de código:

```
t1 = b + c; a = t1 + d;
```

tiene el mismo valor que $t2$ en:

```
t2 = b + c; e = t2 + d;
```

siempre y cuando las variables b y c no se hayan redefinido entre una expresión y la otra. Sin embargo, no reconoce que a y e también son iguales. Es posible encontrar dichas subexpresiones comunes “profundas” al aplicar de nuevo la eliminación de subexpresiones comunes hasta que no se encuentren subexpresiones nuevas en una ronda. También es posible utilizar el marco de trabajo del ejercicio 9.4.2 para atrapar las subexpresiones comunes profundas.

Expresiones invariantes de ciclo

La figura 9.30(b) muestra un ejemplo de una expresión invariante de ciclo. La expresión $b + c$ es una invariante de ciclo, suponiendo que ninguna de las variables b o c se redefinen dentro del ciclo. Podemos optimizar el programa al sustituir todas las reejecuciones en un ciclo mediante un solo cálculo fuera del mismo. Asignamos el cálculo a una variable temporal, por decir t , y después sustituimos la expresión en el ciclo por t . Hay un punto más que debemos considerar al realizar optimizaciones de “movimiento de código” como éstas. No debemos ejecutar ninguna instrucción que no se hubiera ejecutado sin la optimización. Por ejemplo, si es posible salir del ciclo sin ejecutar la instrucción invariante de ciclo, entonces no debemos mover la instrucción fuera del ciclo. Hay dos razones.

1. Si la instrucción genera una excepción, entonces al ejecutarla se puede producir una excepción que no hubiera ocurrido en el programa original.
2. Cuando el ciclo sale antes de tiempo, el programa “optimizado” requiere más tiempo que el programa original.

Para asegurar que las expresiones invariantes de ciclo en los ciclos while puedan optimizarse, por lo general, los compiladores representan la siguiente instrucción:

```
while c {
    S;
}
```

de la misma forma que la siguiente instrucción:

```
if c {
    repeat
        S;
    until not c;
}
```

De esta forma, las expresiones invariantes de ciclo pueden colocarse justo antes de la construcción repeat-until.

A diferencia de la eliminación de subexpresiones comunes, en donde el cálculo de una expresión redundante simplemente se descarta, la eliminación de expresiones invariantes de ciclo requiere que una expresión del interior del ciclo se mueva fuera del mismo. Por ende, esta optimización se conoce como “movimiento de código invariante de ciclo”. Tal vez haya que repetir el movimiento de código invariante de ciclo, ya que una vez que se determina que una variable tiene un valor invariante de ciclo, las expresiones que utilizan dicha variable también pueden convertirse en invariantes de ciclo.

Expresiones parcialmente redundantes

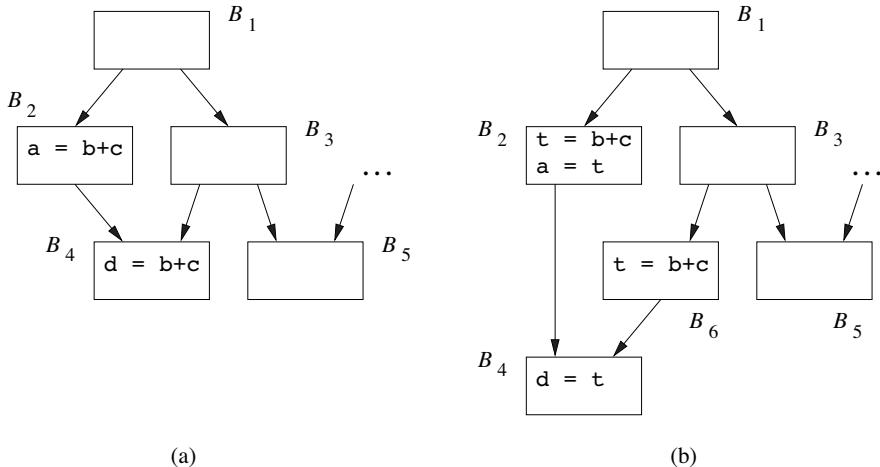
En la figura 9.30(c) se muestra un ejemplo de una expresión parcialmente redundante. La expresión $b + c$ en el bloque B_4 es redundante en el camino $B_1 \rightarrow B_2 \rightarrow B_4$, pero no en el camino $B_1 \rightarrow B_3 \rightarrow B_4$. Podemos eliminar la redundancia en el camino anterior, colocando un cálculo de $b + c$ en el bloque B_3 . Todos los resultados de $b + c$ se escriben en una variable temporal t , y el cálculo en el bloque B_4 se sustituye con t . Así, al igual que el movimiento de código invariante de ciclo, la eliminación de redundancia parcial requiere la colocación de nuevos cálculos de expresiones.

9.5.2 ¿Puede eliminarse toda la redundancia?

¿Es posible eliminar todos los cálculos redundantes a lo largo de todos los caminos? La respuesta es “no”, a menos que podamos modificar el grafo de flujo mediante la creación de nuevos bloques.

Ejemplo 9.28: En el ejemplo que se muestra en la figura 9.31(a), la expresión de $b + c$ se calcula de manera redundante en el bloque B_4 si el programa sigue el camino de ejecución $B_1 \rightarrow B_2 \rightarrow B_4$. Sin embargo, no podemos simplemente mover el cálculo de $b + c$ al bloque B_3 , ya que ello crearía un cálculo adicional de $b + c$ al tomar el camino $B_1 \rightarrow B_3 \rightarrow B_5$.

Lo que tendríamos que hacer es insertar el cálculo de $b + c$ sólo a lo largo de la flecha que va del bloque B_3 al bloque B_4 . Para ello, podemos colocar la instrucción en un nuevo bloque, por decir B_6 , y hacer que el flujo de control de B_3 pase a B_6 antes de llegar a B_4 . La transformación se muestra en la figura 9.31(b). \square

Figura 9.31: $B_3 \rightarrow B_4$ es una flecha crítica

Definimos una *flecha crítica* de un grafo de flujo como cualquier flecha que conduce de un nodo con más de un sucesor, hacia un nodo con más de un predecesor. Al introducir nuevos bloques a lo largo de las flechas críticas, siempre podemos encontrar un bloque para acomodar la colocación de la expresión deseada. Por ejemplo, la flecha que va de B_3 a B_4 en la figura 9.31(a) es crítico, debido a que B_3 tiene dos sucesores, y B_4 tiene dos predecesores.

Tal vez no sea suficiente agregar bloques para permitir la eliminación de todos los cálculos redundantes. Como se muestra en el ejemplo 9.29, puede ser necesario duplicar código para aislar el camino en la que se encuentra la redundancia.

Ejemplo 9.29: En el ejemplo que se muestra en la figura 9.32(a), la expresión de $b + c$ se calcula de manera redundante a lo largo de $B_1 \rightarrow B_2 \rightarrow B_4 \rightarrow B_6$. Tendríamos que eliminar el cálculo redundante de $b + c$ del bloque B_6 en este camino y calcular la expresión sólo a lo largo del camino $B_1 \rightarrow B_3 \rightarrow B_4 \rightarrow B_6$. Sin embargo, no hay un solo punto del programa o flecha en el programa de origen que corresponda en forma única al camino anterior. Para crear dicho punto en el programa, podemos duplicar el par de bloques B_4 y B_6 , en donde se llega a un par a través de B_2 y se llega a otro a través de B_3 , como se muestra en la figura 9.32(b). El resultado de $b + c$ se guarda en la variable t en el bloque B_2 , y se mueve a la variable d en B'_6 , la copia de B_6 a la que se llega desde B_2 . \square

Como el número de caminos es exponencial en el número de bifurcaciones condicionales en el programa, al eliminar todas las expresiones redundantes se puede incrementar de manera considerable el tamaño del código optimizado. Por lo tanto, restringiremos nuestra discusión sobre las técnicas de eliminación de redundancia a aquellas que puedan introducir bloques adicionales, pero que no dupliquen porciones del grafo de flujo de control.

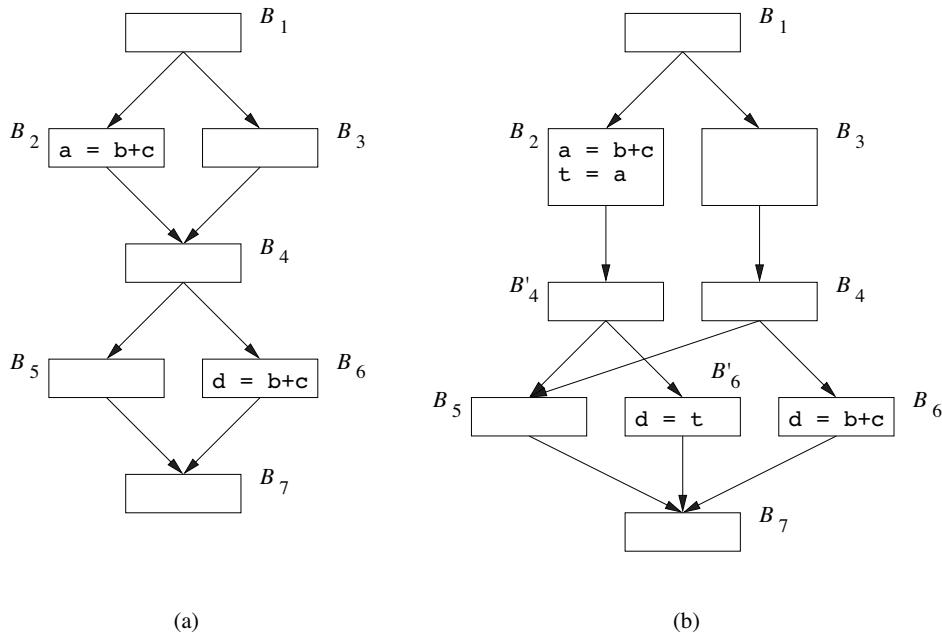


Figura 9.32: Duplicación del código para eliminar redundancias

9.5.3 El problema del movimiento de código diferido

Es conveniente que los programas optimizados con un algoritmo de eliminación de redundancia parcial tengan las siguientes propiedades:

1. Todos los cálculos redundantes de expresiones que puedan eliminarse sin la duplicación de código se eliminan.
2. El programa optimizado no realiza ningún cálculo que no se encuentre en la ejecución del programa original.
3. Las expresiones se calculan lo más tarde que sea posible.

La última propiedad es importante, ya que los valores de las expresiones que son redundantes, por lo general, se mantienen en registros, hasta que se utilizan. Al calcular un valor lo más tarde posible se minimiza su tiempo de vida: la duración entre el tiempo que se define el valor y la última vez que se utilizó, lo cual a su vez disminuye el uso que hace de un registro. Nos referimos a la optimización de eliminar la redundancia parcial, con el objetivo de retrasar los cálculos lo más que sea posible, como *movimiento de código diferido*.

Para elaborar nuestra intuición del problema, primero vamos a ver cómo razonar sobre la redundancia parcial de una sola expresión, a lo largo de un solo camino. Por conveniencia, asumiremos durante el resto de la explicación que toda instrucción es un bloque básico por sí sola.

Redundancia completa

Una expresión e en el bloque B es redundante si, a lo largo de todos los caminos que llegan a B , e se ha evaluado y los operandos de e no se han definido de nuevo posteriormente. Hagamos que S sea el conjunto de bloques, cada uno de los cuales contiene la expresión e , que hacen que e en B sea redundante. El conjunto de flechas que salen de los bloques en S debe formar necesariamente un *conjunto de corte (cutset)*, que si se elimina, desconecta al bloque B de la entrada del programa. Además, no se definen de nuevo operandos de e a lo largo de los caminos que conducen de los bloques en S hacia B .

Redundancia parcial

Si una expresión e en el bloque B es sólo parcialmente redundante, el algoritmo de movimiento de código diferido trata de hacer que e sea completamente redundante en B , colocando copias adicionales de las expresiones en el grafo de flujo. Si el intento tiene éxito, el grafo de flujo optimizado también tendrá un conjunto de bloques básicos S , cada uno de los cuales contendrá la expresión e , y cuyas flechas salientes formen un conjunto de corte entre la entrada y B . Al igual que el caso completamente redundante, no se definen de nuevo los operandos de e a lo largo de los caminos que conducen de los bloques en S hacia B .

9.5.4 Anticipación de las expresiones

Existe una restricción más que se impone en las expresiones insertadas, para asegurar que no se ejecuten operaciones adicionales. Las copias de una expresión deben colocarse sólo en los puntos del programa en donde se *anticipa* la expresión. Decimos que una expresión $b + c$ se *anticipa* en el punto p si todas los caminos que conducen desde el punto p calculan en un momento dado el valor de la expresión $b + c$ a partir de los valores de b y c que están disponibles en ese punto.

Ahora vamos a examinar lo que se requiere para eliminar la redundancia parcial a lo largo de un camino acíclico $B_1 \rightarrow B_2 \rightarrow \dots \rightarrow B_n$. Suponga que la expresión e se evalúa sólo en los bloques B_1 y B_n , y que los operandos de e no se definen de nuevo en los bloques a lo largo del camino. Hay flechas entrantes que se unen al camino y hay flechas salientes que salen de ésta. Podemos ver que e *no* se anticipa a la entrada del bloque B_i si, y sólo si existe una flecha saliente que salga del bloque B_j , $i \leq j < n$, que conduzca hacia un camino de ejecución que no utilice el valor de e . Por ende, la anticipación limita qué tan pronto se puede insertar una expresión.

Podemos crear un conjunto de corte (cutset) que incluya la arista $B_{i-1} \rightarrow B_i$ y que haga a e redundante en B_n , si e está disponible o se anticipa a la entrada de B_i . Si e se anticipa pero no está disponible a la entrada de B_i , debemos colocar una copia de la expresión e a lo largo de la flecha entrante.

Podemos elegir en dónde colocar las copias de la expresión, ya que, por lo general, hay varios conjuntos de corte en el grafo de flujo que cumplen con todos los requerimientos. En el conjunto de corte anterior, el cálculo se introduce a lo largo de las flechas entrantes que van hacia el camino de interés y, por lo tanto, la expresión se calcula lo más cerca posible del uso, sin introducir redundancia. Observe que estas operaciones introducidas pueden ser en sí

parcialmente redundantes con otras instancias de la misma expresión en el programa. Dicha redundancia parcial puede eliminarse al mover estos cálculos más hacia arriba.

En resumen, la anticipación de las expresiones limita qué tan pronto puede colocarse una expresión; no podemos colocar una expresión tan pronto que no se anticipa al lugar en donde la colocamos. Entre más anticipada se coloque una expresión, más será la redundancia que se pueda eliminar, y entre todas las soluciones que eliminan a las mismas redundancias, la que calcule las expresiones lo más tarde minimizará los tiempos de vida de los registros que contienen los valores de las expresiones involucradas.

9.5.5 El algoritmo de movimiento de código diferido

En consecuencia, esta explicación motiva la creación de un algoritmo de cuatro pasos. El primer paso utiliza la anticipación para determinar en dónde pueden colocarse las expresiones; el segundo paso busca el conjunto de corte *más anticipado*, entre aquellos que eliminan todas las operaciones redundantes posibles, sin duplicar el código y sin introducir cálculos indeseables. Este paso coloca los cálculos en puntos del programa en donde los valores de sus resultados se anticipan por primera vez. Después, el tercer paso empuja el conjunto de corte hacia abajo, hasta el punto en donde cualquier retraso adicional alteraría la semántica del programa, o introduciría redundancia. El cuarto paso (final) es una pasada simple para limpiar el código, eliminando asignaciones a variables temporales que se utilizan sólo una vez. Cada paso se realiza con una pasada del flujo de datos: la primera y la cuarta son problemas de flujo hacia atrás, la segunda y la tercera son problemas de flujo hacia delante.

Generalidades del algoritmo

1. Buscar todas las expresiones anticipadas en cada punto del programa, mediante una pasada de flujo de datos hacia atrás.
2. El segundo paso coloca el cálculo en donde se anticipan por primera vez los valores de las expresiones a lo largo de cierto camino. Después de haber colocado copias de una expresión en donde ésta se anticipa por primera vez, la expresión estaría *disponible* en el punto del programa p si se ha anticipado a lo largo de todos los caminos que lleguen a p . La disponibilidad puede resolverse mediante una pasada de flujo de datos hacia delante. Si deseamos colocar las expresiones en las posiciones más anticipadas posibles, podemos simplemente buscar esos puntos del programa en donde se anticipan las expresiones, pero no están disponibles.
3. Al ejecutar una expresión tan pronto como se anticipa se puede producir un valor mucho antes de utilizarlo. Una expresión es *diferible* en un punto del programa si la expresión se ha anticipado y todavía no se utiliza a lo largo de cualquier camino que llegue al punto del programa. Las expresiones diferibles se encuentran mediante una pasada de flujo de datos hacia delante. Colocamos expresiones en esos puntos del programa en donde ya no pueden posponerse.
4. Se utiliza una pasada final simple de flujo de datos hacia atrás para eliminar las asignaciones a las variables temporales que se utilizan sólo una vez en el programa.

Pasos de preprocesamiento

Ahora vamos a presentar el algoritmo completo de movimiento de código diferido. Para mantener este algoritmo simple, vamos a suponer que al principio cada instrucción se encuentra en su propio bloque básico, y sólo introducimos cálculos nuevos de expresiones en los inicios de los bloques. Para asegurar que esta simplificación no reduzca la efectividad de la técnica insertamos un nuevo bloque entre el origen y el destino de un camino si el destino tiene más de un predecesor. Es obvio que al hacer esto también nos hacemos cargo de todos los caminos críticos en el programa.

Abstraemos la semántica de cada bloque B con dos conjuntos: e_uso_B es el conjunto de expresiones calculadas en B y $e_eliminar_B$ es el conjunto de expresiones eliminadas; es decir, el conjunto de expresiones en las que cualquiera de sus operandos están definidos en B . Utilizaremos el ejemplo 9.30 a lo largo de la explicación de los cuatro análisis de flujo de datos, cuyas definiciones se sintetizan en la figura 9.34.

Ejemplo 9.30: En el grafo de flujo de la figura 9.33(a), la expresión $b + c$ aparece tres veces. Como el bloque B_9 es parte de un ciclo, la expresión puede calcularse muchas veces. El cálculo en el bloque B_0 no solo es invariante de ciclo; es también una expresión redundante, ya que su valor se ha utilizado de antemano en el bloque B_7 . Para este ejemplo, debemos calcular $b + c$ sólo dos veces, una en el bloque B_5 y otra a lo largo del camino después de B_2 y antes de B_7 . El algoritmo de movimiento de código diferido colocará los cálculos de la expresión al inicio de los bloques B_4 y B_5 . \square

Expresiones anticipadas

Recuerde que una expresión $b + c$ se anticipa en un punto p del programa si todos los caminos que salen desde el punto p en un momento dado calculan el valor de la expresión $b + c$, a partir de los valores de b y c que están disponibles en ese punto.

En la figura 9.33(a), todos los bloques que se anticipan a $b + c$ en la entrada se muestran como cuadros con un ligero sombreado. La expresión $b + c$ se anticipa en los bloques B_3 , B_4 , B_5 , B_6 , B_7 y B_9 . No se anticipa en la entrada al bloque B_2 , ya que el valor de c se recalcula dentro del bloque y, por lo tanto, el valor de $b + c$ que se calcularía al inicio de B_2 no se utiliza a lo largo de ningún camino. La expresión $b + c$ no se anticipa al entrar a B_1 , ya que es innecesaria a lo largo de la bifurcación de B_1 a B_2 (aunque se utilizaría a lo largo de la camino $B_1 \rightarrow B_5 \rightarrow B_6$). De manera similar, la expresión no se anticipa al inicio de B_8 , debido a la bifurcación de B_8 a B_{11} . La anticipación de una expresión puede oscilar a lo largo de un camino, como se ilustra mediante $B_7 \rightarrow B_8 \rightarrow B_9$.

Las ecuaciones del flujo de datos para el problema de las expresiones anticipadas se muestran en la figura 9.34(a). El análisis es una pasada invertida. Una expresión anticipada en la salida de un bloque B es una expresión anticipada en la entrada, sólo si no se encuentra en el conjunto $e_eliminar_B$. Además, un bloque B genera como nuevos usos el conjunto de expresiones e_uso_B . Al salir del programa, ninguna de las expresiones es anticipada. Como nos interesa encontrar expresiones que sean anticipadas a lo largo de todos los caminos siguientes, el operador de reunión

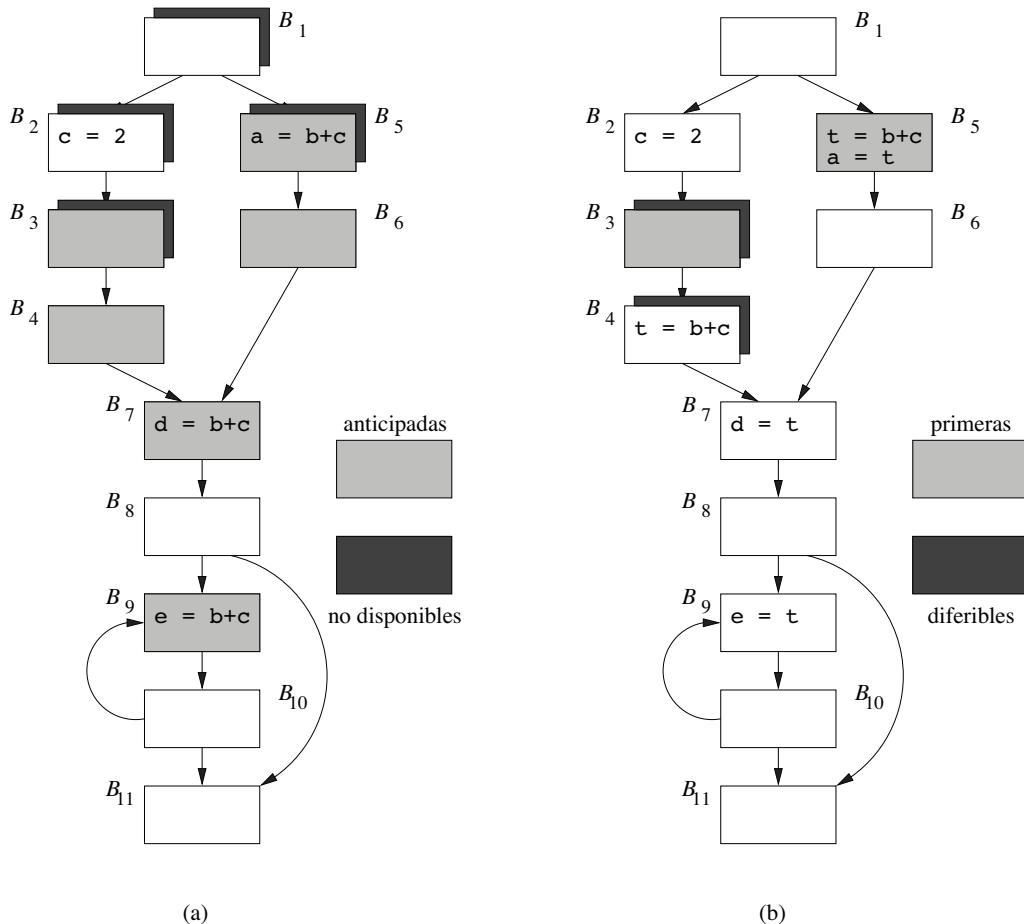


Figura 9.33: Grafo de flujo del ejemplo 9.30

es la intersección de conjuntos. En consecuencia, los puntos interiores deben inicializarse con el conjunto universal U , como vimos para el problema de las expresiones disponibles en la sección 9.2.6.

Expresiones disponibles

Al final de este segundo paso, las copias de una expresión se colocarán en los puntos del programa en donde se anticipó la expresión por primera vez. Si ése es el caso, una expresión estará *disponible* en el punto p del programa si se anticipa a lo largo de todos los caminos que llegan a p . Este problema es similar a las expresiones disponibles descritas en la sección 9.2.6. Aunque la función de transferencia que se utiliza aquí es un poco distinta. Una expresión está disponible al salir de un bloque si está:

	(a) Expresiones anticipadas	(b) Expresiones disponibles
Dominio	Conjuntos de expresiones	Conjuntos de expresiones
Dirección	Hacia atrás	Hacia delante
Función de transferencia	$f_B(x) = e_uso_B \cup (x - e_eliminar_B)$	$f_B(x) = (anticipadas[B].ent \cup x) - e_eliminar_B$
Límite	$ENT[SALIDA] = \emptyset$	$SAL[ENTRADA] = \emptyset$
Reunión (\wedge)	\cap	\cap
Ecuaciones	$ENT[B] = f_B(SAL[B])$ $SAL[B] = \wedge_{S, \ suc(B)} ENT[S]$	$SAL[B] = f_B(ENT[B])$ $ENT[B] = \wedge_{P, \ pred(B)} SAL[P]$
Inicialización	$ENT[B] = U$	$SAL[B] = U$
	(c) Expresiones diferibles	(d) Expresiones usadas
Dominio	Conjuntos de expresiones	Conjuntos de expresiones
Dirección	Hacia delante	Hacia atrás
Función de transferencia	$f_B(x) = (primeras[B] \cup x) - e_uso_B$	$f_B(x) = (e_uso_B \cup x) - ultimas[B]$
Límite	$SAL[ENTRADA] = \emptyset$	$ENT[SALIDA] = \emptyset$
Reunión (\wedge)	\cap	\cup
Ecuaciones	$SAL[B] = f_B(ENT[B])$ $ENT[B] = \wedge_{P, \ pred(B)} SAL[P]$	$ENT[B] = f_B(SAL[B])$ $SAL[B] = \wedge_{S, \ suc(B)} ENT[S]$
Inicialización	$SAL[B] = U$	$ENT[B] = \emptyset$

$$\begin{aligned} primeras[B] &= anticipadas[B].ent - disponibles[B].ent \\ ultimas[B] &= (primeras[B] \cup diferibles[B].ent) \cap \end{aligned}$$

$$(e_uso_B \cup \neg(\cap_{S, \ suc[B]} (primeras[S] \cup diferibles[S].ent)))$$

Figura 9.34: Cuatro pasadas de flujo de datos en la eliminación de redundancia parcial

Completar el cuadrado

Las expresiones anticipadas (también conocidas como “expresiones ocupadas” en cualquier otra parte) son un tipo de análisis de flujo de datos que no hemos visto antes. Aunque hemos visto los marcos de trabajo con flujo hacia atrás como el análisis de variables vivas (sección 9.2.5), y hemos visto marcos de trabajo en donde la reunión es la intersección como las expresiones disponibles (sección 9.2.6), éste es el primer ejemplo de un análisis útil que tiene ambas propiedades. Casi todos los análisis que utilizamos pueden colocarse en uno de cuatro grupos, dependiendo de si fluyen hacia delante o hacia atrás, y dependiendo de si utilizan la unión o la intersección para la reunión. Observe también que los análisis de unión siempre requieren preguntar si existe un camino a través de la cual algo sea verdadero, mientras que los análisis de intersección preguntan si algo es verdadero a lo largo de todos los caminos.

1. Ya sea
 - (a) Disponible.
 - (b) En el conjunto de expresiones anticipadas al momento de entrar (es decir, *podría* hacerse disponible si optamos por calcularla aquí),
- y
2. No se elimina en el bloque.

Las ecuaciones de flujo de datos para las expresiones disponibles se muestran en la figura 9.34 (b). Para evitar confundir el significado de ENT, nos referimos al resultado de un análisis anterior adjuntando “[B].ent” al nombre del análisis anterior.

Con la estrategia de colocación más anticipada, el conjunto de expresiones que se colocan en el bloque B , es decir, *primeras*[B], se define como el conjunto de expresiones anticipadas que no están aún disponibles. Es decir,

$$\text{primeras}[B] = \text{anticipadas}[B].\text{ent} - \text{disponibles}[B].\text{ent}.$$

Ejemplo 9.3.1: La expresión $b + c$ en el grafo de flujo de la figura 9.35 no se anticipa a la entrada del bloque B_3 , pero se anticipa a la entrada del bloque B_4 . Sin embargo, no es necesario calcular la expresión $b + c$ en el bloque B_4 , ya que la expresión se encuentra de antemano disponible, debido al bloque B_2 . \square

Ejemplo 9.3.2: En la figura 9.33(a) se muestran con cuadros sombreados en color oscuro los bloques para los cuales la expresión $b + c$ no está disponible; éstos son B_1 , B_2 , B_3 y B_5 . Las primeras posiciones colocadas se representan mediante los cuadros con sombreado claro y sombras oscuras, que por ende son los bloques B_3 y B_5 . Observe, por ejemplo, que $b + c$ se considera disponible al entrar a B_4 , ya que hay un camino $B_1 \rightarrow B_2 \rightarrow B_3 \rightarrow B_4$ a lo largo de la cual $b + c$ se anticipa por lo menos una vez (en este caso, en B_3) y desde el inicio de B_3 , no se recalcularon b o c . \square

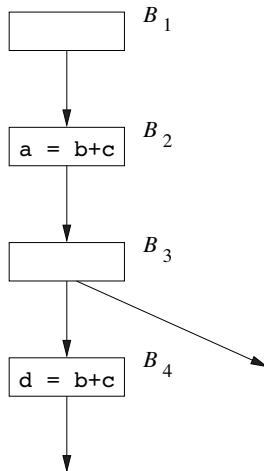


Figura 9.35: Grafo de flujo para el ejemplo 9.31, que ilustra el uso de la disponibilidad

Expresiones diferibles

El tercer paso difiere el cálculo de las expresiones lo más posible, preservando a la vez la semántica del programa original y minimizando la redundancia. El ejemplo 9.33 ilustra la importancia de este paso.

Ejemplo 9.33: En el grafo de flujo que se muestra en la figura 9.36, la expresión $b + c$ se calcula dos veces a lo largo del camino $B_1 \rightarrow B_5 \rightarrow B_6 \rightarrow B_7$. La expresión $b + c$ se anticipa incluso al inicio del bloque B_1 . Si calculáramos la expresión tan pronto como se anticipa, habríamos calculado la expresión $b + c$ en B_1 . El resultado tendría que guardarse desde el inicio, a través de la ejecución del ciclo que comprende los bloques B_2 y B_3 , hasta utilizarlo en el bloque B_7 . En vez de ello, podemos retrasar el cálculo de la expresión $b + c$ hasta el inicio de B_5 y hasta que el flujo de control esté a punto de cambiar de B_4 a B_7 . \square

De manera formal, una expresión $x + y$ es *diferible* a un punto p del programa si se encuentra una colocación anticipada de $x + y$ a lo largo de todos los caminos desde el nodo de entrada hasta p , y no hay un uso subsiguiente de $x + y$ después de esa última colocación.

Ejemplo 9.34: Vamos a considerar de nuevo la expresión $b + c$ en la figura 9.33. Los dos primeros puntos para $b + c$ son B_3 y B_5 ; observe que éstos son los dos bloques que tienen sombreado claro y oscuro en la figura 9.33(a), indicando que $b + c$ se anticipa y no está disponible para estos bloques, y sólo estos bloques. No podemos posponer $b + c$ de B_5 a B_6 , ya que $b + c$ se utiliza en B_5 . Sin embargo, podemos posponerla de B_3 a B_4 .

Pero no podemos posponer $b + c$ de B_4 a B_7 . La razón es que, aunque $b + c$ no se utiliza en B_4 , al colocar su cálculo mejor en B_7 se produciría un cálculo redundante de $b + c$ a lo largo

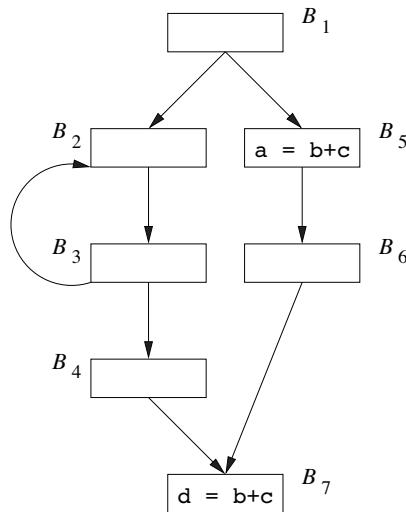


Figura 9.36: Grafo de flujo para el ejemplo 9.33, para ilustrar la necesidad de diferir una expresión

del camino $B_5 \rightarrow B_6 \rightarrow B_7$. Como veremos más adelante, B_4 es uno de los últimos lugares en los que podemos calcular $b + c$. \square

Las ecuaciones de flujo de datos para el problema de las expresiones diferibles se muestran en la figura 9.34(c). El análisis es una pasada hacia delante. No podemos “diferir” una expresión hasta la entrada del programa, por lo que $\text{SAL}[\text{ENTRADA}] = \emptyset$. Una expresión es diferible hasta la salida del bloque B si no se utiliza en el bloque, y si es diferible hasta la entrada de B o si se encuentra en $\text{primeras}[B]$. Una expresión no es diferible hasta la entrada de un bloque a menos que todos sus predecesores incluyan a la expresión en sus conjuntos *diferibles* en sus salidas. Por ende, el operador de reunión es la intersección de conjuntos, y los puntos interiores deben inicializarse con el elemento superior del semi-lattice: el conjunto universal.

Hablando en general, una expresión se coloca en la *frontera*, en donde una expresión cambia de ser diferible a no serlo. Dicho en forma más específica, una expresión e puede colocarse al inicio de un bloque B sólo si la expresión se encuentra en el conjunto *primeras* o *diferibles* de B al momento de entrar. Además, B está en la frontera diferible/no diferible de e si una de las siguientes afirmaciones es válida:

1. e no se encuentra en $\text{diferibles}[B].\text{sal}$. En otras palabras, e está en $e.\text{uso}_B$.
2. e no puede diferirse hacia uno de sus sucesores. En otras palabras, existe un sucesor de B tal que e no se encuentra en el conjunto *primeras* o *diferibles* al momento de entrar a ese sucesor.

La expresión e puede colocarse al frente del bloque B en cualquiera de los dos casos anteriores, debido a los nuevos bloques introducidos por el paso de preprocessamiento en el algoritmo.

Ejemplo 9.35: La figura 9.33(b) muestra el resultado del análisis. Los cuadros con sombreado claro representan a los bloques cuyo conjunto *primeras* incluye a $b + c$. Las sombras oscuras indican aquellos bloques que incluyen $b + c$ en su conjunto *diferibles*. Las últimas colocaciones de las expresiones son, por lo tanto, las entradas de los bloques B_4 y B_5 , ya que:

1. $b + c$ está en el conjunto *diferibles* de B_4 , pero no en B_7 .
2. El conjunto *primeras* de B_5 , incluye a $b + c$ y utiliza a $b + c$.

La expresión se almacena en la variable temporal t en los bloques B_4 y B_5 , y t se utiliza en vez de $b + c$ en cualquier otra parte, como se muestra en la figura. \square

Expresiones usadas

Por último, se utiliza una pasada hacia atrás para determinar si las variables temporales introducidas se utilizan más allá del bloque en el que se encuentran. Decimos que una expresión se *utiliza* en el punto p si existe un camino que conduzca desde p , que utilice la expresión antes de evaluar de nuevo el valor. Este análisis es en esencia el análisis del estado de vida (para expresiones, en vez de variables).

Las ecuaciones de flujo de datos para el problema de las expresiones usadas se muestran en la figura 9.34(d). El análisis es una pasada hacia atrás. Una expresión usada a la salida de un bloque B es una expresión usada al entrar sólo si no se encuentra en el conjunto *últimas*. Un bloque genera, como nuevos usos, el conjunto de expresiones en e_uso_B . A la salida del programa, ninguna de las expresiones es usada. Como nos interesa encontrar expresiones que se utilicen en cualquier camino siguiente, el operador de reunión es la unión de conjuntos. Así, los puntos interiores deben inicializarse con el elemento superior del semi-lattice: el conjunto vacío.

Reunión de todos los conceptos

Todos los pasos del algoritmo se sintetizan en el Algoritmo 9.36.

Algoritmo 9.36: Movimiento de código diferido.

ENTRADA: Un grafo de flujo para el cual se han calculado e_uso_B y $e_eliminar_B$ para cada bloque B .

SALIDA: Un grafo de flujo modificado, que cumple con las cuatro condiciones de movimiento de código diferido de la sección 9.5.3.

MÉTODO:

1. Insertar un bloque vacío a lo largo de todas las flechas que entran a un bloque con más de un predecesor.
2. Encontrar $anticipadas[B].ent$ para todos los bloques B , según su definición en la figura 9.34(a).
3. Encontrar $disponibles[B].ent$ para todos los bloques B , según su definición en la figura 9.34(b).

4. Calcular las primeras colocaciones para todos los bloques B :

$$\text{primeras}[B] = \text{anticipadas}[B].\text{ent} - \text{disponibles}[B].\text{ent}$$

5. Encontrar $\text{diferibles}[B].\text{ent}$ para todos los bloques B , según su definición en la figura 9.34(c).
6. Calcular las últimas colocaciones para todos los bloques B :

$$\begin{aligned} \text{últimas}[B] = & (\text{primeras}[B] \cup \text{diferibles}[B].\text{ent}) \cap \\ & \left(e_{-uso_B} \cup \neg \left(\bigcap_{S \text{ en } \text{suc}[B]} (\text{anticipadas}[S] \cup \text{diferibles}[S].\text{ent}) \right) \right) \end{aligned}$$

Observe que \neg denota la complementación con respecto al conjunto de todas las expresiones calculadas por el programa.

7. Encontrar $\text{usadas}[B].\text{sal}$ para todos los bloques B , según su definición en la figura 9.34(d).
8. Para cada expresión (por decir $x + y$) calculada por el programa, hacer lo siguiente:
- Crear un nuevo valor temporal, por decir t para $x + y$.
 - Para todos los bloques B tales que $x + y$ se encuentre en $\text{últimas}[B] \cap \text{usadas}[B].\text{sal}$, agregar $t = x+y$ al inicio de B .
 - Para todos los bloques B tales que $x + y$ se encuentre en

$$e_{-uso_B} \cap (\neg \text{últimas}[B] \cup \text{usadas}.sal[B])$$

sustituir cada $x + y$ original por t .

□

Resumen

La eliminación de redundancia parcial encuentra muchas formas distintas de operaciones redundantes en un algoritmo unificado. Este algoritmo ilustra cómo pueden usarse múltiples problemas de flujo de datos para encontrar la colocación óptima de las expresiones.

- El análisis de las expresiones anticipadas proporciona las restricciones de colocación; éste es un análisis de flujo de datos *hacia atrás*, con un operador de reunión de intersección de conjuntos, ya que determina si las expresiones se utilizan en forma *siguiente* a cada punto del programa, en *todos* los caminos.
- La primera colocación de una expresión se da mediante los puntos del programa en donde se anticipa la expresión, pero no está disponible. Las expresiones disponibles se encuentran mediante un análisis de flujo de datos *hacia delante*, con un operador de reunión de intersección de conjuntos que calcula si una expresión ha sido anticipada *antes* de cada punto del programa, a lo largo de *todos* los caminos.

3. La última colocación de una expresión se da mediante los puntos del programa en donde ya no puede diferirse una expresión. Las expresiones pueden diferirse en un punto del programa si para *todos* los caminos que *llegan* al punto del programa, no se ha encontrado un uso de la expresión. Las expresiones diferibles se encuentran mediante un análisis del flujo de datos *hacia delante*, con un operador de reunión de intersección de conjuntos.
4. Las asignaciones temporales se eliminan, a menos que las utilice *algún* camino *posteriormente*. Para encontrar las expresiones usadas utilizamos un análisis del flujo de datos *hacia atrás*, esta vez con un operador de reunión de unión de conjuntos.

9.5.6 Ejercicios para la sección 9.5

Ejercicio 9.5.1: Para el grafo de flujo de la figura 9.37:

- a) Calcule *anticipadas* para el inicio y el final de cada bloque.
- b) Calcule *disponibles* para el inicio y el final de cada bloque.
- c) Calcule *primeras* para cada bloque.
- d) Calcule *diferibles* para el inicio y el final de cada bloque.
- e) Calcule *usadas* para el inicio y el final de cada bloque.
- f) Calcule *últimas* para cada bloque.
- g) Introduzca la variable temporal t ; muestre en dónde se calcula y en dónde se utiliza.

Ejercicio 9.5.2: Repita el ejercicio 9.5.1 para el grafo de flujo de la figura 9.10 (vea los ejercicios de la sección 9.1). Puede limitar su análisis a las expresiones $a + b$, $c - a$ y $b * d$.

!! Ejercicio 9.5.3: Los conceptos que vimos en esta sección también pueden aplicarse para eliminar el código parcialmente muerto. Una definición de una variable está *parcialmente muerta* si la variable está viva en ciertos caminos y no en otras. Podemos optimizar la ejecución del programa con sólo realizar la definición a lo largo de los caminos en las que la variable está viva. A diferencia de la eliminación de redundancia parcial, en donde las expresiones se mueven antes de la original, las nuevas definiciones se colocan después de la original. Desarrolle un algoritmo para mover el código parcialmente muerto, de manera que las expresiones se evalúen sólo en donde se utilizarán en un momento dado.

9.6 Ciclos en los grafos de flujo

Hasta ahora en nuestra discusión, no hemos manejado los ciclos de manera distinta; los hemos tratado justo igual que cualquier otro tipo de flujo de control. Sin embargo, los ciclos son importantes ya que los programas invierten la mayor parte de su tiempo ejecutándolos,

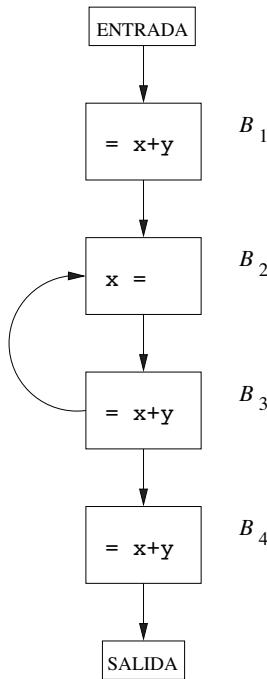


Figura 9.37: Grafo de flujo para el ejercicio 9.5.1

y las optimizaciones que mejoran su rendimiento pueden tener un impacto considerable. Por ende, es esencial que identifiquemos los ciclos y los tratemos de manera especial.

Los ciclos también afectan el tiempo de ejecución de los análisis de los programas. Si un programa no contiene ciclos, podemos obtener las respuestas a los problemas de flujo de datos realizando sólo una pasada a través del programa. Por ejemplo, un problema de flujo de datos hacia delante puede resolverse visitando todos los nodos una vez, en orden topológico.

En esta sección presentaremos los siguientes conceptos: dominadores, orden “primero en profundidad”, aristas posteriores, profundidad de los grafos y capacidad de reducción. Cada uno de estos conceptos se requiere para nuestras explicaciones posteriores acerca de cómo encontrar ciclos y la velocidad de convergencia del análisis de flujos de datos iterativos.

9.6.1 Dominadores

Decimos que el nodo d de un grafo de flujo *domina* al nodo n , lo cual se escribe como $d \text{ dom } n$, si cada camino desde el nodo de entrada del grafo de flujo hasta n pasa a través de d . Observe que bajo esta definición, cada nodo se domina a sí mismo.

Ejemplo 9.37: Considere el grafo de flujo de la figura 9.38, con el nodo de entrada 1. El nodo de entrada domina a todos los nodos (esta instrucción se aplica a todos los grafos de flujo). El nodo 2 se domina sólo a sí mismo, ya que el control puede llegar a cualquier otro nodo, a lo largo de un camino que empieza con $1 \rightarrow 3$. El nodo 3 domina a todos los nodos, excepto

al 1 y al 2. El nodo 4 domina a todos excepto 1, 2 y 3, ya que todos los caminos que parten del nodo 1 deben empezar con $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ o $1 \rightarrow 3 \rightarrow 4$. Los nodos 5 y 6 sólo se dominan a sí mismos, ya que el flujo de control puede saltar a uno, pasando a través del otro. Por último, 7 domina a 7, 8, 9 y 10; 8 domina a 8, 9 y 10; y 9 y 10 sólo se dominan a sí mismos. \square

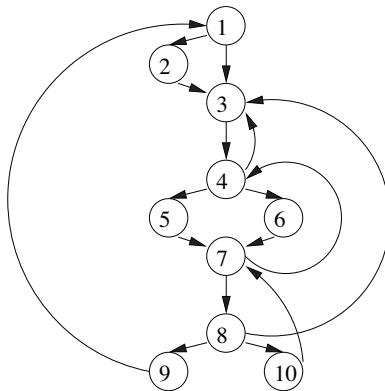


Figura 9.38: Un grafo de flujo

Una manera útil de presentar la información de los dominadores es en un árbol, conocido como el *árbol de dominadores*, en el cual el nodo de entrada es la raíz, y cada nodo d domina sólo a sus descendientes en el árbol. Por ejemplo, la figura 9.39 muestra el árbol de dominadores para el grafo de flujo de la figura 9.38.

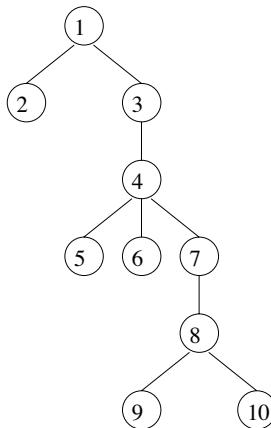


Figura 9.39: Árbol dominador para el grafo de flujo de la figura 9.38

La existencia de árboles de dominadores proviene de una propiedad de dominadores: cada nodo n tiene un *dominador inmediato* m único, que es el último dominador de n en cualquier

camino desde el nodo de entrada hasta n . En términos de la relación dom , el dominador inmediato m tiene la propiedad de que si $d \neq n$ y $d \ dom \ n$, entonces $d \ dom \ m$.

Vamos a proporcionar un simple algoritmo para calcular los dominadores de cada nodo n en un grafo de flujo, con base en el principio de que si p_1, p_2, \dots, p_k son todos los predecesores de n , y $d \neq n$, entonces $d \ dom \ n$ si, y sólo si $d \ dom \ p_i$ para cada i . El problema puede formularse como un análisis de flujo de datos hacia delante. Los valores del flujo de datos son conjuntos de bloques básicos. El conjunto de dominadores de un nodo, aparte de sí mismo, es la intersección de los dominadores de todos sus predecesores; por ende, el operador de reunión es la intersección de conjuntos. La función de transferencia para el bloque B simplemente agrega el mismo B al conjunto de nodos de entrada. La condición delimitadora es que el nodo ENTRADA se domina a sí mismo. Por último, la inicialización de los nodos interiores es el conjunto universal, es decir, el conjunto de todos los nodos.

Algoritmo 9.38: Búsqueda de dominadores.

ENTRADA: Un grafo de flujo G con el conjunto de nodos N , el conjunto de aristas F y el nodo de entrada ENTRADA.

SALIDA: $D(n)$, el conjunto de nodos que dominan al nodo n , para todos los nodos n en N .

MÉTODO: Buscar la solución al problema de flujo de datos, cuyos parámetros se muestran en la figura 9.40. Los bloques básicos son los nodos. $D(n) = SAL[n]$ para todas las n en N . \square

El proceso de buscar dominadores usando este algoritmo de flujo de datos es eficiente. Los nodos en el grafo necesitan visitarse sólo unas cuantas veces, como veremos en la sección 9.6.7.

	Dominadores
Dominio	El conjunto potencia de N
Dirección	Hacia delante
Función de transferencia	$f_B(x) = x \cup \{B\}$
Límite	$SAL[\text{ENTRADA}] = \{\text{ENTRADA}\}$
Reunión (\wedge)	\cap
Ecuaciones	$SAL[B] = f_B(ENT[B])$ $ENT[B] = \bigwedge_{P, \ pred(B)} SAL[P]$
Inicialización	$SAL[B] = N$

Figura 9.40: Un algoritmo de flujo de datos para calcular dominadores

Ejemplo 9.39: Vamos a regresar al grafo de flujo de la figura 9.38, y suponga que el ciclo for de las líneas (4) a (6) en la figura 9.23 visita los nodos en orden numérico. Hagamos que $D(n)$ sea el conjunto de nodos en $SAL[n]$. Como 1 es el nodo de entrada, a $D(1)$ se le asignó $\{1\}$

Propiedades de la relación dom

Una observación clave acerca de los dominadores es que si tomamos cualquier camino acíclico desde la entrada hasta el nodo n , entonces todos los dominadores de n aparecen a lo largo de este camino, y además deben aparecer *en el mismo orden* a lo largo de cualquier camino de este tipo. Para ver por qué, suponga que sólo hubiera un camino acíclico P_1 hacia n , a lo largo de la cual aparecieran los dominadores a y b en ese orden, y otro camino P_2 hacia n , a lo largo de la cual b estuviera antes que a . Entonces, podríamos seguir P_1 hacia a y P_2 hacia n , con lo cual evitaríamos por completo a b . Por ende, b en realidad no dominaría a a .

Este razonamiento nos permite demostrar que dom es transitivo: si $a \ dom \ b$ y $b \ dom \ c$, entonces $a \ dom \ c$. Además, dom es antisimétrico: nunca es posible que tanto $a \ dom \ b$ como $b \ dom \ a$ sean válidas, si $a \neq b$. Lo que es más, si a y b son dos dominadores de n , entonces debe ser válida $a \ dom \ b$ o $b \ dom \ a$. Por último, resulta que cada nodo n (excepto el de entrada) debe tener un dominador inmediato único; el dominador que aparezca más cerca de n a lo largo de cualquier camino acíclico desde la entrada hasta n .

en la línea (1). El nodo 2 sólo tiene 1 como predecesor, por lo que $D(2) = \{2\} \cup D(1)$. Por ende, $D(2)$ se establece en $\{1, 2\}$. Después se considera el nodo 3, con los predecesores 1, 2, 4 y 8. Como todos los nodos interiores se inicializan con el conjunto universal N ,

$$D(3) = \{3\} \cup (\{1\} \cap \{1, 2\} \cap \{1, 2, \dots, 10\} \cap \{1, 2, \dots, 10\}) = \{1, 3\}$$

El resto de los cálculos se muestran en la figura 9.41. Como estos valores no cambian en la segunda iteración a través del ciclo externo de las líneas (3) a (6) de la figura 9.23(a), son las respuestas finales al problema de los dominadores. \square

$$D(4) = \{4\} \cup (D(3) \cap D(7)) = \{4\} \cup (\{1, 3\} \cap \{1, 2, \dots, 10\}) = \{1, 3, 4\}$$

$$D(5) = \{5\} \cup D(4) = \{5\} \cup \{1, 3, 4\} = \{1, 3, 4, 5\}$$

$$D(6) = \{6\} \cup D(4) = \{6\} \cup \{1, 3, 4\} = \{1, 3, 4, 6\}$$

$$D(7) = \{7\} \cup (D(5) \cap D(6) \cap D(10))$$

$$= \{7\} \cup (\{1, 3, 4, 5\} \cap \{1, 3, 4, 6\} \cap \{1, 2, \dots, 10\}) = \{1, 3, 4, 7\}$$

$$D(8) = \{8\} \cup D(7) = \{8\} \cup \{1, 3, 4, 7\} = \{1, 3, 4, 7, 8\}$$

$$D(9) = \{9\} \cup D(8) = \{9\} \cup \{1, 3, 4, 7, 8\} = \{1, 3, 4, 7, 8, 9\}$$

$$D(10) = \{10\} \cup D(8) = \{10\} \cup \{1, 3, 4, 7, 8\} = \{1, 3, 4, 7, 8, 10\}$$

Figura 9.41: Término del cálculo de dominadores para el ejemplo 9.39

9.6.2 Ordenamiento “primero en profundidad”

Como se presentó en la sección 2.3.4, una *búsqueda “primero en profundidad”* de un grafo vista a todos los nodos en el grafo una vez, empezando en el nodo de entrada y visitando los nodos que estén más alejados del nodo de entrada, lo más rápido que sea posible. El camino de la búsqueda en el método “primero en profundidad” forma un *árbol de expansión con búsqueda en profundidad* (DFST). En la sección 2.3.4 vimos que un recorrido preorden visita a un nodo antes de visitar cualquiera de sus hijos, que después visita de manera recursiva, en orden de izquierda a derecha. Además, un recorrido postorden visita a los hijos de un nodo, de manera recursiva y en orden de izquierda a derecha, antes de visitar al nodo en sí.

Hay una variante más de ordenamiento que es importante para el análisis de grafos de flujo: un *ordenamiento “primero en profundidad”* es el inverso de un recorrido postorden. Es decir, en un ordenamiento “primero en profundidad”, visitamos un nodo, después recorremos su hijo de más a la derecha, el hijo a su izquierda, y así en lo sucesivo. No obstante, antes de construir el árbol para el grafo de flujo, tenemos que elegir qué sucesor de un nodo se convierte en el hijo de más a la derecha en el árbol, qué nodo se convierte en el siguiente hijo, y así sucesivamente. Antes de proporcionar el algoritmo para el ordenamiento “primero en profundidad”, vamos a considerar un ejemplo.

Ejemplo 9.40: Una posible presentación tipo “primero en profundidad” del grafo de flujo en la figura 9.38 se ilustra en la figura 9.42. Las aristas sólidas forman el árbol; las aristas punteadas son las otras aristas del grafo de flujo. Un recorrido “primero en profundidad” del árbol se da mediante: $1 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10$, después de regreso a 8, y luego a 9. Regresamos a 8 una vez más, nos retractamos a 7, 6 y 4, y después avanzamos a 5. Nos retractamos de 5 hacia 4, después hacia 3 y luego a 1. De 1 vamos a 2, después nos retractamos de 2, de vuelta a 1, y hemos recorrido el árbol completo.

Así, la secuencia preorden para el recorrido es:

$$1, 3, 4, 6, 7, 8, 10, 9, 5, 2.$$

La secuencia postorden para el recorrido del árbol en la figura 9.42 es:

$$10, 9, 8, 7, 6, 5, 4, 3, 2, 1.$$

El ordenamiento tipo primero en profundidad, que es el inverso de la secuencia postorden, es:

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10.$$

□

Ahora vamos a proporcionar un algoritmo que encuentra un árbol de expansión con búsqueda en profundidad y un ordenamiento “primero en profundidad” de un grafo. Este algoritmo es el que encuentra el DFST de la figura 9.42, a partir de la figura 9.38.

Algoritmo 9.41: Árbol de expansión con búsqueda en profundidad y ordenamiento “primero en profundidad”.

ENTRADA: Un grafo de flujo G .

SALIDA: Un DFST T de G y un ordenamiento de los nodos de G .

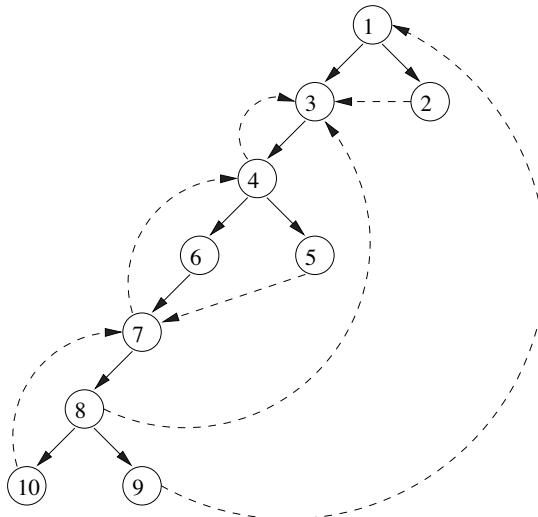


Figura 9.42: Una presentación “primero en profundidad” del grafo de flujo de la figura 9.38

MÉTODO: Utilizamos el procedimiento recursivo $buscar(n)$ de la figura 9.43. El algoritmo inicializa todos los nodos de G a “no visitado”, después llama a $buscar(n_0)$, en donde n_0 es la entrada. Cuando llama a $buscar(n)$, primero marca a n como “visitado”, para evitar agregar n al árbol dos veces. Utiliza c para contar en orden descendente, desde el número de nodos de G hasta 1, asignando los números primero en profundidad $dfn[n]$ a los nodos n , a medida que avanzamos. El conjunto de aristas T forma el árbol de expansión con búsqueda en profundidad para G . \square

Ejemplo 9.42: Para el grafo de flujo de la figura 9.42, el Algoritmo 9.41 establece c en 10 y empieza la búsqueda, llamando a $buscar(1)$. El resto de la secuencia de ejecución se muestra en la figura 9.44. \square

9.6.3 Aristas en un árbol de expansión con búsqueda en profundidad

Al construir un DFST para un grafo de flujo, las aristas del grafo de flujo se dividen en tres categorías.

1. Hay aristas, conocidos como *aristas de avance*, que van de un nodo m hasta un descendiente apropiado de m en el árbol. Todas las aristas en el mismo DFST son aristas de avance. No hay otros aristas de avance en la figura 9.42 pero, por ejemplo, si $4 \rightarrow 8$ fuera una arista, estaría dentro de esta categoría.
2. Hay aristas que pasan de un nodo m a un ancestro de m en el árbol (posiblemente al mismo m). A estas aristas las llamaremos *aristas de retirada*. Por ejemplo, $4 \rightarrow 3$, $7 \rightarrow 4$, $10 \rightarrow 7$ y $9 \rightarrow 1$ son las aristas de retirada en la figura 9.42.

```

void buscar( $n$ ) {
    marcar  $n$  como “visitado”;
    for (cada sucesor  $s$  de  $n$ )
        if ( $s$  es “no visitado”) {
            agregar arista  $n \rightarrow s$  a  $T$ ;
            buscar( $s$ );
        }
     $dfn[n] = c$ ;
     $c = c - 1$ ;
}

main() {
     $T = \emptyset$ ; /* conjunto de aristas */
    for (cada nodo  $n$  de  $G$ )
        marcar  $n$  como “no visitado”;
     $c =$  número de nodos de  $G$ ;
    buscar( $n_0$ );
}

```

Figura 9.43: Algoritmo de búsqueda “primero en profundidad”

3. Hay aristas $m \rightarrow n$ tales que ni m ni n son ancestros uno del otro en el DFST. Las aristas $2 \rightarrow 3$ y $5 \rightarrow 7$ son los únicos ejemplos de este tipo en la figura 9.42. A estas aristas las llamamos *aristas de cruce*. Una propiedad importante de las aristas de cruce es que, si dibujamos el DFST de manera que los hijos de un nodo se dibujen de izquierda a derecha en el orden en el que se agregaron al árbol, entonces todos las aristas de cruce viajan de derecha a izquierda.

Hay que tener en cuenta que $m \rightarrow n$ es una arista de retirada si, y sólo si $dfn[m] \geq dfn[n]$. Para ver por qué, observe que si m es descendiente de n en el DFST, entonces $buscar(m)$ termina antes de $buscar(n)$, por lo que $dfn[m] \geq dfn[n]$. Por el contrario, si $dfn[m] \geq dfn[n]$, entonces $buscar(m)$ termina antes que $buscar(n)$, o $m = n$. Pero $buscar(n)$ debe haber empezado antes de $buscar(m)$ si hay una arista $m \rightarrow n$, o de lo contrario el hecho de que n es un sucesor de m habría convertido a n en descendiente de m en el DFST. En consecuencia, el tiempo que $buscar(m)$ está activa es un subintervalo del tiempo que $buscar(n)$ está activa, de lo cual resulta que n es un ancestro de m en el DFST.

9.6.4 Aristas posteriores y capacidad de reducción

Una *arista posterior* es una arista $a \rightarrow b$ cuya cabeza b domina a su cola a . Para cualquier grafo de flujo, toda arista posterior es de retirada, pero no toda arista de retirada es una arista posterior. Se dice que un grafo de flujo es *reducible* si todas sus aristas de retirada en cualquier árbol de expansión con búsqueda en profundidad son también aristas posteriores. En otras palabras, si un grafo es reducible, entonces todos los DFSTs tienen el mismo conjunto de aristas

- Llama a *buscar*(1) El nodo 1 tiene dos sucesores. Suponga que $s = 3$ se considera primero; agrega la arista $1 \rightarrow 3$ a T .
Agrega la arista $3 \rightarrow 4$ a T .
- Llama a *buscar*(3) El nodo 4 tiene dos sucesores, 4 y 6. Suponga que $s = 6$ se considera primero; agrega la arista $4 \rightarrow 6$ a T .
Agrega $6 \rightarrow 7$ a T .
- Llama a *buscar*(4) El nodo 7 tiene dos sucesores, 4 y 8. Pero 4 ya está marcado como “visitado” por *buscar*(4), por lo que no hace nada cuando $s = 4$. Para $s = 8$, agrega la arista $7 \rightarrow 8$ a T .
- Llama a *buscar*(6) El nodo 8 tiene dos sucesores, 9 y 10. Suponga que $s = 10$ se considera primero; agrega la arista $8 \rightarrow 10$.
- Llama a *buscar*(7) 10 tiene un sucesor, 7, pero 7 ya está marcado como “visitado”. Por ende, *buscar*(10) se completa estableciendo $dfn[10] = 10$ y $c = 9$.
Establece $s = 9$ y agrega la arista $8 \rightarrow 9$ a T .
- Llama a *buscar*(8) El nodo 1, el único sucesor de 9, ya está “visitado”, por lo que establece $dfn[9] = 9$ y $c = 8$.
- Regresa a *buscar*(8) El nodo 3, el último sucesor de 8, está “visitado”, por lo que no hace nada para $s = 3$. En este punto, todos los sucesores de 8 se han considerado, por lo que establece $dfn[8] = 8$ y $c = 7$.
- Regresa a *buscar*(7) Todos los sucesores de 7 se han considerado, por lo que establece $dfn[7] = 7$ y $c = 6$.
- Regresa a *buscar*(6) De manera similar, se han considerado todos los sucesores de 6, por lo que establece $dfn[6] = 6$ y $c = 5$.
- Regresa a *buscar*(4) El sucesor 3 de 4 ha sido “visitado”, pero 5 no, por lo que agrega $4 \rightarrow 5$ al árbol.
- Llama a *buscar*(5) El sucesor 7 de 5 ha sido “visitado”, por lo cual establece $dfn[5] = 5$ y $c = 4$.
- Regresa a *buscar*(4) Se han considerado todos los sucesores de 4, establece $dfn[4] = 4$ y $c = 3$.
- Regresa a *buscar*(3) Establece $dfn[3] = 3$ y $c = 2$.
- Regresa a *buscar*(1) 2 no se ha visitado todavía, por lo que agrega $1 \rightarrow 2$ a T .
- Llama a *buscar*(2) Establece $dfn[2] = 2$, $c = 1$
- Regresa a *buscar*(1) Establece $dfn[1] = 1$ y $c = 0$.

Figura 9.44: Ejecución del Algoritmo 9.41 con el grafo de flujo de la figura 9.43

¿Por qué están las aristas posteriores retirando aristas?

Suponga que $a \rightarrow b$ es una arista posterior, por ejemplo; su cabeza domina a su cola. La secuencia de llamadas de la función *buscar* en la figura 9.43 que lidera al nodo a debe ser un camino en el grafo de flujo. Este camino debe, por supuesto, incluir cualquier dominador de a . Tomando que una llamada a *buscar*(b) debe ser abierta cuando se llama a *buscar*(a). Sin embargo; b está en el árbol cuando a es agregada como un descendiente de b . Por ende, $a \rightarrow b$ debe ser una arista de retirada.

de retirada, y esos son exactamente las aristas posteriores en el grafo. No obstante, si el grafo es *no reducible*, todas las aristas posteriores son aristas de retirada en cualquier DFST, pero cada DFST puede tener aristas de retirada adicionales que no sean aristas posteriores. Estas aristas de retirada pueden ser distintas de un DFST a otro. Por ende, si eliminamos todas las aristas posteriores de un grafo de flujo y el grafo restante es cíclico, entonces el grafo es no reducible, y viceversa.

Los grafos de flujo que se dan en la práctica casi siempre son reducibles. El uso exclusivo de las instrucciones de flujo de control estructuradas, como if-then-else, while-do, continue y break produce programas cuyos grafos de flujo siempre son reducibles. A menudo, incluso hasta los programas escritos que usan instrucciones goto resultan ser reducibles, ya que el programador piensa lógicamente en términos de ciclos y bifurcaciones.

Ejemplo 9.43: El grafo de flujo de la figura 9.38 es reducible. Las aristas de retirada en el grafo son todas aristas posteriores; es decir, sus cabezas dominan a sus respectivas colas. \square

Ejemplo 9.44: Considere el grafo de flujo de la figura 9.45, cuyo nodo inicial es 1. El nodo 1 domina a los nodos 2 y 3, pero 2 no domina a 3, ni viceversa. Así, este grafo de flujo no tiene aristas posteriores, ya que ninguna cabeza de ninguna arista domina a su cola. Hay dos posibles árboles de expansión con búsqueda en profundidad, dependiendo de si elegimos primero llamar a *buscar*(2) o a *buscar*(3), desde *buscar*(1). En el primer caso, la arista $3 \rightarrow 2$ es una arista de retirada pero no una arista posterior; en el segundo caso, $2 \rightarrow 3$ es la arista de retirada, pero no posterior. Por intuición, la única razón por la cual este grafo de flujo no es reducible es que podemos entrar al ciclo 2-3 en dos lugares distintos: los nodos 2 y 3. \square

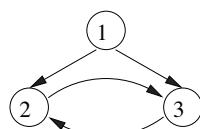


Figura 9.45: El grafo de flujo canónico, no reducible

9.6.5 Profundidad de un grafo de flujo

Dado un árbol de expansión con búsqueda en profundidad para el grafo, la *profundidad* es el número mayor de aristas de retirada en cualquier camino sin ciclos. Podemos probar que la profundidad nunca es mayor de lo que podríamos intuitivamente llamar la profundidad del anidamiento de los ciclos en el grafo de flujo. Si un grafo de flujo es reducible, podemos sustituir “de retirada” por “posterior” en la definición de “profundidad”, ya que las aristas de retirada en cualquier DFST son exactamente las aristas posteriores. La noción de profundidad entonces se vuelve independiente del DFST que se haya elegido, y podemos hablar en verdad de la “profundidad de un grafo de flujo”, en vez de la profundidad de un grafo de flujo en conexión con uno de sus árboles de expansión con búsqueda en profundidad.

Ejemplo 9.45: En la figura 9.42, la profundidad es 3, ya que hay un camino

$$10 \rightarrow 7 \rightarrow 4 \rightarrow 3$$

con tres aristas de retirada, pero no hay un camino sin ciclos con cuatro o más aristas de retirada. Es una coincidencia que el camino “más profunda” aquí sólo tenga aristas de retirada; en general podemos tener una mezcla de aristas de retirada, de avance y de cruce en un camino más profunda. \square

9.6.6 Ciclos naturales

Los ciclos pueden especificarse en un programa fuente de muchas formas distintas: pueden escribirse como ciclos *for*, ciclos *while*, o ciclos *repeat*; pueden incluso definirse mediante etiquetas e instrucciones *goto*. Desde el punto de vista del análisis de un programa, no importa cómo aparecen los ciclos en el código fuente. Lo que importa es si tienen las propiedades que permiten una fácil optimización. En especial, nos preocupamos si un ciclo tiene un solo nodo de entrada; si es así, los análisis del compilador pueden asumir ciertas condiciones iniciales para aplicar al inicio de cada iteración a través del ciclo. Esta oportunidad motiva la necesidad de la definición de un “ciclo natural”.

Un *ciclo natural* se define por dos propiedades esenciales.

1. Debe tener un solo nodo de entrada, llamado *encabezado*. Este nodo de entrada domina a todos los nodos en el ciclo, o de lo contrario no sería la única entrada para el ciclo.
2. Debe haber una arista posterior que entre al encabezado del ciclo. De no ser así, no es posible que el flujo de control regrese al encabezado directamente desde el “ciclo”; es decir, en realidad no hay ciclo.

Dado una arista posterior $n \rightarrow d$, definimos el *ciclo natural de la arista* como d más el conjunto de nodos que pueden llegar a n sin pasar a través de d . Observe que d es el encabezado del ciclo.

Algoritmo 9.46: Construcción del ciclo natural de una arista posterior.

ENTRADA: Un grafo de flujo G y una arista posterior $n \rightarrow d$.

SALIDA: El conjunto *ciclo*, que consiste en todos los nodos en el ciclo natural de $n \rightarrow d$.

MÉTODO: Haga que *ciclo* sea $\{n, d\}$. Marque d como “visitado”, de manera que la búsqueda no vaya más allá de d . Realice una búsqueda por profundidad en el grafo de flujo de control inverso, empezando con el nodo n . Inserte en *ciclo* todos los nodos visitados en esta búsqueda. Este procedimiento encuentra todos los nodos que llegan a n sin pasar a través de d . \square

Ejemplo 9.47: En la figura 9.38, hay cinco aristas posteriores, aquellos cuyas cabezas dominan sus colas: $10 \rightarrow 7$, $7 \rightarrow 4$, $4 \rightarrow 3$, $8 \rightarrow 3$ y $9 \rightarrow 1$. Observe que éstos son exactamente las aristas que consideraríamos que forman ciclos en el grafo de flujo.

La arista posterior $10 \rightarrow 7$ tiene el ciclo natural $\{7, 8, 10\}$, ya que 8 y 10 son los únicos nodos que pueden llegar a 10 sin pasar a través de 7. La arista posterior $7 \rightarrow 4$ tiene un ciclo natural que consiste en $\{4, 5, 6, 7, 8, 10\}$ y, por lo tanto, contiene el ciclo de $10 \rightarrow 7$. Por ende, asumimos que este último es un ciclo interno, contenido dentro del anterior.

Los ciclos naturales de las aristas posteriores $4 \rightarrow 3$ y $8 \rightarrow 3$ tienen el mismo encabezado, el nodo 3, y también tienen el mismo conjunto de nodos: $\{3, 4, 5, 6, 7, 8, 10\}$. Por lo tanto, vamos a combinar estos dos ciclos en uno. Este ciclo contiene los dos ciclos más pequeños que descubrimos antes.

Por último, la arista $9 \rightarrow 1$ tiene a todo el grafo de flujo completo como su ciclo natural, y por ende es el ciclo más externo. En este ejemplo, los cuatro ciclos están anidados uno dentro de otro. Sin embargo, es común tener dos ciclos, en donde ninguno de los dos sea un subconjunto del otro. \square

En los grafos de flujo reducibles, como todos las aristas de retirada son aristas posteriores, podemos asociar un ciclo natural con cada arista de retirada. Esta instrucción no se aplica para los grafos no reducibles. Por ejemplo, el grafo de flujo no reducible en la figura 9.45 tiene un ciclo que consiste en los nodos 2 y 3. Ninguno de las aristas en el ciclo es una arista posterior, por lo que este ciclo no se ajusta a la definición de un ciclo natural. No identificamos el ciclo como natural, y no está optimizado como tal. Esta situación es aceptable, ya que nuestros análisis de los ciclos pueden simplificarse si asumimos que todos los ciclos tienen nodos de una sola entrada, y de todas formas los programas no reducibles son raros en la práctica.

Al considerar sólo los ciclos naturales como “ciclos”, tenemos la útil propiedad de que a menos que dos ciclos tengan el mismo encabezado, están desunidos o uno está anidado dentro del otro. Por ende, tenemos una noción natural de los *ciclos más internos*: ciclos que no contienen otros ciclos.

Cuando dos ciclos naturales tienen el mismo encabezado, como en la figura 9.46, es difícil distinguir cuál es el ciclo interno. Por ende, vamos a suponer que cuando dos ciclos naturales tienen el mismo encabezado, y ninguno de ellos está contenido apropiadamente dentro del otro, se combinan y se tratan como un solo ciclo.

Ejemplo 9.48: Los ciclos naturales de las aristas posteriores $3 \rightarrow 1$ y $4 \rightarrow 1$ en la figura 9.46 son $\{1, 2, 3\}$ y $\{1, 2, 4\}$, respectivamente. Vamos a combinarlos en un solo ciclo, $\{1, 2, 3, 4\}$.

No obstante, si hubiera otra arista posterior $2 \rightarrow 1$ en la figura 9.46, su ciclo natural sería $\{1, 2\}$, un tercer ciclo con el encabezado 1. Este conjunto de nodos está contenido

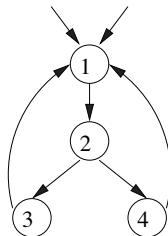


Figura 9.46: Dos ciclos con el mismo encabezado

apropiadamente dentro de $\{1, 2, 3, 4\}$, por lo que no se combinaría con los demás ciclos naturales, sino que se trataría como un ciclo interno, anidado dentro de ellos. \square

9.6.7 Velocidad de convergencia de los algoritmos de flujos de datos iterativos

Ahora estamos listos para hablar sobre la velocidad de convergencia de los algoritmos iterativos. Como vimos en la sección 9.3.3, el número máximo de iteraciones que el algoritmo puede realizar es el producto de la altura del lattice y el número de nodos en el grafo de flujo. Para muchos análisis de flujos de datos, es posible ordenar la evaluación de tal forma que el algoritmo converja en un número mucho más pequeño de iteraciones. La propiedad de interés es si todos los eventos de importancia en un nodo se propagarán a ese nodo, a lo largo de algún camino acíclico. Entre los análisis de flujos de datos que hemos visto hasta ahora, las definiciones de alcance, las expresiones disponibles y las variables vivas tienen esta propiedad, pero la propagación de constantes no. Dicho en forma más específica:

- Si una definición d está en $\text{ENT}[B]$, entonces hay un camino acíclico desde el bloque que contiene d hasta B , de tal forma que d se encuentra en todos los ENT y SAL a lo largo de ese camino.
- Si una expresión $x + y$ no está disponible a la entrada del bloque B , entonces hay un camino acíclico que demuestra que el camino proviene del nodo de entrada y no incluye una instrucción que elimine o genere a $x + y$, o el camino proviene de un bloque que elimina a $x + y$ y a lo largo de ese camino no hay una generación posterior de $x + y$.
- Si x está viva al salir del bloque B , entonces hay un camino acíclico desde B hasta un uso de x , a lo largo de la cual no hay definiciones de x .

Deberíamos verificar que en cada uno de estos casos, los caminos con ciclos no agreguen nada. Por ejemplo, si se llega a un uso de x desde el final del bloque B a lo largo de un camino con un ciclo, podemos eliminar ese ciclo para buscar un camino más corta a lo largo de la cual aún se pueda llegar al uso de x desde B .

En contraste, la propagación de constantes no tiene esta propiedad. Considere un programa simple que tiene un ciclo, el cual contiene un bloque básico con las siguientes instrucciones:

```

L:    a = b
      b = c
      c = 1
      goto L
  
```

La primera vez que visitamos el bloque básico, encontramos que c tiene el valor constante 1, pero tanto a como b no están definidas. Al visitar el bloque por segunda vez, encontramos que b y c tienen los valores constantes de 1. Se requieren tres visitas al bloque básico para que el valor constante 1 se asigne a c para llegar a a .

Si toda la información se propaga a lo largo de caminos acíclicos, tenemos una oportunidad para personalizar el orden en el que visitamos los nodos en los algoritmos de flujo de datos iterativos, para que después de unas cuantas pasadas a través de los nodos podamos estar seguros de que la información ha pasado a lo largo de todos los caminos acíclicos.

En la sección 9.6.3 vimos que si $a \rightarrow b$ es una arista, entonces el número tipo “primero en profundidad” de b es menor que el de a , sólo cuando la arista es de retirada. Para los problemas de flujo de datos hacia delante, es conveniente visitar los nodos de acuerdo con el ordenamiento tipo “primero en profundidad”. De manera específica, modificamos el algoritmo en la figura 9.23(a) sustituyendo la línea (4), la cual visita los bloques básicos en el grafo de flujo con:

```

for (cada bloque  $B$  distinto de ENTRADA, en orden “primero en profundidad”) {
```

Ejemplo 9.49: Suponga que tenemos un camino a lo largo de la cual se propaga una definición d , como:

$$3 \rightarrow 5 \rightarrow 19 \rightarrow 35 \rightarrow 16 \rightarrow 23 \rightarrow 45 \rightarrow 4 \rightarrow 10 \rightarrow 17$$

en donde los enteros representan a los números “primero en profundidad” de los bloques a lo largo del camino. Entonces, la primera vez a través del ciclo de las líneas (4) a (6) en el algoritmo de la figura 9.23(a), d se propagará de SAL[3] hacia ENT[5] y luego hacia SAL[5], y así en lo sucesivo, hasta llegar a SAL[35]. No llegará a ENT[16] en esa ronda, ya que como 16 precede a 35, ya habíamos calculado ENT[16] para cuando d se puso en SAL[35]. Sin embargo, la siguiente vez que pasamos por el ciclo de las líneas (4) a (6), al calcular ENT[16] se incluirá d , ya que está en SAL[35]. La definición d también se propagará a SAL[16], ENT[23] y así en lo sucesivo, hasta SAL[45], en donde debe esperar debido a que ENT[4] ya se calculó en esta ronda. En la tercera pasada, d va hacia ENT[4], SAL[4], ENT[10], SAL[10] y ENT[17], por lo que después de tres pasadas establecemos que d llega al bloque 17. \square

No debe ser difícil extraer el principio general a partir de este ejemplo. Si utilizamos el orden por profundidad en la figura 9.23(a), entonces el número de pasadas necesarias para propagar cualquier definición de alcance a lo largo de algún camino acíclico no es mayor de uno más que el número de aristas a lo largo de ese camino, que van desde un bloque con mayor numeración, hasta uno con menor numeración. Esas aristas son exactamente las aristas de retirada, por lo que el número de pasadas necesarias es uno más la profundidad. Desde luego que el Algoritmo 9.11 no detecta el hecho de que todas las definiciones han llegado hasta donde pueden llegar, hasta que una pasada más ya no produce cambios. Por lo tanto, el límite superior en el número de pasadas que recibe ese algoritmo con ordenamiento de bloques por

Una razón para los grafos de flujo no reducibles

Hay un lugar en el que, por lo general, no podemos esperar que un grafo de flujo sea reducible. Si invertimos las aristas de un grafo de flujo de un programa, como hicimos en el Algoritmo 9.46 para encontrar ciclos naturales, entonces tal vez no obtendríamos un grafo de flujo reducible. La razón intuitiva es que, aunque los programas ordinarios tienen ciclos con una sola entrada, esos ciclos algunas veces tienen varias salidas, que se convierten en entradas cuando invertimos las aristas.

profundidad es en realidad de dos más la profundidad. Un estudio¹⁰ ha demostrado que los grafos de flujo ordinarios tienen una profundidad promedio aproximada de 2.75. Por ende, el algoritmo converge con mucha rapidez.

En el caso de los problemas de flujo hacia atrás, al igual que las variables vivas, visitamos los nodos en el orden inverso al orden tipo “primero en profundidad”. Por ende, podemos propagar un uso de una variable en el bloque 17, en sentido hacia atrás a lo largo del camino:

$$3 \rightarrow 5 \rightarrow 19 \rightarrow 35 \rightarrow 16 \rightarrow 23 \rightarrow 45 \rightarrow 4 \rightarrow 10 \rightarrow 17$$

en una pasada a ENT[4], en donde debemos esperar la siguiente pasada para poder llegar a SAL[45]. En la segunda pasada llega a ENT[16], y en la tercera pasada va desde SAL[35] hasta SAL[3].

En general, basta un número de pasadas igual a uno más la profundidad para llevar el uso de una variable hacia atrás, a lo largo de cualquier camino acíclico. Sin embargo, debemos elegir el inverso del orden tipo “primero en profundidad” para visitar los nodos en una pasada, porque así, los usos se propagan a lo largo de cualquier secuencia decremental en una sola pasada.

El límite descrito hasta ahora es un límite superior en todos los problemas en los que los caminos cíclicos no agregan información al análisis. En problemas especiales como los dominadores, el algoritmo converge incluso con mayor rapidez. En el caso en el que el grafo del flujo de entrada es reducible, el conjunto correcto de dominadores para cada nodo se obtiene en la primera iteración de un algoritmo de flujo de datos que visita los nodos en orden tipo “primero en profundidad”. Si no sabemos que la entrada puede reducirse antes de tiempo, se requiere una iteración adicional para determinar que ha ocurrido la convergencia.

9.6.8 Ejercicios para la sección 9.6

Ejercicio 9.6.1: Para el grafo de flujo de la figura 9.10 (vea los ejercicios de la sección 9.1):

- i. Calcule la relación de los dominadores.
- ii. Encuentre el dominador inmediato de cada nodo.

¹⁰D. E. Knuth, “An empirical study of FORTRAN programs”, *Software – Practice and Experience* 1:2 (1971), pp. 105-133.

- iii. Construya el árbol de dominadores.
- iv. Encuentre un ordenamiento tipo “primero en profundidad” para el grafo de flujo.
- v. Indique las aristas de avance, de retirada, de cruce y de árbol para su respuesta al inciso iv.
- vi. ¿Es reducible el grafo de flujo?
- vii. Calcule la profundidad del grafo de flujo.
- viii. Encuentre los ciclos naturales del grafo de flujo.

Ejercicio 9.6.2: Repita el ejercicio 9.6.1 en los siguientes grafos de flujo:

- a) Figura 9.3.
- b) Figura 8.9.
- c) Su grafo de flujo del ejercicio 8.4.1.
- d) Su grafo de flujo del ejercicio 8.4.2.

! Ejercicio 9.6.3: Demuestre lo siguiente acerca de la relación *dom*:

- a) Si $a \text{ dom } b$ y $b \text{ dom } c$, entonces $a \text{ dom } c$ (condición *transitiva*).
- b) No es posible que tanto $a \text{ dom } b$ como $b \text{ dom } a$ sean válidas, si $a \neq b$ (*anti-simetría*).
- c) Si a y b son dos dominadores de n , entonces una de las relaciones $a \text{ dom } b$ o $b \text{ dom } a$ debe ser válida.
- d) Cada nodo n (excepto la entrada) tiene un *dominador inmediato* único: el dominador que aparece más cerca de n , a lo largo de cualquier camino acíclico desde la entrada hasta n .

! Ejercicio 9.6.4: La figura 9.42 es una presentación “primero en profundidad” del grafo de flujo de la figura 9.38. ¿Cuántas otras presentaciones tipo “primero en profundidad” hay de este grafo de flujo? Recuerde, el orden de los hijos es importante al distinguir las presentaciones tipo “primero en profundidad”.

!! Ejercicio 9.6.5: Demuestre que un grafo de flujo es reducible si, y sólo si al eliminar todas las aristas posteriores (aquellos cuyas cabezas dominan a sus colas), el grafo de flujo resultante es acíclico.

! Ejercicio 9.6.6: Un *grafo de flujo completo* sobre n nodos tiene los arcos $i \rightarrow j$ entre dos nodos i y j cualesquiera (en ambas direcciones). ¿Para qué valores de n es reducible este grafo?

! Ejercicio 9.6.7: Un *grafo de flujo acíclico completo* sobre n nodos $1, 2, \dots, n$ tiene los arcos $i \rightarrow j$ para todos los nodos i y j tales que $i < j$. El nodo 1 es la entrada.

- a) ¿Para qué valores de n es reducible este grafo?
- b) ¿Cambia su respuesta al inciso (a) si agrega autociclos $i \rightarrow i$ para todos los nodos i ?

! Ejercicio 9.6.8: El ciclo natural de una arista posterior $n \rightarrow h$ se definió como h más el conjunto de nodos que pueden llegar a n sin pasar a través de h . Muestre que h domina a todos los nodos en el ciclo natural de $n \rightarrow h$.

!! Ejercicio 9.6.9: Afirmamos que el grafo de flujo de la figura 9.45 no es reducible. Si los arcos se sustituyeran por caminos de nodos separados (excepto los puntos finales, desde luego), entonces el grafo de flujo aún sería no reducible. De hecho, el nodo 1 no necesita ser la entrada; puede ser cualquier nodo alcanzable desde la entrada, a lo largo de un camino cuyos nodos intermedios no forman parte de ninguna de los cuatro caminos que se muestran de manera explícita. Demuestre lo contrario: que todo grafo de flujo no reducible tiene un subgrafo como el de la figura 9.45, pero en donde tal vez los arcos se sustituyan por caminos de nodos separados y el nodo 1 puede ser cualquier nodo alcanzable desde la entrada, por un camino que tiene nodos separados de las otras cuatro caminos.

!! Ejercicio 9.6.10: Muestre que cada presentación “primero en profundidad” para cada grafo de flujo no reducible tiene una arista de retirada que no es una arista posterior.

!! Ejercicio 9.6.11: Muestre que si la siguiente condición:

$$f(a) \wedge g(a) \wedge a \leq f(g(a))$$

se aplica para todas las funciones f y g , y el valor a , entonces el algoritmo iterativo general, el Algoritmo 9.25, con la iteración después de un ordenamiento tipo “primero en profundidad”, converge dentro de un número de pasadas igual a 2 más la profundidad.

! Ejercicio 9.6.12: Encuentre un grafo de flujo no reducible con dos DSFTs distintos, que tengan distintas profundidades.

! Ejercicio 9.6.13: Demuestre lo siguiente:

- a) Si una definición d está en $\text{ENT}[B]$, entonces hay un camino acíclico desde el bloque que contiene d hasta B , de tal forma que d está en todos los ENT y SAL a lo largo de ese camino.
- b) Si una expresión $x + y$ no está disponible en la entrada al bloque B , entonces hay un camino acíclico que demuestra ese hecho; ya sea que el camino provenga del nodo de entrada y no incluya una instrucción para eliminar o generar a $x + y$, o que el camino provenga de un bloque que elimine a $x + y$ y a lo largo del camino no haya una generación subsiguiente de $x + y$.
- c) Si x está viva al salir del bloque B , entonces hay una camino acíclico desde B hasta un uso de x , a lo largo de la cual no hay definiciones de x .

9.7 Análisis basado en regiones

El algoritmo de análisis de flujo de datos iterativo que hemos visto hasta ahora es sólo un método para resolver los problemas de flujos de datos. Aquí hablaremos sobre otro método llamado *análisis basado en regiones*. Recuerde que en el método del análisis iterativo, creamos funciones de transferencia para los bloques básicos, y después encontramos la solución del punto fijo mediante varias pasadas a través de los bloques. En vez de crear funciones de transferencia sólo para bloques individuales, un análisis basado en regiones busca funciones de transferencia que resuman la ejecución de regiones cada vez más grandes del programa. En última instancia, se construyen funciones de transferencia para procedimientos completos y después se aplican, para obtener directamente los valores deseados del flujo de datos.

Mientras que el marco de trabajo del flujo de datos que utiliza un algoritmo iterativo se especifica mediante un semi-lattice de valores del flujo de datos y una familia de funciones de transferencia que se cierran bajo la composición, el análisis basado en regiones requiere más elementos. Un marco de trabajo basado en regiones incluye tanto un semi-lattice de valores del flujo de datos como un semi-lattice de funciones de transferencia que deben poseer un operador de reunión, un operador de composición y un operador de cierre. En la sección 9.7.4 veremos lo que todos estos elementos implican.

Un análisis basado en regiones es muy útil para los problemas de flujo de datos en los que los caminos que tienen ciclos pueden modificar los valores del flujo de datos. El operador de cierre permite sintetizar el efecto de un ciclo de una manera más efectiva que el análisis iterativo. La técnica es también útil para el análisis entre procedimientos, en donde las funciones de transferencia asociadas con la llamada a un procedimiento pueden tratarse como las funciones de transferencia asociadas con los bloques básicos.

Por cuestión de simplicidad, vamos a considerar sólo problemas de flujo de datos hacia delante en esta sección. Primero ilustraremos cómo funciona el análisis basado en regiones, mediante el uso del conocido ejemplo de las definiciones de alcance. En la sección 9.8 mostraremos un uso más convincente de esta técnica, cuando estudiemos el análisis de las variables de inducción.

9.7.1 Regiones

En el análisis basado en regiones, un programa se ve como una jerarquía de *regiones*, que son (aproximadamente) porciones de un grafo de flujo que sólo tienen un punto de entrada. Este concepto de ver el código como una jerarquía de regiones nos debe parecer intuitivo, ya que un procedimiento estructurado por bloques se organiza de manera natural como una jerarquía de regiones. Cada instrucción en un programa estructurado por bloques es una región, ya que el flujo de control sólo puede entrar al inicio de una instrucción. Cada nivel de anidamiento de instrucciones corresponde a un nivel en la jerarquía de regiones.

De manera formal, una *región* de un grafo de flujo es una colección de nodos N y aristas E , de tal forma que:

1. Hay un encabezado h en N que domina a todos los nodos en N .
2. Si algún nodo m puede llegar a un nodo n en N sin pasar a través de h , entonces m también está en N .

3. E es el conjunto de todas las aristas del flujo de control entre los nodos n_1 y n_2 en N , excepto (tal vez) algunos que entran a h .

Ejemplo 9.50: Es evidente que un ciclo natural es una región, pero una región no tiene necesariamente una arista posterior, y no necesita contener ciclos. Por ejemplo, en la figura 9.47, los nodos B_1 y B_2 , junto con la arista $B_1 \rightarrow B_2$ forman una región; lo mismo pasa con los nodos B_1 , B_2 y B_3 con las aristas $B_1 \rightarrow B_2$, $B_2 \rightarrow B_3$ y $B_1 \rightarrow B_3$.

Sin embargo, el subgrafo con los nodos B_2 y B_3 con la arista $B_2 \rightarrow B_3$ no forma una región, ya que el control puede entrar al subgrafo en ambos nodos B_2 y B_3 . Dicho de forma más precisa, B_2 no domina a B_3 ni viceversa, por lo que se viola la condición (1) para una región. Incluso si eligiéramos, por decir, a B_2 para que fuera el “encabezado”, violaríamos la condición (2), ya que podemos llegar a B_3 desde B_1 sin pasar a través de B_2 , y B_1 no se encuentra en la “región”. \square

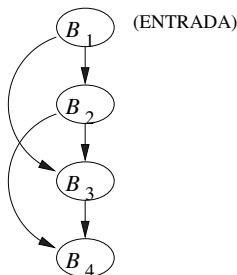


Figura 9.47: Ejemplos de regiones

9.7.2 Jerarquías de regiones para grafos de flujo reducibles

A continuación vamos a suponer que el grafo de flujo es reducible. Si en ocasiones debemos lidiar con grafos de flujo no reducibles, entonces podemos usar una técnica conocida como “división de nodos”, que veremos en la sección 9.7.6.

Para construir una jerarquía de regiones, identificamos los ciclos naturales. En la sección 9.6.6 vimos que en un grafo de flujo reducible, dos ciclos naturales cualesquiera están separados o uno está anidado dentro del otro. El proceso de “analizar” un grafo de flujo reducible en su jerarquía de ciclos empieza con cada bloque, como una región por sí solo. A estas regiones les llamamos *regiones hoja*. Después, ordenamos los ciclos naturales desde el interior hacia fuera; es decir, empezando con los ciclos más internos. Para procesar un ciclo, sustituimos el ciclo completo por un nodo en dos pasos:

1. En primer lugar, el *cuerpo* del ciclo L (todos los nodos y aristas, excepto las aristas posteriores que van al encabezado) se sustituye por un nodo que representa a una región R . Las aristas que van al encabezado de L ahora entran al nodo para R . Una arista proveniente de cualquier salida del ciclo L se sustituye por una arista de R que va al mismo destino. No obstante, si la arista es una arista posterior, entonces se convierte en un ciclo en R . A R le llamamos *región de cuerpo*.

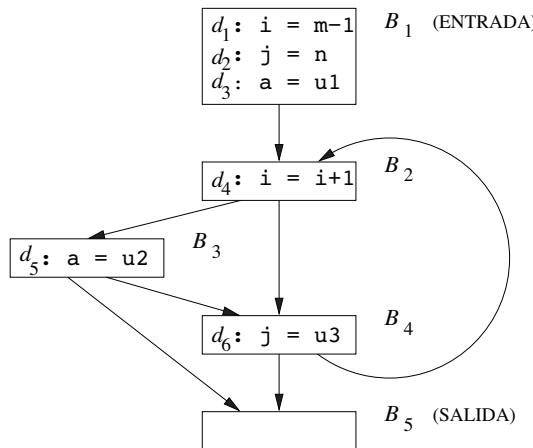
2. A continuación, construimos una región R' que representa a todo el ciclo natural L . A R' le llamamos *región de ciclo*. La única diferencia entre R y R' es que la última incluye las aristas posteriores que van al encabezado del ciclo L . Dicho de otra forma, cuando R' sustituye a R en el grafo de flujo, todo lo que tenemos que hacer es eliminar la arista que va de R a sí misma.

Procedemos de esta forma, reduciendo ciclos cada vez más grandes a nodos individuales, primero con una arista de ciclo y después sin él. Como los ciclos de un grafo de flujo reducible están anidados o separados, el nodo de la región del ciclo puede representar a todos los nodos del ciclo natural en la serie de grafos de flujo que se construyen mediante este proceso de reducción.

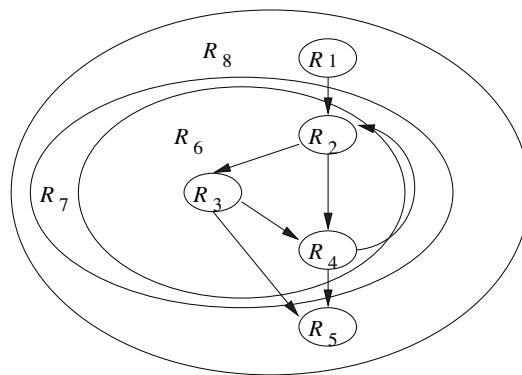
En un momento dado, todos los ciclos naturales se reducen a nodos individuales. En ese punto, el grafo de flujo puede reducirse a un solo nodo, o puede haber varios nodos restantes, sin ciclos; es decir, el grafo de flujo reducido es un grafo acíclico de más de un nodo. En el primer caso terminamos de construir la jerarquía de regiones, mientras que en el último caso construimos una región del cuerpo más para todo el grafo de flujo.

Ejemplo 9.51: Considere el grafo de flujo de control en la figura 9.48(a). Hay una arista posterior en este grafo de flujo, que conduce de B_4 a B_2 . La jerarquía de regiones se muestra en la figura 9.48(b); las aristas que se muestran son las aristas en los grafos de flujo de las regiones. En total, hay 8 regiones:

1. Las regiones R_1, \dots, R_5 son regiones hoja que representan a los bloques B_1 a B_5 , respectivamente. Cada bloque es también un bloque de salida en esta región.
2. La región del cuerpo R_6 representa al cuerpo del único ciclo en el grafo de flujo; consiste en las regiones R_2, R_3 y R_4 , y tres aristas entre regiones: $B_2 \rightarrow B_3, B_2 \rightarrow B_4$ y $B_3 \rightarrow B_4$. Tiene dos bloques de salida, B_3 y B_4 , ya que ambos tienen aristas salientes que no están contenidos en la región. La figura 9.49(a) muestra el grafo de flujo con R_6 reducida a un solo nodo. Observe que, aunque las aristas $R_3 \rightarrow R_5$ y $R_4 \rightarrow R_5$ se han sustituido por la arista $R_6 \rightarrow R_5$, es importante recordar que la última arista representa a las dos primeras aristas, ya que tenemos que propagar las funciones de transferencia a través de esta arista en un momento dado, y debemos saber que lo que sale de ambos bloques B_3 y B_4 llegará al encabezado de R_5 .
3. La región del ciclo R_7 representa a todo el ciclo natural. Incluye una subregión, R_6 , y una arista posterior $B_4 \rightarrow B_2$. También tiene dos nodos de salida, de nuevo B_3 y B_4 . La figura 9.49(b) muestra el grafo de flujo después de que todo el ciclo natural se reduce a R_7 .
4. Por último, la región del cuerpo R_8 es la región superior. Incluye tres regiones, R_1, R_7 , R_5 y tres aristas entre regiones, $B_1 \rightarrow B_2, B_3 \rightarrow B_5$ y $B_4 \rightarrow B_5$. Cuando reducimos el grafo de flujo a R_8 , se convierte en un solo nodo. Como no hay aristas posteriores que vayan a su encabezado, R_1 , no hay necesidad de un paso final que reduzca esta región del cuerpo a una región de ciclo.



(a)



(b)

Figura 9.48: (a) Un grafo de flujo de ejemplo para el problema de las definiciones de alcance y (b) su jerarquía de regiones

□

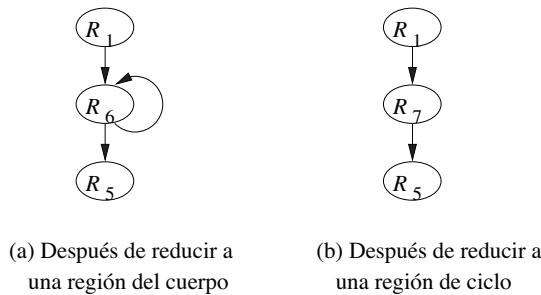


Figura 9.49: Pasos en la reducci  n del grafo de flujo de la figura 9.47 a una sola regi  n

Para resumir el proceso de descomponer los grafos de flujo reducibles en forma jer  quica, ofrecemos el siguiente algoritmo.

Algoritmo 9.52: Construir un orden de regiones de abajo hacia arriba, de un grafo de flujo reducible.

ENTRADA: Un grafo de flujo reducible G .

SALIDA: Una lista de regiones de G que pueden usarse en los problemas de flujo de datos basados en regiones.

M  TODO:

1. Empiece la lista con todas las regiones hoja que consistan en bloques individuales de G , en cualquier orden.
2. Elija en forma repetida un ciclo natural L , de tal forma que si hay ciclos naturales contenidos dentro de L , entonces ya se han agregado las regiones de cuerpo y de ciclo de estos ciclos a la lista. Agregue primero la regi  n consistente en el cuerpo de L (es decir, L sin las aristas posteriores que van al encabezado de L), y despu  s la regi  n de ciclo de L .
3. Si todo el grafo de flujo no es en s   un ciclo natural, agregue al final de la lista la regi  n consistente en todo el grafo de flujo.

□

9.7.3 Generalidades de un an  lisis basado en regiones

Para cada regi  n R , y para cada subregi  n R' dentro de R , calculamos una funci  n de transferencia $f_{R, \text{ENT}[R']}$ que sintetice el efecto de ejecutar todos los caminos posibles que conducen de la entrada de R a la entrada de R' , que al mismo tiempo permanecen dentro de R .

Cuál es el origen de la “reducibilidad”

Ahora podemos ver por qué los grafos de flujo reducibles recibieron ese nombre. Aunque no vamos a demostrar este hecho, la definición de “grafo de flujo reducible” utilizada en este libro, que involucra a las aristas posteriores del grafo, es equivalente a varias definiciones en las que reducimos de manera mecánica el grafo de flujo a un solo nodo. El proceso de colapsar los ciclos naturales descritos en la sección 9.7.2 es uno de ellos. Otra definición interesante es que los grafos de flujo reducibles son los únicos grafos que pueden reducirse a un solo nodo mediante las siguientes dos transformaciones:

T_1 : Eliminar una arista que va de un nodo a sí mismo.

T_2 : Si el nodo n tiene un solo predecesor m , y n no es la entrada del grafo de flujo, combinar m y n .

Decimos que un bloque B dentro de R es un *bloque de salida* de la región R si tiene una arista saliente hacia algún bloque fuera de R . También calculamos una función de transferencia para cada bloque de salida B de R , denotado como $f_{R, \text{SAL}[B]}$, que sintetiza el efecto de ejecutar todos los caminos posibles dentro de R , que conducen de la entrada de R a la salida de B .

Después procedemos hacia arriba en la jerarquía de regiones, calculando funciones de transferencia para regiones cada vez más grandes. Empezamos con regiones que sean bloques individuales, en donde $f_{B, \text{ENT}[B]}$ es sólo la función identidad y $f_{B, \text{SAL}[B]}$ es la función de transferencia para el mismo bloque B . A medida que avanzamos hacia arriba en la jerarquía,

- Si R es una región de cuerpo, entonces las aristas que pertenecen a R forman un grafo acíclico en las subregiones de R . Podemos proceder a calcular las funciones de transferencia en un orden topológico de las subregiones.
- Si R es una región de ciclo, entonces sólo tenemos que tomar en cuenta el efecto de las aristas posteriores que van al encabezado de R .

En un momento dado, llegamos a la parte superior de la jerarquía y calculamos las funciones de transferencia para la región R_n que constituye el grafo completo. En el Algoritmo 9.53 veremos cómo realizar cada uno de estos cálculos.

El siguiente paso es calcular los valores del flujo de datos en la entrada y salida de cada bloque. Procesamos las regiones en el orden inverso, empezando con la región R_n y avanzando hacia abajo en la jerarquía. Para cada región, calculamos los valores del flujo de datos en la entrada. Para la región R_n , aplicamos $f_{R_n, \text{ENT}[R]}$ ($\text{ENT}[\text{ENTRADA}]$) para obtener los valores del flujo de datos en la entrada de las subregiones R en R_n . Repetimos hasta llegar a los bloques básicos en las hojas de la jerarquía de regiones.

9.7.4 Suposiciones necesarias sobre las funciones transformación

Para que el análisis basado en regiones pueda funcionar, debemos hacer varias suposiciones sobre las propiedades del conjunto de funciones de transferencia en el marco de trabajo. En específico, necesitamos tres operaciones primitivas relacionadas con las funciones de transferencia: composición, reunión y cerradura; sólo la primera se requiere para los marcos de trabajo de flujo de datos que utilizan el algoritmo iterativo.

Composición

La función de transferencia de una secuencia de nodos puede derivarse mediante la composición de las funciones que representan a los nodos individuales. Hagamos que f_1 y f_2 sean las funciones de transferencia de los nodos n_1 y n_2 . El efecto de ejecutar n_1 seguido de n_2 se representa mediante $f_2 \circ f_1$. En la sección 9.2.2 hablamos sobre la composición de funciones, y en la sección 9.2.4 se mostró un ejemplo mediante el uso de las definiciones de alcance. Para repasar, hagamos que gen_i y $eliminar_i$ sean los conjuntos gen y $eliminar$ para f_i . Entonces:

$$\begin{aligned} f_2 \circ f_1(x) &= gen_2 \cup ((gen_1 \cup (x - eliminar_1)) - eliminar_2) \\ &= (gen_2 \cup (gen_1 - eliminar_2)) \cup (x - (eliminar_1 \cup eliminar_2)) \end{aligned}$$

Por ende, los conjuntos gen y $eliminar$ para $f_2 \circ f_1$ son $gen_2 \cup (gen_1 - eliminar_2)$ y $eliminar_1 \cup eliminar_2$, respectivamente. La misma idea funciona para cualquier función de transferencia de la forma gen-eliminar. También pueden cerrarse otras funciones de transferencia, pero tenemos que considerar cada caso por separado.

Reunión

Aquí, las mismas funciones de transferencia son valores de un semi-lattice con un operador de reunión \wedge_f . La reunión de dos funciones de transferencia f_1 y f_2 , $f_1 \wedge_f f_2$, se define mediante $(f_1 \wedge_f f_2)(x) = f_1(x) \wedge f_2(x)$, en donde \wedge es el operador de reunión para los valores del flujo de datos. El operador de reunión en las funciones de transferencia se utiliza para combinar el efecto de los caminos alternativas de ejecución con los mismos puntos finales. En donde no sea ambiguo, de ahora en adelante nos referiremos al operador de reunión de las funciones de transferencia también como \wedge . Para el marco de trabajo de las definiciones de alcance, tenemos que:

$$\begin{aligned} (f_1 \wedge f_2)(x) &= f_1(x) \wedge f_2(x) \\ &= (gen_1 \cup (x - eliminar_1)) \cup (gen_2 \cup (x - eliminar_2)) \\ &= (gen_1 \cup gen_2) \cup (x - (eliminar_1 \cap eliminar_2)) \end{aligned}$$

Es decir, los conjuntos gen y $eliminar$ para $f_1 \wedge f_2$ son $gen_1 \cup gen_2$ y $eliminar_1 \cap eliminar_2$, respectivamente. De nuevo, se aplica el mismo argumento para cualquier conjunto de funciones de transferencia gen-eliminar.

Cerradura

Si f representa a la función de transferencia de un ciclo, entonces f^n representa el efecto de recorrer el ciclo n veces. En el caso en el que no se conoce el número de iteraciones, tenemos que suponer que el ciclo puede ejecutarse 0 o más veces. Representamos la función de transferencia de dicho ciclo mediante f^* , la *cerradura* de f , que se define por:

$$f^* = \bigwedge_{n \geq 0} f^n.$$

Observe que f^0 debe ser la función de transferencia identidad, ya que representa el efecto de recorrer el ciclo cero veces; es decir, empezar en la entrada y no moverse. Si dejamos que I represente a la función de transferencia identidad, entonces podemos escribir:

$$f^* = I \wedge \left(\bigwedge_{n > 0} f^n \right).$$

Suponga que la función de transferencia f en un marco de trabajo de definiciones de alcance tiene un conjunto *gen* y un conjunto *eliminar*. Entonces,

$$\begin{aligned} f^2(x) &= f(f(x)) \\ &= \left(\text{gen} \cup \left((\text{gen} \cup (x - \text{eliminar})) - \text{eliminar} \right) \right. \\ &\quad \left. = \text{gen} \cup (x - \text{eliminar}) \right) \\ f^3(x) &= f(f^2(x)) \\ &= \text{gen} \cup (x - \text{eliminar}) \end{aligned}$$

y así sucesivamente: cualquier $f^n(x)$ es $\text{gen} \cup (x - \text{eliminar})$. Es decir, al recorrer un ciclo no se ve afectada la función de transferencia, si es de la forma *gen-eliminar*. Por lo tanto,

$$\begin{aligned} f^*(x) &= I \wedge f^1(x) \wedge f^2(x) \wedge \dots \\ &= x \cup \left(\text{gen} \cup (x - \text{eliminar}) \right) \\ &= \text{gen} \cup x \end{aligned}$$

Es decir, los conjuntos *gen* y *eliminar* para f^* son *gen* y \emptyset , respectivamente. Por intuición, como en definitiva no podemos recorrer un ciclo, cualquier cosa en x alcanzará la entrada al ciclo. En todas las iteraciones siguientes, las definiciones de alcance incluyen a las que están en el conjunto *gen*.

9.7.5 Un algoritmo para el análisis basado en regiones

El siguiente algoritmo resuelve un problema de análisis de flujo de datos hacia delante en un grafo de flujo reducible, de acuerdo con cierto marco de trabajo que cumple con las suposiciones de la sección 9.7.4. Recuerde que $f_{R, \text{ENT}[R']}$ y $f_{R, \text{SAL}[B]}$ se refieren a las funciones de transferencia que transforman los valores del flujo de datos en la entrada a la región R , en el valor correcto en la entrada de la subregión R' y en la salida del bloque de salida B , respectivamente.

Algoritmo 9.53: Análisis basado en regiones.

ENTRADA: Un marco de trabajo del flujo de datos, con las propiedades descritas en la sección 9.7.4 y un grafo de flujo reducible G .

SALIDA: Valores del flujo de datos $\text{ENT}[B]$ para cada bloque B de G .

MÉTODO:

1. Use el Algoritmo 9.52 para construir la secuencia de abajo hacia arriba de regiones de G , por decir R_1, R_2, \dots, R_n , en donde R_n es la región de más arriba.
2. Realice el análisis ascendente para calcular las funciones de transferencia, sintetizando el efecto de ejecutar una región. Para cada región R_1, R_2, \dots, R_n , en el orden ascendente, haga lo siguiente:
 - (a) Si R es una región hoja que corresponde al bloque B , haga que $f_{R, \text{ENT}[B]} = I$ y $f_{R, \text{SAL}[B]} = f_B$, la función de transferencia asociada con el bloque B .
 - (b) Si R es una región de cuerpo, realice el cálculo de la figura 9.50(a).
 - (c) Si R es una región de ciclo, realice el cálculo de la figura 9.50(b).
3. Realice la pasada de arriba hacia abajo para buscar los valores del flujo de datos al inicio de cada región.
 - (a) $\text{ENT}[R_n] = \text{ENT}[\text{ENTRADA}]$.
 - (b) Para cada región R en $\{R_1, \dots, R_{n-1}\}$, en el orden descendente, calcule $\text{ENT}[R] = f_{R', \text{ENT}[R]}(\text{ENT}[R'])$, en donde R' es la región de cerradura inmediata de R .

Primero vamos a ver los detalles del funcionamiento del análisis ascendente. En la línea (1) de la figura 9.50(a) visitamos las subregiones de una región de cuerpo, en cierto orden topológico. La línea (2) calcula la función de transferencia que representa todos los posibles caminos desde el encabezado de R hasta el encabezado de S ; después en las líneas (3) y (4) calculamos las funciones de transferencia que representan todos los posibles caminos que van del encabezado de R a las salidas de R ; es decir, a las salidas de todos los bloques que tienen sucesores fuera de S . Observe que todos los predecesores B' en R deben estar en regiones que precedan a S en el orden topológico construido en la línea (1). Por ende, $f_{R, \text{SAL}[B']}$ ya se habría calculado en la línea (4) de una iteración anterior, a través del ciclo externo.

Para las regiones de ciclo, realizamos los pasos de las líneas (1) a (4) en la figura 9.50(b). La línea (2) calcula el efecto de recorrer la región S de cuerpo del ciclo cero o más veces. Las líneas (3) y (4) calculan el efecto en las salidas del ciclo, después de una o más iteraciones.

En la pasada de arriba hacia abajo del algoritmo, el paso 3(a) primero asigna la condición delimitadora a la entrada de la región de más arriba. Después, si R está contenido inmediatamente en R' , simplemente podemos aplicar la función de transferencia $f_{R', \text{ENT}[R]}$ al valor del flujo de datos $\text{ENT}[R']$ para calcular $\text{ENT}[R]$. \square

- 1) **for** (cada subregión S que está contenida inmediatamente en R , en orden topológico) {
- 2) $f_{R, \text{ENT}[S]} = \bigwedge \text{predecesores } B \text{ en } R \text{ del encabezado de } S f_{R, \text{SAL}[B]}$;
/* si S es el encabezado de la región R , entonces $f_{R, \text{ENT}[S]}$ es la reunión sobre nada, lo cual viene siendo la función identidad */
- 3) **for** (cada bloque de salida B en S)
- 4) $f_{R, \text{SAL}[B]} = f_{S, \text{SAL}[B]} \circ f_{R, \text{ENT}[S]}$;

(a) Construcción de funciones de transferencia para una región de cuerpo R

- 1) **let** S sea la región del cuerpo inmediatamente anidada dentro de R ; es decir, S es R sin las aristas posteriores que van de R al encabezado de R ;
- 2) $f_{R, \text{ENT}[S]} = (\bigwedge \text{predecesores } B \text{ en } R \text{ del encabezado de } S f_{S, \text{SAL}[B]})^*$;
- 3) **for** (cada bloque de salida B en R)
- 4) $f_{R, \text{SAL}[B]} = f_{S, \text{SAL}[B]} \circ f_{R, \text{ENT}[S]}$;

(b) Construcción de funciones de transferencia para una región de ciclo R'

Figura 9.50: Detalles de los cálculos de flujo de datos basado en regiones

Ejemplo 9.54: Vamos a aplicar el Algoritmo 9.53 para buscar definiciones de alcance en el grafo de flujo de la figura 9.48(a). El paso 1 construye el orden ascendente en el que se visitan las regiones; este orden será el orden numérico de sus subíndices, R_1, R_2, \dots, R_n .

A continuación se resumen los valores de los conjuntos *gen* y *eliminar* para los cinco bloques:

B	B_1	B_2	B_3	B_4	B_5
gen_B	$\{d_1, d_2, d_3\}$	$\{d_4\}$	$\{d_5\}$	$\{d_6\}$	\emptyset
$eliminar_B$	$\{d_4, d_5, d_6\}$	$\{d_1\}$	$\{d_3\}$	$\{d_2\}$	\emptyset

Recuerde las reglas simplificadas para las funciones de transferencia *gen*-*eliminar*, de la sección 9.7.4:

- Para tomar la reunión de funciones de transferencia, tome la unión de los conjuntos *gen* y la intersección de los conjuntos *eliminar*.
- Para componer funciones de transferencia, tome la unión de los conjuntos *gen* y *eliminar*. Sin embargo, como excepción, una expresión generada por la primera función, no generada por la segunda, pero eliminada por ésta *no* se encuentra en el conjunto *gen* del resultado.
- Para tomar la cerradura de una función de transferencia, retenga su conjunto *gen* y sustituya el conjunto *eliminar* por \emptyset .

Las primeras cinco regiones R_1, \dots, R_5 son los bloques B_1, \dots, B_5 , respectivamente. Para $1 \leq i \leq 5$, $f_{R_i, \text{ENT}[B_i]}$ es la función de identidad, y $f_{R_i, \text{SAL}[B_i]}$ es la función de transferencia para el bloque B_i :

$$F_{B_i, \text{SAL}[B_i]}(x) = (x - \text{eliminar}_{B_i}) \cup \text{gen}_{B_i}.$$

	Función de transferencia	<i>gen</i>	<i>eliminar</i>
R_6	$f_{R_6, \text{ENT}[B_2]} = I$	\emptyset	\emptyset
	$f_{R_6, \text{SAL}[B_2]} = f_{R_2, \text{SAL}[B_2]} \circ f_{R_6, \text{ENT}[B_2]}$	$\{d_4\}$	$\{d_1\}$
	$f_{R_6, \text{ENT}[B_3]} = f_{R_6, \text{SAL}[B_2]}$	$\{d_4\}$	$\{d_1\}$
	$f_{R_6, \text{SAL}[B_3]} = f_{R_3, \text{SAL}[B_3]} \circ f_{R_6, \text{ENT}[B_3]}$	$\{d_4, d_5\}$	$\{d_1, d_3\}$
	$f_{R_6, \text{ENT}[B_4]} = f_{R_6, \text{SAL}[B_2]} \wedge f_{R_6, \text{SAL}[B_3]}$	$\{d_4, d_5\}$	$\{d_1\}$
	$f_{R_6, \text{SAL}[B_4]} = f_{R_4, \text{SAL}[B_4]} \circ f_{R_6, \text{ENT}[B_4]}$	$\{d_4, d_5, d_6\}$	$\{d_1, d_2\}$
R_7	$f_{R_7, \text{ENT}[R_6]} = f_{R_6, \text{SAL}[B_4]}^*$	$\{d_4, d_5, d_6\}$	\emptyset
	$f_{R_7, \text{SAL}[B_3]} = f_{R_6, \text{SAL}[B_3]} \circ f_{R_7, \text{ENT}[R_6]}$	$\{d_4, d_5, d_6\}$	$\{d_1, d_3\}$
	$f_{R_7, \text{SAL}[B_4]} = f_{R_6, \text{SAL}[B_4]} \circ f_{R_7, \text{ENT}[R_6]}$	$\{d_4, d_5, d_6\}$	$\{d_1, d_2\}$
R_8	$f_{R_8, \text{ENT}[B_1]} = I$	\emptyset	\emptyset
	$f_{R_8, \text{SAL}[B_1]} = f_{R_1, \text{SAL}[B_1]}$	$\{d_1, d_2, d_3\}$	$\{d_4, d_5, d_6\}$
	$f_{R_8, \text{ENT}[R_7]} = f_{R_8, \text{SAL}[B_1]}$	$\{d_1, d_2, d_3\}$	$\{d_4, d_5, d_6\}$
	$f_{R_8, \text{SAL}[B_3]} = f_{R_7, \text{SAL}[B_3]} \circ f_{R_8, \text{ENT}[R_7]}$	$\{d_2, d_4, d_5, d_6\}$	$\{d_1, d_3\}$
	$f_{R_8, \text{SAL}[B_4]} = f_{R_7, \text{SAL}[B_4]} \circ f_{R_8, \text{ENT}[R_7]}$	$\{d_3, d_4, d_5, d_6\}$	$\{d_1, d_2\}$
	$f_{R_8, \text{ENT}[R_5]} = f_{R_8, \text{SAL}[B_3]} \wedge f_{R_8, \text{SAL}[B_4]}$	$\{d_2, d_3, d_4, d_5, d_6\}$	$\{d_1\}$
	$f_{R_8, \text{SAL}[B_5]} = f_{R_5, \text{SAL}[B_5]} \circ f_{R_8, \text{ENT}[R_5]}$	$\{d_2, d_3, d_4, d_5, d_6\}$	$\{d_1\}$

Figura 9.51: Cálculo de funciones de transferencia para el grafo de flujo en la figura 9.48(a), usando el análisis basado en regiones

El resto de las funciones de transferencia construidas en el paso 2 del Algoritmo 9.50 se resumen en la figura 9.51. La región R_6 , que consiste en las regiones R_2, R_3 y R_4 , representa

el cuerpo del ciclo y, por lo tanto, no incluye la arista posterior $B_4 \rightarrow B_2$. El orden para procesar estas regiones será el único orden topológico: R_2, R_3, R_4 . En primer lugar, R_2 no tiene predecesores dentro de R_6 ; recuerde que la arista $B_4 \rightarrow B_2$ va hacia fuera de R_6 . Por lo tanto, $f_{R_6, \text{ENT}[B_2]}$ es la función de identidad,¹¹ y $f_{R_6, \text{SAL}[B_2]}$ es la función de transferencia para el mismo bloque B_2 .

El encabezado de la región B_3 tiene un predecesor dentro de R_6 ; a saber, R_2 . La función de transferencia a su entrada es sólo la función de transferencia a la salida de B_2 , $f_{R_6, \text{SAL}[B_2]}$, que ya se ha calculado. Componemos esta función con la función de transferencia de B_3 dentro de su propia región, para calcular la función de transferencia a la salida de B_3 .

Por último, para la función de transferencia a la entrada de R_4 , debemos calcular:

$$f_{R_6, \text{SAL}[B_2]} \wedge f_{R_6, \text{SAL}[B_3]}$$

ya que tanto B_2 como B_3 son predecesores de B_4 , el encabezado de R_4 . Esta función de transferencia está compuesta con la función de transferencia $f_{R_4, \text{SAL}[B_4]}$ para obtener la función deseada $f_{R_6, \text{SAL}[B_4]}$. Por ejemplo, observe que d_3 no se elimina en esta función de transferencia, ya que el camino $B_2 \rightarrow B_4$ no define de nuevo a la variable a .

Ahora, considere la región de ciclo R_7 . Sólo contiene una subregión R_6 , que representa al cuerpo de su ciclo. Como sólo hay una arista posterior, $B_4 \rightarrow B_2$ que va al encabezado de R_6 , la función de transferencia que representa la ejecución del cuerpo del ciclo 0 o más veces es sólo $f_{R_6, \text{SAL}[B_4]}^*$: el conjunto *gen* es $\{d_4, d_5, d_6\}$ y el conjunto *eliminar* es \emptyset . Hay dos salidas de la región R_7 , los bloques B_3 y B_4 . En consecuencia, esta función de transferencia está compuesta con cada una de las funciones de transferencia de R_6 para obtener las funciones correspondientes de transferencia de R_7 . Por ejemplo, observe cómo d_6 está en el conjunto *gen* para f_{R_7, R_3} debido a caminos como $B_2 \rightarrow B_4 \rightarrow B_2 \rightarrow B_3$, o inclusive $B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2 \rightarrow B_3$.

Por último, considere a R_8 , el grafo de flujo completo. Sus subregiones son R_1, R_7 y R_5 , que consideraremos en ese orden topológico. Como antes, la función de transferencia $f_{R_8, \text{ENT}[B_1]}$ es simplemente la función de identidad, y la función de transferencia $f_{R_8, \text{SAL}[B_1]}$ es sólo $f_{R_1, \text{SAL}[B_1]}$, que a su vez es f_{B_1} .

El encabezado de R_7 , que es B_2 , sólo tiene un predecesor, B_1 , por lo que la función de transferencia a su entrada es simplemente la función de transferencia que sale de B_1 en la región R_8 . Componemos $f_{R_8, \text{SAL}[B_1]}$ con las funciones de transferencia a las salidas de B_3 y B_4 dentro de R_7 , para obtener sus funciones correspondientes de transferencia dentro de R_8 . Por último, consideraremos a R_5 . Su encabezado, B_5 , tiene dos predecesores dentro de R_8 , es decir R_3 y R_4 . Por lo tanto, calculamos $f_{R_8, \text{SAL}[B_3]} \wedge f_{R_8, \text{SAL}[B_4]}$ para obtener $f_{R_8, \text{ENT}[B_5]}$. Como la función de transferencia del bloque B_5 es la función de identidad, $f_{R_8, \text{SAL}[B_5]} = f_{R_8, \text{ENT}[B_5]}$.

El paso 3 calcula las definiciones de alcance actuales de las funciones de transferencia. En el paso 3(a), $\text{ENT}[R_8] = \emptyset$, ya que no hay definiciones de alcance al inicio del programa. La figura 9.52 muestra cómo el paso 3(b) calcula el resto de los valores del flujo de datos. El paso empieza con las subregiones de R_8 . Como se ha calculado la función de transferencia desde el

¹¹ Hablando en sentido estricto, queremos decir $f_{R_6, \text{ENT}[R_2]}$, pero cuando una región como R_2 es un solo bloque, a menudo es más claro si utilizamos el nombre del bloque en vez del nombre de la región en este contexto.

inicio de R_8 hasta el inicio de cada una de sus subregiones, una sola aplicación de la función de transferencia encuentra el valor del flujo de datos al inicio de cada subregión. Repetimos los pasos hasta obtener los valores del flujo de datos de las regiones hoja, que sólo son los bloques básicos individuales. Observe que los valores del flujo de datos que se muestran en la figura 9.52 son exactamente lo que obtendríamos si aplicáramos el análisis de flujo de datos iterativo al mismo grafo de flujo, como desde luego debe ser el caso. \square

$$\begin{aligned}
 \text{ENT}[R_8] &= \emptyset \\
 \text{ENT}[R_1] &= f_{R_8}, \text{ENT}[R_1](\text{ENT}[R_8]) = \emptyset \\
 \text{ENT}[R_7] &= f_{R_8}, \text{ENT}[R_7](\text{ENT}[R_8]) = \{d_1, d_2, d_3\} \\
 \text{ENT}[R_5] &= f_{R_8}, \text{ENT}[R_5](\text{ENT}[R_8]) = \{d_2, d_3, d_4, d_5, d_6\} \\
 \text{ENT}[R_6] &= f_{R_7}, \text{ENT}[R_6](\text{ENT}[R_7]) = \{d_1, d_2, d_3, d_4, d_5, d_6\} \\
 \text{ENT}[R_4] &= f_{R_6}, \text{ENT}[R_4](\text{ENT}[R_6]) = \{d_2, d_3, d_4, d_5, d_6\} \\
 \text{ENT}[R_3] &= f_{R_6}, \text{ENT}[R_3](\text{ENT}[R_6]) = \{d_2, d_3, d_4, d_5, d_6\} \\
 \text{ENT}[R_2] &= f_{R_6}, \text{ENT}[R_2](\text{ENT}[R_6]) = \{d_1, d_2, d_3, d_4, d_5, d_6\}
 \end{aligned}$$

Figura 9.52: Pasos finales del análisis de flujo basado en regiones

9.7.6 Manejo de grafos de flujo no reducibles

Si se espera que los grafos de flujo no reducibles sean comunes para los programas que se van a procesar por un compilador o cualquier otro software de procesamiento de programas, entonces recomendamos el uso de un método iterativo en vez de uno basado en jerarquías, para el análisis de flujos de datos. No obstante, si sólo necesitamos prepararnos para el grafo de flujo no reducible ocasional, entonces la siguiente técnica de “división de nodos” es adecuada.

Construya regiones desde los ciclos naturales hasta la mayor extensión posible. Si el grafo de flujo no es reducible, encontraremos que el grafo resultante de regiones tiene ciclos, pero no aristas posteriores, por lo que no podemos analizar más el grafo. En la figura 9.53(a) se sugiere una situación ordinaria, la cual tiene la misma estructura que el grafo de flujo no reducible de la figura 9.45, pero los nodos en la figura 9.53 pueden de hecho ser regiones complejas, como lo sugieren los nodos más pequeños en su interior.

Elegimos cierta región R que tenga más de un predecesor y que no sea el encabezado del grafo de flujo completo. Si R tiene k predecesores, haga k copias del grafo de flujo R completo, y conecte cada predecesor del encabezado de R a una copia distinta de R . Recuerde que sólo el encabezado de una región podría llegar a tener un predecesor fuera de esa región. Resulta que, aun cuando no lo demostraremos, dicha división de nodos es una reducción de cuando menos uno en el número de regiones, después de identificar aristas nuevas posteriores y construir sus regiones. El grafo resultante puede incluso no ser reducible, pero mediante la acción de alternar una fase de división con una fase en donde se identifican los nuevos ciclos naturales y se colapsan con regiones, en un momento dado nos quedamos con una sola región; es decir, el grafo de flujo se ha reducido.

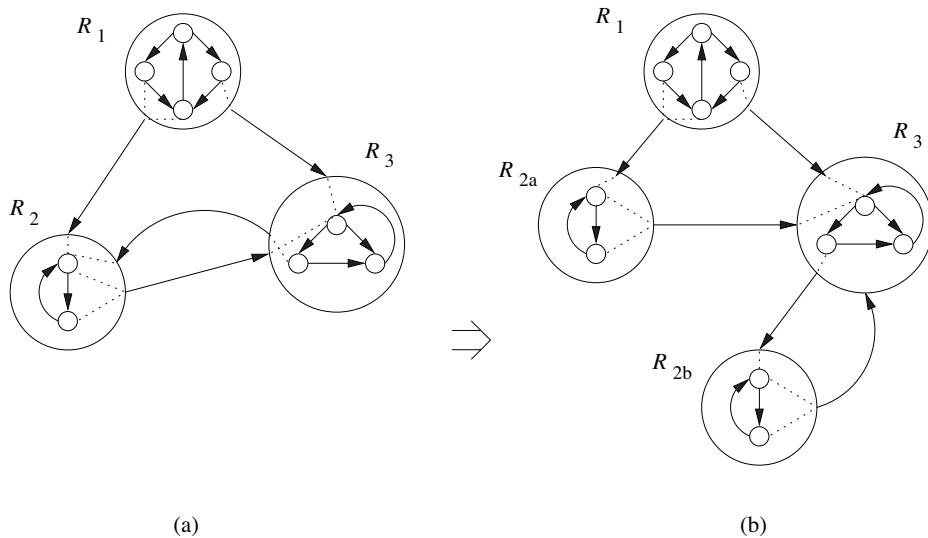


Figura 9.53: Duplicar una región para hacer que un grafo de flujo no reducible pueda reducirse

Ejemplo 9.55: La división que se muestra en la figura 9.53(b) ha convertido la arista $R_{2b} \rightarrow R_3$ en una arista posterior, ya que R_3 ahora domina a R_{2b} . Por lo tanto, estas dos regiones deben combinarse en una sola. Las tres regiones resultantes (R_1 , R_{2a} y la nueva región) forman un grafo acíclico y, por lo tanto, pueden combinarse en una sola región de cuerpo. Por ende, hemos reducido el grafo de flujo completo a una sola región. En general, puede ser necesario realizar divisiones adicionales y, en el peor de los casos, el número total de bloques básicos podría volverse exponencial en el número de bloques en el grafo de flujo original. \square

También debemos pensar acerca de cómo se relaciona el resultado del análisis del flujo de datos en el grafo de flujo dividido con la respuesta que deseamos para el grafo de flujo original. Hay dos métodos que podríamos considerar.

1. Dividir las regiones puede ser benéfico para el proceso de optimización, y sólo debemos modificar el grafo de flujo para tener copias de ciertos bloques. Como sólo se entra a cada bloque duplicado a lo largo de un subconjunto de los caminos que llegaron al original, los valores del flujo de datos en estos bloques duplicados tenderán a contener más información específica de la que estaba disponible en el original. Por ejemplo, pueden llegar menos definiciones a cada uno de los bloques duplicados que al bloque original.
2. Si deseamos retener el grafo de flujo original, sin división, entonces después de analizar el grafo de flujo dividido, analizamos cada bloque B dividido, junto con su conjunto correspondiente de bloques B_1, B_2, \dots, B_k . Podemos calcular $\text{ENT}[B] = \text{ENT}[B_1] \wedge \text{ENT}[B_2] \wedge \dots \wedge \text{ENT}[B_k]$, y de manera similar para los conjuntos SAL.

9.7.7 Ejercicios para la sección 9.7

Ejercicio 9.7.1: Para el grafo de flujo de la figura 9.10 (vea los ejercicios de la sección 9.1):

- i. Encuentre todas las regiones posibles. Sin embargo, puede omitir de la lista las regiones que consistan en un solo nodo y sin aristas.
- ii. Proporcione el conjunto de regiones anidadas construidas por el Algoritmo 9.52.
- iii. Proporcione una reducción T_1-T_2 del grafo de flujo, según la descripción en el recuadro “Cuál es el origen de la ‘reducibilidad’” en la sección 9.7.3.

Ejercicio 9.7.2: Repita el ejercicio 9.7.1 con los siguientes grafos de flujo:

- a) Figura 9.3.
- b) Figura 8.9.
- c) Su grafo de flujo del ejercicio 8.4.1.
- d) Su grafo de flujo del ejercicio 8.4.2.

Ejercicio 9.7.3: Demuestre que todo ciclo natural es una región.

!! Ejercicio 9.7.4: Muestre que un grafo de flujo es reducible si, y sólo si puede transformarse en un nodo individual mediante:

- a) Las operaciones T_1 y T_2 descritas en el cuadro de la sección 9.7.3.
- b) La definición de región introducida en la sección 9.7.3.

! Ejercicio 9.7.5: Muestre que al aplicar la división de nodos a un grafo de flujo no reducible, y después realizar la reducción T_1-T_2 sobre el grafo dividido resultante, siempre terminamos con menos nodos de los que había al empezar.

! Ejercicio 9.7.6: ¿Qué ocurre si aplica la división de nodos y la reducción T_1-T_2 de manera alternativa, para reducir un grafo dirigido completo de n nodos?

9.8 Análisis simbólico

Vamos a usar el análisis simbólico en esta sección para ilustrar el uso del análisis basado en regiones. En este análisis, rastreamos los valores de las variables en programas de manera simbólica, como expresiones de variables de entrada y otras variables, a las cuales llamaremos *variables de referencia*. Al expresar las variables en términos del mismo conjunto de variables de referencia se sacan sus relaciones. El análisis simbólico puede usarse para varios propósitos como la optimización, la paralelización y análisis para la comprensión de los programas.

```

1) x = input();
2) y = x-1;
3) z = y-1;
4) A[x] = 10;
5) A[y] = 11;
6) if (z > x)
7)     z = x;

```

Figura 9.54: Un programa de ejemplo que motiva el análisis simbólico

Ejemplo 9.56: Considere el programa simple de ejemplo de la figura 9.54. Aquí, utilizamos x como la única variable de referencia. El análisis simbólico encontrará que y tiene el valor $x - 1$ y z tiene el valor $x - 2$ después de sus respectivas instrucciones de asignación en las líneas (2) y (3). Por ejemplo, esta información es útil para determinar que las dos asignaciones en las líneas (4) y (5) escriben en distintas ubicaciones de memoria y, por lo tanto, pueden ejecutarse en paralelo. Además, podemos determinar que la condición $z > x$ nunca es verdadera, con lo cual el optimizador puede eliminar la instrucción condicional en las líneas (6) y (7). \square

9.8.1 Expresiones afines de las variables de referencia

Como no podemos crear expresiones simbólicas concisas y de forma cerrada para todos los valores calculados, elegimos un dominio abstracto y aproximamos los cálculos con las expresiones más precisas dentro del dominio. Ya hemos visto un ejemplo de esta estrategia antes: la propagación de constantes. En la propagación de constantes, nuestro dominio abstracto consiste en las constantes, un símbolo UNDEF si no hemos determinado aún que el valor es una constante, y un símbolo especial NAC que se utiliza cada vez que se descubre que una variable no es una constante.

El análisis simbólico que presentamos aquí expresa los valores como expresiones *afines* de variables de referencia siempre que sea posible. Una expresión es afn con respecto a las variables v_1, v_2, \dots, v_n si puede expresarse como $c_0 + c_1v_1 + \dots + c_nv_n$, en donde c_0, c_1, \dots, c_n son constantes. Dichas expresiones se conocen de manera informal como expresiones lineales. Habiendo en sentido estricto, una expresión afín es lineal sólo si c_0 es cero. Estamos interesados en las expresiones afines, ya que a menudo se utilizan para indexar los arreglos en los ciclos; dicha información es útil para las optimizaciones y la paralelización. En el capítulo 11 hablaremos mucho más acerca de este tema.

Variables de inducción

En vez de utilizar variables del programa como variables de referencia, una expresión afín también puede escribirse en términos de la cuenta de iteraciones a través del ciclo. Las variables cuyos valores pueden expresarse como $c_1i + c_0$, en donde i es la cuenta de iteraciones a través del ciclo circundante más cercano, se conocen como *variables de inducción*.

Ejemplo 9.57: Considere el siguiente fragmento de código:

```
for (m = 10; m < 20; m++)
{ x = m*3; A[x] = 0; }
```

Suponga que introducimos una variable para el ciclo, por decir i , para representar el número de iteraciones ejecutadas. El valor i es 0 en la primera iteración del ciclo, 1 en la segunda, y así en lo sucesivo. Podemos expresar la variable m como una expresión afín de i , es decir $m = i + 10$. La variable x , que es $3m$, toma los valores 30, 33, ..., 57 durante iteraciones sucesivas del ciclo. Por ende, x tiene la expresión afín $x = 30 + 3i$. Concluimos que tanto m como x son variables de inducción de este ciclo. \square

Al expresar las variables como expresiones afines de los índices de un ciclo, la serie de valores que se calculan se hace explícita y permite varias transformaciones. La serie de valores que toma una variable de inducción puede calcularse con sumas, en vez de multiplicaciones. Esta transformación se conoce como “reducción de fuerza”, y se presentó en las secciones 8.7 y 9.1. Por ejemplo, podemos eliminar la multiplicación $x=m*3$ del ciclo del ejemplo 9.57, reescribiendo el ciclo como:

```
x = 27;
for (m = 10; m < 20; m++)
{ x = x+3; A[x] = 0; }
```

Además, observe que las ubicaciones a las que se asigna 0 en ese ciclo, $\&A+30$, $\&A+33, \dots$, $\&A+57$, son también expresiones afines del índice de ciclo. De hecho, esta serie de enteros es sólo una que debe calcularse; no necesitamos a m o a x . El código anterior puede sustituirse tan sólo por:

```
for (x = &A+30; x <= &A+57; x = x+3)
*x = 0;
```

Además de agilizar el cálculo, el análisis simbólico también es útil para la paralelización. Cuando los índices de un arreglo en un ciclo son expresiones afines de los índices del ciclo, podemos razonar acerca de las relaciones de los datos a los que se accede a través de las iteraciones. Por ejemplo, podemos saber que las ubicaciones escritas son distintas en cada iteración y, por lo tanto, todas las iteraciones en el ciclo pueden ejecutarse en paralelo, en distintos procesadores. Dicha información se utiliza en los capítulos 10 y 11 para extraer el paralelismo de los programas secuenciales.

Otras variables de referencia

Si una variable no es una función lineal de las variables de referencia que ya se han elegido, tenemos la opción de tratar su valor como referencia para operaciones futuras. Por ejemplo, considere el siguiente fragmento de código:

```
a = f();
b = a + 10;
c = a + 11;
```

Aunque el valor que contiene a después de la llamada a la función no puede en sí expresarse como una función lineal de cualquier variable de referencia, puede usarse como referencia para las instrucciones siguientes. Por ejemplo, si usamos a como variable de referencia, podemos descubrir que c es un número más grande que b al final del programa.

```

1) a = 0;
2) for (f = 100; f < 200; f++) {
3)     a = a + 1;
4)     b = 10 * a;
5)     c = 0;
6)     for (g = 10; g < 20; g++) {
7)         d = b + c;
8)         c = c + 1;
}
}

```

Figura 9.55: Código fuente para el ejemplo 9.58

Ejemplo 9.58: Nuestro ejemplo para esta sección se basa en el código fuente que se muestra en la figura 9.55. Los ciclos interno y externo son fáciles de comprender, ya que f y g no se modifican, excepto cuando lo requieran los ciclos for. Por ende, es posible sustituir f y g por las variables de referencia i y j , que cuentan el número de iteraciones de los ciclos externo e interno, respectivamente. Es decir, podemos dejar que $f = i + 99$ y $g = j + 9$, y sustituir para f y g durante la ejecución. Al traducir a código intermedio, podemos aprovechar el hecho de que cada ciclo itera por lo menos una vez, para así posponer la prueba para $i \leq 100$ y $j \leq 10$ hasta el final de los ciclos. La figura 9.56 muestra el grafo de flujo para el código de la figura 9.55, después de introducir i y j y tratar los ciclos for como si fueran ciclos repeat.

Resulta que a , b , c y d son todas variables de inducción. Las secuencias de valores asignados a las variables en cada línea del código se muestran en la figura 9.57. Como veremos, es posible descubrir las expresiones afines para esas variables, en términos de las variables de referencia i y j . Es decir, en la línea (4) $a = i$, en la línea (7) $d = 10i + j - 1$, y en la línea (8), $c = j$. \square

9.8.2 Formulación del problema de flujo de datos

Este análisis encuentra las expresiones afines de las variables de referencia introducidas (1) para contar el número de iteraciones ejecutadas en cada ciclo, y (2) para guardar valores a la entrada de las regiones, cuando sea necesario. Este análisis también encuentra variables de inducción, invariantes de ciclo, así como constantes, como expresiones afines degeneradas. Tenga en cuenta que este análisis no puede encontrar todas las constantes, ya que sólo rastrea las expresiones afines de las variables de referencia.

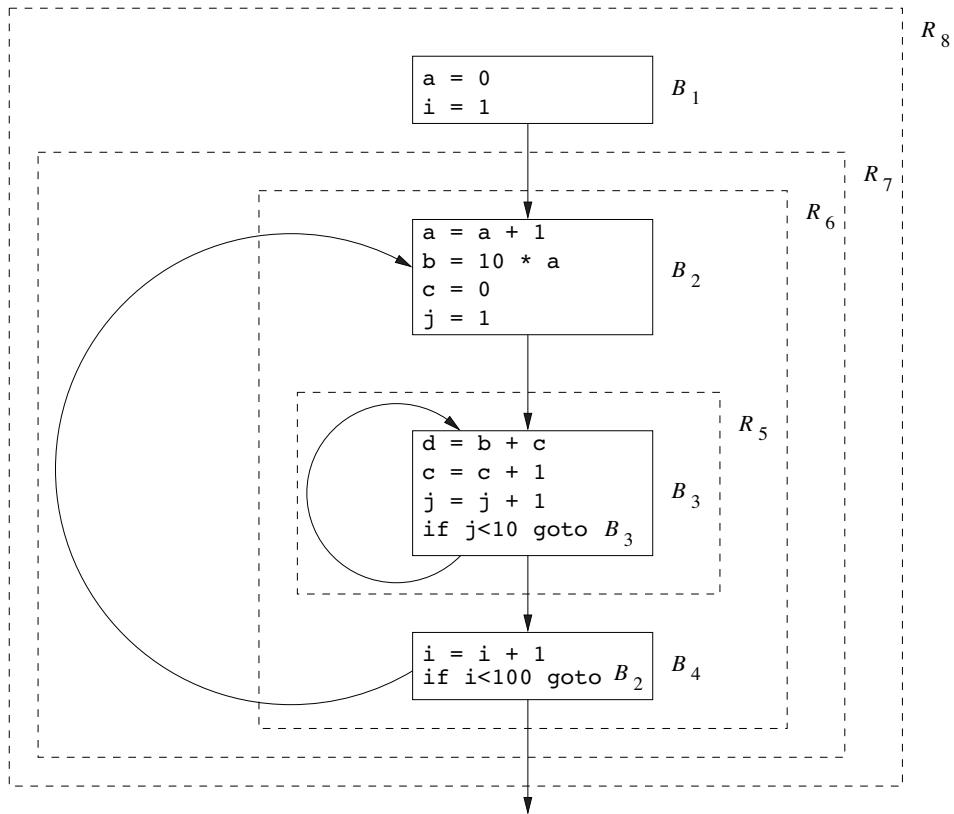


Figura 9.56: Grafo de flujo y su jerarquía de regiones para el ejemplo 9.58

Valores de flujo de datos: mapas simbólicos

El dominio de los valores de flujo de datos para este análisis es el de los mapas simbólicos, que son funciones que asignan cada variable en el programa a un valor. El valor puede ser una función afín de valores de referencia, o el símbolo especial NAA para representar a una expresión no afín. Si sólo hay una variable, el valor inferior del semi-lattice es un mapa que envía la variable a NAA. El semi-lattice para n variables es simplemente el producto de los semi-lattices individuales. Utilizamos m_{NAA} para denotar la parte inferior del semi-lattice, que asigna todas las variables a NAA. Podemos definir el mapa simbólico que envía todas las variables a un valor desconocido para que sea el valor superior del flujo de datos, como hicimos con la propagación de constantes. Sin embargo, no necesitamos valores superiores en el análisis basado en regiones.

Ejemplo 9.59: Los mapas simbólicos asociados con cada bloque para el código del ejemplo 9.58 se muestran en la figura 9.58. Más adelante veremos cómo se descubren estos mapas; son el resultado de realizar un análisis de flujo de datos basado en regiones en el grafo de flujo de la figura 9.56.

línea	var	$i = 1$ $j = 1, \dots, 10$	$i = 2$ $j = 1, \dots, 10$	$1 \leq i \leq 100$ $j = 1, \dots, 10$	$i = 100$ $j = 1, \dots, 10$
2	a	1	2	i	100
3	b	10	20	$10i$	1000
7	d	$10, \dots, 19$	$20, \dots, 29$	$10i, \dots, 10i + 9$	$1000, \dots, 1009$
8	c	$1, \dots, 10$	$1, \dots, 10$	$1 \dots 10$	$1 \dots 10$

Figura 9.57: Secuencia de valores vistos en puntos del programa del ejemplo 9.58

m	$m(a)$	$m(b)$	$m(c)$	$m(d)$
$\text{ENT}[B_1]$	NAA	NAA	NAA	NAA
$\text{SAL}[B_1]$	0	NAA	NAA	NAA
$\text{ENT}[B_2]$	$i - 1$	NAA	NAA	NAA
$\text{SAL}[B_2]$	i	$10i$	0	NAA
$\text{ENT}[B_3]$	i	$10i$	$j - 1$	NAA
$\text{SAL}[B_3]$	i	$10i$	j	$10i + j - 1$
$\text{ENT}[B_4]$	i	$10i$	j	$10i + j - 1$
$\text{SAL}[B_4]$	$i - 1$	$10i - 10$	j	$10i + j - 11$

Figura 9.58: Mapas simbólicos del programa en el ejemplo 9.58

El mapa simbólico asociado con la entrada del programa es m_{NAA} . A la salida de B_1 , el valor de a se establece a 0. Al entrar al bloque B_2 , a tiene el valor 0 en la primera iteración y se incrementa en uno en cada iteración siguiente del ciclo externo. Por ende, a tiene el valor $i - 1$ al entrar a la i -ésima iteración y el valor i al final. El mapa simbólico a la entrada de B_2 asigna las variables b , c , d a NAA, ya que estas variables tienen valores desconocidos al entrar al ciclo interno. Sus valores dependen del número de iteraciones del ciclo externo, hasta ahora. El mapa simbólico al salir de B_2 refleja las instrucciones de asignación a a , b y c en ese bloque. El resto de los mapas simbólicos se puede deducir de una manera similar. Una vez establecida la validez de los mapas en la figura 9.58, podemos sustituir cada una de las asignaciones a a , b , c y d en la figura 9.55 por las expresiones afines apropiadas. Es decir, podemos sustituir la figura 9.55 por el código de la figura 9.59. \square

Función de transferencia de una instrucción

Las funciones de transferencia en este problema de flujo de datos envían mapas simbólicos a los mapas simbólicos. Para calcular la función de transferencia de una instrucción de asignación, interpretamos la semántica de la instrucción y determinamos si las variables asignadas pueden expresarse como una expresión afín de los valores a la derecha de la asignación. Los valores de todas las demás variables permanecen sin cambio.

```

1) a = 0;
2) for (i = 1; i <= 100; i++) {
3)     a = i;
4)     b = 10*i;
5)     c = 0;
6)     for (j = 1; j <= 10; j++) {
7)         d = 10*i + j -1;
8)         c = j;
}
}

```

Figura 9.59: El código de la figura 9.55, con las asignaciones sustituidas por expresiones afines de las variables de referencia i y j

Precauciones para las funciones de transferencia en los mapas de valores

Una sutileza en la forma que definimos las funciones de transferencia en los mapas simbólicos es que tenemos opciones en cuanto a la forma en que se expresan los efectos de un cálculo. Si m es el mapa para la entrada de una función de transferencia, $m(x)$ es en realidad sólo “cualquier valor que la variable x tenga al momento de entrar”. Nos esforzamos en expresar el resultado de la función de transferencia como una expresión afín de los valores que se describen mediante el mapa de entrada.

Hay que observar la interpretación apropiada de expresiones como $f(m)(x)$, en donde f es una función de transferencia, m un mapa, y x una variable. Al igual que la convención en matemáticas, aplicamos las funciones a partir de la izquierda, lo cual significa que primero calculamos $f(m)$, que es un mapa. Como un mapa es una función, podemos entonces aplicarla a una variable x para producir un valor.

La función de transferencia de la instrucción s , denotada por f_s , se define de la siguiente manera:

1. Si s no es una instrucción de asignación, entonces f_s es la función de identidad.
2. Si s es una instrucción de asignación para la variable x , entonces:

$$f_s(m)(x) = \begin{cases} m(v) & \text{para todas las variables } v \neq x \\ c_0 + c_1m(y) + c_2m(z) & \text{si } x \text{ es asignada } c_0 + c_1y + c_2z, \\ & (c_1 = 0, \text{ o } m(y) \neq \text{NAA}), \text{ y} \\ & (c_2 = 0, \text{ o } m(z) \neq \text{NAA}) \\ \text{NAA} & \text{en cualquier otro caso.} \end{cases}$$

El objetivo de la expresión $c_0 + c_1m(y) + c_2m(z)$ es representar todas las posibles formas de expresiones que involucren variables arbitrarias y y z , y que puedan aparecer en el lado derecho de una asignación para x , y que proporcionen a x un valor que sea una transformación afín sobre los valores anteriores de las variables. Estas expresiones son: c_0 , $c_0 + y$, $c_0 - y$, $y + z$, $x - y$, $c_1 * y$ y $y/(1/c_1)$. Observe que en muchos casos, uno o más de los valores de c_0 , c_1 y c_2 son 0.

Ejemplo 9.60: Si la asignación es $x=y+z$, entonces $c_0 = 0$ y $c_1 = c_2 = 1$. Si la asignación es $x=y/5$, entonces $c_0 = c_2 = 0$, y $c_1 = 1/5$. \square

Composición de las funciones de transferencia

Para calcular $f_2 \circ f_1$, en donde f_1 y f_2 se definen en términos del mapa de entrada m , sustituimos el valor de $m(v_i)$ en la definición de f_2 con la definición de $f_1(m)(v_i)$. Sustituimos todas las operaciones sobre los valores NAA con NAA. Es decir,

1. Si $f_2(m)(v) = \text{NAA}$, entonces $(f_2 \circ f_1)(m)(v) = \text{NAA}$.
2. Si $f_2(m)(v) = c_0 + \sum_i c_i m(v_i)$, entonces

$$(f_2 \circ f_1)(m)(v) = \begin{cases} \text{NAA,} & \text{si } f_1(m)(v_i) = \text{NAA} \text{ para cierta } i \neq 0, c_i \neq 0 \\ c_0 + \sum_i c_i f_1(m)(v_i) & \text{en cualquier otro caso} \end{cases}$$

Ejemplo 9.61: Las funciones de transferencia de los bloques en el ejemplo 9.58 pueden calcularse al componer las funciones de transferencia de sus instrucciones constituyentes. Estas funciones de transferencia se definen en la figura 9.60. \square

f	$f(m)(a)$	$f(m)(b)$	$f(m)(c)$	$f(m)(d)$
f_{B_1}	0	$m(b)$	$m(c)$	$m(d)$
f_{B_2}	$m(a) + 1$	$10m(a) + 10$	0	$m(d)$
f_{B_3}	$m(a)$	$m(b)$	$m(c) + 1$	$m(b) + m(c)$
f_{B_4}	$m(a)$	$m(b)$	$m(c)$	$m(d)$

Figura 9.60: Funciones de transferencia del ejemplo 9.58

Solución al problema de flujo de datos

Utilizamos la notación $\text{ENT}_{i,j}[B_3]$ y $\text{SAL}_{i,j}[B_3]$ para referirnos a los valores del flujo de datos de entrada y salida del bloque B_3 en la iteración j del ciclo interno, y la iteración i del ciclo externo. Para los demás bloques, usamos $\text{ENT}_i[B_k]$ y $\text{SAL}_i[B_k]$ para referirnos a estos valores en

$$\begin{aligned}
 \text{SAL}[B_k] &= f_B(\text{ENT}[B_k]), \quad \text{para todos los valores de } B_k \\
 \text{SAL}[B_1] &\geq \text{ENT}_1[B_2] \\
 \text{SAL}_i[B_2] &\geq \text{ENT}_{i,1}[B_3], \quad 1 \leq i \leq 10 \\
 \text{SAL}_{i, j-1}[B_3] &\geq \text{ENT}_{i, j}[B_3], \quad 1 \leq i \leq 100, 2 \leq j \leq 10 \\
 \text{SAL}_{i, 10}[B_3] &\geq \text{ENT}_i[B_4], \quad 2 \leq i \leq 100 \\
 \text{SAL}_{i-1}[B_4] &\geq \text{ENT}_i[B_2], \quad 1 \leq i \leq 100
 \end{aligned}$$

Figura 9.61: Restricciones que se satisfacen en cada iteración de los ciclos anidados

la i -ésima iteración del ciclo externo. Además, podemos ver que los mapas simbólicos que se muestran en la figura 9.58 satisfacen las restricciones impuestas por las funciones de transferencia, que se presentan en la figura 9.61.

La primera restricción dice que el mapa de salida de un bloque básico se obtiene aplicando la función de transferencia del bloque al mapa de entrada. El resto de las restricciones dicen que el mapa de salida de un bloque básico debe ser mayor que, o igual al mapa de entrada de un bloque sucesor en la ejecución.

Observe que nuestro algoritmo de flujo de datos iterativo no puede producir la solución anterior, ya que carece del concepto de expresar los valores del flujo de datos en términos del número de iteraciones ejecutadas. El análisis basado en regiones puede utilizarse para encontrar dichas soluciones, como veremos en la siguiente sección.

9.8.3 Análisis simbólico basado en regiones

Podemos extender el análisis basado en regiones descrito en la sección 9.7 para encontrar expresiones de variables en la i -ésima iteración de un ciclo. Un análisis simbólico basado en regiones tiene una pasada de abajo hacia arriba y una pasada de arriba hacia abajo, al igual que otros algoritmos basados en regiones. La pasada ascendente sintetiza el efecto de una región con una función de transferencia que envía un mapa simbólico a la entrada hacia un mapa simbólico de salida en la salida. En la pasada descendente, los valores de los mapas simbólicos se propagan hacia abajo, a las regiones más internas.

La diferencia recae en la forma en que manejamos los ciclos. En la sección 9.7, el efecto de un ciclo se sintetiza con un operador de cerradura. Dado un ciclo con el cuerpo f , su cerradura f^* se define como una reunión infinita de todos los posibles números de aplicaciones de f . Sin embargo, para encontrar variables de inducción debemos determinar si un valor de una variable es una función afín del número de iteraciones ejecutadas hasta el momento. El mapa simbólico debe parametrizarse en base al número de la iteración que se está ejecutando. Además, cada vez que conocemos el número total de iteraciones ejecutadas en un ciclo, podemos usar ese número para encontrar los valores de las variables de inducción después del ciclo. Por ejemplo, en el ejemplo 9.58 afirmamos que a tiene el valor de i después de ejecutar la i -ésima iteración. Como el ciclo tiene 100 iteraciones, el valor de a debe ser 100 al final del ciclo.

En lo que sigue, primero vamos a definir los operadores primitivos: la reunión y la composición de funciones de transferencia para el análisis simbólico. Despues mostraremos cómo usarlos para realizar un análisis basado en regiones de las variables de inducción.

Reunión de funciones de transferencia

Al calcular la reunión de dos funciones, el valor de una variable es NAA, a menos que las dos funciones asignen la variable al mismo valor, y que el valor no sea NAA.

Por ende,

$$(f_1 \wedge f_2)(m)(v) = \begin{cases} f_1(m)(v) & \text{si } f_1(m)(v) = f_2(m)(v) \\ \text{NAA} & \text{en cualquier otro caso} \end{cases}$$

Composiciones de funciones parametrizadas

Para expresar una variable como una función afín de un índice de ciclo, debemos calcular el efecto de componer una función cierto número de veces. Si el efecto de una iteración se sintetiza mediante la función de transferencia f , entonces el efecto de ejecutar i iteraciones, para cierta $i \geq 0$, se denota como f^i . Observe que cuando $i = 0$, $f^i = f^0 = I$, la función de identidad.

Las variables en el programa se dividen en tres categorías:

1. Si $f(m)(x) = m(x) + c$, en donde c es una constante, entonces $f^i(m)(x) = m(x) + ci$ para cada valor de $i \geq 0$. Decimos que x es una *variable de inducción básica* del ciclo cuyo cuerpo se representa mediante la función de transferencia f .
2. Si $f(m)(x) = m(x)$, entonces $f^i(m)(x) = m(x)$ para todas las $i \geq 0$. La variable x no se modifica y permanece sin cambios al final de cualquier número de iteraciones a través del ciclo con la función de transferencia f . Decimos que x es una *constante simbólica* en el ciclo.
3. Si $f(m)(x) = c_0 + c_1 m(x_1) + \cdots + c_n m(x_n)$, en donde cada x_k es una variable de inducción básica o una constante simbólica, entonces para $i > 0$,

$$f^i(m)(x) = c_0 + c_1 f^i(m)(x_1) + \cdots + c_n f^i(m)(x_n).$$

Decimos que x es también una variable de inducción, aunque no básica. Observe que la fórmula anterior no se aplica si $i = 0$.

4. En todos los demás casos, $f^i(m)(x) = \text{NAA}$.

Para encontrar el efecto de ejecutar un número fijo de iteraciones, simplemente sustituimos i en la ecuación anterior por ese número. En el caso en el que se desconoce el número de iteraciones, el valor al inicio de la última iteración se da mediante f^* . En este caso, las únicas variables cuyos valores pueden aún expresarse en la forma afín son las variables invariantes de ciclo.

$$f^*(m)(v) = \begin{cases} m(v) & \text{si } f(m)(v) = m(v) \\ \text{NAA} & \text{en cualquier otro caso} \end{cases}$$

Ejemplo 9.62: Para el ciclo más interno en el ejemplo 9.58, el efecto de ejecutar i iteraciones, $i > 0$, se sintetiza mediante $f^i B_3$. De la definición de f_{B_3} podemos ver que a y b son constantes simbólicas, c es una variable de inducción básica debido a que se incrementa por cada iteración,

d es una variable de inducción ya que es una función afín la constante simbólica b y la variable de inducción básica c . Por ende,

$$f_{B_3}^i(m)(v) = \begin{cases} m(a) & \text{si } v = a \\ m(b) & \text{si } v = b \\ m(c) + i & \text{si } v = c \\ m(b) + m(c) + i & \text{si } v = d. \end{cases}$$

Si no podemos saber cuántas veces iteró el ciclo del bloque B_3 , entonces no podríamos usar f^i y tendríamos que usar f^* para expresar las condiciones al final del ciclo. En este caso, tendríamos:

$$f_{B_3}^*(m)(v) = \begin{cases} m(a) & \text{si } v = a \\ m(b) & \text{si } v = b \\ \text{NAA} & \text{si } v = c \\ \text{NAA} & \text{si } v = d. \end{cases}$$

□

Un algoritmo basado en regiones

Algoritmo 9.63: Análisis simbólico basado en regiones.

ENTRADA: Un grafo de flujo reducible G .

SALIDA: Los mapas simbólicos $\text{ENT}[B]$ para cada bloque B de G .

MÉTODO: Realizamos las siguientes modificaciones al Algoritmo 9.53.

1. Cambiamos la forma en que construimos la función de transferencia para una región de ciclo. En el algoritmo original usamos la función de transferencia $f_{R, \text{ENT}[S]}$ para asignar el mapa simbólico a la entrada de la región de ciclo R a un mapa simbólico en la entrada del cuerpo de ciclo S , después de ejecutar un número desconocido de iteraciones. Se define como el cierre de la función de transferencia que representa a todos los caminos que conducen de vuelta a la entrada del ciclo, como se muestra en la figura 9.50(b). Aquí definimos $f_{R, i, \text{ENT}[S]}$ para representar el efecto de la ejecución, desde el inicio de la región de ciclo, hasta la entrada de la i -ésima iteración. Por lo tanto,

$$f_{R, i, \text{ENT}[S]} = \left(\bigwedge_{\text{predecesores } B \text{ en } R \text{ del encabezado de } S} f_{S, \text{SAL}[B]} \right)^{i-1}$$

2. Si el número de iteraciones de una región es conocido, el resumen de la región se calcula sustituyendo i por la cuenta actual.
3. En la pasada descendente, calculamos $f_{R, i, \text{ENT}[B]}$ para encontrar el mapa simbólico asociado con la entrada de la i -ésima iteración de un ciclo.

4. En el caso en donde se utiliza el valor de entrada de una variable $m(v)$ del lado derecho de un mapa simbólico en la región R , y $m(v) = \text{NAA}$ al entrar a la región, introducimos una nueva variable de referencia t , agregamos la asignación $t = v$ al inicio de la región R , y todas las referencias de $m(v)$ se sustituyen por t . Si no introdujéramos una variable de referencia en este punto, el valor NAA guardado por v penetraría en los ciclos de más adentro.

□

$$\begin{aligned}
 f_{R_5,j,\text{ENT}[B_3]} &= f_{B_3}^{j-1} \\
 f_{R_5,j,\text{SAL}[B_3]} &= f_{B_3}^j \\
 \\
 f_{R_6,\text{ENT}[B_2]} &= I \\
 f_{R_6,\text{ENT}[R_5]} &= f_{B_2} \\
 f_{R_6,\text{SAL}[B_4]} &= I \circ f_{R_5,10,\text{SAL}[B_3]} \circ f_{B_2} \\
 \\
 f_{R_7,i,\text{ENT}[R_6]} &= f_{R_6,\text{SAL}[B_4]}^{i-1} \\
 f_{R_7,i,\text{SAL}[B_4]} &= f_{R_6,\text{SAL}[B_4]}^i \\
 \\
 f_{R_8,\text{ENT}[B_1]} &= I \\
 f_{R_8,\text{ENT}[R_7]} &= f_{B_1} \\
 f_{R_8,\text{SAL}[B_4]} &= f_{R_7,100,\text{SAL}[B_4]} \circ f_{B_1}
 \end{aligned}$$

Figura 9.62: Relaciones de las funciones de transferencia en la pasada de abajo hacia arriba para el ejemplo 9.58

Ejemplo 9.64: Para el ejemplo 9.58, mostramos cómo se calculan las funciones de transferencia para el programa en la pasada de abajo hacia arriba en la figura 9.62. La región R_5 es el ciclo interno, con el cuerpo B_5 . La función de transferencia que representa el camino que proviene de la entrada de la región R_5 al inicio de la j -ésima iteración, $j \geq 1$, es $f_{B_3}^{j-1}$. La función de transferencia que representa el camino al final de la j -ésima iteración, $j \geq 1$, es $f_{B_3}^j$.

La región R_6 consiste en los bloques B_2 y B_4 , con la región de ciclo R_5 en medio. Las funciones de transferencia que provienen de la entrada de B_2 y R_5 pueden calcularse de la misma forma que en el algoritmo original. La función de transferencia $f_{R_6,\text{SAL}[B_3]}$ representa a la composición del bloque B_2 y toda la ejecución del ciclo interior, ya que f_{B_4} es la función de identidad. Como se sabe que el ciclo interno itera 10 veces, podemos sustituir j por 10 para sintetizar con precisión el efecto del ciclo interno. El resto de las funciones de transferencia puede calcularse de manera similar. Las funciones de transferencia actuales que se calculan se muestran en la figura 9.63.

f	$f(m)(a)$	$f(m)(b)$	$f(m)(c)$	$f(m)(d)$
$f_{R_5,j, \text{ENT}[B_3]}$	$m(a)$	$m(b)$	$m(c) + j - 1$	NAA
$f_{R_5,j, \text{SAL}[B_3]}$	$m(a)$	$m(b)$	$m(c) + j$	$m(b) + m(c) + j - 1$
$f_{R_6, \text{ENT}[B_2]}$	$m(a)$	$m(b)$	$m(c)$	$m(d)$
$f_{R_6, \text{ENT}[R_5]}$	$m(a) + 1$	$10m(a) + 10$	0	$m(d)$
$f_{R_6, \text{SAL}[B_4]}$	$m(a) + 1$	$10m(a) + 10$	10	$10m(a) + 9$
$f_{R_7,i, \text{ENT}[R_6]}$	$m(a) + i - 1$	NAA	NAA	NAA
$f_{R_7,i, \text{SAL}[B_4]}$	$m(a) + i$	$10m(a) + 10i$	10	$10m(a) + 10i + 9$
$f_{R_8, \text{ENT}[B_1]}$	$m(a)$	$m(b)$	$m(c)$	$m(d)$
$f_{R_8, \text{ENT}[R_7]}$	0	$m(b)$	$m(c)$	$m(d)$
$f_{R_8, \text{SAL}[B_4]}$	100	1000	10	1009

Figura 9.63: Funciones de transferencia calculadas en la pasada de abajo hacia arriba para el ejemplo 9.58

El mapa simbólico a la entrada del programa es simplemente m_{NAA} . Utilizamos la pasada de arriba hacia abajo para calcular el mapa simbólico a la entrada hacia regiones anidadas en forma sucesiva, hasta encontrar todos los mapas simbólicos para cada bloque básico. Empezamos por calcular los valores del flujo de datos para el bloque B_1 en la región R_8 :

$$\begin{aligned} \text{ENT}[B_1] &= m_{\text{NAA}} \\ \text{SAL}[B_1] &= f_{B_1}(\text{ENT}[B_1]) \end{aligned}$$

Si descendemos hasta las regiones R_7 y R_6 , obtenemos:

$$\begin{aligned} \text{ENT}_i[B_2] &= f_{R_7, i, \text{ENT}[R_6]}(\text{SAL}[B_1]) \\ \text{SAL}_i[B_2] &= f_{B_2}(\text{ENT}_i[B_2]) \end{aligned}$$

Por último, en la región R_5 , obtenemos:

$$\begin{aligned} \text{ENT}_{i,j}[B_3] &= f_{R_5, \text{ENT}[B_3]}(\text{SAL}_i[B_2]) \\ \text{SAL}_{i,j}[B_3] &= f_{B_3}(\text{ENT}_{i,j}[B_3]) \end{aligned}$$

No es de sorprender que estas ecuaciones produzcan los resultados que mostramos en la figura 9.58. \square

El ejemplo 9.58 muestra un programa simple, en el cual cada variable utilizada en el mapa simbólico tiene una expresión afín. Utilizamos el ejemplo 9.65 para ilustrar por qué y cómo introducimos las variables de referencia en el Algoritmo 9.63.

```

1) for (i = 1; i < n; i++) {
2)     a = input();
3)     for (j = 1; j < 10; j++) {
4)         a = a - 1;
5)         b = j + a;
6)         a = a + 1;
}

```

(a) Un ciclo en donde a fluctúa.

```

for (i = 1; i < n; i++) {
    a = input();
    t = a;
    for (j = 1; j < 10; j++) {
        a = t - 1;
        b = t - 1 + j;
        a = t;
    }
}

```

(b) Una variable de referencia t hace a una b una variable de inducción.

Figura 9.64: La necesidad de introducir variables de referencia

Ejemplo 9.65: Considere el ejemplo simple en la figura 9.64(a). Hagamos que f_j sea la función de transferencia que sintetiza el efecto de ejecutar j iteraciones del ciclo interno. Aun cuando el valor de a puede fluctuar durante la ejecución del ciclo, podemos ver que b es una variable de inducción basada en el valor de a al entrar al ciclo; es decir, $f_j(m)(b) = m(a) - 1 + j$. Como a a se le asigna un valor de entrada, el mapa simbólico al entrara al ciclo interno asigna a a NAA. Introducimos una nueva variable de referencia t para guardar el valor de a al entrar, y realizamos las sustituciones como en la figura 9.64(b). \square

9.8.4 Ejercicios para la sección 9.8

Ejercicio 9.8.1: Para el grafo de flujo de la figura 9.10 (vea los ejercicios de la sección 9.1), proporcione las funciones de transferencia para:

- a) El bloque B_2 .
- b) El bloque B_4 .
- c) El bloque B_5 .

Ejercicio 9.8.2: Considere el ciclo interno de la figura 9.10, que consiste en los bloques B_3 y B_4 . Si i representa el número de veces que se recorre el ciclo, y f es la función de transferencia para el cuerpo del ciclo (es decir, excluyendo la arista de B_4 a B_3) que proviene de la entrada del ciclo (es decir, el inicio de B_3) hasta la salida de B_4 , entonces ¿qué es f^i ? Recuerde que f toma como argumento un mapa m , y m asigna un valor a cada una de las variables a , b , d y e . Denotamos estos valores $m(a)$, y así en lo sucesivo, aunque no conocemos sus valores.

! Ejercicio 9.8.3: Ahora considere el ciclo externo de la figura 9.10, que consiste en los bloques B_2 , B_3 , B_4 y B_5 . Hagamos que g sea la función de transferencia para el cuerpo del ciclo, desde la entrada del ciclo en B_2 hasta su salida en B_5 . Hagamos que i mida el número de iteraciones del ciclo interno de B_3 y B_4 (no podemos saber cuál cuenta de iteraciones), y hagamos que j mida el número de iteraciones del ciclo externo (que tampoco podemos conocer). ¿Qué es g^j ?

9.9 Resumen del capítulo 9

- ◆ *Subexpresiones comunes globales:* Una de las optimizaciones importantes es la de encontrar cálculos de la misma expresión en dos bloques básicos distintos. Si una va antes que la otra, podemos almacenar el resultado la primera vez que se calcula y usar el resultado almacenado en las ocurrencias siguientes.
- ◆ *Propagación de copias:* Una instrucción de copia, $u = v$, asigna una variable v a otra variable, u . En ciertas circunstancias, podemos sustituir todos los usos de u por v , con lo cual se elimina tanto la asignación como u .
- ◆ *Movimiento de código:* Otra optimización es mover un cálculo fuera del ciclo en el que aparece. Este cambio sólo es correcto si el cálculo produce el mismo valor cada vez que se recorre el ciclo.
- ◆ *Variables de inducción:* Muchos ciclos tienen variables de inducción, variables que toman una secuencia lineal de valores cada vez que se recorre el ciclo. Algunas de éstas se utilizan sólo para contar iteraciones, y a menudo pueden eliminarse, con lo cual se reduce el tiempo que se requiere para recorrer el ciclo.
- ◆ *Ánálisis del flujo de datos:* Un esquema de análisis de flujo de datos define un valor en cada punto en el programa. Las instrucciones del programa tienen funciones de transferencia asociadas que relacionan el valor antes de la instrucción con el valor después de la misma. Las instrucciones con más de un predecesor deben definir su valor mediante la combinación de los valores en los predecesores, usando un operador de reunión (o confluencia).
- ◆ *Ánálisis del flujo de datos sobre bloques básicos:* Debido a que la propagación de los valores de un flujo de datos dentro de un bloque es en general bastante simple, es común que las ecuaciones de flujo de datos se preparen para tener dos variables para cada bloque, de nombre ENT y SAL, que representan los valores del flujo de datos al inicio y al final del bloque, respectivamente. Las funciones de transferencia para las instrucciones en un blo-

que son compuestas con el fin obtener la función de transferencia para el bloque como un todo.

- ◆ *Definiciones de alcance:* El marco de trabajo del flujo de datos de las definiciones de alcance tiene valores que son conjuntos de instrucciones en el programa, los cuales definen valores para una o más variables. La función de transferencia para un bloque elimina las definiciones de variables que en definitiva se definen de nuevo en el bloque, y agrega (“genera”) esas definiciones de variables que ocurren dentro del bloque. El operador de confluencia es la unión, ya que las definiciones llegan a un punto si alcanzan cualquier predecesor de ese punto.
- ◆ *Variables vivas:* Otro marco de trabajo del flujo de datos importante calcula las variables que están vivas (se utilizarán antes de la redefinición) en cada punto. El marco de trabajo es similar a las definiciones de alcance, sólo que la función de transferencia trabaja al revés. Una variable está viva al inicio de un bloque si se utiliza antes de la definición en el bloque, o si está viva al final del bloque y no se define de nuevo en el bloque.
- ◆ *Expresiones disponibles:* Para descubrir las subexpresiones comunes globales, determinamos las expresiones disponibles en cada punto (las expresiones que se habían calculado y ninguno de los argumentos de ellas se definieron de nuevo después del último cálculo). El marco de trabajo del flujo de datos es similar a las definiciones de alcance, pero el operador de confluencia es la intersección, en vez de la unión.
- ◆ *Abstracción de los problemas de flujo de datos:* Los problemas comunes de flujo de datos, como los que ya hemos mencionado, pueden expresarse en una estructura matemática común. Los valores son miembros de un semi-lattice, cuya reunión es el operador de confluencia. Las funciones de transferencia asignan elementos del lattice a elementos del lattice. El conjunto de funciones de transferencia permitidas debe cerrarse bajo la composición e incluir la función de identidad.
- ◆ *Marcos de trabajo monótonos:* Un semi-lattice tiene una relación \leq definida por $a \leq b$, si y sólo si $a \wedge b = a$. Los marcos de trabajo monótonos tienen la propiedad de que cada función de transferencia preserva la relación \leq ; es decir, $a \leq b$ implica que $f(a) \leq f(b)$, para todos los elementos a y b del lattice y la función de transferencia f .
- ◆ *Marcos de trabajo distributivos:* Estos marcos de trabajo cumplen con la condición de que $f(a \wedge b) = f(a) \wedge f(b)$, para todos los elementos a y b del lattice y la función de transferencia f . Puede mostrarse que la condición distributiva implica la condición de monotonía.
- ◆ *Solución iterativa a los marcos de trabajo abstractos:* Todos los marcos de flujo de datos monótonos pueden resolverse mediante un algoritmo iterativo, en el que los valores ENT y SAL para cada bloque se inicializan de manera apropiada (dependiendo del marco de trabajo), y se calculan en forma repetida nuevos valores para estas variables, al aplicar las operaciones de transferencia y confluencia. Esta solución siempre es segura (las optimizaciones que sugiere no cambiarán lo que hace el programa), pero sin duda la solución será la mejor posible, sólo si el marco de trabajo es distributivo.

- ◆ *El marco de trabajo de propagación de constantes:* Mientras que los marcos de trabajo básicos, como las definiciones de alcance, son distributivos, hay también marcos de trabajo interesantes que son monótonos pero no distributivos. Uno implica la propagación de constantes mediante el uso de un semi-lattice, cuyos elementos son asignaciones de las variables del programa a constantes, más dos valores especiales que representan “no hay información” y “en definitiva no es constante”.
- ◆ *Eliminación de redundancia parcial:* Muchas optimizaciones útiles, como el movimiento de código y la eliminación de subexpresiones comunes globales, pueden generalizarse a un solo problema, conocido como eliminación de redundancia parcial. Las expresiones que son necesarias, pero que están disponibles sólo a lo largo de algunos de los caminos hacia un punto, se calculan sólo a lo largo de los caminos en donde no están disponibles. La aplicación correcta de esta idea requiere la solución a una secuencia de cuatro problemas de flujo de datos distintos, además de otras operaciones.
- ◆ *Dominadores:* Un nodo en un grafo de flujo domina a otro nodo si cada camino que va al último debe pasar a través del primero. Un dominador propio es uno distinto del nodo en sí. Cada nodo, excepto el nodo de entrada, tiene un dominador inmediato: uno de sus propios dominadores que todos los demás dominadores propios dominan.
- ◆ *Ordenamiento “primero en profundidad” de los grafos de flujo:* Si realizamos una búsqueda en profundidad de un grafo de flujo, empezando en su entrada, producimos un árbol de expansión con búsqueda en profundidad. El orden “primero en profundidad” de los nodos es el inverso de un recorrido postorden de este árbol.
- ◆ *Clasificación de las aristas:* Cuando construimos un árbol de expansión con búsqueda en profundidad, todos las aristas del grafo de flujo pueden dividirse en tres grupos: aristas de avance (los que van del ancestro al descendiente apropiado), aristas de retirada (los que van del descendiente al ancestro), y las aristas de cruce (los demás). Una propiedad importante es que todas las aristas de cruce van de derecha a izquierda en el árbol. Otra propiedad importante es que de estas aristas, sólo los de retirada tienen una cabeza menor que su cola en el orden “primero en profundidad” (postorden inverso).
- ◆ *Aristas posteriores:* En una arista posterior, su cabeza domina a su cola. Cada arista posterior es una arista de retirada, sin importar cuál árbol de expansión con búsqueda en profundidad se elija para su grafo de flujo.
- ◆ *Grafos de flujo reducibles:* Si cada arista de retirada es una arista posterior, sin importar qué árbol de expansión con búsqueda en profundidad se elija, entonces se dice que el grafo de flujo es reducible. La gran mayoría de los grafos de flujo son reducibles; aquellos cuyas únicas instrucciones de flujo de control son las instrucciones usuales para formar ciclos y bifurcar son sin duda reducibles.
- ◆ *Ciclos naturales:* Un ciclo natural es un conjunto de nodos con un nodo de encabezado que domina a todos los nodos en el conjunto, y tiene por lo menos una arista posterior que entra a ese nodo. Dado cualquier arista posterior, podemos construir su ciclo

natural tomando la cabeza de la arista más todos los nodos que puedan llegar a la cola de la arista, sin pasar a través de la cabeza. Dos ciclos naturales con distintos encabezados están separados, o uno de ellos está completamente dentro del otro; este hecho nos permite hablar sobre una jerarquía de ciclos anidados, siempre y cuando los “ciclos” se consideren como ciclos naturales.

- ◆ *El orden “primero en profundidad” hace que el algoritmo iterativo sea eficiente:* El algoritmo iterativo requiere pocas pasadas, siempre y cuando la propagación de información a lo largo de los caminos acíclicos sea suficiente; es decir, los ciclos no agregan nada. Si visitamos los nodos en orden “primero en profundidad”, cualquier marco de trabajo del flujo de datos que propague la información hacia delante, por ejemplo, las definiciones de alcance, convergerá en no más de 2 más el número más grande de aristas de retirada en cualquier camino acíclico. Lo mismo se aplica para los marcos de trabajo de propagación hacia atrás, como las variables vivas, si visitamos en el inverso del orden “primero en profundidad” (es decir, en postorden).
- ◆ *Regiones:* Las regiones son conjuntos de nodos y aristas con un encabezado h que domina a todos los nodos en la región. Los predecesores de cualquier nodo distinto de h en la región también deben encontrarse en la misma. Las aristas de la región son todos los que pasan de un nodo a otro de la región, con la posible excepción de algunos o todos los que entren al encabezado.
- ◆ *Regiones y grafos de flujo reducibles:* Los grafos de flujo reducibles pueden analizarse en una jerarquía de regiones. Estas regiones son regiones de ciclo, que incluyen a todas las aristas que van al encabezado, o regiones de cuerpo que no tienen aristas que van al encabezado.
- ◆ *Ánalysis del flujo de datos basado en regiones:* Una alternativa para el método iterativo del análisis del flujo de datos es ir hacia arriba y hacia abajo de la jerarquía de regiones, calculando las funciones de transferencia desde el encabezado de cada región hasta cada nodo en esa región.
- ◆ *Detección de variables de inducción basada en regiones:* Una aplicación importante del análisis basado en regiones está en un marco de trabajo del flujo de datos que trata de calcular fórmulas para cada variable en una región de ciclo, cuyo valor sea una función afín (lineal) del número de veces que se recorre el ciclo.

9.10 Referencias para el capítulo 9

Dos de los primeros compiladores que realizaban una optimización de código extensa fueron Alpha [7] y Fortran H [16]. El tratado fundamental sobre las técnicas para la optimización de ciclos (por ejemplo, el movimiento de código) es [1], aunque aparecen versiones anteriores de algunas de estas ideas en [8]. Un libro distribuido de manera informal [4] fue una influencia para diseminar las ideas de optimización del código.

La primera descripción del algoritmo iterativo para el análisis de flujo de datos proviene del informe técnico no publicado de Vyssotsky y Wegner [20]. Se dice que el estudio científico del análisis de flujo de datos empieza con un par de documentos realizados por Allen [2] y Cocke [3].

La abstracción teórica del lattice que describimos aquí se basa en el trabajo de Kildall [13]. Estos marcos de trabajo asumían la distributividad, que muchos marcos de trabajo no satisfacen. Después de que surgieron varios de esos marcos de trabajo, en [5] y [11] se incrustó la condición de monotonía en el modelo.

La eliminación de redundancia parcial se manejo por primera vez en [17]. El algoritmo de movimiento de código diferido descrito en este capítulo está basado en [14].

Los dominadores se utilizaron por primera vez en el compilador descrito en [13]. Sin embargo, la idea se remonta a [18].

La noción de los grafos de flujo reducibles proviene de [2]. La estructura de estos grafos de flujo, como se presentan aquí, proviene de [9] y [10]. [12] y [15] conectaron por primera vez la capacidad de reducción de los grafos de flujo con las estructuras comunes anidadas de flujo de control, lo cual explica por qué esta clase de grafos de flujo es tan común.

La definición de capacidad de reducción mediante la reducción T_1-T_2 , como se utiliza en el análisis basado en regiones, proviene de [19]. El método basado en regiones se utilizó por primera vez en un compilador descrito en [21].

La forma de asignación individual estática (SSA) de la representación intermedia que se presenta en la sección 6.1 incorpora tanto el flujo de datos como el flujo de control en su representación. SSA facilita la implementación de muchas transformaciones de optimización, a partir de un marco de trabajo común [6].

1. Allen, F. E., “Program optimization”, *Annual Review in Automatic Programming* **5** (1969), pp. 239-307.
2. Allen, F. E., “Control flow analysis”, *ACM Sigplan Notices* **5:7** (1970), pp. 1-19.
3. Cocke, J., “Global common subexpression elimination”, *ACM SIGPLAN Notices* **5:7** (1970), pp. 20-24.
4. Cocke, J. y J. T. Schwartz, *Programming Languages and Their Compilers: Preliminary Notes*, Courant Institute of Mathematical Sciences, Univ. de Nueva York, Nueva York, 1970.
5. Cousot P. y R. Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”, *Fourth ACM Symposium on Principles of Programming Languages* (1977), pp. 238-252.
6. Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman y F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph”, *ACM Transactions on Programming Languages and Systems* **13:4** (1991), pp. 451-490.

7. Ershov, A. P., "Alpha – an automatic programming system of high efficiency", *J. ACM* **13**:1 (1966), pp. 17-24.
8. Gear, C. W., "High speed compilation of efficient object code", *Comm. ACM* **8**:8 (1965), pp. 483-488.
9. Hecht, M. S. y J. D. Ullman, "Flow graph reducibility", *SIAM J. Computing* **1** (1972), pp. 188-202.
10. Hecht, M. S. y J. D. Ullman, "Characterizations of reducible flow graphs", *J. ACM* **21** (1974), pp. 367-375.
11. Kam, J. B. y J. D. Ullman, "Monotone data flow analysis frameworks", *Acta Informatica* **7**:3 (1977), pp. 305-318.
12. Kasami, T., W. W. Peterson y N. Tokura, "On the capabilities of while, repeat and exit statements", *Comm. ACM* **16**:8 (1973), pp. 503-512.
13. Kildall, G., "A unified approach to global program optimization", *ACM Symposium on Principles of Programming Languages* (1973), pp. 194-206.
14. Knoop, J., "Lazy code motion", *Proc. ACM SIGPLAN 1992 conference on Programming Language Design and Implementation*, pp. 224-234.
15. Kosaraju, S. R., "Analysis of structured programs", *J. Computer and System Sciences* **9**:3 (1974), pp. 232-255.
16. Lowry, E. S. y C. W. Medlock, "Object code optimization", *Comm. ACM* **12**:1 (1969), pp. 13-22.
17. Morel, E. y C. Renvoise, "Global optimization by suppression of partial redundancies", *Comm. ACM* **22** (1979), pp. 96-103.
18. Prosser, R. T., "Application of boolean matrices to the analysis of flow diagrams", *AFIPS Eastern Joint Computer Conference* (1959), Spartan Books, Baltimore MD, pp. 133-138.
19. Ullman, J. D., "Fast algorithms for the elimination of common subexpressions", *Acta Informatica* **2** (1973), pp. 191-213.
20. Vyssotsky, V. y P. Wegner, "A graph theoretical Fortran source language analyzer", informe técnico sin publicar, Bell Laboratories, Murray Hill, NJ, 1963.
21. Wulf, W. A., R. K. Johnson, C. B. Weinstock, S. O. Hobbs y C. M. Geschke, *The Design of an Optimizing Compiler*, Elsevier, Nueva York, 1975.

Capítulo 10

Paralelismo a nivel de instrucción

Todo procesador moderno de alto rendimiento puede ejecutar varias operaciones en un solo ciclo de reloj. La “pregunta del millón de pesos” es: ¿qué tan rápido puede ejecutarse un programa en un procesador con paralelismo a nivel de instrucción? La respuesta depende de:

1. El paralelismo potencial en el programa.
2. El paralelismo disponible en el procesador.
3. Nuestra capacidad para extraer el paralelismo del programa secuencial original.
4. Nuestra capacidad para encontrar la mejor programación en paralelo, dadas las restricciones de programación.

Si todas las operaciones en un programa dependen en gran parte unas de otras, entonces no hay hardware ni técnicas de paralelización que puedan hacer que el programa se ejecute con rapidez en paralelo. Se ha investigado mucho para comprender los límites de la paralelización. Las aplicaciones no numéricas ordinarias tienen muchas dependencias inherentes. Por ejemplo, estos programas tienen muchas bifurcaciones dependientes de datos que no nos permiten predecir qué instrucciones se van a ejecutar, y mucho menos decidir qué operaciones pueden ejecutarse en paralelo. Por lo tanto, el trabajo en esta área se ha enfocado en la relajación de las restricciones de programación, incluyendo la introducción de nuevas características arquitectónicas, en vez de las mismas técnicas de programación.

Las aplicaciones numéricas, como los cálculos científicos y el procesamiento de señales, tienden a presentar más paralelismo. Estas aplicaciones manejan grandes estructuras de datos en conjunto; a menudo, las operaciones sobre los distintos elementos de la estructura son independientes unas de otras y pueden ejecutarse en paralelo. Los recursos de hardware adicionales pueden aprovechar dicho paralelismo, y se proporcionan en máquinas de propósito general y procesadores de señales digitales de alto rendimiento. Estos programas tienen, por lo regular, estructuras de control simples y patrones regulares de acceso a los datos; además se han desarrollado técnicas estáticas para extraer el paralelismo disponible de estos programas. La programación de código para

dichas aplicaciones es interesante y considerable, ya que ofrecen un gran número de operaciones independientes que se asignan a un gran número de recursos.

Tanto la extracción del paralelismo como la programación para la ejecución en paralelo pueden realizarse en forma estática en el software, o en forma dinámica en el hardware. De hecho, hasta las máquinas con programación de hardware se pueden ayudar mediante la programación basada en software. Este capítulo empieza por explicar las cuestiones fundamentales del uso del paralelismo a nivel de instrucción, que es igual sin importar si éste se maneja mediante software o hardware. Despues motivamos los análisis de las dependencias de datos básicas necesarias para la extracción del paralelismo. Estos análisis son útiles para muchas optimizaciones además del paralelismo a nivel de instrucción, como veremos en el capítulo 11.

Por último, presentamos las ideas básicas en la programación de código. Describimos una técnica para programar bloques básicos, un método para manejar el flujo de control con alta dependencia a los datos, que se encuentra en los programas de propósito general, y por último una técnica llamada canalizaciones de software, que se utiliza principalmente para la programación de programas numéricos.

10.1 Arquitecturas de procesadores

Al pensar en el paralelismo a nivel de instrucción, por lo general, imaginamos un procesador que emite varias operaciones en un solo ciclo de reloj. De hecho, es posible para una máquina emitir sólo una operación por reloj¹ y aún así lograr el paralelismo a nivel de instrucción, mediante el concepto de las *canalizaciones*. A continuación, primero explicaremos las canalizaciones y después hablaremos sobre la emisión de varias instrucciones.

10.1.1 Canalizaciones de instrucciones y retrasos de bifurcación

Prácticamente todos los procesadores, ya sean supercomputadoras de alto rendimiento o máquinas estándar, utilizan una *canalización de instrucciones*. Con una canalización de instrucciones, puede obtenerse una nueva instrucción en cada ciclo del reloj mientras que las siguientes instrucciones siguen pasando a través de la canalización. En la figura 10.1 se muestra una canalización simple de instrucciones de 5 etapas: primero obtiene la instrucción (IF), la decodifica (ID), ejecuta la operación (EX), accede a la memoria (MEM) y escribe de vuelta el resultado (WB). La figura muestra cómo las instrucciones i , $i + 1$, $i + 2$, $i + 3$ e $i + 4$ pueden ejecutarse al mismo tiempo. Cada fila corresponde a un pulso del reloj, y cada columna en la figura especifica la etapa que ocupa cada instrucción en cada pulso del reloj.

Si el resultado de una instrucción está disponible para cuando la instrucción que le sucede necesita los datos, el procesador puede emitir una instrucción en cada ciclo del reloj. Las instrucciones de bifurcación son en especial problemáticas, ya que hasta que se obtienen, decodifican y ejecutan, el procesador no sabe qué instrucción se ejecutará a continuación. Muchos procesadores obtienen y decodifican de manera especulativa las instrucciones que siguen justo después, en caso de que no se tome una bifurcación. Cuando se toma una bifurcación, la canalización de instrucciones se vacía y se obtiene el destino de la bifurcación.

¹Nos referiremos a un “pulso” de reloj o ciclo de reloj simplemente como un “ciclo”, cuando la intención esté clara.

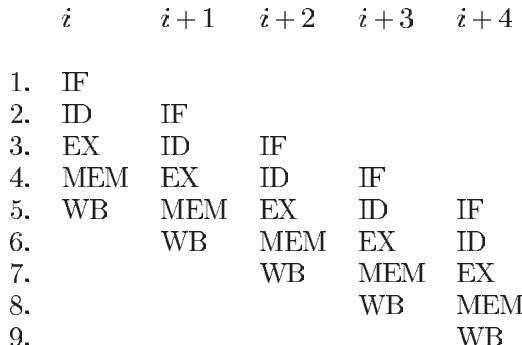


Figura 10.1: Cinco instrucciones consecutivas en una canalización de instrucciones de 5 etapas

Por ende, las bifurcaciones que se toman introducen un retraso en la obtención del destino de la bifurcación, e introducen “pequeños contratiempos” en la canalización de instrucciones. Los procesadores avanzados utilizan hardware para predecir los resultados de las bifurcaciones, con base en su historial de ejecución y para la preobtención de las ubicaciones de destino que se predijeron. No obstante, se observan retrasos de bifurcación si se predicen mal las bifurcaciones.

10.1.2 Ejecución canalizada

Algunas instrucciones requieren varios ciclos para ejecutarse. Un ejemplo común es la operación de carga en memoria. Aun cuando una memoria accede a las coincidencias en la caché, por lo general, se requieren varios ciclos para que la caché devuelva los datos. Decimos que la ejecución de una instrucción está *canalizada* si las instrucciones que le suceden, que no son dependientes del resultado, pueden proceder. En consecuencia, aun cuando un procesador puede emitir sólo una operación por ciclo de reloj, varias operaciones podrían estar en sus etapas de ejecución al mismo tiempo. Si la canalización de ejecución más profunda tiene n etapas, existe la posibilidad de que las n operaciones se encuentren en ejecución al mismo tiempo. Hay que tener en cuenta que no todas las instrucciones se canalizan por completo. Mientras que, por lo general, las sumas y multiplicaciones de punto flotante se canalizan por completo, es común que las divisiones de punto flotante, que son más complejas y se ejecutan con menos frecuencia, no se canalicen.

La mayoría de los procesadores de propósito general detectan en forma dinámica las dependencias entre las instrucciones consecutivas y detienen de manera automática la ejecución de las instrucciones si sus operandos no están disponibles. Algunos procesadores, en especial los que están incrustados en los dispositivos portátiles, dejan la comprobación de dependencias al software para poder mantener el hardware simple y el consumo de energía bajo. En este caso, el compilador es responsable de insertar instrucciones “no-operation” en el código, en caso de ser necesario, para asegurar que los resultados estén disponibles cuando se requieran.

10.1.3 Emisión de varias instrucciones

Al emitir varias operaciones por cada ciclo de reloj, los procesadores pueden mantener aún más operaciones en ejecución. El mayor número de operaciones que pueden ejecutarse en forma simultánea puede calcularse mediante la multiplicación del tamaño de emisión de instrucciones por el número promedio de etapas en la canalización de ejecución.

Al igual que la canalización, el paralelismo en máquinas que emiten varias instrucciones puede administrarse, ya sea mediante software o hardware. Las máquinas que dependen del software para administrar su paralelismo se conocen como máquinas *VLIW* (Very-Long-Instruction-Word, Palabra de instrucción muy larga), mientras que las que administran su paralelismo con el hardware se conocen como máquinas *superescalares*. Las máquinas VLIW, como su nombre lo implica, tienen palabras de instrucciones más grandes de lo normal, que codifican las operaciones a emitir en un solo ciclo de reloj. El compilador decide qué operaciones se van a emitir en paralelo y codifica la información en el código máquina de manera explícita. Por otro lado, las máquinas superescalares tienen un conjunto de instrucciones regular, con una semántica de ejecución secuencial ordinaria. Las máquinas superescalares detectan en forma automática las dependencias entre las instrucciones y las emiten a medida que sus operandos se vuelven disponibles. Algunos procesadores incluyen ambas funcionalidades, VLIW y superescalar.

Los programadores simples de hardware ejecutan las instrucciones en el orden en el que se obtienen. Si un programador se encuentra con una instrucción dependiente, ésta y todas las instrucciones que le siguen deben esperar hasta que se resuelvan las dependencias (es decir, que los resultados necesarios estén disponibles). Es obvio que dichas máquinas pueden beneficiarse de tener un programador estático que coloque las operaciones independientes una enseguida de la otra, en el orden de ejecución.

Los programadores más sofisticados pueden ejecutar las instrucciones “fuera de orden”. Las operaciones se detienen de manera independiente y no se les permite ejecutarse sino hasta que se hayan producido todos los valores de los cuales dependen. Inclusive, hasta estos programadores se benefician de la programación estática, ya que los programadores de hardware sólo tienen un espacio limitado en el cual pueden colocar en un búfer las operaciones que deben detenerse. La programación estática puede colocar las operaciones independientes cerca unas de otras, para lograr una mejor utilización del hardware. Además, sin importar qué tan sofisticado sea un programador dinámico, no puede ejecutar las instrucciones que no haya obtenido. Cuando el procesador tiene que tomar una bifurcación inesperada, sólo puede encontrar paralelismo entre las instrucciones recién obtenidas. El compilador puede mejorar el rendimiento del programador dinámico, al asegurar que estas instrucciones recién obtenidas puedan ejecutarse en paralelo.

10.2 Restricciones de la programación del código

La programación del código es una forma de optimización de los programas, que se aplica al código máquina que produce el generador de código. La programación de código está sujeta a tres tipos de restricciones:

1. *Restricciones de dependencia de control.* Todas las operaciones que se ejecutan en el programa original deben ejecutarse en el programa optimizado.

2. *Restricciones de dependencia de datos.* Las operaciones en el programa optimizado deben producir los mismos resultados que las operaciones correspondientes en el programa original.
3. *Restricciones de recursos.* El programador no debe excederse en las solicitudes de recursos de la máquina.

Estas restricciones de programación garantizan que el programa optimizado producirá los mismos resultados que el original. Sin embargo, como la programación de código modifica el orden en el que se ejecutan las operaciones, el estado de la memoria en cualquier punto dado puede no coincidir con ninguno de los estados de la memoria en una ejecución secuencial. Esta situación es un problema si la ejecución de un programa se interrumpe, por ejemplo, debido al lanzamiento de una excepción o un punto de interrupción insertado por el usuario. Por lo tanto, los programas optimizados son más difíciles de depurar. Observe que este problema no es específico para la programación de código, sino que se aplica a todas las demás optimizaciones, incluyendo la eliminación de redundancia parcial (sección 9.5) y la asignación de registros (sección 8.8).

10.2.1 Dependencia de datos

Es fácil ver que si modificamos el orden de ejecución de dos operaciones que no toquen ninguna de las mismas variables, no es posible que se puedan afectar sus resultados. De hecho, aun cuando estas dos operaciones lean la misma variable, de todas formas podemos permutar su ejecución. Sólo si una operación escribe a una variable leída o escrita por otra, se pueden ver alterados sus resultados al modificar su orden de ejecución. Se dice que dichos pares de operaciones comparten una *dependencia de datos*, y debe preservarse su orden de ejecución relativo. Hay tres tipos de dependencia de datos:

1. *Dependencia verdadera:* lectura después de escritura. Si una escritura va seguida de una lectura de la misma ubicación, la lectura depende del valor escrito; a dicha dependencia se le conoce como dependencia verdadera.
2. *Antidependencia:* escritura después de lectura. Si una lectura va seguida de una escritura a la misma ubicación, decimos que hay una antidependencia de la lectura a la escritura. La escritura no depende de la lectura en sí, pero si la escritura ocurre antes de la lectura, entonces la operación de lectura elegirá el valor incorrecto. La antidependencia es un derivado de la programación imperativa, en donde las mismas ubicaciones de memoria se utilizan para almacenar distintos valores. No es una “verdadera” dependencia y es posible eliminarla si almacenamos los valores en ubicaciones distintas.
3. *Dependencia de salida:* escritura después de escritura. Dos escrituras a la misma ubicación comparten una dependencia de salida. Si se viola esta dependencia, el valor de la ubicación de memoria escrita tendrá el valor incorrecto después de realizar ambas operaciones.

A la antidependencia y a las dependencias de salida se les conoce como *dependencias relacionadas con el almacenamiento*. No son “verdaderas” dependencias, además de que pueden eliminarse

mediante el uso de distintas ubicaciones para almacenar valores distintos. Observe que las dependencias de datos se aplican tanto a los accesos de memoria como a los accesos de registros.

10.2.2 Búsqueda de dependencias entre accesos a memoria

Para comprobar si dos accesos a memoria comparten una dependencia de datos, sólo debemos saber si pueden referirse a la misma ubicación; no es necesario saber a qué ubicación se está accediendo. Por ejemplo, podemos saber que los dos accesos $*p$ y $(*p)+4$ no pueden referirse a la misma ubicación, aun cuando no podemos saber hacia dónde apunta p . Por lo general, la dependencia de datos es indecidible en tiempo de compilación. El compilador debe suponer que las operaciones pueden referirse a la misma ubicación, a menos que se pueda demostrar lo contrario.

Ejemplo 10.1: Dada la siguiente secuencia de código:

- 1) $a = 1;$
- 2) $*p = 2;$
- 3) $x = a;$

a menos que el compilador sepa que no es posible que p apunte a a , debe concluir que las tres operaciones tienen que ejecutarse en serie. Hay una dependencia de salida que fluye de la instrucción (1) hacia la instrucción (2), y hay dos dependencias verdaderas que fluyen de las instrucciones (1) y (2) a la instrucción (3). \square

El análisis de dependencias de datos es altamente sensible al lenguaje de programación que se utiliza en el programa. Para los lenguajes sin seguridad de tipos como C y C++, en donde un apuntador puede convertirse para apuntar a cualquier tipo de objeto, es necesario el análisis sofisticado para probar la independencia entre cualquier par de accesos a memoria basados en apuntadores. Hasta podemos acceder de manera indirecta a las variables escalares locales o globales, a menos que podamos demostrar que ninguna de las instrucciones del programa ha almacenado sus direcciones en ninguna parte. En los lenguajes con seguridad de tipos, como Java, los objetos de tipos diferentes son necesariamente distintos unos de otros. De manera similar, las variables primitivas locales en la pila no pueden tener alias con accesos a través de otros nombres.

Un descubrimiento correcto de las dependencias de datos requiere una variedad de formas de análisis. Nos enfocaremos en las preguntas principales que deben resolverse si el compilador va a detectar todas las dependencias que existan en un programa, y en cómo utilizar esta información en la programación de código. En los siguientes capítulos mostraremos cómo se realizan estos análisis.

Análisis de dependencia de datos en arreglos

La dependencia de los datos en arreglos es el problema de eliminar la ambigüedad entre los valores de los índices en los accesos a los elementos de un arreglo. Por ejemplo, el siguiente ciclo:

```
for (i = 0; i < n; i++)
    A[2*i] = A[2*i+1];
```

copia los elementos impares en el arreglo A a los elementos pares que están justo después de ellos. Como todas las ubicaciones leídas y escritas en el ciclo son distintas unas de otras, no hay dependencias entre los accesos y todas las iteraciones en el ciclo pueden ejecutarse en paralelo. El análisis de dependencias de datos en arreglos, que a menudo se le conoce simplemente como *análisis de dependencias de datos*, es muy importante para la optimización de aplicaciones numéricas. En la sección 11.6 hablaremos sobre este tema con detalle.

Análisis de alias de apuntadores

Decimos que dos apuntadores tienen el mismo alias si pueden hacer referencia al mismo objeto. El análisis de alias de apuntadores es difícil, ya que existen muchos apuntadores que pueden tener el mismo alias potencialmente en un programa, y cada uno de ellos puede apuntar a un número ilimitado de objetos dinámicos a través del tiempo. Para obtener alguna precisión, el análisis de alias de apuntadores debe aplicarse a través de todas las funciones en un programa. En la sección 12.4 empezaremos a hablar sobre este tema.

Análisis entre procedimientos

Para los lenguajes que pasan parámetros por referencia, es necesario el análisis entre procedimientos para determinar si la misma variable se pasa como dos o más argumentos distintos. Dichos alias pueden crear dependencias entre parámetros aparentemente distintos. De manera similar, pueden usarse las variables globales como parámetros y, por ende, crear dependencias entre los accesos a los parámetros y los accesos a las variables globales. El análisis entre procedimientos, que veremos en el capítulo 12, es necesario para determinar estos alias.

10.2.3 Concesiones entre el uso de registros y el paralelismo

En este capítulo vamos a suponer que la representación intermedia independiente de la máquina del programa fuente utiliza un número ilimitado de *seudoregistros* para representar variables que pueden asignarse a los registros. Estas variables incluyen a las variables escalares en el programa fuente a las que no se puede hacer referencia mediante otros nombres, así como las variables temporales que el compilador genera para guardar los resultados parciales en las expresiones. A diferencia de las ubicaciones de memoria, los registros tienen nombres únicos. En consecuencia, pueden generarse con facilidad restricciones de dependencia de datos precisas para los accesos a los registros.

El número ilimitado de seudoregistros utilizados en la representación intermedia debe, en un momento dado, asignarse al pequeño número de registros físicos disponibles en la máquina de destino. La asignación de varios seudoregistros al mismo registro físico tiene el desafortunado efecto adicional de crear dependencias de almacenamiento artificiales, que restringen el paralelismo a nivel de instrucción. Por el contrario, la ejecución de las instrucciones en paralelo crea la necesidad de más almacenamiento para guardar los valores que se calculan al mismo tiempo. Por ende, el objetivo de minimizar el número de registros utilizados entra en conflicto directo con el objetivo de incrementar al máximo el paralelismo a nivel de instrucción. Los ejemplos 10.2 y 10.3 que se muestran a continuación, ilustran esta concesión clásica entre el almacenamiento y el paralelismo.

Renombramiento de registros de hardware

El paralelismo a nivel de instrucción se utilizó por primera vez en las arquitecturas computacionales como un medio para agilizar el código de máquina secuencial ordinario. Los compiladores de ese tiempo no estaban al tanto del paralelismo a nivel de instrucción en la máquina y estaban diseñados para optimizar el uso de los registros. Reordenaban en forma deliberada las instrucciones para minimizar el número de registros utilizados y, como resultado, también disminuían la cantidad de paralelismo disponible. El ejemplo 10.3 ilustra cómo la minimización del uso de registros en el cálculo de árboles de expresiones también limita su paralelismo.

Quedaba tan poco paralelismo en el código secuencial que los arquitectos de computadoras inventaron el concepto de *renombramiento de registros de hardware* para deshacer los efectos de la optimización de los registros en los compiladores. El renombramiento de los registros de hardware cambia en forma dinámica la asignación de registros, a medida que se ejecuta el programa. Interpreta el código máquina, almacena los valores destinados para el mismo registro en distintos registros internos, y actualiza todos sus usos para que hagan referencia a los registros correctos, de manera acorde.

Como el compilador introdujo primero las restricciones de dependencia de registros artificiales, pueden eliminarse mediante el uso de un algoritmo de asignación de registros que está al corriente del paralelismo a nivel de instrucción. El renombramiento de registros de hardware sigue siendo útil en el caso en el que el conjunto de instrucciones de una máquina sólo pueda hacer referencia a un pequeño número de registros. Esta capacidad permite una implementación de la arquitectura para asignar el pequeño número de registros arquitectónicos en el código, a un número mucho mayor de registros internos, en forma dinámica.

Ejemplo 10.2: El siguiente código copia los valores de las variables en las ubicaciones **a** y **c** a las variables en las ubicaciones **b** y **d**, respectivamente, usando los seudoregistros **t1** y **t2**.

```
LD t1, a    // t1 = a
ST b, t1    // b  = t1
LD t2, c    // t2 = c
ST d, t2    // d  = t2
```

Si sabemos que todas las ubicaciones de memoria a las que se accedió son distintas, entonces las copias pueden proceder en paralelo. No obstante, si a **t1** y **t2** se les asigna el mismo registro de forma que minimice el número de registros usados, es necesario serializar las copias. □

Ejemplo 10.3: Las técnicas tradicionales de repartición de registros están orientadas a minimizar el número de registros utilizados cuando se realiza un cálculo. Considere la siguiente expresión:

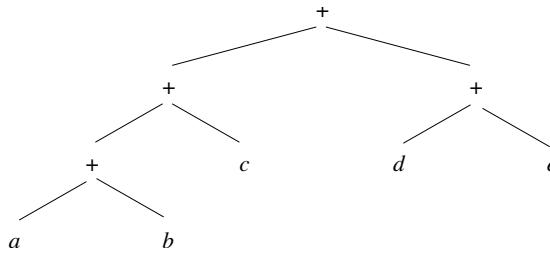


Figura 10.2: Árbol de la expresión del ejemplo 10.3

$$(a + b) + c + (d + e)$$

que se muestra como un árbol sintáctico en la figura 10.2. Es posible realizar este cálculo usando tres registros, como se ilustra mediante el código máquina de la figura 10.3.

```

LD r1, a      // r1 = a
LD r2, b      // r2 = b
ADD r1, r1, r2 // r1 = r1+r2
LD r2, c      // r2 = c
ADD r1, r1, r2 // r1 = r1+r2
LD r2, d      // r2 = d
LD r3, e      // r3 = e
ADD r2, r2, r3 // r2 = r2+r3
ADD r1, r1, r2 // r1 = r1+r2
  
```

Figura 10.3: Código máquina para la expresión de la figura 10.2

Sin embargo, la reutilización de registros serializa el cálculo. Las únicas operaciones a las que se les permite ejecutarse en paralelo son las cargas de los valores en las ubicaciones **a** y **b**, y las cargas de los valores en las ubicaciones **d** y **e**. Por ende, se requiere un total de 7 pasos para completar el cálculo en paralelo.

Si hubiéramos usado distintos registros para cada suma parcial, la expresión podría evaluarse en 4 pasos, lo cual constituye la altura del árbol de expresión de la figura 10.2. La figura 10.4 sugiere los cálculos en paralelo. \square

$r1 = a$	$r2 = b$	$r3 = c$	$r4 = d$	$r5 = e$
$r6 = r1+r2$	$r7 = r4+r5$			
$r8 = r6+r3$				
$r9 = r8+r7$				

Figura 10.4: Evaluación en paralelo de la expresión de la figura 10.2

10.2.4 Ordenamiento de fases entre la asignación de registros y la programación de código

Si los registros se asignan antes de la programación, el código resultante tiene por lo regular muchas dependencias de almacenamiento que limitan la programación del código. Por otro lado, si el código se programa antes de la asignación de los registros, el programa creado puede requerir tantos registros que un *derrame* de registros (almacenamiento del contenido de un registro en una ubicación de memoria, de manera que el registro pueda usarse para otro fin) puede negar las ventajas del paralelismo a nivel de instrucción. ¿Debe un compilador asignar los registros primero, antes de programar el código?, ¿o debe ser al revés? ¿O tenemos que lidiar con estos dos problemas al mismo tiempo?

Para responder a las preguntas anteriores, debemos considerar las características de los programas que se van a compilar. Muchas aplicaciones no numéricas no tienen tanto paralelismo disponible. Basta con dedicar un pequeño número de registros para guardar los resultados temporales en las expresiones. Podemos aplicar primero un algoritmo de coloración, como en la sección 8.8.4, para asignar registros a todas las variables no temporales, después programar el código y por último asignar registros a las variables temporales.

Este método no funciona para las aplicaciones numéricas, en las que hay expresiones mucho más extensas. Podemos usar un método jerárquico, en el cual el código se optimiza desde el interior hacia fuera, empezando con los ciclos más internos. Las instrucciones se programan primero, suponiendo que a cada seudoregistro se le asigna su propio registro físico. La asignación de registros se aplica después de la programación y el código de derrame se agrega en donde sea necesario, y después el código se reprograma. Este proceso se repite para el código en los ciclos externos. Cuando se consideran varios ciclos internos en conjunto en un ciclo externo común, a la misma variable se le pueden haber asignado distintos registros. Podemos cambiar la asignación de registros para evitar tener que copiar los valores de un registro a otro. En la sección 10.5, vamos a hablar sobre la interacción entre la repartición de registros y la programación con más detalle, en el contexto de un algoritmo de programación específico.

10.2.5 Dependencia del control

La programación de las operaciones dentro de un bloque básico es relativamente simple, debido a que se garantiza que todas las instrucciones se ejecutan una vez que el flujo de control llega al inicio del bloque. Las instrucciones en un bloque básico pueden reordenarse en forma arbitraria, siempre y cuando se satisfagan todas las dependencias de datos. Por desgracia, los bloques básicos, en especial en los programas no numéricos, son en general muy pequeños; en promedio, hay aproximadamente sólo cinco instrucciones en un bloque básico. Además, a menudo las operaciones en el mismo bloque están muy relacionadas y, por ende, tienen poco paralelismo. Por lo tanto, la explotación del paralelismo entre los bloques básicos es crucial.

Un programa optimizado debe ejecutar todas las operaciones en el programa original. Puede ejecutar más instrucciones que el original, siempre y cuando las instrucciones adicionales no cambien lo que hace el programa. ¿Por qué la ejecución de instrucciones adicionales agiliza la ejecución de un programa? Si sabemos que hay la probabilidad de que se vaya a ejecutar una instrucción, y hay un recurso inactivo disponible para realizar la operación “sin costo”, podemos ejecutar la instrucción en forma *especulativa*. El programa se ejecuta con más rapidez cuando la especulación resulta ser correcta.

Se dice que una instrucción i_1 es *dependiente del control* en la instrucción i_2 si el resultado de i_2 determina si se va a ejecutar i_1 . La noción de dependencia del control corresponde al concepto de anidar niveles en los programas estructurados por bloques. En específico, en la siguiente instrucción if-else:

```
if (c) s1; else s2;
```

$s1$ y $s2$ son dependientes del control en c . De manera similar, en la siguiente instrucción while:

```
while (c) s;
```

el cuerpo es dependiente del control en c .

Ejemplo 10.4: En el siguiente fragmento de código:

```
if (a > t)
    b = a*a;
    d = a+c;
```

las instrucciones $b = a*a$ y $d = a+c$ no tienen dependencia de datos con ninguna otra parte del fragmento. La instrucción $b = a*a$ depende de la comparación $a > t$. Sin embargo, la instrucción $d = a+c$ no depende de la comparación y puede ejecutarse en cualquier momento. Suponiendo que la multiplicación $a * a$ no produzca efectos colaterales, puede realizarse en forma especulativa, siempre y cuando b se escriba sólo después de encontrar que a es mayor que t . \square

10.2.6 Soporte de ejecución especulativa

Las cargas de memoria son un tipo de instrucción que puede beneficiarse en forma considerable de la ejecución especulativa. Desde luego, las cargas en memoria son bastante comunes. Tienen latencias de ejecución bastante extensas, las direcciones que se utilizan en las cargas están disponibles comúnmente desde antes, y el resultado puede almacenarse en una nueva variable temporal sin destruir el valor de cualquier otra variable. Por desgracia, las cargas en memoria pueden producir excepciones si sus direcciones son ilegales, por lo que si accedemos ilegalmente a las direcciones en forma especulativa, podemos provocar que un programa correcto se detenga de manera inesperada. Además, las cargas de memoria mal pronosticadas pueden provocar fallas adicionales en la caché y fallas de página, que son en extremo costosas.

Ejemplo 10.5: En el siguiente fragmento:

```
if (p != null)
    q = *p;
```

al desreferenciar p en forma especulativa, este programa correcto se detendrá con un error si p es `null`. \square

Muchos procesadores de alto rendimiento proporcionan características especiales para soportar los accesos especulativos a memoria. A continuación mencionaremos los más importantes.

Preobtención

La instrucción de *preobtención* (*prefetch*) se inventó para llevar los datos de la memoria a la caché antes de usarlos. Una instrucción *preobtención* indica al procesador que es probable que el programa utilice una palabra de memoria específica en el futuro cercano. Si la ubicación especificada es inválida o si al acceder a ésta se produce una falla de página, el procesador sólo ignora la operación. En cualquier otro caso, el procesador lleva los datos de la memoria a la caché, si no se encuentran ya ahí.

Bits venenosos

Hay otra característica de arquitectura, conocida como *bits venenosos*, que se inventó para permitir la carga especulativa de datos, de la memoria al archivo de registro. Cada registro en la máquina se aumenta con un bit *venenoso*. Si se accede a la memoria ilegal o la página a la que se accedió no se encuentra en memoria, el procesador no produce la excepción de inmediato, sino que sólo establece el bit venenoso del registro de destino. Una excepción se produce sólo si se utiliza el contenido del registro con un bit venenoso marcado.

Ejecución predicada

Como las bifurcaciones son costosas, y las bifurcaciones mal pronosticadas lo son aún más (vea la sección 10.1), se inventaron las *instrucciones predicadas* para reducir el número de bifurcaciones en un programa. Una instrucción predicada es como una instrucción normal, sólo que tiene un operando predicado adicional para proteger su ejecución; la instrucción se ejecuta sólo si el predicado es verdadero.

Como ejemplo, una instrucción de movimiento condicional CMOVZ R2, R3, R1 tiene la semántica de que el contenido del registro R3 se mueve hacia el registro R2 sólo si el registro R1 es cero. El código como:

```
if (a == 0)
    b = c+d;
```

puede implementarse con dos instrucciones de máquina, suponiendo que *a*, *b*, *c* y *d* se asignan a los registros R1, R2, R4, R5 respectivamente, como se muestra a continuación:

```
ADD    R3, R4, R5
CMOVZ R2, R3, R1
```

Esta conversión sustituye una serie de instrucciones que comparten una dependencia de control con las instrucciones que comparten sólo dependencias de datos. Así, estas instrucciones pueden combinarse con bloques básicos adyacentes para crear un bloque básico más grande. Lo más importante es que con este código el procesador no tiene oportunidad de dar un mal pronóstico, con lo cual se garantiza que la canalización de instrucciones funcionará de manera uniforme.

La ejecución predicada tiene un costo. Las instrucciones predicadas se obtienen y se decodifican, aun cuando tal vez no se ejecuten al final. Los programadores estáticos deben reservar todos los recursos necesarios para su ejecución y asegurar que se cumpla con todas las

Máquinas con programación dinámica

El conjunto de instrucciones de una máquina con programación estática define en forma explícita lo que se puede ejecutar en paralelo. Sin embargo, en la sección 10.1.2 vimos que ciertas arquitecturas de máquinas permiten realizar la decisión en tiempo de ejecución, acerca de lo que puede ejecutarse en paralelo. Con la programación dinámica, el mismo código de máquina puede ejecutarse en distintos miembros de la misma familia (máquinas que implementan el mismo conjunto de instrucciones) que tienen cantidades variadas de soporte de ejecución en paralelo. De hecho, la compatibilidad del código de máquina es una de las principales ventajas de las máquinas con programación dinámica.

Los programadores estáticos, que se implementan en el compilador mediante software, pueden ayudar a los programadores dinámicos (que se implementan en el hardware de la máquina) a utilizar mejor los recursos de la máquina. Para construir un programador estático para una máquina con programación dinámica, podemos usar casi el mismo algoritmo de programación que para las máquinas con programación estática, sólo que no se deben generar de manera explícita las instrucciones *no-op* que quedan en el programa. En la sección 10.4.7 hablaremos con más detalle sobre esta cuestión.

dependencias de datos potenciales. La ejecución predicable no debe usarse en forma agresiva, a menos que la máquina tenga muchos recursos más de los que podrían utilizarse en cualquier otro caso.

10.2.7 Un modelo de máquina básico

Muchas máquinas pueden representarse mediante el siguiente modelo simple. Una máquina $M = \langle R, T \rangle$, consiste en:

1. Un conjunto de tipos de operación T , como cargas, almacenamientos, operaciones aritméticas, etcétera.
2. Un vector $R = [r_1, r_2, \dots]$ que representa a los recursos de hardware, en donde r_i es el número de unidades disponibles del i -ésimo tipo de recurso. Algunos ejemplos de tipos de recursos comunes son: unidades de acceso a memoria, ALUs y unidades funcionales de punto flotante.

Cada operación tiene un conjunto de operandos de entrada, un conjunto de operandos de salida y un requerimiento de recursos. Con cada operando de entrada hay una latencia de entrada asociada, la cual indica cuándo debe estar disponible el valor de entrada (relativo al inicio de la operación). Los operandos de entrada comunes tienen cero latencia, lo cual significa que los valores se necesitan de inmediato, en el ciclo de reloj en el que se emite la operación. De manera similar, con cada operando de salida hay una latencia de salida asociada, la cual indica cuándo debe estar disponible el resultado, relativo al inicio de la operación.

El uso de recursos para cada tipo de operación t de la máquina se modela mediante una *tabla de reservación de recursos* bidimensional, RT_t . La anchura de la tabla es el número de

tipos de recursos en la máquina, y su longitud es la duración a través de la cual la operación utiliza los recursos. La entrada $RT_t[i, j]$ es el número de unidades del j -ésimo recurso utilizado por una operación de tipo t , i ciclos de reloj después de emitirse. Por cuestión de simplicidad en la notación, vamos a suponer que $RT_t[i, j] = 0$, si i se refiere a una entrada inexistente en la tabla (es decir, si i es mayor que el número de ciclos que se requieren para ejecutar la operación). Desde luego que, para cualquier t , i y j , $RT_t[i, j]$ debe ser menor o igual que $R[j]$, el número de recursos de tipo j que tiene la máquina.

Las operaciones comunes de la máquina sólo ocupan una unidad de recurso al momento en que se emite una operación. Algunas operaciones pueden usar más de una unidad funcional. Por ejemplo, una operación de multiplicación y suma puede usar un multiplicador en el primer ciclo de reloj, y un sumador en el segundo. Algunas operaciones, como la división, tal vez necesiten ocupar un recurso durante varios ciclos de reloj. Las operaciones *canalizadas por completo* son aquellas que pueden emitirse en cada ciclo de reloj, aun cuando sus resultados no estén disponibles sino hasta cierto número de ciclos después. No tenemos que modelar los recursos de cada etapa de canalización en forma explícita; basta con una sola unidad para representar la primera etapa. Cualquier operación que ocupe la primera etapa de una canalización tiene garantizado el derecho de proceder a las etapas siguientes en los ciclos de reloj posteriores.

- 1) $a = b$
- 2) $c = d$
- 3) $b = c$
- 4) $d = a$
- 5) $c = d$
- 6) $a = b$

Figura 10.5: Una secuencia de asignaciones que exhibe dependencias de datos

10.2.8 Ejercicios para la sección 10.2

Ejercicio 10.2.1: Las asignaciones en la figura 10.5 tienen ciertas dependencias. Para cada uno de los siguientes pares de instrucciones, clasifique la dependencia como (i) dependencia verdadera, (ii) antidependencia, (iii) dependencia de salida, o (iv) sin dependencia (es decir, las instrucciones pueden aparecer en cualquier orden):

- a) Instrucciones (1) y (4).
- b) Instrucciones (3) y (5).
- c) Instrucciones (1) y (6).
- d) Instrucciones (3) y (6).
- e) Instrucciones (4) y (6).

Ejercicio 10.2.2: Evalúe la expresión $((u+v)+(w+x))+(y+z)$ tal como están los paréntesis (es decir, no utilice las leyes conmutativas o asociativas para reordenar las sumas). Proporcione el código de máquina a nivel de registro para obtener el máximo paralelismo posible.

Ejercicio 10.2.3: Repita el ejercicio 10.2.2 para las siguientes expresiones:

a) $(u + (v + (w + x))) + (y + z)$.

b) $(u + (v + w)) + (x + (y + z))$.

Si en vez de maximizar el paralelismo, minimizamos el número de registros, ¿cuántos pasos requeriría el cálculo? ¿Cuántos pasos nos ahorramos al utilizar el paralelismo máximo?

Ejercicio 10.2.4: La expresión del ejercicio 10.2.2 puede ejecutarse mediante la secuencia de instrucciones que se muestran en la figura 10.6. Si tenemos todo el paralelismo que necesitamos, ¿cuántos pasos se necesitan para ejecutar las instrucciones?

```

1) LD r1, u          // r1 = u
2) LD r2, v          // r2 = v
3) ADD r1, r1, r2   // r1 = r1 + r2
4) LD r2, w          // r2 = w
5) LD r3, x          // r3 = x
6) ADD r2, r2, r3   // r2 = r2 + r3
7) ADD r1, r1, r2   // r1 = r1 + r2
8) LD r2, y          // r2 = y
9) LD r3, z          // r3 = z
10) ADD r2, r2, r3  // r2 = r2 + r3
11) ADD r1, r1, r2  // r1 = r1 + r2

```

Figura 10.6: Implementación de una expresión aritmética con el número mínimo de registros

Ejercicio 10.2.5: Traduzca el fragmento de código que vimos en el ejemplo 10.4, usando la instrucción de copia condicional CMOVZ de la sección 10.2.6. ¿Cuáles son las dependencias de datos en su código de máquina?

10.3 Programación de bloques básicos

Ahora estamos listos para empezar a hablar sobre los algoritmos de programación de código. Empezaremos con el problema más sencillo: programar las operaciones en un bloque básico que consiste en instrucciones de máquina. La solución óptima para este problema es NP-completo. Pero en la práctica, un bloque básico ordinario sólo tiene un pequeño número de operaciones altamente restringidas, por lo que basta con las técnicas simples de programación. Para este problema vamos a presentar un algoritmo simple pero muy efectivo, conocido como *programación por lista*.

10.3.1 Grafos de dependencia de datos

Representamos cada bloque básico de instrucciones de máquina mediante un *grafo de dependencia de datos*, $G = (N, E)$, el cual tiene un conjunto de nodos N que representan las operaciones en las instrucciones de máquina en el bloque, y un conjunto de aristas E dirigidos que representan las restricciones de dependencia de datos entre las operaciones. Los nodos y las aristas de G se construyen de la siguiente manera:

1. Cada operación n en N tiene una tabla de reservación de recursos RT_n , cuyo valor es simplemente la tabla de reservación de recursos asociada con el tipo de operación de n .
2. Cada arista e en E se etiqueta con el retraso d_e , lo cual indica que el nodo de destino no debe emitirse antes de d_e ciclos después de emitir el nodo de origen. Suponga que la operación n_1 va seguida de la operación n_2 , y que ambas acceden a la misma ubicación, con las latencias l_1 y l_2 , respectivamente. Es decir, el valor de la ubicación se produce l_1 ciclos después de que empieza la primera instrucción, y la segunda instrucción necesita ese valor l_2 ciclos después de iniciar (observe que es común tener $l_1 = 1$ y $l_2 = 0$). Entonces, hay una arista $n_1 \rightarrow n_2$ en E etiquetada con el retraso $l_1 - l_2$.

Ejemplo 10.6: Considere una máquina simple que puede ejecutar dos operaciones en cada ciclo de reloj. La primera debe ser una operación de bifurcación, o una operación de la ALU, de la siguiente forma:

```
OP dst, orig1, orig2
```

La segunda debe ser una operación de carga o almacenamiento, de la siguiente forma:

```
LD dst, direc
ST direc, orig
```

La operación de carga (LD) está canalizada por completo y requiere dos ciclos de reloj. Sin embargo, justo después de una carga puede ir un almacenamiento ST que escriba a la ubicación de memoria que se leyó. Todas las demás operaciones se completan en un ciclo.

En la figura 10.7 se muestra el grafo de dependencia de un ejemplo de un bloque básico y su requerimiento de recursos. Podríamos imaginar que R1 es un apuntador de pila, que se utiliza para acceder a los datos en la pila con desplazamientos como 0 o 12. La primera instrucción carga el registro R2, y el valor cargado no está disponible sino hasta dos ciclos de reloj después. Esta observación explica la etiqueta 2 en las aristas que van de la primera a la segunda y quinta instrucciones, cada una de las cuales necesita el valor de R2. De manera similar, hay un retraso de 2 en la arista que va de la tercera instrucción a la cuarta; el valor que se carga en R3 lo necesita la cuarta instrucción, y no está disponible sino hasta dos ciclos después de que empieza la tercera instrucción.

Como no sabemos cómo se relacionan los valores de R1 y R7, tenemos que considerar la posibilidad de que una dirección como 8(R1) sea igual que la dirección 0(R7). Es decir, la última

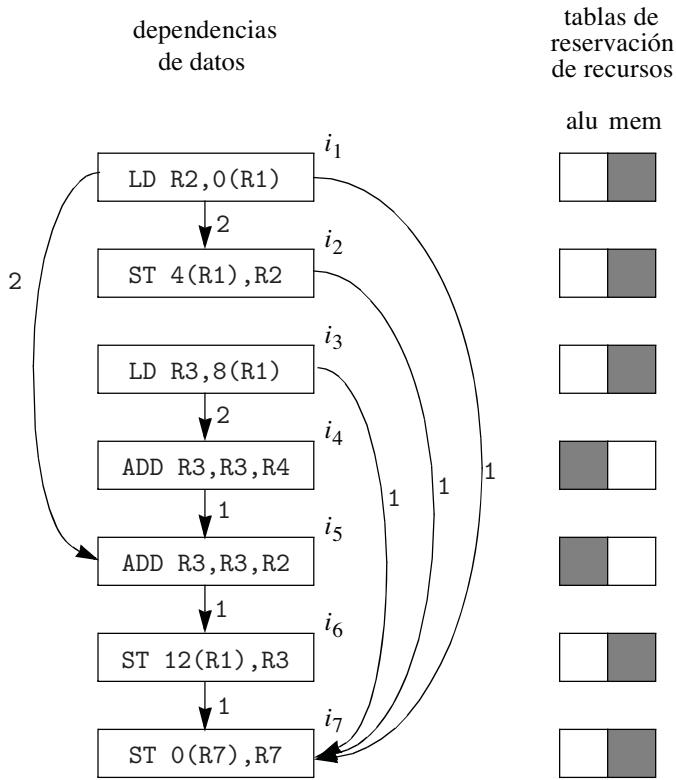


Figura 10.7: Grafo de dependencia de datos para el ejemplo 10.6

instrucción puede estar almacenando datos en la misma dirección de la que la tercera instrucción carga datos. El modelo de máquina que estamos usando nos permite almacenar datos en una ubicación un ciclo después de cargar datos de esa ubicación, aun cuando el valor que se va a cargar no aparecerá en un registro sino hasta un ciclo de reloj después. Esta observación explica la etiqueta 1 en la arista que va de la tercera instrucción a la última. El mismo razonamiento explica las aristas y las etiquetas que van de la primera instrucción a la última. Las otras aristas con la etiqueta 1 se explican mediante una dependencia, o posible dependencia, condicionada en el valor de R7. \square

10.3.2 Programación por lista de bloques básicos

El método más simple para programar bloques básicos implica la acción de visitar cada nodo del grafo de dependencia de datos en “orden topológico priorizado”. Como no puede haber ciclos en un grafo de dependencia de datos, siempre hay por lo menos un orden topológico para los nodos. Sin embargo, entre los posibles órdenes topológicos, algunos pueden ser preferibles a otros. En la sección 10.3.3 hablaremos sobre algunas de las estrategias para elegir un orden

Tablas de reservación de recursos ilustradas

Con frecuencia, es útil visualizar una tabla de reservación de recursos para una operación, mediante una rejilla de cuadros llenos y vacíos. Cada columna corresponde a uno de los recursos de la máquina, y cada fila corresponde a uno de los ciclos de reloj durante los cuales se ejecuta la operación. Suponiendo que la operación nunca necesita más de una unidad de cualquier recurso, podemos representar los 1s mediante cuadros llenos y los 0s mediante cuadros vacíos. Además, si la operación está canalizada por completo, entonces sólo debemos indicar los recursos utilizados en la primera fila, y la tabla de reservación de recursos se convierte en una sola fila.

Por ejemplo, esta representación se utiliza en el ejemplo 10.6. En la figura 10.7 podemos ver las tablas de reservación de recursos como filas. Las dos operaciones de suma requieren el recurso “alu”, mientras que las operaciones de carga y almacenamiento requieren el recurso “mem”.

topológico, pero por el momento sólo supondremos que existe un algoritmo para elegir un orden preferido.

El algoritmo de programación por lista que vamos a describir a continuación visita los nodos en el orden topológico priorizado que hayamos elegido. Los nodos pueden o no terminar siendo programados en el mismo orden en el que se visitan. Pero las instrucciones se colocan en el programa lo más pronto posible, por lo que hay una tendencia para que las instrucciones se programen aproximadamente en el orden visitado.

Visto con más detalle, el algoritmo calcula la primera ranura de tiempo en la que puede ejecutarse cada nodo, de acuerdo con sus restricciones de dependencia de datos con los nodos que se programaron previamente. A continuación, se comprueban los recursos que el nodo necesita contra una tabla de reservación de recursos que recolecta todos los recursos asignados hasta ese momento. El nodo se programa en la primera ranura de tiempo que tenga suficientes recursos.

Algoritmo 10.7: Programación por lista de un bloque básico.

ENTRADA: Un vector de recursos de máquina $R = [r_1, r_2, \dots]$, en donde r_i es el número de unidades disponibles del i -ésimo tipo de recurso, y un grafo de dependencia de datos $G = (N, E)$. Cada operación n en N se etiqueta con su tabla de reservación de recursos RT_n ; cada arista $e = n_1 \rightarrow n_2$ en E se etiqueta con d_e , lo cual indica que n_2 no debe ejecutarse antes de d_e ciclos de reloj después que n_1 .

SALIDA: Un programa S que asigna las operaciones en N en ranuras de tiempo, en donde las operaciones pueden iniciarse y cumplir con todas las restricciones de recursos y de datos.

MÉTODO: Ejecute el programa de la figura 10.8. En la sección 10.3.3 veremos una explicación de lo que podría ser el “orden topológico priorizado”. \square

```

 $RT$  = una tabla de reservación vacía;
for (cada  $n$  en  $N$  en orden topológico priorizado) {
     $s = \max_{e=p \rightarrow n \text{ en } E} (S(p) + d_e);$ 
    /* Busca el momento más temprano en el que podría empezar esta instrucción,
       ya cuando empezaron sus predecesores. */
    while (exista una  $i$  tal que  $RT[s + i] + RT_n[i] > R$ )
         $s = s + 1;$ 
        /* Retrasa aún más la instrucción, hasta que estén disponibles
           los recursos necesarios. */
     $S(n) = s;$ 
    for (todas las  $i$ )
         $RT[s + i] = RT[s + i] + RT_n[i]$ 
}

```

Figura 10.8: Un algoritmo de programación por lista

10.3.3 Órdenes topológicos priorizados

La programación por lista no da marcha atrás; programa cada nodo sólo una vez. Utiliza una función de prioridad heurística para elegir de entre los nodos que están listos para programarse a continuación. He aquí algunas observaciones acerca de los posibles ordenamientos priorizados de los nodos:

- Sin restricciones de recursos, el programa más corto se da mediante la *ruta crítica*, la ruta más larga a través del grafo de dependencia de datos. Una medida útil como función de prioridad es la *altura* del nodo, que es la longitud de una ruta más larga en el grafo, que se origina desde el nodo.
- Por otro lado, si todas las operaciones son independientes, entonces la longitud del programa se ve restringida por los recursos disponibles. El recurso crítico es el que tiene la proporción más grande de usos, para el número de unidades disponibles de ese recurso. Las operaciones que utilizan recursos más críticos pueden recibir una prioridad más alta.
- Por último, podemos usar el ordenamiento de origen para romper los vínculos entre las operaciones; la operación que aparezca primero en el programa fuente debe programarse primero.

Ejemplo 10.8: Para el grafo de dependencia de datos de la figura 10.7, la ruta crítica, incluyendo el tiempo para ejecutar la última instrucción, es de 6 ciclos. Es decir, la ruta crítica consta de los últimos cinco nodos, desde la carga de R3 hasta el almacenamiento en R7. El total de los retrasos en las aristas a lo largo de esta ruta es de 5, a lo cual sumamos 1 por el ciclo de reloj requerido para la última instrucción.

Usando la altura como la función de prioridad, el Algoritmo 10.7 encuentra un programa óptimo, como se muestra en la figura 10.9. Observe que programamos primero la carga de R3, ya que tiene la mayor altura. La suma de R3 y R4 tiene los recursos para programarse en el

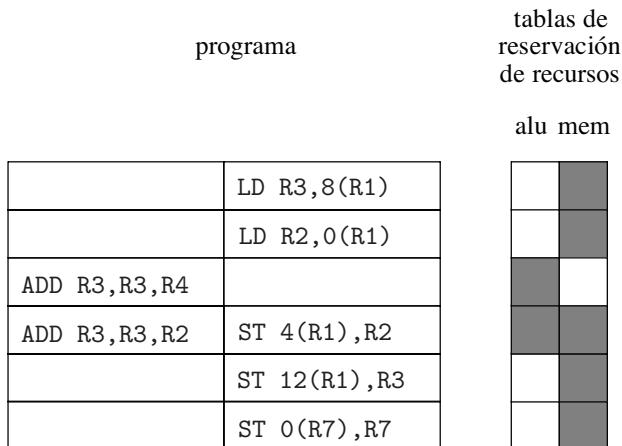


Figura 10.9: Resultado de aplicar la programación por lista al ejemplo de la figura 10.7

segundo ciclo de reloj, pero el retraso de 2 para una carga nos obliga a esperar hasta el tercer ciclo de reloj para programar esta suma. Es decir, no podemos estar seguros de que $R3$ tendrá su valor necesario, sino hasta el inicio del ciclo 3. \square

1)	LD R1, a	LD R1, a	LD R1, a
2)	LD R2, b	LD R2, b	LD R2, b
3)	SUB R3, R1, R2	SUB R1, R1, R2	SUB R3, R1, R2
4)	ADD R2, R1, R2	ADD R2, R1, R2	ADD R4, R1, R2
5)	ST a, R3	ST a, R1	ST a, R3
6)	ST b, R2	ST b, R2	ST b, R4

Figura 10.10: Código máquina para el ejercicio 10.3.1

10.3.4 Ejercicios para la sección 10.3

Ejercicio 10.3.1: Para cada uno de los fragmentos de código de la figura 10.10, dibuje el grafo de dependencia de datos.

Ejercicio 10.3.2: Suponga que tenemos una máquina con un recurso ALU (para las operaciones ADD y SUB) y un recurso MEM (para las operaciones LD y ST). Suponga que todas las operaciones requieren un ciclo de reloj, excepto LD, la cual requiere dos. Sin embargo, como en el ejemplo 10.6, una operación ST en la misma ubicación de memoria puede comenzar un ciclo de reloj después del inicio de una operación LD sobre esa ubicación. Busque la programación más corta para cada uno de los fragmentos de la figura 10.10.

Ejercicio 10.3.3: Repita el ejercicio 10.3.2, suponiendo que:

- i. La máquina tiene un recurso ALU y dos recursos MEM.
- ii. La máquina tiene dos recursos ALU y un recurso MEM.
- iii. La máquina tiene dos recursos ALU y dos recursos MEM.

- 1) LD R1, a
- 2) ST b, R1
- 3) LD R2, c
- 4) ST c, R1
- 5) LD R1, d
- 6) ST d, R2
- 7) ST a, R1

Figura 10.11: Código máquina para el ejercicio 10.3.4

Ejercicio 10.3.4: Tomando el modelo de máquina del ejemplo 10.6 (como en el ejercicio 10.3.2):

- a) Dibuje el grafo de dependencia de datos para el código de la figura 10.11.
- b) ¿Cuáles son todas las rutas críticas en su grafo de la parte (a)?
- c) Si consideramos recursos MEM ilimitados, ¿cuáles son todas las programaciones posibles para las siete instrucciones?

10.4 Programación de código global

Para una máquina con una cantidad moderada de paralelismo a nivel de instrucción, los programas creados por la compactación de bloques básicos individuales tienden a dejar muchos recursos inactivos. Para poder utilizar mejor los recursos de la máquina, es necesario considerar estrategias de generación de código que muevan las instrucciones de un bloque básico a otro. Las estrategias que consideran más de un bloque básico a la vez se conocen como algoritmos de *programación global*. Para realizar la programación global de manera correcta, debemos considerar no sólo las dependencias de datos, sino también las dependencias de control. Debemos asegurarnos de que:

1. Todas las instrucciones en el programa original se ejecuten en el programa optimizado, y
2. Aunque el programa optimizado puede ejecutar instrucciones adicionales en forma especulativa, estas instrucciones no deben tener efectos adicionales no deseados.

10.4.1 Movimiento de código primitivo

Vamos a estudiar primero las cuestiones que tienen que ver con el movimiento de las operaciones, mediante un ejemplo simple.

Ejemplo 10.9: Suponga que tenemos una máquina que puede ejecutar dos operaciones cualesquiera en un solo ciclo de reloj. Cada operación se ejecuta con un retraso de un ciclo de reloj, excepto la operación de carga, que tiene una latencia de dos ciclos. Por cuestión de simplicidad, suponemos que todos los accesos a memoria en el ejemplo son válidos, y que habrá coincidencias en la caché. La figura 10.12(a) muestra un grafo de flujo simple con tres bloques básicos. El código se expande en las operaciones de máquina de la figura 10.12(b). Todas las instrucciones en cada bloque básico deben ejecutarse en serie, debido a las dependencias de datos; de hecho, hay que insertar una instrucción “no-op” en cada bloque básico.

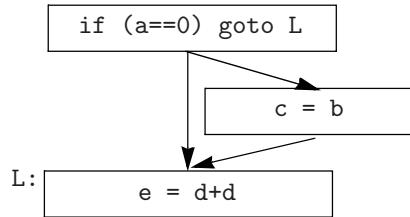
Suponga que las direcciones de las variables a , b , c , d y e son distintas y que estas direcciones se almacenan en los registros del R1 al R5, respectivamente. Por lo tanto, los cálculos de los distintos bloques básicos no comparten dependencias de datos. Observamos que se ejecutan todas las operaciones en el bloque B_3 , sin importar si se toma la bifurcación y, por lo tanto, pueden ejecutarse en paralelo con las operaciones del bloque B_1 . No podemos mover las operaciones de B_1 hacia B_3 , ya que se necesitan para determinar los resultados de la bifurcación.

Las operaciones en el bloque B_2 son dependientes del control en la prueba del bloque B_1 . Podemos realizar la carga de B_2 de manera especulativa en el bloque B_1 sin costo, y ahorrarnos dos ciclos del tiempo de ejecución cada vez que se tome la bifurcación.

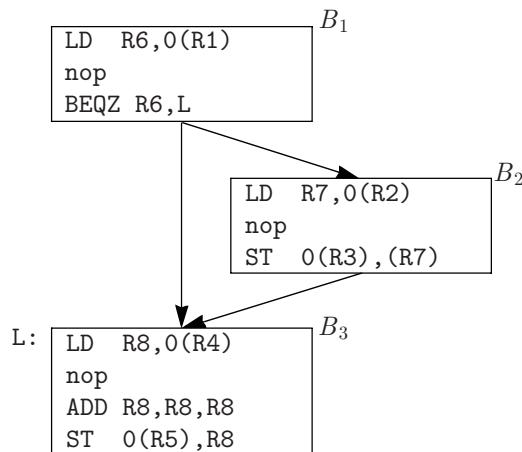
Las operaciones de almacenamiento no deben realizarse en forma especulativa, ya que sobrescriben el antiguo valor en una ubicación en memoria. Sin embargo, es posible retrasar una operación de almacenamiento. No podemos simplemente colocar la operación de almacenamiento del bloque B_2 en el bloque B_3 , ya que sólo debería ejecutarse si el flujo de control pasa a través del bloque B_2 . Sin embargo, podemos colocar la operación de almacenamiento en una copia duplicada de B_3 . La figura 10.12(c) muestra un programa optimizado de este tipo. El código optimizado se ejecuta en 4 ciclos de reloj, que es lo mismo que el tiempo requerido para ejecutar B_3 por sí solo. \square

El ejemplo 10.9 muestra que es posible mover las operaciones a lo largo de una ruta de ejecución. Cada par de bloques básicos en este ejemplo tiene una “relación de predominio” distinta, y por ende son distintas las consideraciones de cuándo y cómo deben moverse las instrucciones entre cada par. Como vimos en la sección 9.6.1, se dice que un bloque B domina al bloque B' si cada ruta proveniente de la entrada del grafo de flujo de control hacia B' pasa a través de B . De manera similar, un bloque B posdomina al bloque B' si cada ruta que va de B' a la salida del grafo pasa a través de B . Cuando B domina a B' y B' posdomina a B , decimos que B y B' son de *control equivalente*, lo cual significa que uno se ejecuta sólo cuando el otro también se ejecuta. Para el ejemplo de la figura 10.12, suponiendo que B_1 sea la entrada y B_3 la salida:

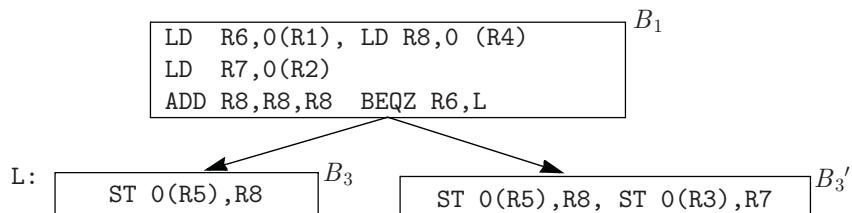
1. B_1 y B_3 son de control equivalente: B_1 domina a B_3 y B_3 posdomina a B_1 .
2. B_1 domina a B_2 , pero B_2 no posdomina a B_1 .



(a) Programa fuente



(b) Código máquina programado en forma local



(c) Código máquina programado en forma global

Figura 10.12: Grafos de flujo antes y después de la programación global en el ejemplo 10.9

3. B_2 no domina a B_3 , pero B_3 posdomina a B_2 .

Es posible también que un par de bloques a lo largo de una ruta no compartan una relación de dominio ni de posdominio.

10.4.2 Movimiento de código hacia arriba

Ahora vamos a examinar con cuidado lo que significa mover una operación hacia arriba en un camino. Suponga que deseamos mover una operación del bloque *org* hacia arriba, por una ruta de flujo de control hasta el bloque *dst*. Asumimos que dicho movimiento no viola ninguna dependencia de datos, y que hace que las rutas a través de *dst* y *org* se ejecuten con más rapidez. Si *dst* domina a *org*, y *org* posdomina a *dst*, entonces la operación que se movió se ejecuta sólo una vez, cuando es debido.

Si *org* no posdomina a *dst*

Entonces, existe un camino que pasa a través de *dst* y que no llega a *org*. En este caso se habría ejecutado una operación adicional. Este movimiento de código es ilegal, a menos que la operación que se movió no tenga efectos adicionales no deseables. Si la operación que se movió se ejecuta “sin costo” (es decir, si utiliza sólo recursos que de otra forma estarían inactivos), entonces este movimiento no tiene costo. Es benéfico sólo si el flujo de control llega a *org*.

Si *dst* no domina a *org*

Entonces, existe un camino que llega a *org* sin pasar primero a través de *dst*. Debemos insertar copias de la operación que se movió a lo largo de dichas rutas. Sabemos cómo lograr exactamente eso debido a nuestra explicación sobre la eliminación de redundancia parcial de la sección 9.5. Colocamos copias de la operación a lo largo de los bloques básicos que forman un conjunto de corte, separando el bloque de entrada de *org*. En cada lugar en el que se inserta la operación, se deben cumplir las siguientes restricciones:

1. Los operandos de la operación deben contener los mismos valores que en la operación original.
2. El resultado no sobrescribe un valor que todavía se necesita.
3. La misma operación no se sobrescribe más adelante, antes de llegar a *org*.

Estas copias hacen que la instrucción original en *org* sea totalmente redundante y, por lo tanto, puede eliminarse.

Nos referimos a las copias adicionales de la operación como *código de compensación*. Como vimos en la sección 9.5, pueden insertarse bloques básicos a lo largo de las aristas críticas con el fin de crear lugares para almacenar dichas copias. El código de compensación puede llegar a provocar que algunas rutas se ejecuten con más lentitud. Por ende, el movimiento de código mejora la ejecución del programa sólo si los caminos optimizados se ejecutan con más frecuencia que las no optimizadas.

10.4.3 Movimiento de código hacia abajo

Suponga que estamos interesados en mover una operación del bloque *org*, hacia abajo por un camino de flujo de control hasta el bloque *dst*. Podemos razonar acerca de dicho movimiento de código en la misma forma que la sección anterior.

Si *org* no domina a *dst*

Entonces, existe un camino que llega a *dst* sin visitar primero a *org*. De nuevo, en este caso se ejecutará una operación adicional. Por desgracia, el movimiento de código hacia abajo se aplica con frecuencia a las operaciones de escritura, que tienen los efectos adicionales de sobrescribir los valores antiguos. Podemos lidiar con este problema mediante la replicación de los bloques básicos a lo largo de las rutas de *org* a *dst*, y colocando la operación sólo en la nueva copia de *dst*. Otro método, si está disponible, es usar instrucciones predicadas. Protegemos la operación que se movió con el predicado que protege al bloque *org*. Tenga en cuenta que la instrucción predicada debe programarse sólo en un bloque dominado por el cálculo del predicado, ya que éste no estará disponible en cualquier otro caso.

Si *dst* no posdomina a *org*

Como en la explicación anterior, hay que insertar código de compensación para que la operación que se movió se ejecute en todas las rutas que no visitan a *dst*. De nuevo, esta transformación es análoga a la eliminación de redundancia parcial, sólo que las copias se colocan debajo del bloque *org* en un conjunto de corte que separa a *org* de la salida.

Resumen del movimiento de código hacia arriba y hacia abajo

De esta explicación podemos ver que hay un rango de posibles movimientos de código globales, que varían en términos de beneficio, costo y complejidad de implementación. La figura 10.13 muestra un resumen de estos diversos movimientos de código; las líneas corresponden a los cuatro casos siguientes:

	arriba: <i>org</i> posdomina a <i>dst</i>	<i>dst</i> domina a <i>org</i>	especulación	código de compensación
	abajo: <i>org</i> domina a <i>dst</i>	<i>dst</i> posdomina a <i>org</i>	dup. de código	
1	sí	sí	no	no
2	no	sí	sí	no
3	sí	no	no	sí
4	no	no	sí	sí

Figura 10.13: Resumen de movimientos de código

1. Es más simple y efectivo en costo mover las instrucciones entre los bloques de control equivalente. Nunca se ejecutan operaciones adicionales y no se necesita código de compensación.

2. Pueden ejecutarse operaciones adicionales si el origen no posdomina (domina) al destino en el movimiento de código hacia arriba (abajo). Este movimiento de código es benéfico si las operaciones adicionales pueden ejecutarse sin costo, y se ejecuta la ruta que pasa a través del bloque de origen.
3. Se requiere código de compensación si el destino no domina (posdomina) al origen en el movimiento de código hacia arriba (abajo). Las rutas con el código de compensación pueden reducir su velocidad, por lo que es importante que las rutas optimizadas se ejecuten con más frecuencia.
4. El último caso combina las desventajas de los casos segundo y tercero: pueden ejecutarse operaciones adicionales y se requiere código de compensación.

10.4.4 Actualización de las dependencias de datos

Como se ilustra mediante el ejemplo 10.10 a continuación, el movimiento de código puede cambiar las relaciones de dependencia de datos entre las operaciones. Por ende, las dependencias de datos se deben actualizar después de cada movimiento de código.

Ejemplo 10.10: Para el grafo de flujo que se muestra en la figura 10.14, cualquiera de las asignaciones a x puede moverse hacia arriba hasta el bloque superior, ya que todas las dependencias en el programa original se preservan con esta transformación. No obstante, una vez que hemos movido una asignación hacia arriba, no podemos mover la otra. Dicho en forma más específica, podemos ver que la variable x no está viva al salir en el bloque superior antes del movimiento de código, pero lo está después del movimiento. Si una variable está viva en un punto del programa, entonces no podemos mover las definiciones especulativas a la variable por encima de ese punto del programa. \square

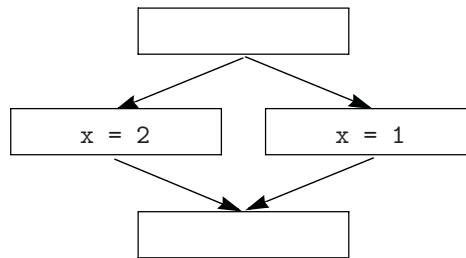


Figura 10.14: Ejemplo que ilustra el cambio en las dependencias de datos, debido al movimiento de código

10.4.5 Algoritmos de programación global

En la última sección vimos que el movimiento de código puede beneficiar a ciertos caminos, mientras que afecta al rendimiento de otras. Lo bueno es que no todas las instrucciones se crean de la misma forma. De hecho, está bien establecido que más del 90% del tiempo de ejecución de

un programa se invierte en menos del 10% del código. Por ende, debemos lograr que los caminos que se ejecutan con frecuencia lo hagan con más rapidez, mientras que existe la posibilidad de que los caminos menos frecuentes se ejecuten con menos velocidad.

Hay una variedad de técnicas que un compilador puede usar para estimar las frecuencias de ejecución. Es razonable suponer que las instrucciones en los ciclos más internos se ejecutan más a menudo que el código en los ciclos externos, y que es más probable tomar las bifurcaciones que van hacia atrás que no hacerlo. Además, es poco probable que se tomen las instrucciones de bifurcación que protegen las salidas del programa o las rutinas de manejo de excepciones. Sin embargo, las mejores estimaciones de frecuencia provienen de los perfiles dinámicos. En esta técnica, los programas se instrumentan para registrar los resultados de las bifurcaciones condicionales, a medida que se ejecutan. Después, los programas se ejecutan con entradas representativas, para determinar cómo se van a comportar en general. Los resultados que se obtienen de esta técnica han demostrado ser bastante precisos. Dicha información puede retroalimentarse al compilador, para que la utilice en sus optimizaciones.

Programación basada en regiones

Ahora describiremos un programador global simple, que soporta las dos formas más sencillas de movimiento de código:

1. Mover las operaciones hacia arriba, a los bloques básicos de control equivalente.
2. Mover las operaciones en forma especulativa una bifurcación hacia arriba, hasta un predecesor dominante.

En la sección 9.7.1 vimos que una región es un subconjunto de un grafo de flujo de control, al cual puede llegar sólo a través de un bloque de entrada. Podemos representar cualquier procedimiento como una jerarquía de regiones. El procedimiento completo constituye la región de nivel superior; en éste se anidan las subregiones que representan a los ciclos naturales en la función. Suponemos que el grafo de control de flujo es reducible.

Algoritmo 10.11: Programación basada en regiones.

ENTRADA: Un grafo de flujo de control y una descripción de los recursos de una máquina.

SALIDA: Un programa S que asigna cada instrucción a un bloque básico y una ranura de tiempo.

MÉTODO: Ejecute el programa de la figura 10.15. Debe ser aparente cierta terminología de abreviación: $ControlEquiv(B)$ es el conjunto de bloques que tiene control equivalente para el bloque B , y $SucDominados$, que se aplica a un conjunto de bloques, es el conjunto de bloques que son sucesores de por lo menos un bloque en el conjunto, y están dominados por todos.

La programación de código en el Algoritmo 10.11 procede de las regiones más internas a las más externas. Al programar una región, cada subregión anidada se trata como una caja negra; no se permite a las instrucciones moverse hacia dentro o hacia fuera de una subregión. Sin embargo, pueden moverse alrededor de una subregión, siempre y cuando se cumpla con sus dependencias de datos y de control.

```

for (cada región  $R$  en orden topológico, de manera que las regiones internas
      se procesen antes que las regiones externas) {
  calcular las dependencias de datos;
  for (cada bloque básico  $B$  de  $R$  en orden topológico priorizado) {
     $BloquesCand = ControlEquiv(B) \cup$ 
       $SucDominados(ControlEquiv(B));$ 
     $InstCand =$  instrucciones listas en  $BloquesCand$ ;
    for ( $t = 0, 1, \dots$  hasta que se programen todas las instrucciones de  $B$ ) {
      for (cada instrucción  $n$  en  $InstCand$  en orden de prioridad)
        if ( $n$  no tiene conflictos de recursos en el tiempo  $t$ ) {
           $S(n) = \langle B, t \rangle;$ 
          actualizar asignaciones de recursos;
          actualizar dependencias de datos;
        }
      actualizar  $InstCand$ ;
    }
  }
}

```

Figura 10.15: Un algoritmo de programación global basado en regiones

Se ignoran todos las aristas de control y de dependencia que fluyen de regreso al encabezado de la región, por lo que los grafos resultantes de flujo de control y dependencia de datos son acíclicos. Los bloques básicos en cada región se visitan en orden topológico. Este ordenamiento garantiza que un bloque básico no se programe sino hasta que se hayan programado todas las instrucciones de las que depende. Las instrucciones que se van a programar en un bloque básico B se arrastran de todos los bloques que son de control equivalente para B (incluyendo a B), así como de todos sus sucesores inmediatos dominados por B .

Se utiliza un algoritmo de programación por lista para crear el programa para cada bloque básico. El algoritmo mantiene una lista de instrucciones candidatas, $InstCand$, que contiene todas las instrucciones en los bloques candidatos cuyos predecesores hayan sido todos programados. Crea el programa de ciclo en ciclo del reloj. Para cada ciclo de reloj, comprueba cada instrucción de $InstCand$ en orden de prioridad y la programa en ese ciclo de reloj, si los recursos lo permiten. Después, el Algoritmo 10.11 actualiza $InstCand$ y repite el proceso, hasta que se programan todas las instrucciones de B .

El orden de prioridad en *InstCand* utiliza una función de prioridad similar a la que vimos en la sección 10.3. No obstante, hicimos una modificación importante. Dimos a las instrucciones de los bloques que tienen control equivalente para B una mayor prioridad que a las de los bloques sucesores. La razón es que las instrucciones en esta última categoría sólo se ejecutan en forma especulativa en el bloque B . \square

Desenrollamiento de ciclos

En la programación basada en regiones, el límite de una iteración de un ciclo es una barrera para el movimiento de código. Las operaciones de una iteración no pueden traslaparse con las de otra. Una técnica simple, pero muy efectiva para mitigar este problema, es la de desenrollar el ciclo un pequeño número de veces antes de la programación del código. Un ciclo for como:

```
for (i = 0; i < N; i++) {
    S(i);
}
```

puede escribirse como en la figura 10.16(a). De manera similar, un ciclo repeat como:

```
repeat
    S;
until C;
```

puede escribirse como en la figura 10.16(b). El desenrollamiento crea más instrucciones en el cuerpo del ciclo, lo cual permite a los algoritmos de programación global encontrar más paralelismo.

```
for (i = 0; i+4 < N; i+=4) {
    S(i);
    S(i+1);
    S(i+2);
    S(i+3);
}
for ( ; i < N; i++) {
    S(i);
}
```

(a) Desenrollamiento de un ciclo for.

```
repeat {
    S;
    if (C) break;
    S;
    if (C) break;
    S;
    if (C) break;
    S;
} until C;
```

(b) Desenrollamiento de un ciclo repeat.

Figura 10.16: Ciclos desenrollados

Compactación de las proximidades

El Algoritmo 10.11 sólo soporta las primeras dos formas de movimiento de código descritas en la sección 10.4.1. Los movimientos de código que requieren la introducción de código de compensación pueden ser útiles algunas veces. Una manera de soportar dichos movimientos de código es colocar una pasada simple después de la programación basada en regiones. En esta pasada, podemos examinar cada par de bloques básicos que se ejecutan uno después del otro, y comprobar si alguna operación puede moverse hacia arriba o hacia abajo entre ellos, para mejorar el tiempo de ejecución de estos bloques. Si se encuentra un par de este tipo, comprobamos si la instrucción que se va a mover necesita duplicarse a lo largo de otras rutas. El movimiento de código se realiza si se produce la ganancia neta esperada.

Esta extensión simple puede ser bastante efectiva para mejorar el rendimiento de los ciclos. Por ejemplo, puede mover una operación al principio de una iteración hasta el final de la siguiente iteración, al tiempo que mueve la operación de la primera iteración hacia fuera del ciclo. Esta optimización es muy atractiva para los ciclos estrechos, que son ciclos que ejecutan sólo algunas instrucciones por cada iteración. Sin embargo, el impacto de esta técnica se limita por el hecho de que cada decisión de movimiento de código se realiza en forma local e independiente.

10.4.6 Técnicas avanzadas de movimiento de código

Si nuestra máquina destino se programa en forma estática y tiene mucho paralelismo a nivel de instrucción, tal vez necesitemos un algoritmo más agresivo. He aquí una descripción de alto nivel de extensiones adicionales:

1. Para facilitar las siguientes extensiones, podemos agregar nuevos bloques básicos a lo largo de los flancos de flujo de control que se originan de bloques con más de un predecesor. Estos bloques básicos se eliminarán al final de la programación de código, si están vacíos. Una heurística útil es mover las instrucciones fuera de un bloque básico que está casi vacío, para que el bloque pueda eliminarse por completo.
2. En el Algoritmo 10.11, el código a ejecutar en cada bloque básico se programa de una vez por todas, a medida que se visita cada bloque. Este método simple es suficiente, ya que el algoritmo sólo puede mover las operaciones hacia arriba, a los bloques dominantes. Para permitir movimientos que requieran la adición de código de compensación, tomamos un método ligeramente distinto. Al visitar el bloque B , sólo programamos las instrucciones provenientes de B y todos sus bloques de control equivalente. Primero tratamos de colocar estas instrucciones en bloques predecesores, que ya se han visitado y para los cuales ya existe un programa parcial. Tratamos de encontrar un bloque de destino que nos conduzca a una mejora en una ruta de ejecución frecuente y después colocamos copias de la instrucción en otras rutas, para garantizar que sea lo correcto. Si las instrucciones no pueden moverse hacia arriba, se programan en el bloque básico actual, como antes.
3. Es más difícil implementar el movimiento de código hacia abajo en un algoritmo que visite los bloques básicos en orden topológico, ya que todavía no se han programado los

bloques de destino. Sin embargo, de cualquier forma hay relativamente menos oportunidades para dicho movimiento de código. Movemos todas las operaciones que:

- (a) puedan moverse, y
- (b) no puedan ejecutarse sin costo en su bloque nativo.

Esta estrategia simple funciona bien si la máquina de destino contiene muchos recursos de hardware sin uso.

10.4.7 Interacción con los programadores dinámicos

Un programador dinámico tiene la ventaja de que puede crear nuevos programas de acuerdo con las condiciones en tiempo de ejecución, sin tener que codificar todos estos posibles programas antes de tiempo. Si una máquina destino tiene un programador dinámico, la función principal del programador estático es asegurar que las instrucciones con alta latencia se obtengan antes de tiempo, para que el programador dinámico pueda emitirlas lo antes posible.

Las fallas en caché son una clase de eventos impredecibles que pueden hacer una gran diferencia en el rendimiento de un programa. Si hay instrucciones de preobtención de datos disponibles, el programador estático puede ayudar al programador dinámico de manera considerable, al colocar estas instrucciones de preobtención con la suficiente anticipación como para que los datos se encuentren en la caché, para cuando se les requiere. Si no hay instrucciones de preobtención disponibles, es útil para un compilador estimar qué operaciones tienen probabilidad de fallar, y tratar de emitirlas antes de tiempo.

Si la programación dinámica no está disponible en la máquina destino, el programador estático debe ser conservador y separar cada par dependiente de datos de operaciones, en base al retraso mínimo. No obstante, si la programación dinámica está disponible, el compilador sólo debe colocar las operaciones dependientes de datos en el orden correcto, para asegurar que el programa esté correcto. Para un mejor rendimiento, el compilador debe asignar retrasos extensos a las dependencias que tienen probabilidad de ocurrir, y retrasos cortos a las dependencias que no tengan probabilidad de ocurrir.

El mal pronóstico de las bifurcaciones es una causa importante de pérdida en el rendimiento. Debido al extenso castigo por mal pronóstico, las instrucciones en rutas que se ejecutan raras veces aún pueden tener un efecto considerable sobre el tiempo total de ejecución. Se debe dar una mayor prioridad a dichas instrucciones, para reducir el costo del mal pronóstico.

10.4.8 Ejercicios para la sección 10.4

Ejercicio 10.4.1: Muestre cómo desenrollar el siguiente ciclo while genérico:

```
while (C)
    S;
```

! Ejercicio 10.4.2: Considere el siguiente fragmento de código:

```
if (x == 0) a = b;
else a = c;
d = a;
```

Suponga que tiene una máquina que utiliza el modelo de retraso del ejemplo 10.6 (las cargas requieren dos ciclos de reloj, y todas las demás instrucciones requieren un ciclo). Suponga además que la máquina puede ejecutar dos instrucciones cualesquiera a la vez. Encuentre una ejecución que sea lo más corta posible para este fragmento. No olvide considerar qué registro se utiliza mejor para cada uno de los pasos de copia. Además, recuerde explotar la información que proporcionan los descriptores de los registros, como se describió en la sección 8.6, para evitar operaciones innecesarias de carga y almacenamiento.

10.5 Canalización por software

Como vimos en la introducción de este capítulo, las aplicaciones numéricas tienen, por lo regular, mucho paralelismo. En especial, a menudo tienen ciclos cuyas iteraciones son por completo independientes unas de otras. Estos ciclos, conocidos como ciclos de *ejecución total (do-all)*, son bastante atractivos desde la perspectiva de la paralelización, ya que sus iteraciones pueden ejecutarse en paralelo para lograr una agilización lineal en cuanto al número de iteraciones en el ciclo. Los ciclos de ejecución total con muchas iteraciones tienen suficiente paralelismo para saturar a todos los recursos en un procesador. Depende del programador si aprovecha o no por completo el paralelismo disponible. Esta sección describe un algoritmo, conocido como *canalización por software*, que programa un ciclo completo a la vez, aprovechando por completo el paralelismo entre las iteraciones.

10.5.1 Introducción

Vamos a usar el ciclo de ejecución total en el ejemplo 10.12 a lo largo de esta sección, para explicar la canalización por software. Primero mostraremos que la programación entre iteraciones es de gran importancia, ya que hay relativamente poco paralelismo entre las operaciones en una sola iteración. A continuación, vamos a mostrar que el desenrollamiento de los ciclos mejora el rendimiento, al traslapar el cálculo de las iteraciones desenrolladas. Sin embargo, el límite del ciclo desenrollado sigue actuando como barrera para el movimiento de código, y el desenrollamiento aún deja mucho rendimiento “en la mesa”. Por otro lado, la técnica de canalización por software traslapa un número de iteraciones consecutivas en forma continua, hasta que se agotan las iteraciones. Esta técnica permite que la canalización por software produzca código compacto y muy eficiente.

Ejemplo 10.12: He aquí un típico ciclo de ejecución total:

```
for (i = 0; i < n; i++)
    D[i] = A[i]*B[i] + c;
```

Las iteraciones en el ciclo anterior escriben en distintas ubicaciones de memoria, que por sí solas son diferentes de cualquiera de las ubicaciones leídas. Por lo tanto, no hay dependencias de memoria entre las iteraciones, y todas pueden proceder en paralelo.

Adoptaremos el siguiente modelo como nuestra máquina destino a lo largo de esta sección. En este modelo:

- La máquina puede emitir en un solo ciclo de reloj: una operación de carga, una de almacenamiento, una aritmética y una bifurcación.
- La máquina tiene una operación de regreso al ciclo de la siguiente forma:

BL R, L

la cual decrementa al registro R y, a menos que el resultado sea 0, bifurca hacia la ubicación L .

- Las operaciones de memoria tienen un modo de direccionamiento con autoincremento, denotado por $++$ después del registro. El registro se incrementa de manera automática para apuntar a la siguiente dirección consecutiva después de cada acceso.
- Las operaciones aritméticas están canalizadas por completo; pueden iniciarse en cada ciclo del reloj, pero sus resultados no están disponibles sino hasta 2 ciclos de reloj después. Todas las demás instrucciones tienen una latencia de un solo ciclo de reloj.

Si las iteraciones se programan una a la vez, el mejor programa que podemos obtener en nuestro modelo de máquina se muestra en la figura 10.17. Algunas suposiciones acerca de la distribución de los datos también se indican en esa figura: los registros $R1$, $R2$ y $R3$ contienen las direcciones de los inicios de los arreglos A , B y D , el registro $R4$ contiene la constante c , y el registro $R10$ contiene el valor $n - 1$, que se ha calculado fuera del ciclo. El cálculo es en su mayor parte serial, y requiere un total de 7 ciclos; sólo la instrucción de regreso al ciclo se traslape con la última operación en la iteración.

```

//  R1, R2, R3 = &A, &B, &D
//  R4          = c
//  R10         = n-1

L:   LD  R5, 0(R1++)
      LD  R6, 0(R2++)
      MUL R7, R5, R6
      nop
      ADD R8, R7, R4
      nop
      ST  0(R3++), R8      BL R10, L

```

Figura 10.17: Código programado en forma local para el ejemplo 10.12

En general, obtenemos un mejor uso del hardware al desenrollar varias iteraciones de un ciclo. Sin embargo, al hacer esto se incrementa el tamaño del código, lo que a su vez puede tener un impacto negativo sobre el rendimiento en general. Por ende, tenemos que comprometernos, elegir un número de veces para desenrollar un ciclo que obtenga la mayoría de la mejora en el rendimiento, y que no expanda el código demasiado. El siguiente ejemplo ilustra esta concesión.

Ejemplo 10.13: Aunque es difícil encontrar paralelismo en cada iteración del ciclo del ejemplo 10.12, hay mucho paralelismo a través de las iteraciones. El desenrollamiento del ciclo coloca varias iteraciones del mismo en un bloque básico grande, y puede utilizarse un algoritmo simple de programación por lista para programar las operaciones a ejecutar en paralelo. Si desenrollamos el ciclo en nuestro ejemplo cuatro veces, y aplicamos el Algoritmo 10.7 al código, podemos obtener el programa que se muestra en la figura 10.18. (Por cuestión de simplicidad, ignoramos los detalles de la asignación de registros por ahora.) El ciclo se ejecuta en 13 ciclos de reloj, o una iteración cada 3.25 ciclos.

Un ciclo que se desenrolla k veces requiere por lo menos $2k + 5$ ciclos de reloj, con lo cual se logra una tasa de transferencia de una iteración por cada $2 + 5/k$ ciclos de reloj. Por ende, entre más iteraciones desenrollemos, más rápido se ejecutará el ciclo. A medida que $n \rightarrow \infty$, un ciclo desenrollado por completo se puede ejecutar en promedio una iteración cada dos ciclos de reloj. Sin embargo, entre más iteraciones desenrollemos, más grande se hará el código. Es evidente que no podemos darnos el lujo de desenrollar todas las iteraciones en un ciclo. Si desenrollamos el ciclo 4 veces se produce código con 13 instrucciones, o un 163% del valor óptimo; si desenrollamos el ciclo 8 veces se produce código con 21 instrucciones, o un 131% del valor óptimo. Por el contrario, si deseamos operar, por decir a sólo un 110% del valor óptimo, debemos desenrollar el ciclo 25 veces, lo cual produciría un código con 55 instrucciones. \square

10.5.2 Canalización de los ciclos mediante software

La canalización por software proporciona una manera conveniente de obtener un uso óptimo de los recursos y un código compacto al mismo tiempo. Vamos a ilustrar esta idea con nuestro ejemplo abierto.

Ejemplo 10.14: En la figura 10.19 está el código del ejemplo 10.12, desenrollado cinco veces. De nuevo, omitiremos la consideración del uso de los registros. En la fila i se muestran todas las operaciones que se emiten en el ciclo de reloj i ; en la columna j se muestran todas las operaciones de la iteración j . Observe que cada iteración tiene el mismo programa relativo a su inicio, y observe también que cada iteración se inicia dos ciclos de reloj después de la que le precede. Es fácil ver que este programa cumple con todas las restricciones de recursos y dependencia de datos.

Podemos observar que las operaciones que se ejecutan en los ciclos 7 y 8 son las mismas que se ejecutan en los ciclos 9 y 10. Los ciclos de reloj 7 y 8 ejecutan las operaciones de las primeras cuatro iteraciones en el programa original. Los relojes 9 y 10 también ejecutan operaciones de cuatro iteraciones, esta vez de las iteraciones 2 a la 5. De hecho, podemos seguir ejecutando el mismo par de instrucciones con múltiples operaciones para obtener el efecto de retirar la iteración más antigua y agregar una nueva, hasta que se agoten las iteraciones.

Dicho comportamiento dinámico puede codificarse brevemente con el código que se muestra en la figura 10.20, si asumimos que el ciclo tiene por lo menos 4 iteraciones. Cada fila en la figura corresponde a una instrucción de máquina. Las líneas 7 y 8 forman un ciclo de dos ciclos de reloj, el cual se ejecuta $n - 3$ veces, en donde n es el número de iteraciones en el ciclo original. \square

```

L:  LD
    LD
        LD
    MUL  LD
        MUL  LD
    ADD   LD
        ADD   LD
    ST    MUL  LD
        ST    MUL
            ADD
            ADD
    ST
        ST    BL (L)

```

Figura 10.18: Código desenrollado para el ejemplo 10.12

Clock	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$
1	LD				
2	LD				
3	MUL	LD			
4		LD			
5		MUL	LD		
6	ADD		LD		
7			MUL	LD	
8	ST	ADD		LD	
9				MUL	LD
10		ST	ADD		LD
11					MUL
12			ST	ADD	
13					
14				ST	ADD
15					
16					ST

Figura 10.19: Cinco iteraciones desenrolladas del código del ejemplo 10.12

```

1)      LD
2)      LD
3)      MUL  LD
4)          LD
5)          MUL  LD
6)      ADD   LD
7) L:          MUL  LD
8)      ST   ADD   LD   BL (L)
9)          MUL
10)     ST   ADD
11)
12)     ST   ADD
13)
14)     ST

```

Figura 10.20: Código con canalización por software para el ejemplo 10.12

La técnica antes descrita se conoce como *canalización por software*, ya que es la analogía de software de una técnica que se utiliza para la programación de canalizaciones por hardware. Podemos considerar el programa ejecutado por cada iteración en este ejemplo como una canalización de 8 etapas. Se puede iniciar una nueva iteración en la canalización cada 2 ciclos de reloj. Al principio, sólo hay una iteración en la canalización. Cuando la primera iteración procede a la etapa tres, la segunda iteración empieza a ejecutarse en la primera etapa de la canalización.

Para el ciclo de reloj 7, la canalización está llena por completo con las primeras cuatro iteraciones. En el estado estable, se ejecutan cuatro iteraciones consecutivas al mismo tiempo. Se inicia una nueva iteración cuando se retira la iteración más antigua en la canalización. Cuando se agotan las iteraciones, la canalización se drena y todas las iteraciones en la canalización se ejecutan hasta completarse. La secuencia de instrucciones que se utiliza para llenar la canalización, las líneas 1 a 6 en nuestro ejemplo, se conoce como el *prólogo*; las líneas 7 y 8 son el *estado estable*; y la secuencia de instrucciones que se utiliza para drenar la canalización, las líneas de la 9 a 14, es el *epílogo*.

Para este ejemplo, sabemos que el ciclo no puede ejecutarse a una velocidad mayor de 2 ciclos de reloj por iteración, ya que la máquina sólo puede emitir una lectura por cada ciclo de reloj, y hay dos lecturas en cada iteración. El ciclo anterior canalizado por software se ejecuta en $2n + 6$ ciclos de reloj, en donde n es el número de iteraciones en el ciclo original. A medida que $n \rightarrow \infty$, la tasa de transferencia se aproxima a la velocidad de una iteración por cada dos ciclos de reloj. Por ende, a diferencia del desenrollamiento, la programación por software tiene el potencial de codificar el programa óptimo con una secuencia de código muy compacta.

Observe que el programa adoptado por cada iteración individual no es el más corto posible. La comparación con el programa optimizado en forma local que se presenta en la figura 10.17 muestra que se introduce un retraso antes de la operación ADD. El retraso se coloca de manera estratégica, para que el programa pueda iniciarse cada dos ciclos de reloj sin conflictos en los recursos. Si nos hubiéramos apegado al programa compactado en forma local, el intervalo de

iniciación tendría que extenderse hasta 4 ciclos de reloj para evitar conflictos en los recursos, y la velocidad de la tasa de transferencia se reduciría a la mitad. Este ejemplo ilustra un principio importante en la programación por canalización: el programa debe elegirse con cuidado para poder optimizar la tasa de transferencia. Un programa compactado en forma local, aunque disminuye al mínimo el tiempo para completar una iteración, puede producir una tasa de transferencia por debajo de la óptima a la hora de canalizarse.

10.5.3 Asignación de recursos y generación de código

Vamos a empezar por hablar sobre la asignación de registros para el ciclo canalizado por software del ejemplo 10.14.

Ejemplo 10.15: En el ejemplo 10.14, el resultado de la operación de multiplicación en la primera iteración se produce en el ciclo de reloj 3 y se utiliza en el ciclo 6. Entre estos ciclos de reloj se genera un nuevo resultado debido a la operación de multiplicación en la segunda iteración, en el ciclo de reloj 5; este valor se utiliza en el ciclo 8. Los resultados de estas dos iteraciones deben guardarse en registros diferentes para evitar que interfieran uno con el otro. Como la interferencia ocurre sólo entre pares adyacentes de iteraciones, puede evitarse con el uso de dos registros, uno para las iteraciones impares y otro para las iteraciones pares. Como el código para las iteraciones impares es distinto del de las pares, se duplica el tamaño del ciclo en el estado estable. Este código puede usarse para ejecutar cualquier ciclo que tenga un número impar de iteraciones mayor o igual a 5.

```

if (N >= 5)
    N2 = 3 + 2 * floor((N-3)/2);
else
    N2 = 0;
for (i = 0; i < N2; i++)
    D[i] = A[i]* B[i] + c;
for (i = N2; i < N; i++)
    D[i] = A[i]* B[i] + c;

```

Figura 10.21: Desenrollamiento a nivel de código fuente del ciclo del ejemplo 10.12

Para manejar ciclos que tengan menos de 5 iteraciones y ciclos con un número par de iteraciones, generamos el código cuyo equivalente a nivel de código fuente se muestra en la figura 10.21. El primer ciclo está canalizado, como puede verse en el equivalente a nivel de máquina de la figura 10.22. El segundo ciclo de la figura 10.21 no necesita optimizarse, ya que puede iterar cuando mucho cuatro veces. \square

10.5.4 Ciclos de ejecución cruzada

La canalización por software también puede aplicarse a los ciclos cuyas iteraciones comparten dependencias de datos. Dichos ciclos se conocen como *ciclos de ejecución cruzada*.

```

1.      LD R5,0(R1++)
2.      LD R6,0(R2++)
3.      LD R5,0(R1++)  MUL R7,R5,R6
4.      LD R6,0(R2++)
5.      LD R5,0(R1++)  MUL R9,R5,R6
6.      LD R6,0(R2++)  ADD R8,R7,R4
7.  L:   LD R5,0(R1++)  MUL R7,R5,R6
8.      LD R6,0(R2++)  ADD R8,R9,R4  ST 0(R3++),R8
9.      LD R5,0(R1++)  MUL R9,R5,R6
10.     LD R6,0(R2++)  ADD R8,R7,R4  ST 0(R3++),R8  BL R10,L
11.          MUL R7,R5,R6
12.          ADD R8,R9,R4  ST 0(R3++),R8
13.
14.          ADD R8,R7,R4  ST 0(R3++),R8
15.
16.          ST 0(R3++),R8

```

Figura 10.22: El código después de la canalización por software y la asignación de recursos en el ejemplo 10.15

Ejemplo 10.16: El siguiente código:

```

for (i = 0; i < n; i++) {
    suma = suma + A[i];
    B[i] = A[i] * b;
}

```

tiene una dependencia de datos entre iteraciones consecutivas, ya que el valor anterior de **suma** se suma a $A[i]$ para crear un nuevo valor de **suma**. Es posible ejecutar la suma en un tiempo determinado por $O(\log n)$ si la máquina puede producir el paralelismo suficiente, pero por el bien de esta explicación, sólo asumiremos que se deben obedecer todas las dependencias secuenciales, y que todas las sumas deben realizarse en el orden secuencial original. Como nuestro supuesto modelo de máquina requiere dos ciclos de reloj para completar una operación ADD, el ciclo no se puede ejecutar con más rapidez que la de una iteración por cada dos ciclos de reloj. Si proporcionamos a la máquina más sumadores o multiplicadores, el ciclo no se ejecutará más rápido. La tasa de transferencia de los ciclos de ejecución cruzada como éste se limita en base a la cadena de dependencias a través de las iteraciones.

En la figura 10.23(a) se muestra el mejor programa compactado en forma local para cada iteración, y en la figura 10.23(b) el código con canalización por software. Este ciclo canalizado por software empieza una iteración cada dos ciclos, y por ende opera a la velocidad óptima. \square

```

// R1 = &A; R2 = &B
// R3 = suma
// R4 = b
// R10 = n-1

L: LD R5, 0(R1++)
    MUL R6, R5, R4
    ADD R3, R3, R4
    ST R6, 0(R2++)
    BL R10, L

```

(a) El mejor programa compactado en forma local.

```

// R1 = &A; R2 = &B
// R3 = suma
// R4 = b
// R10 = n-2

LD R5, 0(R1++)
MUL R6, R5, R4
L: ADD R3, R3, R4      LD R5, 0(R1++)
    ST R6, 0(R2++)    MUL R6, R5, R4    BL R10, L
                        ADD R3, R3, R4
                        ST R6, 0(R2++)

```

(b) La versión canalizada por software.

Figura 10.23: Canalización por software de un ciclo de ejecución cruzada

10.5.5 Objetivos y restricciones de la canalización por software

El principal objetivo de la canalización por software es incrementar al máximo la tasa de transferencia de un ciclo de larga duración. Un objetivo secundario es mantener el código generado de un tamaño razonablemente pequeño. En otras palabras, el ciclo canalizado por software debe tener un pequeño estado estable de la canalización. Podemos lograr un pequeño estado estable al requerir que el programa relativo de cada iteración sea el mismo, y que las iteraciones se inicien en un intervalo constante. Como la tasa de transferencia del ciclo es simplemente el inverso del intervalo de iniciación, el objetivo de la canalización por software es minimizar este intervalo.

Un programa de canalización por software para un grafo de dependencia de datos $G = (N, E)$ puede especificarse mediante:

1. Un intervalo de iniciación T .
2. Un programa S relativo que especifique, para cada operación, cuándo se va a ejecutar esa operación, en forma relativa al inicio de la iteración a la cual pertenece.

Así, una operación n en la i -ésima iteración, contando a partir de 0, se ejecuta en el ciclo $i \times T + S(n)$. Al igual que todos los demás problemas de programación, la canalización por software tiene dos tipos de restricciones: recursos y dependencias de datos. A continuación hablaremos sobre cada una de ellas con detalle.

Reservación modular de recursos

Hagamos que los recursos de una máquina se representen mediante $R = [r_1, r_2, \dots]$, en donde r_i es el número de unidades del i -ésimo tipo de recurso disponible. Si una iteración de un ciclo requiere n_i unidades del recurso i , entonces el intervalo de iniciación promedio de un ciclo canalizado es, por lo menos, de $\max_i(n_i/r_i)$ ciclos de reloj. La canalización por software requiere que los intervalos de iniciación entre cualquier par de iteraciones tengan un valor constante. Por ende, el intervalo de iniciación debe tener por lo menos $\max_i[n_i/r_i]$ ciclos de reloj. Si $\max_i(n_i/r_i)$ es menor que 1, es útil desenrollar el código fuente un pequeño número de veces.

Ejemplo 10.17: Vamos a regresar a nuestro ciclo canalizado por software, que se muestra en la figura 10.20. Recuerde que la máquina destino puede emitir una operación de carga, una operación aritmética, una de almacenamiento y una bifurcación de regreso de ciclo, por cada ciclo de reloj. Como el ciclo tiene dos cargas, dos operaciones aritméticas y una operación de almacenamiento, el intervalo mínimo de iniciación basado en las restricciones de los recursos es de 2 ciclos de reloj.

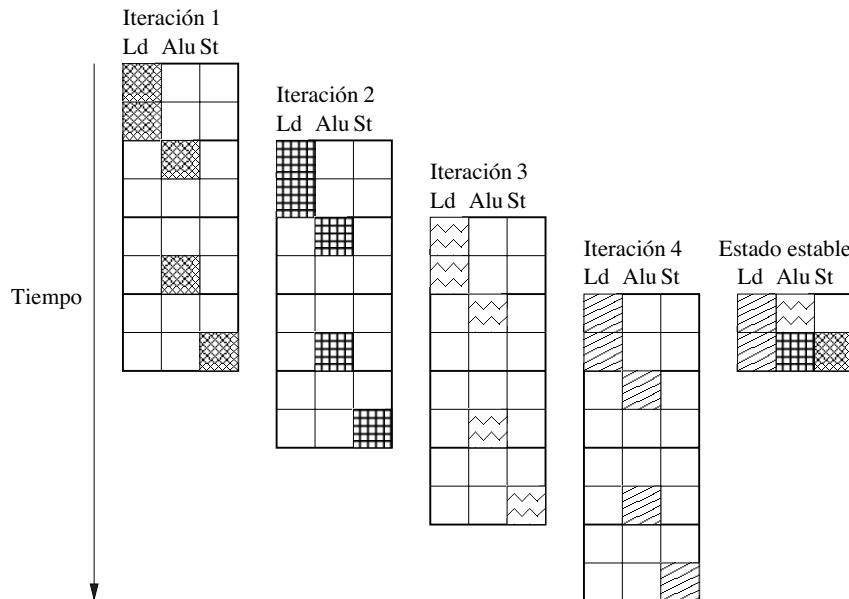


Figura 10.24: Requerimientos de recursos de cuatro iteraciones consecutivas del ejemplo en la figura 10.13

La figura 10.24 muestra los requerimientos de recursos de cuatro iteraciones consecutivas a través del tiempo. Se utilizan más recursos a medida que se inician más iteraciones, culminando

en la asignación máxima de recursos en el estado estable. Suponga que RT es la tabla de reservación de recursos que representa la asignación de una iteración, y suponga que RT_S representa la asignación del estado estable. RT_S combina la asignación de cuatro iteraciones consecutivas que empiezan T ciclos de reloj aparte. La asignación de la fila 0 en la tabla RT_S corresponde a la suma de los recursos asignados en $RT[0]$, $RT[2]$, $RT[4]$ y $RT[6]$. De manera similar, la asignación de la fila 1 en la tabla corresponde a la suma de los recursos asignados en $RT[1]$, $RT[3]$, $RT[5]$ y $RT[7]$. Es decir, los recursos asignados en la i -ésima fila en el estado estable se dan mediante:

$$RT_S[i] = \sum_{\{t \mid (t \bmod 2)=i\}} RT[t].$$

Nos referimos a la tabla de reservación de recursos que representan el estado estable como la *tabla de reservación de recursos modular* del ciclo canalizado.

Para comprobar si el programa de canalización por software tiene conflictos de recursos, podemos simplemente comprobar la asignación de la tabla de reservación de recursos. Sin duda, si puede cumplirse la asignación en el estado estable, también se pueden cumplir las asignaciones en el prólogo y el epílogo, las porciones de código antes y después del ciclo en estado estable. \square

En general, dado un intervalo de iniciación T y una tabla de reservación de recursos de una iteración RT , el programa canalizado no tiene conflictos de recursos en una máquina con el vector de recursos R , si y sólo si $RT_S[i] \leq R$ para todas las $i = 0, 1, \dots, T - 1$.

Restricciones de dependencia de datos

Las dependencias de datos en la canalización por software son distintas de las que nos hemos encontrado hasta ahora, debido a que pueden formar ciclos. Una operación puede depender en el resultado de la misma operación de una iteración anterior. Ya no es adecuado etiquetar una arista de dependencia sólo con base en el retraso; también debemos diferenciar entre las instancias de la misma operación en distintas iteraciones. Etiquetamos una arista de dependencia $n_1 \rightarrow n_2$ con la etiqueta $\langle \delta, d \rangle$ si la operación n_2 en la iteración i debe retrasarse por lo menos d ciclos de reloj después de la ejecución de la operación n_1 en la iteración $i - \delta$. Suponga que S , una función que va de los nodos del grafo de dependencia de datos a enteros, es el programa de canalización por software, y suponga que T es el destino del intervalo de iniciación. Entonces:

$$(\delta \times T) + S(n_2) - S(n_1) \geq d.$$

La diferencia de la iteración, δ , debe ser no negativa. Además. Dado un ciclo de aristas de dependencia de datos, por lo menos una de las aristas tiene una diferencia de iteración positiva.

Ejemplo 10.18: Considere el siguiente ciclo, y suponga que no conocemos los valores de p y q :

```
for (i = 0; i < n; i++)
    *(p++) = *(q++) + c;
```

Debemos suponer que cualquier par de accesos $*(p++)$ y $*(q++)$ puede referirse a la misma ubicación de memoria. Por ende, todas las lecturas y escrituras deben ejecutarse en el mismo orden secuencial original. Suponiendo que la máquina destino tiene las mismas características que las descritas en el ejemplo 10.13, los flancos de dependencia de datos para este código son como se muestra en la figura 10.25. Sin embargo, observe que ignoramos las instrucciones de control de ciclo que tendrían que estar presentes, ya sea calculando y evaluando i , o realizando la prueba con base en el valor de R1 o R2. \square

```
// R1, R2 = q, p
// R3 = c
```

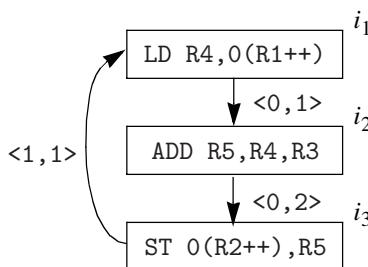


Figura 10.25: Grafo de dependencia de datos para el ejemplo 10.18

La diferencia de iteración entre las operaciones relacionadas puede ser mayor que uno, como se muestra en el siguiente ejemplo:

```
for (i = 2; i < n; i++)
    A[i] = B[i] + A[i-2]
```

Aquí, el valor escrito en la iteración i se utiliza dos iteraciones después. La arista de dependencia entre la operación de almacenamiento de $A[i]$ y la carga de $A[i - 2]$ tiene, por lo tanto, una diferencia de 2 iteraciones.

La presencia de ciclos de dependencia de datos en un ciclo impone otro límite más en cuanto a su tasa de transferencia de ejecución. Por ejemplo, el ciclo de dependencia de datos en la figura 10.25 impone un retraso de 4 ciclos de reloj entre las operaciones de carga de las iteraciones consecutivas. Es decir, los ciclos no pueden ejecutarse a una velocidad mayor que una iteración cada 4 ciclos de reloj.

El intervalo de iniciación de un ciclo canalizado no es menor que:

$$\max_{c \text{ es un ciclo en } G} \left\lceil \frac{\sum_{e \text{ en } c} d_e}{\sum_{e \text{ en } c} \delta_e} \right\rceil$$

En resumen, el intervalo de iniciación de cada ciclo canalizado por software está delimitado por el uso de recursos en cada iteración. Es decir, el intervalo de iniciación no debe ser menor que la proporción de unidades necesarias de cada recurso y las unidades disponibles en la máquina. Además, si los ciclos tienen ciclos de dependencia de datos, entonces el intervalo de

iniciación se restringe aún más debido a la suma de los retrasos en el ciclo, dividida por la suma de las diferencias de iteración. La mayor de estas cantidades define un límite inferior sobre el intervalo de iniciación.

10.5.6 Un algoritmo de canalización por software

El objetivo de la canalización por software es encontrar un programa con el menor intervalo de iniciación posible. El problema es NP-completo, y puede formularse como un problema de programación lineal de enteros. Hemos mostrado que si conocemos el intervalo de iniciación mínimo, el algoritmo de programación puede evitar conflictos de recursos mediante el uso de la tabla de reservación de recursos modular para colocar cada operación. Pero no sabremos cuál es el intervalo de iniciación mínimo sino hasta que podamos encontrar un programa. ¿Cómo resolvemos esta circularidad?

Sabemos que el intervalo de iniciación debe ser mayor que el límite calculado a partir del requerimiento de recursos de un ciclo y los ciclos de dependencia, como vimos antes. Si podemos encontrar un programa que cumpla con este límite, hemos encontrado el programa óptimo. Si no podemos encontrar un programa así, podemos intentar de nuevo con intervalos de iniciación más largos hasta encontrar un programa. Observe que si se utiliza la heurística en vez de la búsqueda exhaustiva, tal vez este proceso no encuentre el programa óptimo.

La probabilidad de encontrar un programa cerca del límite inferior depende de las propiedades del grafo de dependencia de datos y de la arquitectura de la máquina de destino. Podemos encontrar con facilidad el programa óptimo si el grafo de dependencia es acíclico y si cada instrucción de máquina sólo necesita una unidad de un recurso. También es fácil encontrar un programa cerca del límite inferior si hay más recursos de hardware de los que pueden usarse por los grafos con ciclos de dependencia. Para tales casos, es aconsejable empezar con el límite inferior como el destino del intervalo de iniciación inicial, y después seguir incrementando el destino mediante sólo un ciclo de reloj con cada intento de programación. Otra posibilidad es buscar el intervalo de iniciación usando una búsqueda binaria. Podemos emplear como un límite superior sobre el intervalo de iniciación la longitud del programa para una iteración producida mediante la programación por listas.

10.5.7 Programación de grafos de dependencia de datos acíclicos

Por cuestión de simplicidad, vamos a suponer por ahora que el ciclo que se va a canalizar por software contiene sólo un bloque básico. Haremos esta suposición más flexible en la sección 10.5.11.

Algoritmo 10.19: Canalización por software de un grafo de dependencia acíclico.

ENTRADA: Un vector de recursos de máquina $R = [r_1, r_2, \dots]$, en donde r_i es el número de unidades disponibles del i -ésimo tipo de recurso, y un grafo de dependencia de datos $G = (N, E)$. Cada operación n en N se etiqueta con su tabla de reservación de recursos $R T_n$; cada flanco $e = n_1 \rightarrow n_2$ en E se etiqueta con $\langle \delta_e, d_e \rangle$, lo cual indica que n_2 no se debe ejecutar antes de d_e ciclos después del nodo n_1 , de la δ_e -ésima iteración siguiente.

SALIDA: Un programa S canalizado por software y un intervalo de iniciación T .

MÉTODO: Ejecute el programa de la figura 10.26. \square

```

main() {
     $T_0 = \max_j \left\lceil \frac{\sum_{n,i} RT_n(i,j)}{r_j} \right\rceil;$ 
    for ( $T = T_0, T_0 + 1, \dots$ , hasta que se programen todos los nodos en  $N$ ) {
         $RT$  = una tabla de reservación vacía con  $T$  filas;
        for (cada  $n$  en  $N$ , en orden topológico priorizado) {
             $s_0 = \max_{e=p \rightarrow n \text{ en } E} (S(p) + d_e);$ 
            for ( $s = s_0, s_0 + 1, \dots, s_0 + T - 1$ )
                if ( $NodoProgramado(RT, T, n, s)$  break;
                if ( $n$  no puede programarse en  $RT$ ) break;
            }
        }
    }
}

NodoProgramado( $RT, T, n, s$ ) {
     $RT' = RT;$ 
    for (cada fila  $i$  en  $RT_n$ )
         $RT'[(s + i) \bmod T] = RT'[(s + i) \bmod T] + RT_n[i];$ 
    if (para todas las  $i$ ,  $RT'(i) \leq R$ ) {
         $RT = RT';$ 
         $S(n) = s;$ 
        return true;
    }
    else return false;
}

```

Figura 10.26: Algoritmo de canalización por software para los grafos acíclicos

El Algoritmo 10.19 canaliza por software los grafos de dependencia de datos acíclicos. El algoritmo busca primero un límite en el intervalo de iniciación, T_0 , con base en los requerimientos de recursos de las operaciones en el grafo. Después trata de encontrar un programa canalizado por software, empezando con T_0 como el intervalo de iniciación de destino. El algoritmo se repite con intervalos de iniciación cada vez más grandes si no puede encontrar un programa.

El algoritmo utiliza un método de programación por lista en cada intento. Utiliza una tabla RT de reservación de recursos modular para llevar la cuenta de la asignación de recursos en el estado estable. Las operaciones se programan en orden topológico, para que las dependencias de datos siempre puedan satisfacerse mediante las operaciones de retraso. Para programar una operación, primero busca un límite inferior s_0 de acuerdo con las restricciones de dependencia de datos. Después invoca a $NodoProgramado$ para comprobar posibles conflictos de recursos en el estado estable. Si hay un conflicto de recursos, el algoritmo trata de programar la operación en el siguiente ciclo de reloj. Si se descubre que la operación tiene conflictos durante

T ciclos de reloj consecutivos, debido a la naturaleza modular de la detección de conflictos de recursos, se garantiza que los siguientes intentos serán en vano. En ese punto, el algoritmo considera el intento como una falla y se prueba otro intervalo de iniciación.

La heurística de las operaciones de programación tiende lo más pronto posible a disminuir al mínimo la longitud del programa para una iteración. Sin embargo, programar una instrucción lo más pronto posible, puede alargar los tiempos de vida de algunas variables. Por ejemplo, las cargas de datos tienden a programarse antes, algunas veces mucho antes de utilizarse. Una heurística simple es la de programar el grafo de dependencia al revés, ya que, por lo general, hay más cargas que almacenamientos.

10.5.8 Programación de grafos de dependencia cíclicos

Los ciclos de dependencia complican la canalización por software de manera considerable. Cuando se programan operaciones en un grafo acíclico en orden topológico, las dependencias de datos con las operaciones programadas pueden imponer sólo un límite inferior en cuanto a la colocación de cada operación. Como resultado, siempre es posible satisfacer las restricciones de dependencia de datos al retrasar las operaciones. El concepto de “orden topológico” no se aplica a los grafos acíclicos. De hecho, dado un par de operaciones que comparten un ciclo, al colocar una operación se impondrá tanto un límite inferior como superior en la colocación de la segunda operación.

Hagamos que n_1 y n_2 sean dos operaciones en un ciclo de dependencia, que S sea un programa de canalización por software, y que T sea el intervalo de iniciación para el programa. Una arista de dependencia $n_1 \rightarrow n_2$ con la etiqueta $\langle \delta_1, \delta_1 \rangle$ impone la siguiente restricción sobre $S(n_1)$ y $S(n_2)$:

$$(\delta_1 \times T) + S(n_2) - S(n_1) \geq d_1.$$

De manera similar, un flanco de dependencia (n_1, n_2) con la etiqueta $\langle \delta_2, d_2 \rangle$ impone la siguiente restricción:

$$(\delta_2 \times T) + S(n_1) - S(n_2) \geq d_2.$$

Así,

$$S(n_1) + d_1 - (\delta_1 \times T) \leq S(n_2) \leq S(n_1) - d_2 + (\delta_2 \times T).$$

Un *componente fuertemente conectado* (Strongly Connected Component, SCC) en un grafo es un conjunto de nodos, en donde se puede llegar a cada uno en el componente desde cualquier otro nodo en el mismo. La programación de un nodo en un SCC limitará el tiempo de los demás nodos en el componente, tanto de arriba como de abajo. Por transición, si existe una ruta p que conduce de n_1 a n_2 , entonces:

$$S(n_2) - S(n_1) \geq \sum_{e \text{ en } p} (d_e - (\delta_e \times T)) \quad (10.1)$$

Observe que:

- Alrededor de cualquier ciclo, la suma de las δ debe ser positiva. Si fuera 0 o negativa, entonces indicaría que una operación en el ciclo tendría que precederse a sí misma o ejecutarse en el mismo ciclo de reloj para todas las iteraciones.
- El programa de operaciones dentro de una iteración es igual para todas las iteraciones; en esencia, ese requerimiento es el significado de una “canalización por software”. Como resultado, la suma de los retrasos (segundos componentes de las etiquetas de las aristas en un grafo de dependencia de datos) alrededor de un ciclo es un límite inferior sobre el intervalo de iniciación T .

Cuando combinamos estos dos puntos, podemos ver que para cualquier intervalo de iniciación T viable, el valor del lado derecho de la ecuación (10.1) debe ser negativo o cero. Como resultado, las restricciones más fuertes en la colocación de los nodos se obtienen de los caminos *simples*; aquellas rutas que no contienen ciclos.

Así, para cada T viable, calcular el efecto transitivo de las dependencias de datos en cada par de nodos equivale a encontrar la longitud de la ruta simple más larga, desde el primer nodo hasta el segundo. Además, como los ciclos no pueden aumentar la longitud de un camino, podemos usar un algoritmo simple de programación dinámica para encontrar los caminos más largos sin el requerimiento del “camino simple”, y estar seguros de que las longitudes resultantes también serán las longitudes de los caminos simples más largos (vea el ejercicio 10.5.7).

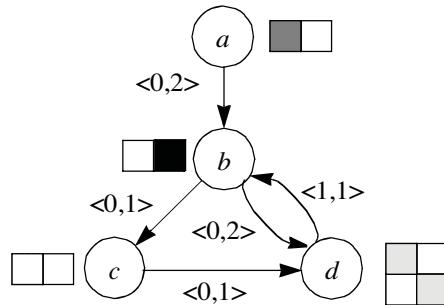


Figura 10.27: Grafo de dependencia y requerimiento de recursos del ejemplo 10.20

Ejemplo 10.20: La figura 10.27 muestra un grafo de dependencia de datos con cuatro nodos a, b, c, d . A cada nodo se le adjunta su tabla de reservación de recursos; a cada arista se le adjunta su diferencia de iteración y su retraso. Para este ejemplo, suponga que la máquina de destino tiene una unidad de cada tipo de recurso. Como hay tres usos del primer recurso y dos del segundo, el intervalo de iniciación no debe ser menor de 3 ciclos de reloj. Hay dos SCCs en este grafo: el primero es un componente trivial que consiste en el nodo a por sí solo, y el segundo consiste en los nodos b, c y d . El ciclo más largo, b, c, d, b , tiene un retraso total de 3 ciclos de reloj que conecta a los nodos que están una iteración aparte. Por ende, el límite inferior en el intervalo de iniciación producido por las restricciones del ciclo de dependencia de datos es también de 3 ciclos de reloj.

Al colocar uno de los nodos b , c o d en un programa, se restringe a todos los demás nodos en el componente. Suponga que T es el intervalo de iniciación. La figura 10.28 muestra las dependencias transitivas. La parte (a) muestra el retraso y la diferencia de iteración δ , para cada arista. El retraso se representa directamente, pero δ se representa “sumando” al retraso el valor $-\delta T$.

La figura 10.28(b) muestra la longitud del camino simple más largo entre dos nodos, si es que existe dicho camino; sus entradas son las sumas de las expresiones dadas por la figura 10.28(a), para cada flanco a lo largo del camino. Por ende, en (c) y (d) podemos ver las expresiones de (b) con los dos valores relevantes de T , es decir, 3 y 4, que se sustituyen por T . La diferencia entre el programa de dos nodos $S(n_2) - S(n_1)$ no debe ser menor que el valor que se proporciona en la entrada (n_1, n_2) en cada una de las tablas (c) o (d), dependiendo del valor elegido de T .

Por ejemplo, considere la entrada en la figura 10.28 para el camino (simple) más largo de c a b , que es $2 - T$. El camino simple más largo de c a b es $c \rightarrow d \rightarrow b$. El retraso total es de 2 a lo largo de este camino, y la suma de las δ es 1, lo cual representa el hecho de que el número de la iteración debe incrementarse en 1. Como T es el tiempo durante el cual cada iteración sigue a la anterior, el ciclo de reloj en el cual b debe programarse es por lo menos de $2 - T$ ciclos de reloj *después* del ciclo de reloj en el que se programa c . Como T es por lo menos de 3, en realidad estamos diciendo que b puede programarse $T - 2$ ciclos de reloj *antes* que c , o después de ese ciclo de reloj, pero no antes.

Observe que si consideramos rutas no simples de c a b , no se produce una restricción más fuerte. Podemos agregar al camino $c \rightarrow d \rightarrow b$ cualquier número de iteraciones del ciclo que involucren a d y b . Si agregamos k ciclos de ese tipo, obtenemos una longitud de $2 - T + k(3 - T)$, ya que el retraso total a lo largo del camino es de 3, y la suma de las δ es 1. Como $T \geq 3$, esta longitud no puede exceder a $2 - T$; es decir, el límite inferior más fuerte en el reloj de b relativo al ciclo de reloj de c es de $2 - T$, el límite que obtenemos al considerar el camino simple más largo.

Por ejemplo, de las entradas (b, c) y (c, b) , podemos ver que:

$$\begin{aligned} S(c) - S(b) &\geq 1 \\ S(b) - S(c) &\geq 2 - T. \end{aligned}$$

Es decir,

$$S(b) + 1 \leq S(c) \leq S(b) - 2 + T.$$

Si $T = 3$,

$$S(b) + 1 \leq S(c) \leq S(b) + 1.$$

De manera equivalente, c debe programarse un ciclo de reloj después de b . No obstante, si $T = 4$,

$$S(b) + 1 \leq S(c) \leq S(b) + 2.$$

Es decir, c se programa uno o dos ciclos de reloj después de b .

Dada toda la información del camino más largo de todos los puntos, podemos calcular con facilidad el rango en donde es válido colocar un nodo debido a las dependencias de datos. Podemos ver que no hay espacio cuando $T = 3$, y que el espacio aumenta a medida que T se incrementa. \square

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
<i>a</i>		2		
<i>b</i>			1	2
<i>c</i>				1
<i>d</i>		$1-T$		

(a) Aristas originales.

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
<i>a</i>		2	3	4
<i>b</i>			1	2
<i>c</i>		$2-T$		1
<i>d</i>		$1-T$	$2-T$	

(b) Caminos simples más largos.

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
<i>a</i>		2	3	4
<i>b</i>			1	2
<i>c</i>		-1		1
<i>d</i>		-2	-1	

(c) Caminos simples más largos ($T=3$).

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
<i>a</i>		2	3	4
<i>b</i>			1	2
<i>c</i>		-2		1
<i>d</i>		-3	-2	

(d) Caminos simples más largos ($T=3$).

Figura 10.28: Dependencias transitivas del ejemplo 10.20

Algoritmo 10.21: Canalización por software.

ENTRADA: Un vector de recursos de máquina $R = [r_1, r_2, \dots]$, en donde r_i es el número de unidades disponibles en el i -ésimo tipo de recurso, y un grafo de dependencia de datos $G = (N, E)$. Cada operación n en N se etiqueta con su tabla de reservación de recursos RT_n ; cada arista $e = n_1 \rightarrow n_2$ en E se etiqueta con (δ_e, d_e) , indicando que n_2 no debe ejecutarse antes de d_e ciclos después del nodo n_1 , de la δ_e -ésima iteración siguiente.

SALIDA: Un programa S canalizado por software y un intervalo de iniciación T .

MÉTODO: Ejecute el programa de la figura 10.29. \square

El Algoritmo 10.21 tiene una estructura de alto nivel similar a la del Algoritmo 10.19, que sólo maneja grafos acíclicos. El intervalo de iniciación mínimo en este caso se delimita no sólo por los requerimientos de recursos, sino también por los ciclos de dependencia de datos en el grafo, el cual se programa un componente fuertemente conectado a la vez. Al tratar a cada componente como una unidad, las aristas entre estos componentes forman necesariamente un grafo acíclico. Mientras que el ciclo de nivel superior en el Algoritmo 10.19 programa los nodos en el grafo en orden topológico, el ciclo de nivel superior en el Algoritmo 10.21 programa los componentes fuertemente conectados en orden topológico. Como antes, si el algoritmo no puede programar todos los componentes, entonces se prueba con un intervalo de iniciación más largo. Observe que el Algoritmo 10.21 se comporta en forma idéntica al Algoritmo 10.19, si recibe un grafo de dependencia de datos acíclico.

El Algoritmo 10.21 calcula dos conjuntos más de aristas: E' es el conjunto de todos las aristas cuya diferencia de iteración es 0, E^* representa a todos los flancos de ruta más larga de todos

```

main() {
     $E' = \{e \mid e \text{ en } E, \delta_e = 0\};$ 
     $T_0 = \max_j \left( \max \left[ \frac{\sum_{n,i} RT_n(i, j)}{r_j} \right], \max_c \left[ \frac{\sum_{e \text{ en } c} d_e}{\sum_{e \text{ en } c} \delta_e} \right] \right);$ 
    for ( $T = T_0, T_0 + 1, \dots$  o hasta que se programen todos los SCCs en  $G$ ) {
         $RT =$  una tabla de reservación vacía con  $T$  filas;
         $E^* = RutaMasLargaTodosLosPares(G, T);$ 
        for (cada SCC  $C$  en  $G$ , en orden topológico priorizado) {
            for (todas las  $n$  en  $C$ )
                 $s_0(n) = \max_{e=p \rightarrow n \text{ en } E^*, p \text{ programado}} (S(p) + d_e);$ 
                 $primero =$  cierta  $n$  tal que  $s_0(n)$  sea un mínimo;
                 $s_0 = s_0(primero);$ 
                for ( $s = s_0; s < s_0 + T; s = s + 1$ )
                    if ( $SccProgramado(RT, T, C, primero, s)$ ) break;
                    if ( $C$  no puede programarse en  $RT$ ) break;
            }
        }
    }

    if ( $SccProgramado(RT, T, c, primero, s)$ ) {
         $RT' = RT;$ 
        if (not  $NodoProgramado(RT', T, primero, s)$ ) return false;
        for (cada  $n$  restante en  $c$ , en orden topológico
            priorizado de aristas en  $E'$ ) {
             $s_t = \max_{e=n' \rightarrow n \text{ en } E^*, n' \text{ en } c, n' \text{ programado}} S(n') + d_e - (\delta_e \times T);$ 
             $s_u = \min_{e=n \rightarrow n' \text{ en } E^*, n' \text{ en } c, n' \text{ programado}} S(n') - d_e + (\delta_e \times T);$ 
            for ( $s = s_l \leq \min(s_u, s_l + T - 1); s = s + 1$ )
                if  $NodoProgramado(RT', T, n, s)$  break;
            if ( $n$  no puede programarse en  $RT'$ ) return false;
        }
         $RT = RT';$ 
        return true;
    }
}

```

Figura 10.29: Un algoritmo de canalización por software para las grafos de dependencia cílicos

los puntos. Es decir, para cada par de nodos (p, n) , hay una arista e en E^* cuya distancia d_e asociada es la longitud del camino simple más largo de p a n , siempre y cuando haya por lo menos un camino de p a n . E^* se calcula para cada valor de T , el destino del intervalo de iniciación. También es posible realizar este cálculo sólo una vez con un valor simbólico de T y después sustituir para T en cada iteración, como hicimos en el ejemplo 10.20.

El Algoritmo 10.20 utiliza el rastreo hacia atrás. Si no puede programar un SCC, trata de reprogramar todo el SCC un ciclo de reloj después. Estos intentos de programación continúan hasta T ciclos de reloj. El rastreo hacia atrás es importante ya que, como se muestra en el ejemplo 10.20, la colocación del primer nodo en un SCC puede dictar por completo el programa de todos los demás nodos. Si el programa no se ajusta al programa creado hasta ahora, el intento falla.

Para programar un SCC, el algoritmo determina el primer tiempo en el que pueda programarse cada nodo en el componente, que cumpla con las dependencias de datos transitivas en E^* . Después elige el que tenga el tiempo de inicio más anticipado como el *primer* nodo para programar. Después, el algoritmo invoca a *SccProgramado* para tratar de programar el componente en el tiempo inicial más anticipado. El algoritmo realiza cuando mucho T intentos con tiempos iniciales cada vez mayores. Si falla, entonces el algoritmo prueba con otro intervalo de iniciación.

El algoritmo *SccProgramado* se asemeja al Algoritmo 10.19, pero tiene tres diferencias importantes.

1. El objetivo de *SccProgramado* es programar el componente fuertemente conectado en la ranura de tiempo s dada. Si el *primer* nodo del componente no puede programarse en s , *SccProgramado* devuelve falso. La función *main* puede invocar a *SccProgramado* otra vez con una ranura de tiempo posterior, si se desea.
2. Los nodos en el componente fuertemente conectado se programan en orden topológico, con base en las aristas en E' . Como las diferencias de iteración en todas las aristas en E' son 0, estas aristas no cruzan ningún límite de iteración y no pueden formar ciclos. (Las aristas que cruzan los límites de iteración se conocen como *acarreados por ciclo*.) Sólo las dependencias acarreadas por ciclo colocan límites superiores en donde pueden programarse las operaciones. Así, este orden de programación, junto con la estrategia de programar cada operación lo antes posible, maximiza los rangos en los que pueden programarse los nodos siguientes.
3. Para los componentes fuertemente conectados, las dependencias imponen un límite tanto inferior como superior en el rango en el que puede programarse un nodo. *SccProgramado* calcula estos rangos y los utiliza para limitar más los intentos de programación.

Ejemplo 10.22: Vamos a aplicar el Algoritmo 10.21 al grafo de dependencia de datos cíclico del ejemplo 10.20. El algoritmo calcula primero que el límite sobre el intervalo de iniciación para este ejemplo sea de 3 ciclos de reloj. Observamos que no es posible cumplir con este límite inferior. Cuando el intervalo de iniciación T es de 3, las dependencias transitivas en la figura 10.28 dictan que $S(d) - S(b) = 2$. Al programar los nodos b y d dos ciclos de reloj

aparte se producirá un conflicto en una tabla de reservación de recursos modular, con una longitud de 3.

Intento	Intervalo de iniciación	Nodo	Rango	Programa	Reservación de recursos modular
1	$T = 3$	a	$(0, \infty)$	0	
		b	$(2, \infty)$	2	
		c	$(3, 3)$	---	
2	$T = 3$	a	$(0, \infty)$	0	
		b	$(2, \infty)$	3	
		c	$(4, 4)$	4	
		d	$(5, 5)$	---	
3	$T = 3$	a	$(0, \infty)$	0	
		b	$(2, \infty)$	4	
		c	$(5, 5)$	5	
		d	$(6, 6)$	---	
		a	$(0, \infty)$	0	
4	$T = 4$	b	$(2, \infty)$	2	
		c	$(3, 4)$	3	
		d	$(4, 5)$	---	
		a	$(0, \infty)$	0	
5	$T = 4$	b	$(2, \infty)$	3	
		c	$(4, 5)$	5	
		d	$(5, 5)$	---	
		a	$(0, \infty)$	0	
		b	$(2, \infty)$	4	
6	$T = 4$	c	$(5, 6)$	5	
		d	$(6, 7)$	6	

Figura 10.30: Comportamiento del Algoritmo 10.21 en el ejemplo 10.20

La figura 10.30 muestra cómo se comporta el Algoritmo 10.21 con este ejemplo. Primero trata de encontrar un programa con un intervalo de iniciación de 3 ciclos de reloj. El intento empieza por programar los nodos a y b lo antes posible. Sin embargo, una vez que el nodo b se coloca en el ciclo de reloj 2, el nodo c sólo puede colocarse en el ciclo 3, lo cual entra en conflicto con el uso de recursos del nodo a . Es decir, tanto a como c necesitan el primer recurso en los ciclos de reloj que tienen un residuo de 0 módulo 3.

El algoritmo rastrea hacia atrás y trata de programar el componente fuertemente conectado $\{b, c, d\}$ un ciclo de reloj más tarde. Esta vez el nodo b se programa en el ciclo 3, y el nodo c se programa con éxito en el ciclo de reloj 4. Sin embargo, el nodo d no puede programarse en el

ciclo de reloj 5. Es decir, tanto b como d necesitan el segundo recurso en los ciclos de reloj que tienen un residuo de 0 módulo 3. Observe que es sólo una coincidencia que los dos conflictos descubiertos hasta ahora se encuentren en ciclos de reloj con un residuo de 0 módulo 3; el conflicto podría haber ocurrido en los ciclos de reloj con un residuo de 1 o 2 en otro ejemplo.

El algoritmo se repite retrasando el inicio del SCC $\{b, c, d\}$ un ciclo más. Pero, como vimos antes, este SCC nunca puede programarse con un intervalo de iniciación de 3 ciclos de reloj, por lo que el intento está propenso a fallar. En este punto, el algoritmo se da por vencido y trata de encontrar un programa con un intervalo de iniciación de 4 ciclos de reloj. En un momento dado, el algoritmo encuentra el programa óptimo en su sexto intento. \square

10.5.9 Mejoras a los algoritmos de canalización

El Algoritmo 10.21 es un algoritmo bastante simple, aunque se ha encontrado que funciona bien en los destinos de máquinas actuales. Los elementos importantes en este algoritmo son:

1. El uso de una tabla de reservación de recursos modular para comprobar conflictos de recursos en el estado estable.
2. La necesidad de calcular las relaciones de dependencia transitivas para encontrar el rango válido en el que puede programarse un nodo, en presencia de los ciclos de dependencia.
3. El rastreo hacia atrás es útil, y los nodos en *ciclos críticos* (ciclos que colocan el límite inferior más alto en el intervalo de iniciación T) deben reprogramarse en conjunto, ya que no hay espacio entre ellos.

Hay muchas formas de mejorar el Algoritmo 10.21. Por ejemplo, el algoritmo tarda cierto tiempo en darse cuenta de que un intervalo de iniciación de 3 ciclos de reloj no es factible para el ejemplo 10.22 simple. Podemos programar primero los componentes fuertemente conectados de manera independiente, con el fin de determinar si el intervalo de iniciación es factible para cada componente.

También podemos modificar el orden en el que se programan los nodos. El orden utilizado en el Algoritmo 10.21 tiene algunas desventajas. En primer lugar, como los SCCs no triviales son más difíciles de programar, es conveniente programarlos primero. En segundo lugar, algunos de los registros pueden tener tiempos de vida innecesariamente largos. Es conveniente acercar más las definiciones a los usos. Una posibilidad es programar primero los componentes fuertemente conectados con los ciclos críticos y después extender el programa en ambos extremos.

10.5.10 Expansión modular de variables

Se dice que una variable escalar es *privatizable* en un ciclo si su rango de vida está dentro de una iteración del ciclo. En otras palabras, una variable privatizable no debe vivir a la entrada o a la salida de cualquier iteración. Estas variables se llaman así debido a que los distintos procesadores que ejecutan distintas iteraciones en un ciclo pueden tener sus propias copias privadas, y, por lo tanto, no interfieren unas con otras.

La *expansión de variables* se refiere a la transformación de convertir una variable escalar privatizable en un arreglo, y hacer que la i -ésima iteración del ciclo lea y escriba el i -ésimo

¿Hay alternativas para la heurística?

Podemos formular el problema de encontrar al mismo tiempo un programa de canalización por software óptimo y la asignación de registros como un problema de programación lineal de enteros. Aunque muchos programas lineales enteros pueden resolverse con rapidez, algunos de ellos pueden requerir una cantidad de tiempo exorbitante. Para usar un solucionador de programación lineal entera en un compilador, debemos tener la capacidad de abortar el procedimiento, si éste no se completa dentro de un límite preestablecido.

Dicho método se ha probado en una máquina de destino (la SGI R8000) en forma empírica, y se descubrió que el solucionador podía encontrar la solución óptima para un gran porcentaje de los programas en el experimento, dentro de un rango de tiempo razonable. El resultado fue que los programas producidos mediante el uso de un método heurístico también estuvieron cerca de lo óptimo. Los resultados sugieren que, por lo menos para esa máquina, no tiene sentido utilizar el método de programación lineal entera, en especial desde la perspectiva de la ingeniería de software. Como el solucionador lineal entero tal vez no termine, aún es necesario implementar cierto tipo de solucionador heurístico en el compilador. Una vez establecido dicho solucionador heurístico, hay poco incentivo para implementar también un programador basado en las técnicas de programación entera.

elemento. Esta transformación elimina las restricciones antidependientes entre las lecturas en una iteración y las escrituras en las iteraciones siguientes, así como las dependencias de salida entre las escrituras de distintas iteraciones. Si se eliminan todas las dependencias acarreadas por ciclo, todas las iteraciones en el ciclo pueden ejecutarse en paralelo.

La eliminación de las dependencias acarreadas por ciclo, y por ende la eliminación de ciclos en el grafo de dependencia de datos, puede mejorar bastante la efectividad de la canalización por software. Como se ilustra mediante el ejemplo 10.15, no necesitamos expandir una variable privatizable por completo mediante el número de iteraciones en el ciclo. Sólo un pequeño número de iteraciones puede estar ejecutándose a la vez, y las variables privatizables pueden estar vivas en forma simultánea, en un número aún más pequeño de iteraciones. Por ende, el mismo almacenamiento se reutiliza para guardar variables con tiempos de vida que no se traslapen. Dicho en forma más específica, si el tiempo de vida de un registro es de l ciclos, y el intervalo de iniciación es T , entonces sólo puede haber $q = \lceil \frac{l}{T} \rceil$ variables vivas en un punto dado. Podemos asignar q registros a la variable, en donde la variable en la i -ésima iteración utiliza el $(i \bmod q)$ -ésimo registro. A esta transformación se le conoce como *expansión modular de variables*.

Algoritmo 10.23: Canalización por software con expansión modular de variables.

ENTRADA: Un grafo de dependencia de datos y una descripción de recursos de la máquina.

SALIDA: Dos ciclos, uno canalizado por software y uno sin canalización.

MÉTODO:

1. Elimine las antidependencias acarreadas por ciclo y las dependencias de salida asociadas con las variables privatizables del grafo de dependencia de datos.
2. Canalice por software el grafo de dependencia resultante, usando el Algoritmo 10.21. Haga que T sea el intervalo de iniciación para el cual se encuentra un programa, y que L sea la longitud del programa para una iteración.
3. Del programa resultante, calcule q_v , el mínimo número de registros necesarios para cada variable privatizable v . Haga que $Q = \max_v q_v$.
4. Genere dos ciclos: un ciclo canalizado por software y uno sin canalización. El ciclo canalizado por software tiene

$$\left\lceil \frac{L}{T} \right\rceil + Q - 1$$

copias de las iteraciones, colocadas T ciclos aparte. Tiene un prólogo con

$$\left(\left\lceil \frac{L}{T} \right\rceil - 1 \right) T$$

instrucciones, un estado estable con QT instrucciones, y un epílogo de $L - T$ instrucciones. Inserte una instrucción de regreso de ciclo que bifurque desde la parte inferior del estado estable hasta la parte superior del mismo.

El número de registros asignados a la variable privatizable v es

$$q'_v = \begin{cases} q_v & \text{si } Q \bmod q_v = 0 \\ Q & \text{en cualquier otro caso} \end{cases}$$

La variable v en la iteración i utiliza el $(i \bmod q'_v)$ -ésimo registro asignado.

Haga que n sea la variable que representa el número de iteraciones en el ciclo de origen. El ciclo canalizado por software se ejecuta si

$$n \geq \left\lceil \frac{L}{T} \right\rceil + Q - 1.$$

El número de veces que se toma la bifurcación de regreso de ciclo es

$$n_1 = \left\lfloor \frac{n - \left\lceil \frac{L}{T} \right\rceil + 1}{Q} \right\rfloor.$$

Por ende, el número de iteraciones de origen ejecutadas por el ciclo canalizado por software es:

$$n_2 = \begin{cases} \left\lceil \frac{L}{T} \right\rceil - 1 + Qn_1 & \text{si } n \geq \left\lceil \frac{L}{T} \right\rceil + Q - 1 \\ 0 & \text{en cualquier otro caso} \end{cases}$$

El número de iteraciones ejecutadas por el ciclo sin canalización es $n_3 = n - n_2$. \square

Ejemplo 10.24: Para el ciclo canalizado por software de la figura 10.22, $L = 8$, $T = 2$ y $Q = 2$. El ciclo canalizado por software tiene 7 copias de las iteraciones, en donde el prólogo, el estado estable y el epílogo tienen 6, 4 y 6 instrucciones, respectivamente. Haga que n sea el número de iteraciones en el ciclo de origen. El ciclo canalizado por software se ejecuta si $n \geq 5$, en cuyo caso la bifurcación de regreso de ciclo se toma

$$\left\lfloor \frac{n-3}{2} \right\rfloor$$

veces, y el ciclo canalizado por software es responsable de

$$3 + 2 \times \left\lfloor \frac{n-3}{2} \right\rfloor$$

de las iteraciones en el ciclo de origen. \square

La expansión modular incrementa el tamaño del estado estable por un factor de Q . A pesar de este incremento, el código generado por el Algoritmo 10.23 sigue siendo bastante compacto. En el peor de los casos, el ciclo canalizado por software requeriría hasta tres veces más instrucciones que el del programa para una iteración. Aproximadamente, junto con el ciclo adicional generado para manejar las iteraciones restantes, el tamaño total del código es de casi 4 veces el del original. Por lo general, esta técnica se aplica a los ciclos internos estrechos, por lo que este incremento es razonable.

El Algoritmo 10.23 minimiza la expansión de código a expensas de usar más registros. Podemos reducir el uso de registros al generar más código. Podemos usar el mínimo de q_v registros para cada variable v si utilizamos un estado estable con

$$T \times \text{LCM}_v q_v$$

instrucciones. Aquí, LCM_v representa la operación de tomar el *mínimo múltiplo común* de todas las q_v , a medida que v varía sobre todas las variables privatizables (es decir, el menor entero que sea un múltiplo entero de todas las q_v). Por desgracia, el mínimo múltiplo común puede ser bastante grande, incluso para unas cuantas q_v pequeñas.

10.5.11 Instrucciones condicionales

Si las instrucciones predicadas están disponibles, podemos convertir las instrucciones dependientes de control en predicadas. Las instrucciones predicadas pueden canalizarse por software, al igual que cualquier otra operación. No obstante, si hay una gran cantidad de flujo de control dependiente de datos dentro del cuerpo del ciclo, tal vez sean más apropiadas las técnicas de programación descritas en la sección 10.4.

Si una máquina no tiene instrucciones predicadas, podemos usar el concepto de *reducción jerárquica*, descrito a continuación, para manejar una pequeña cantidad de flujo de control dependiente de datos. Al igual que el Algoritmo 10.11, en la reducción jerárquica las construcciones de control en el ciclo se programan de adentro hacia fuera, empezando con las estructuras

más profundamente anidadas. A medida que se programa cada construcción, toda la construcción se reduce a un solo nodo, el cual representa todas las restricciones de programación de sus componentes con respecto a las otras partes del programa. Así, este nodo se puede programar como si fuera un nodo simple dentro de la construcción de control circundante. El proceso de programación está completo cuando todo el programa se reduce a un solo nodo.

En el caso de una instrucción condicional con bifurcaciones “then” y “else”, programamos cada una de las bifurcaciones de manera independiente. Entonces:

1. Las restricciones de toda la instrucción condicional se consideran, en forma conservadora, como la unión de las restricciones de ambas bifurcaciones.
2. Su utilización de los recursos es el máximo de los recursos utilizados en cada bifurcación.
3. Sus restricciones de precedencia son la unión de las de cada bifurcación, que se obtienen pretendiendo que ambas bifurcaciones se ejecutan.

Por ende, este nodo puede programarse como cualquier otro. Se generan dos conjuntos de código, que corresponden a las dos bifurcaciones. Cualquier código programado en paralelo con la instrucción condicional se duplica en ambas bifurcaciones. Si se traslapan varias instrucciones condicionales, debe generarse un código separado para cada combinación de bifurcaciones ejecutadas en paralelo.

10.5.12 Soporte de hardware para la canalización por software

Se ha propuesto el soporte para hardware especializado para minimizar el tamaño del código canalizado por software. El *archivo de registro giratorio* en la arquitectura Itanium es uno de esos ejemplos. Un archivo de registro giratorio tiene un *registro base*, que se agrega al número de registro especificado en el código para derivar el registro actual al que se va a acceder. Podemos obtener distintas iteraciones en un ciclo para usar distintos registros, con sólo cambiar el contenido del registro base en el límite de cada iteración. La arquitectura Itanium también tiene un soporte extenso para las instrucciones predicadas. La predicación no sólo puede usarse para convertir la dependencia de control a la dependencia de datos, sino también para evitar generar los prólogos y epílogos. El cuerpo de un ciclo canalizado por software contiene un superconjunto de las instrucciones emitidas en el prólogo y epílogo. Podemos tan sólo generar el código para el estado estable y la predicación de manera apropiada, con el fin de eliminar las operaciones adicionales para obtener los efectos de tener un prólogo y un epílogo.

Aunque el soporte de hardware del Itanium mejora la densidad del código canalizado por software, también debemos tener en cuenta que el soporte no es económico. Como la canalización por software es una técnica orientada a los ciclos estrechos más internos, los ciclos canalizados tienden a ser pequeños de todas formas. El soporte especializado para la canalización por software se garantiza sobre todo para las máquinas destinadas a ejecutar muchos ciclos canalizados por software, y en situaciones en las que es muy importante disminuir al mínimo el tamaño del código.

```

1) L: LD R1, a(R9)
2) ST b(R9), R1
3) LD R2, c(R9)
4) ADD R3, R1, R2
5) ST c(R9), R3
6) SUB R4, R1, R2
7) ST b(R9), R4
8) BL R9, L

```

Figura 10.31: Código máquina para el ejercicio 10.5.2

10.5.13 Ejercicios para la sección 10.5

Ejercicio 10.5.1: En el ejemplo 10.20 mostramos cómo establecer los límites en los ciclos de reloj relativos en los cuales se programan b y c . Calcule los límites para cada uno de los otros cinco pares de nodos (i) para T general (ii) para $T = 3$ (iii) para $T = 4$.

Ejercicio 10.5.2: En la figura 10.31 está el cuerpo de un ciclo. Las direcciones como $a(R9)$ están diseñadas para ser ubicaciones de memoria, en donde a es una constante, y $R9$ es el registro que cuenta las iteraciones a través del ciclo. Puede suponer que cada iteración del ciclo accede a distintas ubicaciones, ya que $R9$ tiene un valor distinto. Usando el modelo de máquina del ejemplo 10.12, programe el ciclo de la figura 10.31 de las siguientes maneras:

- Manteniendo cada iteración lo más estrecha posible (es decir, sólo introduzca una `nop` después de cada operación aritmética), desenrolle el ciclo dos veces. Programe la segunda iteración para que comience lo más pronto posible, sin violar la restricción de que la máquina sólo puede realizar una carga, un almacenamiento, una operación aritmética, y una bifurcación en cualquier ciclo de reloj.
- Repita la parte (a), pero desenrolle el ciclo tres veces. De nuevo, empiece cada iteración lo más pronto que pueda, sujeta a las restricciones de la máquina.
- Construya código completamente canalizado, sujeto a las restricciones de la máquina. En esta parte, puede introducir instrucciones `nop` si es necesario, pero debe empezar una nueva iteración cada dos ciclos de reloj.

Ejercicio 10.5.3: Cierto ciclo requiere 5 cargas, 7 almacenamientos y 8 operaciones aritméticas. ¿Cuál es el intervalo de iniciación mínimo para una canalización por software de este ciclo, en una máquina que ejecuta cada operación en un ciclo de reloj, y tiene suficientes recursos para hacerlo, en un ciclo de reloj?:

- 3 cargas, 4 almacenamientos y 5 operaciones aritméticas.
- 3 cargas, 3 almacenamientos y 3 operaciones aritméticas.

! Ejercicio 10.5.4: Usando el modelo de máquina del ejemplo 10.12, encuentre el intervalo de iniciación mínimo y un programa uniforme para las iteraciones, para el siguiente ciclo:

```
for (i = 1; 1 < n; i++) {
    A[i] = B[i-1] + 1;
    B[i] = A[i-1] + 2;
}
```

Recuerde que la cuenta de iteraciones se maneja mediante el autoincremento de los registros, y no se necesitan operaciones que sean sólo para el conteo asociado con el ciclo for.

! Ejercicio 10.5.5: Demuestre que el Algoritmo 10.19, en el caso especial en donde cada operación requiere sólo una unidad de un recurso, siempre puede encontrar un programa de canalización por software que cumpla con el límite inferior.

! Ejercicio 10.5.6: Suponga que tenemos un grafo de dependencia de datos cíclico con los nodos a , b , c y d . Hay flancos que van de a hacia b y de c hacia d con la etiqueta $\langle 0, 1 \rangle$, y hay flancos que van de b hacia c y de d hacia a con la etiqueta $\langle 1, 1 \rangle$. No hay otros flancos.

- Dibuje el grafo de dependencia cíclico.
- Calcule la tabla de caminos simples más largos entre los nodos.
- Muestre las longitudes de los caminos simples más largos, si el intervalo de iniciación T es 2.
- Repita (c) si $T = 3$.
- Para $T = 3$, ¿cuáles son las restricciones en los tiempos relativos en los que puede programarse cada una de las instrucciones representadas por a , b , c y d ?

! Ejercicio 10.5.7: Proporcione un algoritmo $O(n)^3$ para encontrar la longitud del camino simple más largo en un grafo con n nodos, suponiendo que ningún ciclo tenga una longitud positiva. *Sugerencia:* Adapte el algoritmo de Floyd para los caminos más cortos (vea, por ejemplo, A. V. Aho y J. D. Ullman, *Foundations of Computer Science*, Computer Science Press, Nueva York, 1992).

!! Ejercicio 10.5.8: Suponga que tenemos una máquina con tres tipos de instrucciones, a las cuales llamaremos A , B y C . Todas las instrucciones requieren un ciclo de reloj, y la máquina puede ejecutar una instrucción de cada tipo en cada ciclo de reloj. Suponga que un ciclo consiste en seis instrucciones, dos de cada tipo. Entonces, es posible ejecutar el ciclo en una canalización por software con un intervalo de iniciación de dos. No obstante, algunas secuencias de las seis instrucciones requieren la inserción de un retraso, y algunas requieren la inserción de dos retrasos. De las 90 secuencias posibles de dos A s, dos B s y dos C s, ¿cuántas no requieren retraso? ¿Cuántas requieren un retraso? *Sugerencia:* Hay simetría entre los tres tipos de instrucciones, por lo que dos secuencias que pueden transformarse entre sí mediante la permutación de los nombres A , B y C deben requerir el mismo número de retrasos. Por ejemplo, $ABBCAC$ debe ser igual que $BCCABA$.

10.6 Resumen del capítulo 10

- ◆ *Cuestiones de arquitectura:* La programación de código optimizado aprovecha las características de las arquitecturas de las computadoras modernas. A menudo, dichas máquinas permiten la ejecución canalizada, en donde varias instrucciones están en distintas etapas de ejecución al mismo tiempo. Algunas máquinas también permiten que varias instrucciones empiecen a ejecutarse al mismo tiempo.
- ◆ *Dependencias de datos:* Al programar instrucciones, debemos tener en cuenta el efecto que las instrucciones tienen sobre cada ubicación de memoria y registro. Las verdaderas dependencias de datos ocurren cuando una instrucción debe leer una ubicación, después de que otra ha escrito en ella. Las antidependencias ocurren cuando hay una escritura después de una lectura, y las dependencias de salida ocurren cuando hay dos escrituras en la misma ubicación.
- ◆ *Eliminación de dependencias:* Al utilizar ubicaciones adicionales para almacenar los datos, se pueden eliminar las antidependencias y las dependencias de salida. Sólo las verdaderas dependencias no pueden eliminarse y deben sin duda respetarse cuando el código se programa.
- ◆ *Grafos de dependencia de datos para bloques básicos:* Estos grafos representan las restricciones de sincronización entre las instrucciones de un bloque básico. Los nodos corresponden a las instrucciones. Un flanco de n a m etiquetado como d indica que la instrucción m debe empezar por lo menos d ciclos de reloj después de que empieza la instrucción n .
- ◆ *Órdenes topológicos priorizados:* El grafo de dependencia de datos para un bloque básico siempre es acíclico y, por lo general, hay muchos órdenes topológicos consistentes con el grafo. Se puede utilizar una de varias heurísticas para seleccionar un orden topológico preferido para un grafo dado; por ejemplo, elegir primero los nodos con el camino crítico más largo.
- ◆ *Programación por lista:* Dado un orden topológico priorizado para un grafo de dependencia de datos, podemos considerar los nodos en ese orden. Programe cada nodo en el ciclo de reloj más anticipado que sea consistente con las restricciones de tiempo implicadas por las aristas del grafo, los programas de todos los nodos programados con anterioridad, y las restricciones de recurso de la máquina.
- ◆ *Movimiento de código entre bloques:* Bajo ciertas circunstancias, es posible mover instrucciones del bloque en el que aparecen, a un bloque predecesor o sucesor. La ventaja es que puede haber oportunidades para ejecutar instrucciones en paralelo en la nueva ubicación, que no existen en la ubicación original. Si no hay una relación de predominio entre las ubicaciones antigua y nueva, tal vez sea necesario insertar código de compensación a lo largo de ciertas rutas, para poder asegurar que se ejecute exactamente la misma secuencia de instrucciones, sin importar el flujo del control.
- ◆ *Ciclos de ejecución total (do-all):* Un ciclo de ejecución total no tiene dependencias entre iteraciones, por lo que cualquier iteración puede ejecutarse en paralelo.

- ◆ *Canalización por software de los ciclos de ejecución total:* La canalización por software es una técnica que explota la habilidad de una máquina para ejecutar varias instrucciones a la vez. Programamos las iteraciones del ciclo para empezar en pequeños intervalos, tal vez colocando instrucciones no-op en las iteraciones para evitar conflictos entre iteraciones de los recursos de la máquina. El resultado es que el ciclo puede ejecutarse con rapidez, con un preámbulo, un coda, y (por lo general) un ciclo interno diminuto.
- ◆ *Ciclos de ejecución cruzada:* La mayoría de los ciclos tienen dependencias de datos que van de cada iteración a iteraciones posteriores. A éstos se les conoce como ciclos de ejecución cruzada.
- ◆ *Grafos de dependencia de datos para ciclos de ejecución cruzada:* Para representar las dependencias entre instrucciones de un ciclo de ejecución cruzada, se requiere que las aristas se etiqueten mediante un par de valores: el retraso requerido (como para los grafos que representan bloques básicos) y el número de iteraciones que transcurren entre las dos instrucciones que tienen una dependencia.
- ◆ *Programación por lista de los ciclos:* Para programar un ciclo, debemos elegir el programa para todas las iteraciones, y también elegir el intervalo de iniciación en el cual deben comenzar las iteraciones sucesivas. El algoritmo implica derivar las restricciones en los programas relativos de las diversas instrucciones en el ciclo, buscando la longitud de las rutas acíclicas más largas entre los dos nodos. Estas longitudes tienen el intervalo de iniciación como un parámetro, y por ende imponen un límite menor en el intervalo de iniciación.

10.7 Referencias para el capítulo 10

Para una discusión más detallada sobre la arquitectura y el diseño de procesadores, recomendamos a Hennessy y Patterson [5].

El concepto de la dependencia de datos se discutió por primera vez en Kuck, Muraoka y Chen [6], y en Lamport [8] dentro del contexto de la compilación de código para multiprocesadores y máquinas de vectores.

La programación de instrucciones se utilizó por primera vez en la programación de microcódigo horizontal ([2, 3, 11 y 12]). El trabajo de Fisher en relación con la compactación de microcódigo lo llevó a proponer el concepto de una máquina VLIW, en donde los compiladores pueden controlar directamente la ejecución paralela de las operaciones [3]. Gross y Hennessy [4] utilizaron la programación de instrucciones para manejar las bifurcaciones retrasadas en el primer conjunto de instrucciones MIPS RISC. El algoritmo de este capítulo se basa en el tratamiento más general de Bernstein y Rodeh [1] de la programación de operaciones para máquinas con paralelismo a nivel de instrucción.

Patel y Davidson [9] desarrollaron por primera vez la idea básica de la canalización por software, para programar canalizaciones por hardware. Rau y Glaeser [10] utilizaron por primera vez la canalización por software para compilar para una máquina con hardware especializado, diseñado para soportar la canalización por software. El algoritmo aquí descrito se basa en Lam [7], que no supone ningún soporte de hardware especializado.

1. Bernstein, D. y M. Rodeh, "Global instruction scheduling for superscalar machines", *Proc. ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 241-255.
2. Dasgupta, S., "The organization of microprogram stores", *Computing Surveys* **11**:1 (1979), pp. 39-65.
3. Fisher, J.A., "Trace scheduling: a technique for global microcode compaction", *IEEE Trans. On Computers* **C-30**:7 (1981), pp. 478-490.
4. Gross, T. R. y Hennessy, J. L., "Optimizing delayed branches", *Proc. 15th Annual Workshop on Microprogramming* (1982), pp. 114-120.
5. Hennessy, J. L. y D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Tercera Edición, Morgan Kaufman, San Francisco, 2003.
6. Kuck, D., Y. Muraoka y S. Chen, "On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup", *IEEE Transactions on Computers* **C-21**:12 (1972), pp. 1293-1310.
7. Lam, M. S., "Software pipelining: an effective scheduling technique for VLIW machines", *Proc. ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318-328.
8. Lamport, L., "The parallel execution of DO loops", *Comm. ACM* **17**:2 (1974), pp. 83-93.
9. Patel, J. H. y E. S. Davidson, "Improving the throughput of a pipeline by insertion of delays", *Proc. Third Annual Symposium on Computer Architecture* (1976), pp. 159-164.
10. Rau, B. R. y C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing", *Proc. 14th Annual Workshop on Microprogramming* (1981), pp. 183-198.
11. Tokoro, M., E. Tamura y T. Takizuka, "Optimization of microprograms", *IEEE Trans. on Computers* **C-30**:7 (1981), pp. 491-504.
12. Word, G., "Global optimization of microprograms through modular control constructs", *Proc. 12th Annual Workshop in Microprogramming* (1979), pp. 1-6.

Capítulo 11

Optimización para el paralelismo y la localidad

Este capítulo muestra cómo un compilador puede mejorar el paralelismo y la localidad en los programas con un uso intensivo de cálculos, que involucren arreglos para agilizar los programas destino que se ejecuten en sistemas con multiprocesadores. Muchas aplicaciones científicas, de ingeniería y comerciales tienen una gran necesidad de ciclos computacionales. Entre algunos ejemplos se encuentran el pronóstico del clima, el plegamiento de proteínas para diseñar fármacos, la dinámica de fluidos para diseñar sistemas de autopropulsión, y la cromodinámica cuántica para estudiar las interacciones fuertes en la física de alta energía.

Una manera de agilizar un cálculo es utilizar el paralelismo. Por desgracia, no es fácil desarrollar software que pueda aprovechar las máquinas paralelas. Ya es bastante difícil dividir el cómputo en unidades que puedan ejecutarse en distintos procesadores en paralelo; ni siquiera eso garantiza un aumento en la velocidad. Además, debemos minimizar la comunicación entre procesadores, ya que ¡es muy fácil que la sobrecarga de comunicación haga que el código en paralelo se ejecute aún con más lentitud que la ejecución secuencial!

El proceso de minimizar la comunicación puede considerarse como un caso especial de mejorar la *localidad de los datos* de un programa. En general, decimos que un programa tiene buena localidad de los datos si un procesador accede con frecuencia a los mismos datos que ha utilizado recientemente. Sin duda, si un procesador en una máquina paralela tiene buena localidad, no necesita comunicarse con otros procesadores con frecuencia. Por ende, debemos considerar el paralelismo y la localidad de los datos, juntos. La localidad de los datos, por sí sola, es también importante para el rendimiento de los procesadores individuales. Los procesadores modernos tienen uno o más niveles de cachés en la jerarquía de memoria; un acceso a memoria puede requerir cientos de ciclos de máquina, mientras que una coincidencia en la caché sólo requeriría unos cuantos ciclos. Si un programa no tiene una buena localidad de datos y se producen fallas en la caché con frecuencia, su rendimiento disminuirá.

Otra razón por la que el paralelismo y la localidad se tratan en conjunto en este mismo capítulo es que comparten la misma teoría. Si sabemos cómo optimizar la localidad de los datos, sabemos en dónde se encuentra el paralelismo. En este capítulo verá que el modelo de programa

que utilizamos para el análisis de flujo de datos en el capítulo 9 es inadecuado para la paralelización y la optimización de la localidad. La razón es que el trabajo en el análisis del flujo de datos supone que no hay diferencias entre las formas en que se llega a una instrucción dada, y en realidad estas técnicas del capítulo 9 aprovechan el hecho de que no hacemos diferencia entre las distintas ejecuciones de la misma instrucción, por ejemplo, en un ciclo. Para paralelizar un código, debemos razonar acerca de las dependencias entre las distintas ejecuciones dinámicas de la misma instrucción, para determinar si pueden ejecutarse en diferentes procesadores al mismo tiempo.

Este capítulo se enfoca en las técnicas para optimizar la clase de aplicaciones numéricas que utilizan arreglos como estructuras de datos, y para acceder a ellos con patrones regulares simples. Dicho en forma más específica, estudiaremos los programas que tienen accesos a arreglos *afines* con respecto a los índices de los ciclos circundantes. Por ejemplo, si i y j son las variables índice de los ciclos circundantes, entonces $Z[i][j]$ y $Z[i][i + j]$ son accesos afines. Una función de una o más variables, i_1, i_2, \dots, i_n es *afín* si puede expresarse como una suma de una constante, más los múltiplos constantes de las variables, por decir, $c_0 + c_1x_1 + c_2x_2 + \dots + c_nx_n$, en donde c_0, c_1, \dots, c_n son constantes. Por lo general, las funciones afines se conocen como funciones lineales, aunque hablando en sentido estricto, las funciones lineales no tienen el término c_0 .

He aquí un ejemplo simple de un ciclo en este dominio:

```
for (i = 0; i < 10; i++) {
    Z[i] = 0;
}
```

Como las iteraciones del ciclo se escriben en distintas ubicaciones, distintos procesadores pueden ejecutar distintas iteraciones en forma concurrente. Por otro lado, si hay otra instrucción $Z[j] = 1$ en ejecución, tenemos que preocuparnos acerca de si i podría en algún momento ser igual que j , y de ser así, en qué orden debemos ejecutar esas instancias de las dos instrucciones que comparten un valor común del índice del arreglo.

Es importante saber qué iteraciones pueden hacer referencia a la misma ubicación de memoria. Este conocimiento nos permite especificar las dependencias de datos que deben respetarse al programar código, tanto para uniprocesadores como para multiprocesadores. Nuestro objetivo es encontrar un programa que respete todas las dependencias de datos, de tal forma que las operaciones que acceden a la misma ubicación y líneas de caché se ejecuten cerca una de la otra si es posible, y en el mismo procesador, en el caso de los multiprocesadores.

La teoría que presentamos en este capítulo tiene sus bases en el álgebra lineal y las técnicas de programación entera. Modelamos las iteraciones en un anidamiento de ciclos de n niveles como un poliedro n -dimensional, cuyos límites se especifican mediante los límites de los ciclos en el código. Hay funciones afines que asignan cada iteración a las ubicaciones del arreglo a las que accede. Podemos usar la programación lineal entera para determinar si existen dos iteraciones que puedan hacer referencia a la misma ubicación.

El conjunto de transformaciones de código que veremos aquí se pueden clasificar en dos categorías: *particionamiento afín* y *bloqueo*. El particionamiento afín divide los poliedros de iteraciones en componentes, para ejecutarse ya sea en distintas máquinas o una por una, en

secuencia. Por otro lado, el bloqueo crea una jerarquía de iteraciones. Suponga que recibimos un ciclo que recorre un arreglo fila por fila. En vez de ello, podemos subdividir el arreglo en bloques y visitar todos los elementos en un bloque antes de avanzar al siguiente. El código resultante consistirá en ciclos externos que recorren los bloques, y después ciclos internos para recorrer los elementos dentro de cada bloque. Se utilizan técnicas del álgebra lineal para determinar tanto las mejores particiones afines, como los mejores esquemas de bloqueo.

A continuación, primero empezaremos con una visión general de los conceptos en la computación paralela y la optimización de la localidad en la sección 11.1. Después, la sección 11.2 es un ejemplo concreto extendido (multiplicación de matrices) que muestra cómo las *transformaciones de ciclos* que reordenan el cómputo dentro de un ciclo pueden mejorar tanto la localidad como la efectividad de la parallelización.

Las secciones 11.3 a 11.6 presentan la información preliminar necesaria para las transformaciones de los ciclos. La sección 11.3 muestra cómo modelamos las iteraciones individuales en un anidamiento de ciclos; la sección 11.4 muestra cómo modelamos las funciones de los índices de arreglos que asignan cada iteración del ciclo a las ubicaciones de arreglo a las que accede la iteración; La sección 11.5 muestra cómo determinar qué iteraciones en un ciclo hacen referencia a la misma ubicación en el arreglo o la misma línea de caché, usando algoritmos estándar de álgebra lineal; y la sección 11.6 muestra cómo encontrar todas las dependencias de datos entre las referencias a un arreglo en un programa.

El resto del capítulo aplica estas preliminares para obtener las optimizaciones. La sección 11.7 primero analiza el problema más simple de buscar paralelismo que no requiera sincronización. Para encontrar el mejor particionamiento afín, sólo buscamos la solución a la restricción de que las operaciones que comparten una dependencia de datos deben asignarse al mismo procesador.

Pocos programas pueden parallelizarse sin requerir sincronización. Por lo tanto, en las secciones 11.8 a 11.9.9 consideraremos el caso general de buscar paralelismo que requiera sincronización. Presentamos el concepto de las canalizaciones, mostramos cómo buscar el particionamiento afín que maximice el grado de canalización permitido por un programa. Mostramos cómo optimizar la localidad en la sección 11.10. Por último, hablaremos sobre cómo las transformaciones afines son útiles para optimizar otras formas de paralelismo.

11.1 Conceptos básicos

Esta sección presenta los conceptos básicos relacionados con la parallelización y la optimización de la localidad. Si las operaciones pueden ejecutarse en paralelo, también pueden reordenarse para otros objetivos, como la localidad. Por el contrario, si las dependencias de datos en un programa dictan que las instrucciones en un programa deben ejecutarse en serie, entonces es obvio que no hay paralelismo, ni la oportunidad de reordenar las instrucciones para mejorar la localidad. Por ende, el análisis de la parallelización también busca las oportunidades disponibles para el movimiento de código, para mejorar la localidad de los datos.

Para disminuir al mínimo la comunicación en el código en paralelo, agrupamos todas las operaciones relacionadas y las asignamos al mismo procesador. El código resultante debe, por lo tanto, tener localidad de los datos. Un método primitivo para obtener una buena localidad de los datos en un uniprocesador es hacer que el procesador ejecute el código asignado a cada procesador en sucesión.

En esta introducción, empezaremos con un panorama general sobre las arquitecturas de computadora en paralelo. Después mostraremos los conceptos básicos de la paralelización, el tipo de transformaciones que pueden hacer una gran diferencia, así como los conceptos útiles para la paralelización. Más tarde hablaremos sobre cómo las consideraciones similares pueden usarse para optimizar la localidad. Por último, presentaremos de manera informal los conceptos matemáticos que usamos en este capítulo.

11.1.1 Multiprocesadores

La arquitectura de máquina paralela más popular es el multiprocesador simétrico (Symmetric Multiprocessor, SMP). A menudo, las computadoras personales de alto rendimiento tienen dos procesadores, y muchas máquinas servidor tienen cuatro, ocho e incluso hasta decenas de procesadores. Además, como ya es posible acomodar varios procesadores de alto rendimiento en un solo chip, los multiprocesadores han empezado a tener mucha popularidad.

Los procesadores en un multiprocesador simétrico comparten el mismo espacio de direcciones. Para comunicarse, un procesador sólo necesita escribir en una ubicación de memoria, que después otro procesador lee. Los multiprocesadores simétricos se llaman así debido a que todos los procesadores pueden acceder a toda la memoria en el sistema con un tiempo de acceso uniforme. La figura 11.1 muestra la arquitectura de alto nivel de un multiprocesador. Los procesadores pueden tener su propia caché de primer nivel, de segundo nivel y, en algunos casos, hasta de tercer nivel. Las cachés de más alto nivel se conectan a la memoria física, por lo general, a través de un bus compartido.

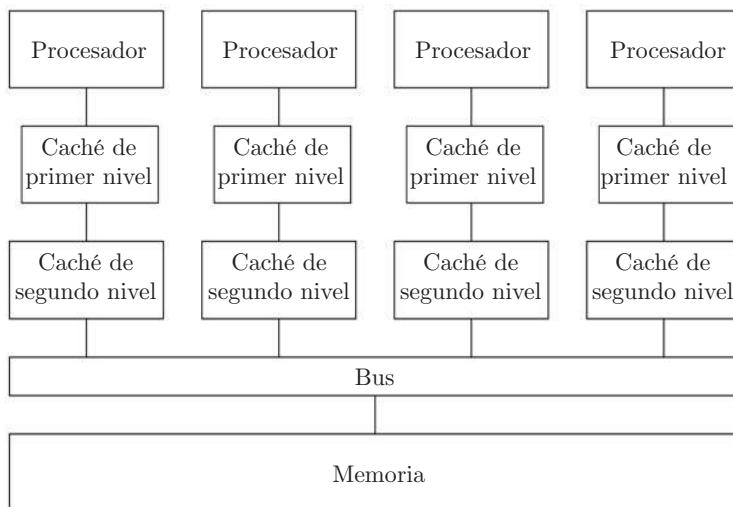


Figura 11.1: La arquitectura del multiprocesador simétrico

Los multiprocesadores simétricos utilizan un *protocolo de caché coherente* para ocultar la presencia de cachés al programador. Bajo dicho protocolo, se permite a varios procesadores

mantener copias de la misma línea¹ de caché al mismo tiempo, siempre y cuando sólo estén leyendo los datos. Cuando un procesador desea escribir en una línea de caché, se eliminan las copias de todas las demás cachés. Cuando un procesador solicita datos que no encuentra en su caché, la solicitud pasa al bus compartido, y los datos se obtienen ya sea de la memoria o de la caché de otro procesador.

El tiempo que requiere un procesador para comunicarse con otro es de aproximadamente el doble del costo de un acceso a memoria. Los datos, en unidades de líneas de caché, primero deben escribirse desde la caché del primer procesador a la memoria, y después se obtienen de la memoria a la caché del segundo procesador. Tal vez piense que la comunicación entre procesadores es muy económica, ya que sólo es aproximadamente el doble de lenta que un acceso a memoria. Sin embargo, debe recordar que los accesos a memoria son muy costosos cuando se les compara con las ocurrencias en la caché; pueden ser cientos de veces más lentos. Este análisis demuestra con claridad la similitud entre la paralelización eficiente y el análisis de la localidad. Para que un procesador trabaje bien, ya sea por su cuenta o dentro del contexto de un multiprocesador, debe encontrar en su caché la mayoría de los datos sobre los que opera.

A principios de la década del 2000, el diseño de los multiprocesadores simétricos ya no escalaba más allá de varias decenas de procesadores, debido a que el bus compartido, o cualquier otro tipo de interconexión relacionada, no podía operar a la par con el número cada vez mayor de procesadores. Para hacer que los diseños de los procesadores fueran escalables, los arquitectos introdujeron otro nivel más en la jerarquía de memoria. En vez de tener memoria que de igual forma está alejada de cada procesador, distribuyeron las memorias de manera que cada procesador pudiera acceder a su memoria local con rapidez, como se muestra en la figura 11.2. Por ende, las memorias remotas constituyeron el siguiente nivel de la jerarquía de memoria; son en conjunto más grandes, pero también requieren más tiempo para su acceso. De manera análoga al principio en el diseño de jerarquías de memoria, que establece que las operaciones de almacenamiento rápidas son necesariamente pequeñas, las máquinas que soportan la comunicación rápida entre procesadores tienen necesariamente un pequeño número de procesadores.

Existen dos variantes de una máquina paralela con memorias distribuidas: las máquinas NUMA (Nonuniform memory access, acceso a memoria no uniforme) y las máquinas que pasan mensajes. Las arquitecturas NUMA proporcionan un espacio de direcciones compartido al software, lo cual permite a los procesadores comunicarse mediante la lectura y la escritura de la memoria compartida. Sin embargo, en las máquinas que pasan mensajes, los procesadores tienen espacios de direcciones separados, y se comunican enviándose mensajes unos a otros. Observe que, aun cuando es más simple escribir código para las máquinas con memoria compartida, el software debe tener una buena localidad para cualquier tipo de máquina, para poder trabajar bien.

11.1.2 Paralelismo en las aplicaciones

Utilizamos dos métricas de alto nivel para estimar qué tan bueno será el rendimiento de una aplicación en paralelo: la *cobertura paralela*, que representa el porcentaje del cómputo que se ejecuta en paralelo, y la *granularidad del paralelismo*, que es la cantidad de cómputo que cada procesador puede ejecutar sin sincronizarse o comunicarse con otros. Un objetivo bastante

¹Tal vez desee repasar la explicación sobre las cachés y las líneas de caché en la sección 7.4.

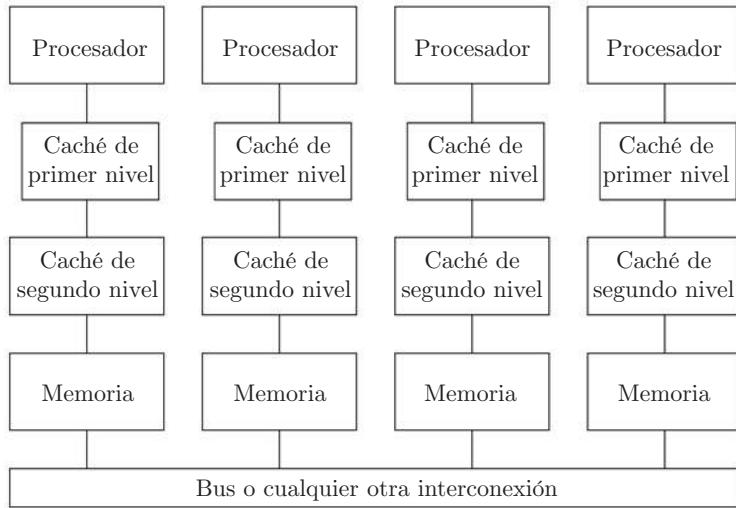


Figura 11.2: Máquinas de memoria distribuidas

atractivo de la paralelización son los ciclos: un ciclo puede tener muchas iteraciones, y si son independientes unas de otras, hemos encontrado una gran fuente de paralelismo.

Ley de Amdahl

La importancia de la cobertura del paralelismo se muestra en forma sintetizada mediante la Ley de Amdahl. La *ley de Amdahl* establece que, si f es la fracción del código paralelizado, y si la versión paralelizada se ejecuta en una máquina con p procesadores sin sobrecarga de comunicación o paralelización, el aumento en velocidad es:

$$\frac{1}{(1 - f) + (f/p)}.$$

Por ejemplo, si la mitad del cómputo es secuencial, el cómputo sólo puede aumentar su velocidad al doble, sin importar cuántos procesadores utilicemos. El aumento en velocidad que puede lograrse es un factor de 1.6 si tenemos 4 procesadores. Incluso si la cobertura del paralelismo es del 90%, obtenemos cuando mucho un factor de aumento de 3 en 4 procesadores, y un factor de 10 en un número ilimitado de procesadores.

Granularidad del paralelismo

Es ideal si todo el cómputo de una aplicación puede particionarse en muchas tareas independientes menos específicas, ya que podemos tan sólo asignar las diferentes tareas a distintos procesadores. Uno de esos ejemplos es el proyecto SETI (Search for extra-terrestrial Intelligence, Búsqueda de inteligencia extraterrestre), el cual es el experimento que utiliza computadoras personales conectadas a través de Internet para analizar distintas porciones de datos de radiotelescopios en

paralelo. Cada unidad de trabajo, que requiere sólo una pequeña cantidad de datos de entrada y genera una pequeña cantidad de datos de salida, puede ejecutarse de manera independiente a las demás. Como resultado, dichos cálculos se ejecutan bien en máquinas através de Internet, que tiene una latencia de comunicación (retraso) relativamente alta y un ancho de banda bajo.

La mayoría de las aplicaciones requieren más comunicación e interacción entre los procesadores, y aún así permiten un paralelismo menos específico. Por ejemplo, consideremos el servidor Web responsable de atender un gran número de solicitudes, en su mayoría independientes, usando una base de datos común. Podemos ejecutar la aplicación en un multiprocesador, con un hilo que implemente la base de datos y varios hilos más que atiendan las solicitudes de los usuarios. Otros ejemplos incluyen el diseño de medicamentos o la simulación de perfiles aerodinámicos, en donde los resultados de muchos parámetros distintos pueden evaluarse en forma independiente. Algunas veces, hasta la evaluación de sólo un conjunto de parámetros en una simulación tarda tanto tiempo, que es conveniente aumentar su velocidad mediante la paralelización. A medida que se reduce la granularidad del paralelismo disponible en una aplicación, se requiere un mejor soporte para las comunicaciones entre los procesadores y un mayor esfuerzo de programación.

Muchas aplicaciones científicas y de ingeniería que se ejecutan durante grandes periodos, con sus estructuras de control simples y sus extensos conjuntos de datos, pueden parallelizarse con más facilidad y con mayor detalle que las aplicaciones antes mencionadas. Por ende, este capítulo se dedica principalmente a las técnicas que se aplican a las aplicaciones numéricas, y en particular, a los programas que invierten la mayor parte de su tiempo manipulando datos en arreglos multidimensionales. A continuación examinaremos esta clase de programas.

11.1.3 Paralelismo a nivel de ciclo

Los ciclos son el objetivo principal para la paralelización, en especial en las aplicaciones que utilizan arreglos. Las aplicaciones que se ejecutan por largos periodos tienen, por lo regular, arreglos extensos, los cuales conducen a ciclos que tienen muchas iteraciones, una para cada elemento en el arreglo. Es común encontrar ciclos cuyas iteraciones son independientes entre sí. Podemos dividir el gran número de iteraciones de dichos ciclos entre los procesadores. Si la cantidad de trabajo realizada en cada iteración es casi la misma, con sólo dividir las iteraciones de manera uniforme a través de los procesadores se logrará el máximo paralelismo. El ejemplo 11.1 es un ejemplo bastante simple, que muestra cómo podemos sacar ventaja del paralelismo a nivel de ciclo.

Ejemplo 11.1: El ciclo

```
for (i = 0; i < n; i++) {  
    Z[i] = X[i] - Y[i];  
    Z[i] = Z[i] * Z[i];  
}
```

calcula el cuadrado de las diferencias entre los elementos en los vectores X y Y , y lo almacena en Z . El ciclo es paralelizable, ya que cada iteración accede a un conjunto distinto de datos. Podemos ejecutar el ciclo en una computadora con M procesadores, al proporcionar a cada procesador un ID único $p = 0, 1, \dots, M - 1$ y hacer que cada procesador ejecute el mismo código:

Paralelismo a nivel de tarea

Es posible encontrar paralelismo fuera de las iteraciones en un ciclo. Por ejemplo, podemos asignar dos invocaciones a funciones distintas, o dos ciclos independientes, a dos procesadores. A esta forma de paralelismo se le conoce como *paralelismo a nivel de tarea*. El nivel de tarea no es una fuente de paralelismo tan atractiva como el nivel de ciclo. La razón es que el número de tareas independientes es una constante para cada programa y no se escala con el tamaño de los datos, como lo hace el número de iteraciones de un ciclo típico. Además, las tareas por lo general no son de igual tamaño, por lo que es difícil mantener todos los procesadores ocupados todo el tiempo.

```

b = ceil(n/M);
for (i = b*p; i < min(n,b*(p+1)); i++) {
    Z[i] = X[i] - Y[i];
    Z[i] = Z[i] * Z[i];
}

```

Dividimos las iteraciones en el ciclo de manera uniforme entre los procesadores; el p -ésimo procesador recibe la p -ésima franja de iteraciones a ejecutar. Observe que el número de iteraciones tal vez no pueda dividirse entre M , por lo que nos aseguramos de que el último procesador no ejecute más allá del límite del ciclo original, introduciendo una operación mínima. \square

El código en paralelo que se muestra en el ejemplo 11.1 es un programa SPMD (Single program multiple data, Un solo programa, varios datos). Todos los procesadores ejecutan el mismo código, pero se parametriza mediante un identificador único para cada procesador, por lo que distintos procesadores pueden realizar diferentes acciones. Por lo general, un procesador, conocido como *maestro*, ejecuta toda la parte serial de los cálculos. El procesador maestro, al llegar a una sección paralelizada del código, despierta a todos los procesadores *esclavos*. Todos los procesadores ejecutan las regiones paralelizadas del código. Al final de cada región paralelizada de código, todos los procesadores participan en una *sincronización de barrera*. Se garantiza que cualquier operación que se ejecute antes de que un procesador entre a una barrera de sincronización se completará antes de permitir que cualquier otro procesador deje la barrera y ejecute operaciones que vengan después de la misma.

Si paralelizamos sólo los ciclos pequeños como los del ejemplo 11.1, entonces es probable que el código resultante tenga una cobertura baja de paralelismo y un paralelismo relativamente menos específico. Preferimos paralelizar los ciclos más externos en un programa, ya que eso produce el paralelismo más general. Por ejemplo, considere la aplicación de una transformación FFT bidimensional que opera sobre un conjunto de datos $n \times n$. Dicho programa realiza n FFTs en las filas de los datos, y después otras n FFTs en las columnas. Es preferible asignar las n FFTs independientes a un procesador cada una, en vez de tratar de usar varios procesadores para que colaboren con una FFT. El código es más fácil de escribir, la cobertura de paralelismo

para el algoritmo es del 100%, y el código tiene una buena localidad de los datos, ya que no requiere comunicación alguna mientras calcula una FFT.

En muchas aplicaciones los ciclos más externos no son grandes, y no pueden parallelizarse. Sin embargo, el tiempo de ejecución de estas aplicaciones se ve dominado a menudo por los *núcleos (kernels)* que consumen mucho tiempo, que pueden tener cientos de líneas de código consistentes en ciclos con distintos niveles de anidamiento. Algunas veces es posible tomar el núcleo, reorganizar su computación y particionarlo en unidades casi independientes, enfocándose en su localidad.

11.1.4 Localidad de los datos

Existen dos nociones bastante distintas de localidad de datos, que debemos tener en consideración al parallelizar los programas. La localidad *temporal* ocurre cuando se utilizan los mismos datos varias veces dentro de un periodo corto. La localidad *espacial* ocurre cuando distintos elementos de datos, que están ubicados unos cerca de otros, se utilizan dentro de un periodo corto. Una forma importante de la localidad espacial ocurre cuando todos los elementos que aparecen en una línea de caché se utilizan en conjunto. La razón es que, tan pronto como se necesita un elemento de una línea de caché, todos los elementos en la misma línea se llevan a la caché y probablemente sigan ahí si se utilizan pronto. El efecto de esta localidad espacial es que se disminuyen al mínimo los fallos en la caché, lo cual produce un importante aumento en la velocidad del programa.

A menudo, los núcleos se pueden escribir en muchas formas equivalentes de manera semántica, pero con localidades de datos y rendimientos que varían en gran medida. El ejemplo 11.2 muestra una forma alternativa de expresar los cálculos del ejemplo 11.1.

Ejemplo 11.2: Al igual que el ejemplo 11.1, el siguiente código también encuentra los cuadrados de las diferencias entre los elementos de los vectores X y Y .

```
for (i = 0; i < n; i++)
    Z[i] = X[i] - Y[i];
for (i = 0; i < n; i++)
    Z[i] = Z[i] * Z[i];
```

El primer ciclo encuentra las diferencias, el segundo encuentra los cuadrados. En los programas reales aparece a menudo código como éste, ya que ésta es la forma en la que podemos optimizar un programa para las *máquinas de vectores*, que son supercomputadoras con instrucciones que realizan varias operaciones aritméticas simples sobre vectores a la vez. Podemos ver que los cuerpos de los dos ciclos aquí se *fusionan* en uno, en el ejemplo 11.1.

Dado que los dos programas realizan los mismos cálculos, ¿cuál tiene un mejor rendimiento? El ciclo fusionado en el ejemplo 11.1 tiene un mejor rendimiento, ya que tiene una mejor localidad de los datos. Cada diferencia se eleva al cuadrado de inmediato, tan pronto como se produce; de hecho, podemos guardar la diferencia en un registro, elevarla al cuadrado y escribir el resultado sólo una vez en la ubicación de memoria $Z[i]$. En contraste, el código en el ejemplo 11.1 obtiene $Z[i]$ una vez, y lo escribe dos veces. Además, si el tamaño del arreglo es más grande que la caché, $Z[i]$ tiene que volver a obtenerse de memoria la segunda vez que se utiliza en este ejemplo. Por ende, este código puede ejecutarse considerablemente más lento. \square

```
for (j = 0; j < n; j++)
    for (i = 0; i < n; i++)
        Z[i,j] = 0;
```

(a) Poner un arreglo en ceros, columna por columna.

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        Z[i,j] = 0;
```

(b) Poner un arreglo en ceros, fila por fila.

```
b = ceil(n/M);
for (i = b*p; i < min(n,b*(p+1)); i++)
    for (j = 0; j < n; j++)
        Z[i,j] = 0;
```

(c) Poner un arreglo en ceros, fila por fila en paralelo.

Figura 11.3: Código secuencial y paralelo para poner un arreglo en ceros

Ejemplo 11.3: Suponga que queremos establecer todos los elementos del arreglo Z , almacenados en orden por filas (recuerde la sección 6.4.3), en cero. Las figuras 11.3(a) y (b) recorren el arreglo, columna por columna y fila por fila, respectivamente. Podemos transponer los ciclos en la figura 11.3(a) para llegar a la figura 11.3(b). En términos de localidad espacial, es preferible poner en ceros el arreglo fila por fila, ya que todas las palabras en una línea de caché se ponen en ceros en forma consecutiva. En el método de columna por columna, aun cuando cada línea de caché se reutiliza mediante iteraciones consecutivas del ciclo externo, se limpiará el contenido de las líneas de caché antes de la reutilización si el tamaño de una columna es mayor que el tamaño de la caché. Para un mejor rendimiento, paralelizamos el ciclo externo de la figura 11.3(b) de una manera similar a la que utilizamos en el ejemplo 11.1. \square

Los dos ejemplos anteriores ilustran varias características importantes asociadas con las aplicaciones numéricas que operan sobre los arreglos:

- A menudo, el código de los arreglos tiene muchos ciclos paralelizables.
- Cuando los ciclos tienen paralelismo, sus iteraciones pueden ejecutarse en orden arbitrario; pueden reordenarse para mejorar de manera considerable la localidad de los datos.
- A medida que creamos unidades extensas de cálculos en paralelo que son independientes entre sí, la ejecución de éstas en serie tiende a producir una buena localidad de los datos.

11.1.5 Introducción a la teoría de la transformación afín

Es difícil escribir programas secuenciales correctos y eficientes; es aún más difícil escribir programas en paralelo que sean correctos y eficientes. El nivel de dificultad se incrementa a medida

que disminuye la granularidad del paralelismo explotado. Como podemos ver en la sección anterior, los programadores deben poner atención a la localidad de los datos para obtener un alto rendimiento. Además, la tarea de tomar un programa secuencial existente y paralelizarlo es en extremo difícil. Es difícil atrapar todas las dependencias en el programa, en especial si no es un programa con el que estemos familiarizados. Es todavía más difícil depurar un programa en paralelo, ya que los errores pueden ser no deterministas.

Lo ideal sería que un compilador paralelizador tradujera en forma automática los programas secuenciales ordinarios en programas eficientes en paralelo, y que optimizara la localidad de estos programas. Por desgracia, los compiladores que no tienen un conocimiento de alto nivel sobre la aplicación, sólo pueden preservar la semántica del algoritmo original, que puede no ser susceptible a la paralelización. Además, tal vez los programadores hayan realizado elecciones arbitrarias que limiten el paralelismo del programa.

Se han demostrado éxitos en la paralelización y las optimizaciones de localidad para aplicaciones numéricas en Fortran, que operan sobre arreglos con accesos afines. Sin apuntadores ni aritmética de apuntadores, Fortran es más fácil de analizar. Tenga en cuenta que no todas las aplicaciones tienen accesos afines; lo más notable es que muchas aplicaciones numéricas operan sobre matrices poco densas, cuyos elementos se utilizan en forma indirecta a través de otro arreglo. Este capítulo se enfoca en la paralelización y las optimizaciones de los núcleos, que consisten en su mayor parte en decenas de líneas.

Como muestran los ejemplos 11.2 y 11.3, la paralelización y la optimización de la localidad requieren que razonemos acerca de las distintas instancias de un ciclo y sus relaciones entre sí. Esta situación es muy distinta al análisis del flujo de datos, en donde combinamos la información asociada con todas las instancias en conjunto.

Para el problema de optimizar ciclos con accesos a arreglos, utilizamos tres tipos de espacios. Cada espacio puede considerarse como puntos en una rejilla de una o más dimensiones.

1. El *espacio de iteraciones* es el conjunto de las instancias de ejecución dinámicas en un cálculo; es decir, el conjunto de combinaciones de valores que reciben los índices del ciclo.
2. El *espacio de datos* es el conjunto de elementos del arreglo a los que se accede.
3. El *espacio de procesadores* es el conjunto de procesadores en el sistema. Por lo general, a estos procesadores se les asignan números enteros o vectores de enteros para diferenciarlos.

Como entrada se proporcionan un orden secuencial en el que se ejecutan las iteraciones, y funciones afines de acceso a arreglos (por ejemplo, $X[i, j + 1]$) que especifican cuáles instancias en el espacio de iteraciones acceden a cuáles elementos en el espacio de datos.

Los resultados de la optimización, que de nuevo se representan como funciones afines, definen lo que hace cada procesador, y cuándo lo hace. Para especificar qué hace cada procesador, utilizamos una función afín para asignar a los procesadores instancias en el espacio de iteraciones original. Para especificar cuándo, usamos una función afín para asignar instancias en el espacio de iteraciones a un nuevo ordenamiento. El programa se deriva mediante el análisis de las funciones de acceso a los arreglos, en búsqueda de dependencias de datos y patrones de reutilización.

El siguiente ejemplo ilustrará los tres espacios (de iteraciones, de datos y de procesadores). También introducirá de manera informal los conceptos y cuestiones importantes que debemos tratar al usar estos espacios para paralelizar el código. En secciones posteriores cubriremos con detalle cada uno de los conceptos.

Ejemplo 11.4: La figura 11.4 ilustra el uso de los distintos espacios y sus relaciones en el siguiente programa:

```
float Z[100];  
for (i = 0; i < 10; i++)  
    Z[i+10] = Z[i];
```

Los tres espacios y las asignaciones entre ellos son:

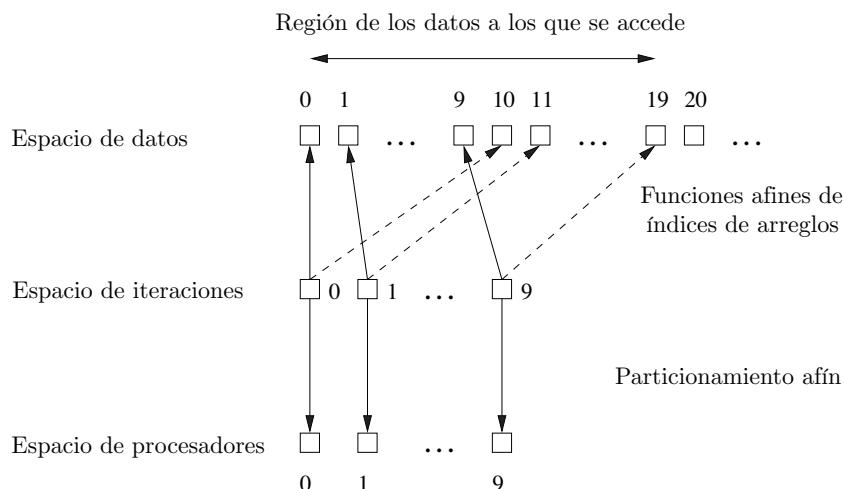


Figura 11.4: Espacio de iteraciones, de datos y de procesadores para el ejemplo 11.4

1. *Espacio de iteraciones:* El espacio de iteraciones es el conjunto de iteraciones, cuyos IDs se proporcionan por los valores que guardan las variables de índice de ciclo. Un *anidamiento de ciclos* con d niveles (es decir, d ciclos anidados) tiene d variables índice, y por ende se modela mediante un espacio d -dimensional. El espacio de iteraciones se delimita mediante los límites inferior y superior de los índices del ciclo. El ciclo de este ejemplo define un espacio unidimensional de 10 iteraciones, etiquetadas mediante los valores del índice del ciclo: $i = 0, 1, \dots, 9$.
 2. *Espacio de datos:* El espacio de datos lo proporcionan directamente las declaraciones del arreglo. En este ejemplo, los elementos en el arreglo se indexan mediante $a = 0, 1, \dots, 99$. Aun cuando todos los arreglos se aplanan en el espacio de direcciones de un programa, tratamos a los arreglos n -dimensionales como espacios n -dimensionales, y suponemos que los índices individuales permanecen dentro de sus límites. En este ejemplo, el arreglo es unidimensional de todas formas.

3. *Espacio de procesadores:* Pretendemos que hay un número ilimitado de procesadores virtuales en el sistema como nuestro objetivo inicial de paralelización. Los procesadores se organizan en un espacio multidimensional, una dimensión para cada ciclo en el anidamiento que deseamos paralelizar. Después de la paralelización, si tenemos menos procesadores físicos que virtuales, dividimos los procesadores virtuales en bloques pares, y asignamos un bloque a cada procesador. En este ejemplo, sólo necesitamos diez procesadores, uno para cada iteración del ciclo. En la figura 11.4 suponemos que los procesadores están organizados en un espacio unidimensional y se enumeran así: 0, 1, ..., 9, en donde la iteración i del ciclo se asigna al procesador i . Si hubiera, por decir, sólo cinco procesadores, podríamos asignar las iteraciones 0 y 1 al procesador 0, las iteraciones 2 y 3 al procesador 1, y así sucesivamente. Como las iteraciones son independientes, no importa cómo las asignemos, siempre y cuando cada uno de los cinco procesadores obtenga dos iteraciones.
4. *Función afín de índice de arreglo:* Cada acceso al arreglo en el código especifica una asignación de una iteración en el espacio de iteraciones, a un elemento del arreglo en el espacio de datos. La función de acceso es afín si implica la multiplicación de las variables índice del ciclo por constantes y la suma de las constantes. Las funciones $i + 10$ e i del índice del arreglo son afines. De la función de acceso podemos conocer la *dimensión* de los datos a los que se accede. En este caso, como cada función de índice tiene una variable de ciclo, el espacio de los elementos del arreglo a los que se accede es unidimensional.
5. *Particionamiento afín:* Para paralelizar un ciclo, usamos una función afín para asignar iteraciones en un espacio de iteraciones a los procesadores en el espacio de procesadores. En nuestro ejemplo, sólo asignamos la iteración i al procesador i . También podemos especificar un nuevo orden de ejecución con las funciones afines. Si deseamos ejecutar el ciclo anterior en forma secuencial, pero a la inversa, podemos especificar la función de ordenamiento brevemente con una expresión afín $10 - i$. Por ende, la iteración 9 es la 1^a iteración en ejecutarse, y así sucesivamente.
6. *Región de los datos a los que se accede:* Para encontrar el mejor particionamiento afín, es útil conocer la región de los datos a los que accede una iteración. Podemos obtener la región de los datos a los que accede si combinamos la información del espacio de iteraciones con la función del índice del arreglo. En este caso, el acceso $Z[i + 10]$ al arreglo entra en contacto con la región $\{a \mid 10 \leq a < 20\}$, y el acceso $Z[i]$ entra en contacto con la región $\{a \mid 0 \leq a < 10\}$.
7. *Dependencia de datos:* Para determinar si el ciclo es paralelizable, preguntamos si hay una dependencia de datos que cruce el límite de cada iteración. Para este ejemplo, primero consideramos las dependencias de los accesos de escritura en el ciclo. Como la función de acceso $Z[i + 10]$ asigna distintas iteraciones a distintas ubicaciones del arreglo, no hay dependencias en relación con el orden en el que las diversas iteraciones escriben valores en el arreglo. ¿Hay una dependencia entre los accesos de lectura y de escritura? Como sólo se escribe en $Z[10], Z[11], \dots, Z[19]$ (mediante el acceso $Z[i + 10]$), y sólo se lee en $Z[0], Z[1], \dots, Z[9]$ (mediante el acceso $Z[i]$), no puede haber dependencias en relación con el orden relativo de una lectura y una escritura. Por ende, este ciclo es paralelizable.

Es decir, cada iteración del ciclo es independiente de todas las demás iteraciones, y podemos ejecutar las iteraciones en paralelo, o en cualquier orden que seleccionemos. Sin embargo, tenga en cuenta que si realizamos un pequeño cambio, por decir al incrementar el límite superior en el índice i del ciclo a 10 o más, entonces habría dependencias, ya que se escribiría en algunos elementos del arreglo Z en una iteración y después se leería 10 iteraciones más adelante. En ese caso, el ciclo no podría paralelizarse por completo, y tendríamos que pensar con cuidado cómo se particionaron las iteraciones entre los procesadores, y cómo ordenamos las iteraciones.

□

Al formular el problema en términos de espacios multidimensionales y asignaciones afines entre estos espacios, podemos usar técnicas matemáticas estándar para resolver el problema de la paralelización y la optimización de la localidad en forma general. Por ejemplo, la región de datos a los que se accede puede encontrarse mediante la eliminación de variables, usando el algoritmo de eliminación de Fourier-Motzkin. La dependencia de datos demuestra ser equivalente al problema de la programación lineal entera. Por último, para encontrar el particionamiento afín es necesario resolver un conjunto de restricciones lineales. No se preocupe si no está familiarizado con estos conceptos, ya que los explicaremos a partir de la sección 11.3.

11.2 Multiplicación de matrices: un ejemplo detallado

Vamos a presentar muchas de las técnicas utilizadas por los compiladores paralelos en un ejemplo extendido. En esta sección, exploraremos el conocido algoritmo de multiplicación de matrices, para mostrar que no es común optimizar ni siquiera un programa paralelizable simple y sencillo. Veremos cómo al rescribir el código se puede mejorar la localidad de los datos; es decir, los procesadores pueden hacer su trabajo con mucho menos comunicación (con la memoria global o con otros procesadores, dependiendo de la arquitectura) que si se elige el programa directo. También hablaremos acerca de cómo el conocimiento de la existencia de líneas de caché que guardan varios elementos consecutivos de datos puede mejorar el tiempo de ejecución de programas como la multiplicación de matrices.

11.2.1 El algoritmo de multiplicación de matrices

En la figura 11.5 podemos ver un programa típico de multiplicación de matrices.² Recibe dos matrices de $n \times n$, X y Y , y produce su producto en una tercera matriz de $n \times n$, Z . Recuerde que Z_{ij} (el elemento de la matriz Z que está en la fila i y la columna j) debe convertirse en $\sum_{k=1}^n X_{ik}Y_{kj}$.

²En los programas en seudocódigo de este capítulo, por lo general, utilizaremos la sintaxis de C, pero para que los accesos a arreglos multidimensionales (la cuestión central para la mayor parte del capítulo) sean más fáciles de leer, utilizaremos referencias a arreglos al estilo Fortran; es decir, $Z[i, j]$ en vez de $Z[i][j]$.

```

for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) {
    Z[i,j] = 0.0;
    for (k = 0; k < n; k++)
      Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
  }
}

```

Figura 11.5: El algoritmo básico de multiplicación de matrices

El código de la figura 11.5 genera n^2 resultados, cada uno de las cuales es un producto interno entre una fila y una columna de los dos operandos tipo matriz. Sin duda, los cálculos de cada uno de los elementos de Z son independientes y pueden ejecutarse en paralelo.

Entre más grande sea n , más veces entrará el algoritmo en contacto con cada elemento. Es decir, hay $3n^2$ ubicaciones entre las tres matrices, pero el algoritmo realiza n^3 operaciones, cada una de las cuales multiplica un elemento de X por un elemento de Y , y suma el producto a un elemento de Z . Por ende, el algoritmo hace un uso intensivo de cálculos y los accesos a memoria no deben, en principio, constituir un cuello de botella.

Ejecución serial de la multiplicación de matrices

Vamos a considerar primero cómo se comporta este programa cuando se ejecuta de manera secuencial en un uniprocesador. El ciclo más interno lee y escribe en el mismo elemento de Z , y utiliza una fila de X y una columna de Y . $Z[i, j]$ puede almacenarse con facilidad en un registro y no requiere accesos a memoria. Suponga, sin perder la generalidad, que la matriz se distribuye en orden por filas, y que c es el número de elementos del arreglo en una línea de caché.

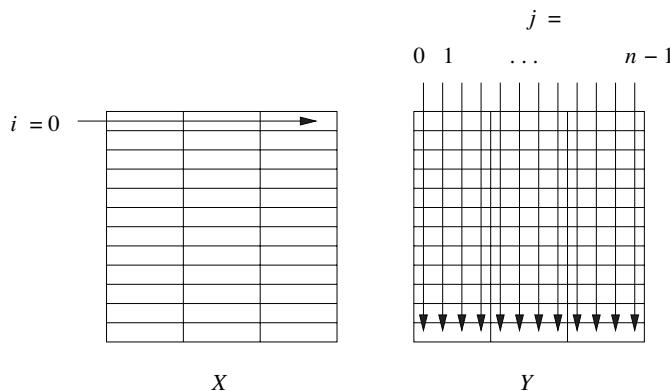


Figura 11.6: El patrón de acceso de datos en la multiplicación de matrices

La figura 11.6 sugiere el patrón de acceso a medida que ejecutamos una iteración del ciclo externo de la figura 11.5. En especial, la imagen muestra la primera iteración, con $i = 0$. Cada

vez que avanzamos de un elemento de la primera fila de X al siguiente, visitamos cada elemento en una sola columna de Y . En la figura 11.6 podemos ver la supuesta organización de las matrices en líneas de caché. Es decir, cada rectángulo pequeño representa a una línea de caché que contiene cuatro elementos del arreglo (es decir, $c = 4$ y $n = 12$ en la imagen).

El acceso a X impone una carga ligera sobre la caché. Una fila de X se esparce entre sólo n/c líneas de caché. Suponiendo que todos estos elementos caben en la caché, sólo se producen n/c fallos en la caché para un valor fijo del índice i , y el número total de fallos para todos los elementos de X es de n^2/c , el mínimo posible (suponemos que n es divisible entre c , por conveniencia).

Sin embargo, al usar una fila de X , el algoritmo de multiplicación de matrices accede a todos los elementos de Y , columna por columna. Es decir, cuando $j = 0$, el ciclo interno lleva a la caché la primera columna completa de Y . Observe que los elementos de esa columna se almacenan entre n líneas distintas de caché. Si la caché es lo bastante grande (o si n es lo bastante pequeña) para guardar n líneas de caché, y ningún otro uso de la caché obliga a expulsar algunas de estas líneas de caché, entonces la columna para $j = 0$ aún estará en la caché cuando necesitemos la segunda columna de Y . En ese caso, no habrá otros n fallos de caché al leer Y , hasta $j = c$, momento en el cual debemos llevar a la caché un conjunto totalmente distinto de líneas de caché para Y . Así, para completar la primera iteración del ciclo externo (con $i = 0$) se requieren entre n^2/c y n^2 fallos de caché, dependiendo de si las columnas de las líneas de caché pueden sobrevivir de una iteración del segundo ciclo a la siguiente.

Además, al completar el ciclo externo, para $i = 1, 2, \dots$ y así sucesivamente, podemos tener muchos fallos adicionales en la caché al leer Y , o ninguno en lo absoluto. Si la caché es lo bastante grande como para que puedan residir en ella todas las n^2/c líneas de caché que contienen a Y , entonces no necesitamos más fallos de caché. En consecuencia, el número total de fallos de caché es $2n^2/c$, la mitad para X y la mitad para Y . No obstante, si la caché puede guardar una columna de Y , pero no todo el contenido de Y , entonces debemos llevar a todos los elementos de Y en la caché de nuevo, cada vez que realicemos una iteración del ciclo externo. Es decir, el número de fallos en la caché es de $n^2/c + n^3/c$; el primer término es para X y el segundo para Y . Peor aún, si no podemos guardar ni siquiera una columna de Y en la caché, entonces tenemos n^2 fallos de caché por cada iteración del ciclo externo, y un total de $n^2/c + n^3$ fallos de caché.

Paralelización fila por fila

Ahora vamos a considerar como podríamos usar cierto número de procesadores, por decir p procesadores, para agilizar la ejecución de la figura 11.5. Un método obvio para la paralelización de la multiplicación de matrices es asignar distintas filas de Z a distintos procesadores. Un procesador es responsable de n/p filas consecutivas (suponemos que n es divisible entre p , por conveniencia). Con esta división del trabajo, cada procesador debe acceder a n/p filas de las matrices X y Z , pero a toda la matriz Y completa. Un procesador calculará n^2/p elementos de Z , realizando para ello n^3/p operaciones de multiplicación y suma.

Aunque, en consecuencia, el tiempo de cálculo disminuye en proporción a p , el costo de comunicación se eleva en proporción a p . Es decir, cada uno de p procesadores tiene que leer n^2/p elementos de X , pero todos los n^2 elementos de Y . El número total de líneas de caché que deben entregarse a las cachés de los p procesadores es $n^2/c + pm^2/c$; los dos términos son

para entregar X y copias de Y , respectivamente. A medida que p se acerca a n , el tiempo de cálculo se convierte en $O(n^2)$ mientras que el costo de comunicación es $O(n^3)$. Es decir, el bus en el que se mueven los datos entre la memoria y las cachés de los procesadores se convierte en el cuello de botella. En consecuencia, con la distribución propuesta de los datos, si utilizamos un gran número de procesadores para compartir los cálculos en realidad se podría disminuir la velocidad de los cálculos, en vez de agilizarlos.

11.2.2 Optimizaciones

El algoritmo de multiplicación de matrices de la figura 11.5 muestra que, aun cuando un algoritmo puede *reutilizar* los mismos datos, puede tener una mala localidad de los mismos. La reutilización de datos produce una ocurrencia en la caché, sólo si la reutilización se lleva a cabo con la suficiente prontitud, antes de que los datos se desplacen de la caché. En este caso, n^2 operaciones de multiplicación-suma separan la reutilización del mismo elemento de datos en la matriz Y , por lo que la localidad es mala. De hecho, n operaciones separan la reutilización de la misma línea de caché en Y . Además, en un multiprocesador, la reutilización puede producir una ocurrencia en la caché sólo si el mismo procesador reutiliza los datos. Cuando consideramos una implementación en paralelo en la sección 11.2.1, vimos que cada procesador tenía que utilizar los elementos de Y . Por ende, la reutilización de Y no se convierte en localidad.

Modificación de la distribución de los datos

Una manera de mejorar la localidad de un programa es modificar la distribución de sus estructuras de datos. Por ejemplo, la acción de almacenar Y en orden por columnas hubiera mejorado la reutilización de las líneas de caché para la matriz Y . La capacidad de aplicación de este método es limitada, debido a que, por lo general, se utiliza la misma matriz en distintas operaciones. Si Y jugara el papel de X en otra multiplicación de matrices, entonces sufriría por estar almacenada en orden por columnas, ya que la primera matriz en una multiplicación se almacena mejor en orden por filas.

Uso de bloques

Algunas veces es posible modificar el orden de ejecución de las instrucciones para mejorar la localidad de los datos. Sin embargo, la técnica de intercambiar ciclos no mejora la rutina de multiplicación de matrices. Suponga que la rutina se escribiera para generar una columna de la matriz Z a la vez, en vez de una fila a la vez. Es decir, hacer que el ciclo j sea el ciclo externo y que el ciclo i sea el interno. Suponiendo que las matrices están almacenadas en orden por fila, la matriz Y disfruta de una mejor localidad espacial y temporal, pero sólo a expensas de la matriz X .

El *uso de bloques* es otra forma de reordenar las iteraciones en un ciclo, que puede mejorar en forma considerable la localidad de un programa. En vez de calcular el resultado una fila o columna a la vez, dividimos la matriz en submatrices, o *bloques*, como lo sugiere la figura 11.7, y ordenamos las operaciones de manera que se utilice todo un bloque completo durante un corto periodo. Por lo general, los bloques son cuadros con un lado de longitud B . Si B divide uniformemente a n , entonces todos los bloques son cuadrados. Si B no divide uniformemente

a n , entonces los bloques en los extremos inferior y derecho tendrán uno o ambos lados de longitud menor a B .

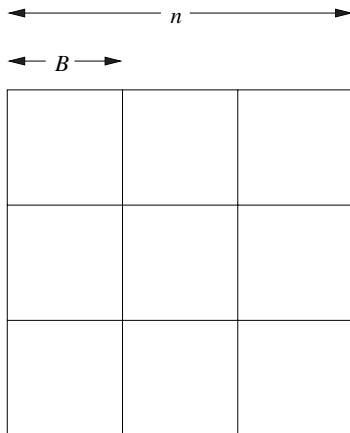


Figura 11.7: Una matriz dividida en bloques de tamaño B

La figura 11.8 muestra una versión del algoritmo básico de multiplicación de matrices, en donde las tres matrices se han dividido en cuadros de tamaño B . Como en la figura 11.5, se asume que se han inicializado a 0 todos los elementos en Z . Suponemos que B divide a n ; si no es así, entonces debemos modificar la línea (4) para que el límite superior sea $\min(ii + B, n)$, y lo hacemos de manera similar para las líneas (5) y (6).

```

1) for (ii = 0; ii < n; ii = ii+B)
2)     for (jj = 0; jj < n; jj = jj+B)
3)         for (kk = 0; kk < n; kk = kk+B)
4)             for (i = ii; i < ii+B; i++)
5)                 for (j = jj; j < jj+B; j++)
6)                     for (k = kk; k < kk+B; k++)
7)                         Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];

```

Figura 11.8: Multiplicación de matrices con uso de bloques

Los tres ciclos externos, las líneas (1) a (3), utilizan los índices ii , jj y kk , que siempre se incrementan mediante B y, por lo tanto, siempre marcan la arista izquierda o superior de algunos bloques. Con valores fijos de ii , jj y kk , las líneas (4) a (7) permiten que los bloques con las esquinas superiores izquierdas $X[ii, kk]$ y $Y[kk, jj]$ realicen todas las posibles contribuciones al bloque con la esquina superior izquierda $Z[ii, jj]$.

Si elegimos B en forma apropiada, podemos reducir de manera considerable el número de fallos de caché, comparados con el algoritmo básico, cuando no pueden caber todos los elementos de X , Y o Z en la caché. Elija B de forma que sea posible acomodar un bloque de cada una

Otra visión de la multiplicación de matrices basada en bloques

Podemos imaginar que las matrices X , Y y Z de la figura 11.8 no son matrices de $n \times n$ de números de punto flotante, sino matrices de $(n/B) \times (n/B)$ cuyos elementos son en sí matrices de $B \times B$ de números de punto flotante. Las líneas (1) a (3) de la figura 11.8 son entonces como los tres ciclos del algoritmo básico de la figura 11.5, pero con n/B como el tamaño de las matrices, en vez de n . Entonces, podemos considerar que las líneas (4) a (7) de la figura 11.8 implementan una sola operación de multiplicar y sumar de la figura 11.5. Observe que en esta operación, el paso de multiplicación individual es un paso de multiplicación de matrices, y utiliza el algoritmo básico de la figura 11.5 en los números de punto flotante que son elementos de las dos matrices involucradas. La suma de matrices es la suma a nivel de elementos de los números de punto flotante.

de las matrices en la caché. Debido al orden de los ciclos, en realidad necesitamos cada bloque Z en la caché sólo una vez, por lo que (como en el análisis del algoritmo básico de la sección 11.2.1) no contaremos los fallos de caché debido a Z .

Para llevar un bloque de X o Y a la caché, se requieren B^2/c fallos de caché; recuerde que c es el número de elementos en una línea de caché. Sin embargo, con bloques fijos de X y Y , realizamos B^3 operaciones de multiplicación y suma en las líneas (4) a (7) de la figura 11.8. Como toda la multiplicación de matrices requiere n^3 operaciones de multiplicación y suma, el número de veces que necesitamos llevar un par de bloques a la caché es n^3/B^3 . Como requerimos $2B^2/c$ fallos de caché cada vez que lo hacemos, el número total de fallos de caché es $2n^3/Bc$.

Es interesante comparar $2n^3/Bc$ con los estimados que se proporcionan en la sección 11.2.1. Ahí vimos que si pueden caber todas las matrices completas en la caché, entonces basta con $O(n^2/c)$ fallos de caché. Sin embargo, en ese caso podemos elegir $B = n$; es decir, hacer que cada matriz sea un solo bloque. De nuevo, obtenemos $O(n^2/c)$ como nuestra estimación de los fallos de caché. Por otro lado, observamos que si no caben las matrices completas en la caché, requerimos $O(n^3/c)$ fallos de caché, o incluso $O(n^3)$ fallos de caché. En ese caso, suponiendo que aún podemos elegir un valor de B considerablemente grande (por ejemplo, B podría ser 200, y de todas formas podríamos meter tres bloques de números de 8 bytes en una caché de un megabyte), hay una gran ventaja en cuanto al uso de bloques en una multiplicación de matrices.

La técnica de los bloques puede volver a aplicarse en cada nivel de la jerarquía de memoria. Por ejemplo, tal vez queramos optimizar el uso de los registros, guardando los operandos de una multiplicación de matrices de 2×2 en los registros. Elegimos tamaños de bloques cada vez más grandes para los distintos niveles de cachés y la memoria física.

De manera similar, podemos distribuir los bloques entre los procesadores para disminuir al mínimo el tráfico de datos. Los experimentos demostraron que dichas optimizaciones pueden mejorar el rendimiento de un uniprocesador por un factor de 3, y el aumento en velocidad en un multiprocesador está cerca de ser lineal, con respecto al número de procesadores utilizados.

11.2.3 Interferencia de la caché

Por desgracia, hay algo más en la historia de la utilización de la caché. La mayoría de las cachés no son completamente asociativas (vea la sección 7.4.2). En una caché con asignación directa, si n es un múltiplo del tamaño de la caché, entonces todos los elementos en la misma fila de un arreglo de $n \times n$ competirán por la misma ubicación en la caché. En ese caso, al llevar el segundo elemento de una columna se descartará la línea de caché del primero, aun cuando la caché tiene la capacidad de mantener ambas líneas al mismo tiempo. A esta situación se le conoce como *interferencia de la caché*.

Hay varias soluciones para este problema. La primera es reordenar los datos de una vez por todas, para que los datos a los que se acceda se distribuyan en ubicaciones de datos consecutivas. La segunda es incrustar el arreglo de $n \times n$ en un arreglo más grande de $m \times n$, en donde m se elige de manera que se disminuya al mínimo el problema de la interferencia. La tercera es que, en algunos casos podemos elegir un tamaño de bloque que garantice evitar la interferencia.

11.2.4 Ejercicios para la sección 11.2

Ejercicio 11.2.1: El algoritmo de multiplicación de matrices basado en bloques de la figura 11.8 no inicializa la matriz Z a cero, como lo hace el código de la figura 11.5. Agregue los pasos que inicializan Z a ceros en la figura 11.8.

11.3 Espacios de iteraciones

La motivación para este estudio es explotar las técnicas que, en escenarios simples como la multiplicación de matrices de la sección 11.2, fueron bastante simples y directas. En el escenario más general se aplican las mismas técnicas, pero son mucho menos intuitivas. Pero mediante la aplicación de ciertas técnicas del álgebra lineal, podemos hacer que todo funcione en el escenario general.

Como vimos en la sección 11.1.5, hay tres tipos de espacios en nuestro modelo de transformación: espacio de iteraciones, espacio de datos y espacio de procesadores. Aquí empezaremos con el espacio de iteraciones. El espacio de iteraciones de un anidamiento de ciclos se define como todas las combinaciones de valores de índices de los ciclos en el anidamiento.

A menudo, el espacio de iteraciones es rectangular, como en el ejemplo de multiplicación de matrices de la figura 11.5. Ahí, cada uno de los ciclos anidados tenía un límite inferior de 0 y un límite superior de $n - 1$. Sin embargo, en anidamientos de ciclos más complicados, pero aun así bastante realistas, los límites inferiores y superiores en un índice de ciclo pueden depender de los valores de los índices de los ciclos externos. En breve veremos un ejemplo.

11.3.1 Construcción de espacios de iteraciones a partir de anidamientos de ciclos

Para empezar, vamos a describir el tipo de anidamientos de ciclos que pueden manejarse mediante las técnicas a desarrollar. Cada ciclo tiene un solo índice de ciclo, el cual suponemos se incrementa en 1 durante cada iteración. Esta suposición es sin pérdida de generalidad, ya que si el incremento es en base al entero $c > 1$, siempre podemos sustituir los usos del índice i por

los usos de $ci + a$ para cierta constante a positiva o negativa, y después incrementar i por 1 en el ciclo. Los límites del ciclo deben escribirse como expresiones afines de los índices del ciclo externo.

Ejemplo 11.5: Consideré el siguiente ciclo:

```
for (i = 2; i <= 100; i = i+3)
    Z[i] = 0;
```

que incrementa a i en 3 cada vez que se recorre el ciclo. El efecto es asignar 0 a cada uno de los elementos $Z[2], Z[5], Z[8], \dots, Z[98]$. Podemos obtener el mismo efecto con:

```
for (j = 0; j <= 32; j++)
    Z[3*j+2] = 0;
```

Es decir, sustituimos $3j + 2$ por i . El límite inferior $i = 2$ se convierte en $j = 0$ (sólo hay que resolver $3j + 2 = 2$ para j), y el límite superior $i \leq 100$ se convierte en $j \leq 32$ (hay que simplificar $3j + 2 \leq 100$ para obtener $j \leq 32.67$ y redondear, ya que j tiene que ser un entero). \square

Por lo general, utilizaremos ciclos for en los anidamientos de ciclos. Un ciclo while o repeat puede sustituirse por un ciclo for si hay un índice y límites superior e inferior para el índice, como sería el caso en algo parecido al ciclo de la figura 11.9(a). Un ciclo for como `for (i=0; i<100; i++)` sirve para el mismo propósito exacto.

Sin embargo, algunos ciclos while o repeat no tienen un límite evidente. Por ejemplo, la figura 11.9(b) puede o no terminar, pero no hay forma de saber qué condición sobre i en el cuerpo no visto del ciclo hace que éste interrumpa su ejecución. La figura 11.9(c) es otro caso problema. Por ejemplo, la variable n podría ser un parámetro de una función. Sabemos que el ciclo itera n veces, pero no sabemos cuál es el valor de n en tiempo de compilación, y de hecho podemos esperar que varias ejecuciones distintas del ciclo se ejecutarán distintos números de veces. En casos como (b) y (c), debemos tratar al límite superior sobre i como infinito.

Un anidamiento de ciclos con d niveles puede modelarse mediante un espacio d -dimensional. Las dimensiones están ordenadas, en donde la k -ésima dimensión representa al k -ésimo ciclo anidado, contando desde el ciclo más externo, hacia dentro. Un punto (x_1, x_2, \dots, x_d) en este espacio representa valores para todos los índices de ciclo; el índice del ciclo más externo tiene el valor x_1 , el índice del segundo ciclo tiene el valor x_2 , y así en lo sucesivo. El índice del ciclo más interno tiene el valor x_d .

Pero no todos los puntos en este espacio representan combinaciones de índices que ocurren realmente durante la ejecución de anidamientos de ciclos. Como una función afín de los índices de los ciclos externos, cada límite de ciclo inferior y superior define una desigualdad que divide el espacio de iteraciones en dos medios espacios: aquellas que son iteraciones en el ciclo (el medio espacio *positivo*) y las que no lo son (el medio espacio *negativo*). La conjunción (AND lógico) de todas las igualdades lineales representa la intersección de los medios espacios positivos, lo cual define a un poliedro convexo, que llamaremos el *espacio de iteraciones* para el anidamiento de ciclos. Un *poliedro convexo* tiene la propiedad de que si hay dos puntos en el

```

i = 0;
while (i<100) {
    <algunas instrucciones que no implican a i>
    i = i+1;
}

```

(a) Un ciclo while con límites evidentes.

```

i = 0;
while (1) {
    <algunas instrucciones>
    i = i+1;
}

```

(b) No está claro cuándo termina este ciclo, o si alguna vez lo hará.

```

i = 0;
while (i<n) {
    <algunas instrucciones que no implican a i o a n>
    i = i+1;
}

```

(c) No sabemos el valor de n , por lo que no sabemos cuándo termina este ciclo.

Figura 11.9: Algunos ciclos while

poliedro, todos los puntos en la línea entre ellos también se encuentran en el poliedro. Todas las iteraciones en el ciclo se representan mediante los puntos con las coordenadas enteras que se encuentran dentro del poliedro descrito por las desigualdades de los límites del ciclo. Y por el contrario, todos los puntos dentro del poliedro representan iteraciones del anidamiento de ciclos en cierto momento.

```

for (i = 0; i <= 5; i++)
    for (j = i; j <= 7; j++)
        Z[j,i] = 0;

```

Figura 11.10: Un anidamiento de ciclos bidimensional

Ejemplo 11.6: Considere el anidamiento de ciclos de 2 dimensiones de la figura 11.10. Podemos modelar este anidamiento de ciclos de 2 niveles mediante el poliedro bidimensional que se muestra en la figura 11.11. Los dos ejes representan los valores de los índices de ciclo i y j . El índice i puede recibir cualquier valor entero entre 0 y 5; el índice j puede recibir cualquier valor tal que $i \leq j \leq 7$. \square

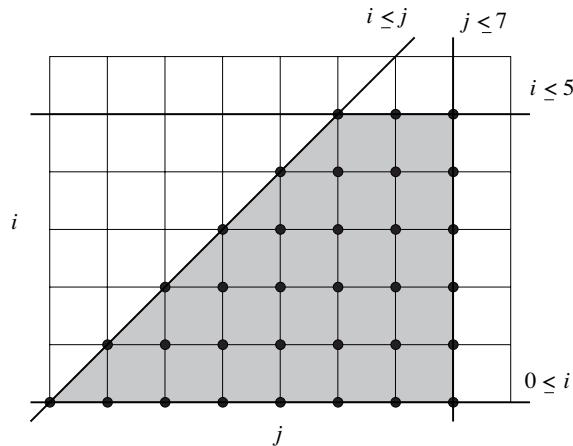


Figura 11.11: El espacio de iteraciones del ejemplo 11.6

Espacios de iteraciones y accesos a arreglos

En el código de la figura 11.10, el espacio de iteraciones también es la parte del arreglo A a la que accede el código. Ese tipo de acceso, en donde los índices de arreglo son también índices de ciclo en cierto orden, es muy común. Sin embargo, no debemos confundir el espacio de las iteraciones, cuyas dimensiones son índices de ciclo, con el espacio de datos. Si hubiéramos utilizado en la figura 11.10 un acceso a arreglo como $A[2*i, i + j]$ en vez de $A[i, j]$, la diferencia habría sido aparente.

11.3.2 Orden de ejecución para los anidamientos de ciclos

Una ejecución secuencial de un anidamiento de ciclos recorre las iteraciones en su espacio de iteraciones, en orden lexicográfico ascendente. Un vector $\mathbf{i} = [i_0, i_1, \dots, i_n]$ es *lexicográficamente menor* que otro vector $\mathbf{i}' = [i'_0, i'_1, \dots, i'_{n'}]$, lo cual se escribe como $\mathbf{i} \prec \mathbf{i}'$, y si sólo existe un valor de $m < \min(n, n')$ tal que $[i_0, i_1, \dots, i_m] = [i_0, i'_1, \dots, i'_m]$ y que $i_{m+1} < i'_{m+1}$. Observe que $m = 0$ es posible, y de hecho común.

Ejemplo 11.7: Con i como el ciclo externo, las iteraciones en el anidamiento de ciclos del ejemplo 11.6 se ejecutan en el orden mostrado en la figura 11.12. \square

11.3.3 Formulación de matrices de desigualdades

Las iteraciones en un ciclo de n niveles puede representarse en forma matemática como:

$$\{\mathbf{i} \text{ en } Z^d \mid \mathbf{Bi} + \mathbf{b} \geq \mathbf{0}\} \quad (11.1)$$

$$\begin{array}{cccccccccc}
 [0,0], & [0,1], & [0,2], & [0,3], & [0,4], & [0,5], & [0,6], & [0,7] \\
 [1,1], & [1,2], & [1,3], & [1,4], & [1,5], & [1,6], & [1,7] \\
 [2,2], & [2,3], & [2,4], & [2,5], & [2,6], & [2,7] \\
 [3,3], & [3,4], & [3,5], & [3,6], & [3,7] \\
 [4,4], & [4,5], & [4,6], & [4,7] \\
 [5,5], & [5,6], & [5,7]
 \end{array}$$

Figura 11.12: Orden de iteración para el anidamiento de ciclos de la figura 11.10

Aquí,

1. \mathbb{Z} , como es convencional en matemáticas, representa al conjunto de enteros: positivos, negativos y cero.
2. \mathbf{B} es una matriz entera de $d \times d$.
3. \mathbf{b} es un vector entero de longitud d .
4. $\mathbf{0}$ es un vector de d 0's.

Ejemplo 11.8: Podemos escribir las desigualdades del ejemplo 11.6 como en la figura 11.13. Es decir, el rango de i se describe mediante $i \geq 0$ y $i \leq 5$; el rango de j se describe mediante $j \geq i$ y $j \leq 7$. Debemos colocar cada una de estas desigualdades en la forma $ui + vj + w \geq 0$. Así, $[u, v]$ se convierte en una fila de la matriz \mathbf{B} en la desigualdad (11.1), y w se convierte en el componente correspondiente del vector \mathbf{b} . Por ejemplo, $i \geq 0$ es de esta forma, con $u = 1$, $v = 0$ y $w = 0$. Esta desigualdad se representa mediante la primera fila de \mathbf{B} y el elemento superior de \mathbf{b} en la figura 11.13.

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 5 \\ 0 \\ 7 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Figura 11.13: La multiplicación de matriz-vector y una desigualdad de vectores representan las desigualdades que definen un espacio de iteraciones

Como otro ejemplo, la desigualdad $i \leq 5$ es equivalente a $(-1)i + (0)j + 5 \geq 0$, y se representa mediante la segunda fila de \mathbf{B} y \mathbf{b} en la figura 11.13. Además, $j \geq i$ se convierte en $(-1)i + (1)j + 0 \geq 0$ y se representa mediante la tercera fila. Por último, $j \leq 7$ se convierte en $(0)i + (-1)j + 7 \geq 0$ y es la última fila de la matriz y el vector. \square

Manipulación de desigualdades

Para convertir las desigualdades, como en el ejemplo 11.8, podemos realizar transformaciones en forma muy parecida a lo que hacemos con las desigualdades; por ejemplo, sumar o restar de ambos lados, o multiplicar ambos lados por una constante. La única regla especial que debemos recordar es que, cuando multiplicamos ambos lados por un número negativo, tenemos que invertir la dirección de la desigualdad. Por ende, $i \leq 5$, multiplicada por -1 , se convierte en $-i \geq -5$. Al sumar 5 a ambos lados obtenemos $-i + 5 \geq 0$, que en esencia es la segunda fila de la figura 11.13.

11.3.4 Incorporación de constantes simbólicas

Algunas veces debemos optimizar un anidamiento de ciclos que involucra a ciertas variables que son invariantes de ciclo para todos los ciclos en el anidamiento. A dichas variables les podemos llamar *constantes simbólicas*, pero para describir los límites de un espacio de iteraciones tenemos que tratarlas como variables y crear una entrada para ellas en el vector de índices de ciclo; es decir, el vector \mathbf{i} en la formulación general de desigualdades (11.1).

Ejemplo 11.9: Consideré el siguiente ciclo simple:

```
for (i = 0; i <= n; i++) {
    ...
}
```

Este ciclo define un espacio de iteraciones unidimensional, con el índice i , delimitado por $i \geq 0$ e $i \leq n$. Como n es una constante simbólica, debemos incluirla como una variable, lo cual nos proporciona un vector de índices de ciclo $[i, n]$. En forma de matriz-vector, este espacio de iteraciones se define mediante lo siguiente:

$$\left\{ i \text{ en } \mathbb{Z} \mid \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ n \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\}.$$

Observe que, aunque el vector de índices de arreglo tiene dos dimensiones, sólo la primera de éstas, que representa a i , forma parte de la salida: el conjunto de puntos que yacen con el espacio de iteraciones. \square

11.3.5 Control del orden de ejecución

Las desigualdades lineales extraídas de los límites inferior y superior de un cuerpo de ciclo definen a un conjunto de iteraciones sobre un poliedro convexo. Como tal, la representación no asume un orden de ejecución entre las iteraciones dentro del espacio de iteraciones. El programa original impone un orden secuencial en las iteraciones, que viene siendo el orden lexicográfico con respecto a las variables índice de ciclo, ordenadas desde el más externo hasta

el más interno. Sin embargo, las iteraciones en el espacio pueden ejecutarse en cualquier orden, siempre y cuando se respeten sus dependencias de datos (es decir, no cambia el orden en el que se llevan a cabo las escrituras y lecturas de cualquier elemento de un arreglo mediante las diversas instrucciones de asignación dentro del anidamiento de ciclos).

Es difícil elegir un ordenamiento que respete las dependencias de datos y optimice para la localidad de los datos y el paralelismo, y lo trataremos más adelante, empezando en la sección 11.7. Aquí supondremos que se da un ordenamiento válido y conveniente, y mostraremos cómo generar código para asegurar que se cumpla ese orden. Vamos a empezar por mostrar un ordenamiento alternativo para el ejemplo 11.6.

Ejemplo 11.10: No hay dependencias entre las iteraciones en el programa del ejemplo 11.6. Por lo tanto, podemos ejecutar las iteraciones en orden arbitrario, en secuencia o en forma concurrente. Como la iteración $[i, j]$ accede al elemento $Z[i, j]$ en el código, el programa original visita el arreglo en el orden de la figura 11.14(a). Para mejorar la localidad espacial, preferimos visitar palabras contiguas en el arreglo en forma consecutiva, como en la figura 11.14(b).

Este patrón de acceso se obtiene si ejecutamos las iteraciones en el orden mostrado en la figura 11.14(c). Es decir, en vez de recorrer el espacio de iteraciones en la figura 11.11 en forma horizontal, lo recorremos en forma vertical, por lo que j se convierte en el índice del ciclo externo. El código que ejecuta las iteraciones en el orden anterior es:

```
for (j = 0; j <= 7; j++)
    for (i = 0; i <= min(5,j); i++)
        Z[j,i] = 0;
```

□

Dado un poliedro convexo y un ordenamiento de las variables índice, ¿cómo generamos los límites de ciclo que recorren el espacio en orden lexicográfico de las variables? En el ejemplo anterior, la restricción $i \leq j$ se muestra como un límite inferior para el índice j en el ciclo interno del programa original, pero como un límite superior para el índice i , de nuevo en el ciclo interno, en el programa transformado.

Los límites del ciclo más externo, expresados como combinaciones lineales de constantes simbólicas y constantes, definen el rango de todos los valores posibles que puede tomar. Los límites para las variables del ciclo interno se expresan como combinaciones lineales de variables índice del ciclo externo, constantes simbólicas y constantes. Definen el rango que puede tomar la variable para cada combinación de valores en las variables del ciclo externo.

Proyección

Hablando en sentido geométrico, podemos encontrar los límites de ciclo del índice del ciclo externo en un anidamiento de ciclos con dos niveles, *proyectando* el poliedro convexo que representa el espacio de iteraciones hacia la dimensión externa del espacio. La proyección de un poliedro en un espacio con menores dimensiones es, por intuición, la sombra que proyecta el objeto hacia ese espacio. La proyección del espacio de iteraciones bidimensional en la figura 11.11 hacia el eje i es la línea vertical de 0 a 5; y la proyección hacia el eje j es la línea horizontal

$$\begin{array}{cccccccc}
 Z[0,0], & Z[1,0], & Z[2,0], & Z[3,0], & Z[4,0], & Z[5,0], & Z[6,0], & Z[7,0] \\
 Z[1,1], & Z[2,1], & Z[3,1], & Z[4,1], & Z[5,1], & Z[6,1], & Z[1,7] \\
 Z[2,2], & Z[3,2], & Z[4,2], & Z[5,2], & Z[6,2], & Z[7,2] \\
 Z[3,3], & Z[4,3], & Z[5,3], & Z[6,3], & Z[7,3] \\
 Z[4,4], & Z[5,4], & Z[6,4], & Z[7,4] \\
 Z[5,5], & Z[6,5], & Z[7,5]
 \end{array}$$

(a) Orden de acceso original.

$$\begin{array}{cccc}
 Z[0,0] \\
 Z[1,0], & Z[1,1] \\
 Z[2,0], & Z[2,1], & Z[2,2] \\
 Z[3,0], & Z[3,1], & Z[3,2], & Z[3,3] \\
 Z[4,0], & Z[4,1], & Z[4,2], & Z[4,3], & Z[4,4] \\
 Z[5,0], & Z[5,1], & Z[5,2], & Z[5,3], & Z[5,4], & Z[5,5] \\
 Z[6,0], & Z[6,1], & Z[6,2], & Z[6,3], & Z[6,4], & Z[6,5] \\
 Z[7,0], & Z[7,1], & Z[7,2], & Z[7,3], & Z[7,4], & Z[7,5]
 \end{array}$$

(b) Orden preferido de acceso.

$$\begin{array}{cccccc}
 [0,0] \\
 [0,1], & [1,1] \\
 [0,2], & [1,2], & [2,2] \\
 [0,3], & [1,3], & [2,3], & [3,3] \\
 [0,4], & [1,4], & [2,4], & [3,4], & [4,4] \\
 [0,5], & [1,5], & [2,5], & [3,5], & [4,5], & [5,5] \\
 [0,6], & [1,6], & [2,6], & [3,6], & [4,6], & [5,6] \\
 [0,7], & [1,7], & [2,7], & [3,7], & [4,7], & [5,7]
 \end{array}$$

(c) Orden preferido de iteraciones.

Figura 11.14: Reordenamiento de los accesos e iteraciones para un anidamiento de ciclos

de 0 a 7. Cuando proyectamos un objeto tridimensional a lo largo del eje z hacia un plano x y y bidimensional, eliminamos la variable z , perdiendo la altura de los puntos individuales y sólo registrando la huella bidimensional del objeto en el plano x - y .

La generación de límites de ciclo es sólo uno de los diversos usos de la proyección. La proyección puede definirse formalmente de la siguiente manera. Suponga que S es un poliedro n -dimensional. La proyección de S hacia la primera m de sus dimensiones es el conjunto de puntos (x_1, x_2, \dots, x_m) tales que para ciertos valores $x_{m+1}, x_{m+2}, \dots, x_n$, el vector $[x_1, x_2, \dots, x_n]$ está en S . Podemos calcular la proyección mediante el uso de la *eliminación de Fourier-Motzkin*, como se muestra a continuación:

Algoritmo 11.11: Eliminación de Fourier-Motzkin.

ENTRADA: Un poliedro S con las variables x_1, x_2, \dots, x_n . Es decir, S es un conjunto de restricciones lineales que involucran a las variables x_i . Una variable x_m dada se especifica como la variable a eliminar.

SALIDA: Un poliedro S' con las variables $x_1, \dots, x_{m-1}, x_{m+1}, \dots, x_n$ (es decir, todas las variables de S excepto x_m) que es la proyección de S en otras dimensiones distintas de la m -ésima.

MÉTODO: Haga que C sea todas las restricciones en S que involucran a x_m . Haga lo siguiente:

1. Para cada par de un límite inferior y un límite superior sobre x_m en C , tal como:

$$L \leq \begin{array}{c} c_1 x_m \\ c_2 x_m \end{array} \leq U$$

cree la nueva restricción

$$c_2 L \leq c_1 U$$

Observe que c_1 y c_2 son enteros, pero L y U pueden ser expresiones con variables distintas a x_m .

2. Si los enteros c_1 y c_2 tienen un factor común, divida ambos lados entre ese factor.
3. Si la nueva restricción no puede cumplirse, entonces no hay solución para S ; es decir, los poliedros S y S' son espacios vacíos.
4. S' es el conjunto de restricciones $S - C$, más todas las restricciones generadas en el paso 2.

Por cierto, tenga en cuenta que si x_m tiene u límites inferiores y v límites superiores, al eliminar x_m se producen hasta uv desigualdades, pero no más. \square

Las restricciones agregadas en el paso (1) del Algoritmo 11.11 corresponden a las complicaciones de las restricciones C en el resto de las variables en el sistema. Por lo tanto, hay una

solución en S' si, y sólo si existe por lo menos una solución correspondiente en S . Dada una solución en S' , puede encontrarse el rango de la x_m correspondiente al sustituir todas las variables, excepto x_m , en las restricciones C por sus valores actuales.

Ejemplo 11.12: Considere las desigualdades que definen el espacio de iteraciones de la figura 11.11. Suponga que deseamos utilizar la eliminación de Fourier-Motzkin para proyectar el espacio bidimensional lejos de la dimensión i y hacia la dimensión j . Hay un límite inferior sobre i : $0 \leq i$ y dos límites superiores: $i \leq j$ e $i \leq 5$. Esto genera dos restricciones: $0 \leq j$ y $0 \leq 5$. Esta última es por lo común verdadera, y puede ignorarse. La primera proporciona el límite inferior sobre j , y el límite superior original $j \leq 7$ proporciona el límite superior. \square

Generación de límites de ciclo

Ahora que hemos definido la eliminación de Fourier-Motzkin, el algoritmo para generar los límites de ciclo para iterar a través de un poliedro convexo (Algoritmo 11.13) es simple. Calculamos los límites de ciclo en orden, desde el más interno hasta los ciclos externos. Todas las desigualdades que involucran a las variables índice del ciclo más interno se escriben como los límites inferior o superior de la variable. Despues proyectamos hacia fuera la dimensión que representa el ciclo más interno y obtenemos un poliedro con una dimensión menos. Repetimos hasta que se hayan encontrado todos los límites para todas las variables índice del ciclo.

Algoritmo 11.13: Cálculo de los límites para un orden dado de variables.

ENTRADA: Un poliedro convexo S sobre las variables v_1, \dots, v_n .

SALIDA: Un conjunto de límites inferiores L_i y límites superiores U_i para cada v_i , expresada sólo en términos de las v_j , para $j < i$.

MÉTODO: El algoritmo se describe en la figura 11.15. \square

Ejemplo 11.14: Aplicamos el Algoritmo 11.13 para generar los límites de ciclo que recorren el espacio de iteraciones de la figura 11.11 en sentido vertical. Las variables se ordenan como j, i . El algoritmo genera estos límites:

$$\begin{aligned} L_i &: 0 \\ U_i &: 5, j \\ L_j &: 0 \\ U_j &: 7 \end{aligned}$$

Debemos satisfacer todas las restricciones, por lo que el límite sobre i es $\min(5, j)$. No hay redundancias en este ejemplo. \square

```

 $S_n = S$ ; /* Usar el Algoritmo 11.11 para encontrar los límites */
for (  $i = n$ ;  $i \geq 1$ ;  $i --$  ) {
     $L_{v_i}$  = todos los límites inferiores sobre  $v_i$  en  $S_i$ ;
     $U_{v_i}$  = todos los límites superiores sobre  $v_i$  en  $S_i$ ;
     $S_{i-1}$  = Restricciones devueltas al aplicar el Algoritmo 11.11
        para eliminar  $v_i$  de las restricciones  $S_i$ ;
}
/* Eliminar redundancias */
 $S' = \emptyset$ ;
for (  $i = 1$ ;  $i \leq n$ ;  $i ++$  ) {
    Eliminar límites en  $L_{v_i}$  y  $U_{v_i}$ , implicados por  $S'$ ;
    Agregar las restricciones restantes de  $L_{v_i}$  y  $U_{v_i}$  sobre  $v_i$  a  $S'$ ;
}

```

Figura 11.15: Código para expresar los límites de las variables con respecto a un ordenamiento dado de las variables

[0,0],	[1,1],	[2,2],	[3,3],	[4,4],	[5,5]
[0,1],	[1,2],	[2,3],	[3,4],	[4,5],	[5,6]
[0,2],	[1,3],	[2,4],	[3,5],	[4,6],	[5,7]
[0,3],	[1,4],	[2,5],	[3,6],	[4,7]	
[0,4],	[1,5],	[2,6],	[3,7]		
[0,5],	[1,6],	[2,7]			
[0,6],	[1,7]				
[0,7]					

Figura 11.16: Ordenamiento por diagonales del espacio de iteraciones de la figura 11.11

11.3.6 Cambio de ejes

Hay que tener en cuenta que los recorridos horizontal y vertical del espacio de iteraciones, como vimos antes, son sólo dos de las formas más comunes de visitar el espacio de iteraciones. Existen muchas otras posibilidades; por ejemplo, podemos recorrer el espacio de iteraciones en el ejemplo 11.6 diagonal por diagonal, como se describe a continuación en el ejemplo 11.15.

Ejemplo 11.15: Podemos recorrer el espacio de iteraciones mostrado en la figura 11.11 en sentido diagonal, usando el orden que se muestra en la figura 11.16. La diferencia entre las coordenadas j e i en cada diagonal es una constante, que empieza con 0 y termina con 7. Así, definimos una nueva variable $k = j - i$ y recorremos el espacio de iteraciones en orden lexicográfico con respecto a k y j . Si sustituimos $i = j - k$ en las desigualdades obtenemos:

$$\begin{array}{rcccl} 0 & \leq & j - k & \leq 5 \\ j - k & \leq & j & \leq 7 \end{array}$$

Para crear los límites de ciclo para el orden antes descrito, podemos aplicar el Algoritmo 11.13 al conjunto anterior de desigualdades con el ordenamiento de variables k, j .

$$\begin{aligned} L_j &: k \\ U_j &: 5+k, 7 \\ L_k &: 0 \\ U_k &: 7 \end{aligned}$$

De estas desigualdades generamos el siguiente código, sustituyendo i por $j - k$ en los accesos al arreglo.

```
for (k = 0; k <= 7; k++)
    for (j = k; j <= min(5+k,7); j++)
        Z[j, j-k] = 0;
```

□

En general, podemos cambiar los ejes de un poliedro creando nuevas variables índice de ciclo que representen combinaciones afines de las variables originales, y definiendo un orden sobre esas variables. Lo difícil está en elegir los ejes correctos para satisfacer las dependencias de datos y lograr al mismo tiempo los objetivos de paralelismo y localidad. En la sección 11.7 hablaremos sobre este problema. Lo que hemos establecido aquí es que, una vez que se eligen los ejes, es muy sencillo generar el código deseado, como se muestra en el ejemplo 11.15.

Hay muchos otros órdenes de recorrido de iteraciones que esta técnica no maneja. Por ejemplo, tal vez queramos visitar todas las filas impares en un espacio de iteraciones antes de visitar las filas pares. O tal vez queramos empezar con las iteraciones en la parte media del espacio de iteraciones, y avanzar hacia los contornos. Sin embargo, para las aplicaciones que tienen funciones de acceso afines, las técnicas aquí descritas cubren la mayoría de los ordenamientos de iteraciones convenientes.

11.3.7 Ejercicios para la sección 11.3

Ejercicio 11.3.1: Convierta cada uno de los siguientes ciclos a una forma en la que cada uno de los índices de ciclo se incremente en 1:

- for (i=10; i<50; i=i+7) X[i, i+1] = 0;.
- for (i= -3; i<=10; i=i+2) X[i] = x[i+1];.
- for (i=50; i>=10; i--) X[i] = 0;.

Ejercicio 11.3.2: Dibuje o describa los espacios de iteraciones para cada uno de los siguientes anidamientos de ciclos:

- El anidamiento de ciclos de la figura 11.17(a).
- El anidamiento de ciclos de la figura 11.17(b).
- El anidamiento de ciclos de la figura 11.17(c).

```
for (i = 1; i < 30; i++)
    for (j = i+2; j < 40-i; j++)
        X[i,j] = 0;
```

(a) Anidamiento de ciclos para el ejercicio 11.3.2(a).

```
for (i = 10; i <= 1000; i++)
    for (j = i; j < i+10; j++)
        X[i,j] = 0;
```

(b) Anidamiento de ciclos para el ejercicio 11.3.2(b).

```
for (i = 0; i < 100; i++)
    for (j = 0; j < 100+i; j++)
        for (k = i+j; k < 100-i-j; k++)
            X[i,j,k] = 0;
```

(c) Anidamiento de ciclos para el ejercicio 11.3.2(c).

Figura 11.17: Anidamientos de ciclos para el ejercicio 11.3.2

Ejercicio 11.3.3: Escriba las restricciones implicadas por cada uno de los anidamientos de ciclos de la figura 11.17 en la forma de la ecuación (11.1). Es decir, proporcione los valores de los vectores **i** y **b** y de la matriz **B**.

Ejercicio 11.3.4: Invierta cada uno de los órdenes de anidamiento de ciclos para los anidamientos de la figura 11.17.

Ejercicio 11.3.5: Use el algoritmo de eliminación de Fourier-Motzkin para eliminar i de cada uno de los conjuntos de restricciones obtenidos en el ejercicio 11.3.3.

Ejercicio 11.3.6: Use el algoritmo de eliminación de Fourier-Motzkin para eliminar j de cada uno de los conjuntos de restricciones obtenidos en el ejercicio 11.3.3.

Ejercicio 11.3.7: Para cada uno de los anidamientos de ciclos de la figura 11.17, modifique el código de manera que se sustituya el eje i por la diagonal mayor; es decir, la dirección del eje está caracterizada por $i = j$. El nuevo eje debe corresponder al ciclo más externo.

Ejercicio 11.3.8: Repita el ejercicio 11.3.7 para los siguientes cambios de ejes:

- Sustituya i por $i + j$; es decir, la dirección del eje consiste en las líneas para las cuales $i + j$ es una constante. El nuevo eje corresponde al ciclo más externo.
- Sustituya j por $i - 2j$. El nuevo eje corresponde al ciclo más externo.

Ejercicio 11.3.9: Haga que A , B y C sean constantes enteras en el siguiente ciclo, con $C > 1$ y $B > A$;

```
for (i = A; i <= B; i = i + C)
    Z[i] = 0;
```

Vuelva a escribir el ciclo de manera que el incremento de la variable de ciclo sea 1 y que la inicialización sea a 0; es decir, que sea de la forma:

```
for (j = 0; j <= D; j++)
    Z[E*j + F] = 0;
```

para los enteros D , E y F . Exprese D , E y F en términos de A , B y C .

Ejercicio 11.3.10: Par un anidamiento genérico de dos ciclos:

```
for (i = 0; i <= A; i++)
    for(j = B*i+C; j <= D*i+E; j++)
```

en donde desde A hasta E son constantes enteras, escriba las restricciones que definen el espacio de iteraciones del anidamiento de ciclos en forma matriz-vector; es decir, en la forma $\mathbf{B}i + \mathbf{b} = \mathbf{0}$.

Ejercicio 11.3.11: Repita el ejercicio 11.3.10 para un anidamiento genérico de dos ciclos con las constantes enteras simbólicas m y n , como en:

```
for (i = 0; i <= m; i++)
    for(j = A*i+B; j <= C*i+n; j++)
```

Como antes, A , B y C representan constantes enteras específicas. Sólo i , j , m y n deben mencionarse en el vector de valores desconocidos. Además, recuerde que sólo i y j son variables de salida para la expresión.

11.4 Índices de arreglos afines

El enfoque de este capítulo es sobre la clase de accesos a arreglos afines, en donde cada índice de arreglo se expresa en forma de expresiones afines de índices de ciclo y constantes simbólicas. Las funciones afines proporcionan una asignación sintetizada del espacio de iteraciones al espacio de datos, ayudando a determinar qué iteraciones se asignan a los mismos datos o a la misma línea de caché.

Así como los límites afines superior e inferior de un ciclo pueden representarse como un cálculo de matriz-vector, podemos hacer lo mismo para las funciones de acceso afines. Una vez colocadas en la forma matriz-vector, podemos aplicar el álgebra lineal estándar para encontrar la información pertinente, como las dimensiones de los datos a los que se accedió, y qué iteraciones se refieren a los mismos datos.

11.4.1 Accesos afines

Decimos que el acceso a un arreglo en un ciclo es *afín* si:

1. Los límites del ciclo se expresan como expresiones afines de las variables y constantes simbólicas del ciclo circundante.
2. El índice para cada dimensión del arreglo es también una expresión afín de variables y constantes simbólicas del ciclo circundante.

Ejemplo 11.16: Suponga que i y j son variables índices de ciclo delimitadas por expresiones afines. Algunos ejemplos de accesos a arreglos afines son $Z[i]$, $Z[i + j + 1]$, $Z[0]$, $Z[i, i]$ y $Z[2 * i + 1, 3 * j - 10]$. Si n es una constante simbólica para un anidamiento de ciclos, entonces $Z[3 * n, n - j]$ es otro ejemplo de un acceso afín a un arreglo. Sin embargo, $Z[i * j]$ y $Z[n * j]$ no son accesos afines. \square

Cada acceso afín a un arreglo puede describirse mediante dos matrices y dos vectores. El primer par matriz-vector es el de \mathbf{B} y \mathbf{b} , que describen el espacio de iteraciones para el acceso, como en la desigualdad de la Ecuación (11.1). El segundo par que, por lo general, conocemos como \mathbf{F} y \mathbf{f} , representa la(s) función(es) de las variables índice de ciclo que producen el(los) índice(s) de arreglo utilizado(s) en las diversas dimensiones del acceso al arreglo.

De manera formal, representamos a un acceso al arreglo en un anidamiento de ciclos que utiliza un vector de variables índice \mathbf{i} mediante el cuadrado $\mathcal{F} = \langle \mathbf{F}, \mathbf{f}, \mathbf{B}, \mathbf{b} \rangle$; asigna un vector \mathbf{i} dentro de los límites:

$$\mathbf{Bi} + \mathbf{b} \geq \mathbf{0}$$

a la siguiente ubicación del elemento del arreglo:

$$\mathbf{Fi} + \mathbf{f}$$

Ejemplo 11.17: En la figura 11.18 hay algunos accesos comunes a arreglos, expresados en notación de matrices. Los dos índices de ciclo son i y j , y éstos forman el vector \mathbf{i} . Además, X , Y y Z son arreglos con 1, 2 y 3 dimensiones, respectivamente.

El primer acceso, $A[i - 1]$, se representa mediante una matriz \mathbf{F} de 1×2 y un vector \mathbf{f} de longitud 1. Observe que cuando realizamos la multiplicación matriz-vector y le sumamos el vector \mathbf{f} , nos quedamos con una sola función, $i - 1$, que es exactamente la fórmula para el acceso al arreglo unidimensional X . Observe además el tercer acceso, $Y[j, j + 1]$, que después de la multiplicación y suma de matriz-vector, produce un par de funciones, $(j, j + 1)$. Éstos son los índices de las dos dimensiones del acceso al arreglo.

Por último, observemos el cuarto acceso $Y[1, 2]$. Este acceso es una constante, y no es de sorprender que la matriz \mathbf{F} conste sólo de 0s. Por ende, el vector de los índices de ciclo, \mathbf{i} , no aparece en la función de acceso. \square

ACCESO	EXPRESIÓN AFÍN
$X[i-1]$	$\begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \end{bmatrix}$
$Y[i, j]$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
$Y[j, j+1]$	$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
$Y[1, 2]$	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix}$
$Z[1, i, 2*i+j]$	$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$

Figura 11.18: Algunos accesos a arreglos y sus representaciones matriz-vector

11.4.2 Accesos afines y no afines en la práctica

Hay ciertos patrones de acceso de datos comunes que se encuentran en los programas numéricos que no pueden ser afines. Los programas que involucran matrices poco densas son un ejemplo importante. Una representación popular para las matrices poco densas es almacenar sólo los elementos distintos de cero en un vector, y se utilizan arreglos de índices auxiliares para marcar en dónde empieza una fila y cuáles columnas contienen valores distintos de cero. Los accesos indirectos a los arreglos se utilizan para acceder a dichos datos. Un acceso de este tipo, como $X[Y[i]]$, es un acceso no afín al arreglo X . Si la poca densidad es regular, como en las matrices en banda que tienen valores distintos de cero sólo alrededor de la diagonal, entonces pueden usarse arreglos densos para representar las subregiones con elementos distintos de cero. En ese caso, los accesos pueden ser afines.

Otro ejemplo común de accesos no afines son los arreglos linealizados. Algunas veces, los programadores utilizan un arreglo lineal para almacenar un objeto lógicamente multidimensional. Una razón del por qué éste es el caso es que las dimensiones del arreglo tal vez no se conozcan en tiempo de compilación. Un acceso que, por lo general, se vería como $Z[i, j]$, se expresaría como $Z[i * n + j]$, que es una función cuadrática. Podemos convertir el acceso lineal

en un acceso multidimensional, si cada acceso puede descomponerse en dimensiones separadas con la garantía de que ninguno de sus componentes excederá su límite. Por último, observamos que los análisis de las variables de inducción pueden usarse para convertir ciertos accesos no afines en afines, como se muestra en el ejemplo 11.18.

Ejemplo 11.18: Podemos escribir el siguiente código:

```
j = n;
for (i = 0; i <= n; i++) {
    Z[j] = 0;
    j = j+2;
}
```

como

```
j = n;
for (i = 0; i <= n; i++) {
    Z[n+2*i] = 0;
}
```

para hacer que el acceso a la matriz Z sea afín. \square

11.4.3 Ejercicios para la sección 11.4

Ejercicio 11.4.1: Para cada uno de los siguientes accesos a un arreglo, proporcione el vector \mathbf{f} y la matriz \mathbf{F} que los describe. Suponga que el vector de índices \mathbf{i} es i, j, \dots , y que todos los índices de ciclo tienen límites afines.

- a) $X[2*i + 3, 2*j - i]$.
- b) $Y[i - j, j - k, k - i]$.
- c) $Z[3, 2*j, k - 2*i + 1]$.

11.5 Reutilización de datos

De las funciones de acceso a arreglos derivamos dos tipos de información útiles para la optimización de la localidad y la paralelización:

1. *Reutilización de datos*: para la optimización de la localidad, deseamos identificar conjuntos de iteraciones que accedan a los mismos datos o la misma línea de caché.
2. *Dependencia de datos*: para que la paralelización y las transformaciones de ciclo de localidad sean correctas, debemos identificar *todas* las dependencias de datos en el código. Recuerde que dos accesos (no necesariamente distintos) tienen una dependencia de datos si las instancias de los accesos pueden referirse a la misma ubicación de memoria, y por lo menos uno de ellos es una escritura.

En muchos casos, cada vez que identificamos iteraciones que reutilizan los mismos datos, hay dependencias de datos entre ellos.

Cada vez que hay una dependencia de datos, es obvio que se reutilizan los mismos datos. Por ejemplo, en la multiplicación de matrices, el mismo elemento en el arreglo de salida se escribe $O(n)$ veces. Las operaciones de escritura deben ejecutarse en el orden de ejecución original;³ hay reutilización debido a que podemos asignar el mismo elemento a un registro.

Sin embargo, no toda la reutilización puede explotarse en las optimizaciones de localidad; he aquí un ejemplo que ilustra esta cuestión.

Ejemplo 11.19: Considere el siguiente ciclo:

```
for (i = 0; i < n; i++)
    Z[7i+3] = Z[3i+5];
```

Observamos que el ciclo escribe a una ubicación distinta en cada iteración, por lo que no hay reutilizaciones ni dependencias en las distintas operaciones de escritura. Sin embargo, el ciclo lee las ubicaciones 5, 8, 11, 14, 17, ..., y escribe en las ubicaciones 3, 10, 17, 24, Las iteraciones de lectura y escritura acceden a los mismos elementos 17, 38 y 59, y así sucesivamente. Es decir, los enteros de la forma $17 + 21j$ para $j = 0, 1, 2, \dots$ son todos aquellos enteros que pueden escribirse como $7i_1 = 3$ y como $3i_2 + 5$, para algunos enteros i_1 e i_2 . Sin embargo, esta reutilización ocurre raras veces y, cuando ocurre, no puede explotarse con facilidad. \square

La dependencia de datos es diferente del análisis de la reutilización en que uno de los accesos que comparten una dependencia de datos debe ser un acceso de escritura. Lo más importante es que la dependencia de datos debe ser tanto correcta como precisa. Tiene que encontrar todas las dependencias en búsqueda de exactitud, y no debe encontrar dependencias ilegítimas, ya que pueden provocar una serialización innecesaria.

Con la reutilización de datos, sólo debemos buscar en dónde se encuentra la mayoría de las reutilizaciones explotables. Esto es mucho más simple, por lo que trataremos este tema en esta sección, y manejaremos las dependencias de datos en la siguiente. Simplificamos el análisis de la reutilización al ignorar los límites de ciclo, ya que pocas veces cambian la forma de la reutilización. La mayor parte de la reutilización que puede explotarse mediante el particionamiento afín reside entre las instancias de los mismos accesos a arreglos, y los accesos que comparten la misma *matriz de coeficientes* (que, por lo general, llamamos **F** en la función de índices afín). Como se muestra en párrafos anteriores, los patrones de acceso como $7i + 3$ y $3i + 5$ no tienen reutilización de interés.

11.5.1 Tipos de reutilización

Empezaremos primero con el ejemplo 11.20 para ilustrar los distintos tipos de reutilizaciones de datos. A continuación, debemos diferenciar entre el acceso como una instrucción en un programa;

³Aquí hay un punto delicado. Debido a la propiedad commutativa de la suma, obtendríamos la misma respuesta a la suma sin importar el orden en el que la realizáramos. Sin embargo, este caso es muy especial. En general, es demasiado complejo para que el compilador determine qué cálculo se está realizando mediante una secuencia de pasos aritméticos seguidos de escrituras, y no podemos confiar en que haya reglas algebraicas que nos ayuden a reordenar los pasos con seguridad.

por ejemplo, $x = Z[i, j]$, de la ejecución de esta instrucción muchas veces, a medida que ejecutamos el anidamiento de ciclos. Por cuestión de énfasis, podemos referirnos a la misma instrucción como un *acceso estático*, mientras que las diversas iteraciones de la instrucción, a medida que ejecutamos su anidamiento de ciclos, se conocen como *accesos dinámicos*.

Las reutilizaciones pueden clasificarse como *propias* y *de grupo*. Si las iteraciones que reutilizan los mismos datos provienen del mismo acceso estático, nos referimos a la reutilización como propia; si provienen de distintos accesos, nos referimos a la reutilización como de grupo. La reutilización es *temporal* si se hace referencia a la misma ubicación exacta; es *espacial* si se hace referencia a la misma línea de caché.

Ejemplo 11.20: Considere el siguiente anidamiento de ciclos:

```
float Z[n];
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        Z[j+1] = (Z[j] + Z[j+1] + Z[j+2])/3;
```

Cada uno de los accesos $Z[j]$, $Z[j + 1]$ y $Z[j + 2]$ tiene reutilización espacial propia, ya que las iteraciones consecutivas del mismo acceso hacen referencia a elementos contiguos del arreglo. Es muy probable que los elementos contiguos residan en la misma línea de caché. Además, todos tienen reutilización temporal propia, ya que los elementos exactos se utilizan una y otra vez en cada iteración en el ciclo externo. Asimismo, todos tienen la misma matriz de coeficientes y por ende, tienen reutilización de grupo. Hay una reutilización de grupo, tanto temporal como espacial, entre los distintos accesos. Aunque hay $4n^2$ accesos en este código, si la reutilización puede explotarse, sólo debemos llevar cerca de n/c líneas de caché a la caché, en donde c es el número de palabras en una línea de caché. Dejamos un factor de n debido a la reutilización espacial propia, un factor de c debido a la localidad espacial, y por último un factor de 4 debido a la reutilización de grupo. \square

A continuación mostraremos cómo podemos usar el álgebra lineal para extraer la información de reutilización de los accesos afines a un arreglo. Nos interesa no sólo encontrar cuántos ahorros potenciales hay, sino también qué iteraciones están reutilizando los datos, de forma que podamos tratar de acercarlas entre sí para explotar la reutilización.

11.5.2 Reutilización propia

Puede haber ahorros considerables en los accesos a memoria al explotar la reutilización propia. Si los datos referenciados por un acceso estático tienen k dimensiones y el acceso está anidado en un ciclo con profundidad d , para cierta $d > k$, entonces los mismos datos pueden reutilizarse n^{d-k} veces, en donde n es el número de iteraciones en cada ciclo. Por ejemplo, si un anidamiento de ciclos con 3 niveles accede a una columna de un arreglo, entonces hay un factor de ahorro potencial de n^2 accesos. Resulta que la dimensionalidad de un acceso corresponde al concepto del *rango* de la matriz de coeficientes en el acceso, y podemos encontrar qué iteraciones se refieren a la misma ubicación, al buscar el *espacio nulo* de la matriz, como se explica a continuación.

Rango de una matriz

El rango de una matriz \mathbf{F} es el mayor número de columnas (o de manera equivalente, filas) de \mathbf{F} que son linealmente independientes. Un conjunto de vectores es *linealmente independiente* si ninguno de los vectores puede escribirse como una combinación lineal de un número finito de muchos otros vectores en el conjunto.

Ejemplo 11.21: Considere la siguiente matriz:

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 4 & 5 & 6 \\ 2 & 1 & 0 \end{bmatrix}$$

Observe que la segunda fila es la suma de las filas primera y tercera, mientras que la cuarta fila es la tercera fila menos el doble de la primera. Sin embargo, las filas primera y tercera son linealmente independientes; ninguna es múltiplo de la otra. Por ende, el rango de la matriz es 2.

También podríamos sacar esta conclusión al examinar las columnas. La tercera columna es el doble de la segunda columna menos la primera. Por otro lado, dos columnas cualesquiera son linealmente independientes. De nuevo, concluimos que el rango es 2. \square

Ejemplo 11.22: Vamos a analizar los accesos a los arreglos en la figura 11.18. El primer acceso, $X[i - 1]$, tiene una dimensión de 1, ya que el rango de la matriz $[1 \ 0]$ es 1. Es decir, la única fila es linealmente independiente, al igual que la primera columna.

El segundo acceso, $Y[i, j]$, tiene una dimensión de 2. La razón es que la matriz:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

tiene dos filas independientes (y, por lo tanto, dos columnas independientes, desde luego). El tercer acceso, $Y[j, j + 1]$, es de dimensión 1, ya que la matriz:

$$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$$

tiene un rango de 1. Observe que las dos filas son idénticas, por lo que sólo una es linealmente independiente. En forma equivalente, la primera columna es 0 veces la segunda columna, por lo que las columnas no son independientes. Por intuición, en un arreglo Y grande y cuadrado, los únicos elementos a los que se accede se encuentran a lo largo de una línea unidimensional, justo encima de la diagonal principal.

El cuarto acceso, $Y[1, 2]$ tiene dimensión 0, ya que una matriz que sólo tiene 0s tiene un rango de 0. Observe que para dicha matriz, no podemos encontrar una suma lineal ni siquiera de una fila que sea distinta de cero. Finalmente, el último acceso $Z[i, i, 2 * i + j]$, tiene una dimensión de 2. Observe que en la matriz para este acceso:

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

las últimas dos filas son linealmente independientes; ninguna es múltiplo de la otra. No obstante, la primera fila es una “suma” lineal de las otras dos filas, con ambos coeficientes 0. \square

Espacio nulo de una matriz

Una referencia en un anidamiento de ciclos con d niveles y rango r accede a $O(n^r)$ elementos de datos en $O(n^d)$ iteraciones, por lo que en promedio, $O(n^{d-r})$ iteraciones deben hacer referencia al mismo elemento del arreglo. ¿Cuáles iteraciones acceden a los mismos datos? Suponga que un acceso en este anidamiento de ciclos se representa mediante una combinación matriz-vector \mathbf{F} y \mathbf{f} . Suponga que \mathbf{i} e \mathbf{i}' son dos iteraciones que hacen referencia al mismo elemento del arreglo. Entonces, $\mathbf{F}\mathbf{i} + \mathbf{f} = \mathbf{F}\mathbf{i}' + \mathbf{f}$. Al reordenar los términos, obtenemos:

$$\mathbf{F}(\mathbf{i} - \mathbf{i}') = \mathbf{0}.$$

Hay un concepto reconocido del álgebra lineal, que caracteriza cuando \mathbf{i} e \mathbf{i}' cumplen con la ecuación anterior. El conjunto de todas las soluciones a la ecuación $\mathbf{F}\mathbf{v} = \mathbf{0}$ se conoce como el *espacio nulo* de \mathbf{F} . Por ende, dos iteraciones se refieren al mismo elemento del arreglo si la diferencia de sus vectores índice de ciclo pertenece al espacio nulo de la matriz \mathbf{F} .

Es fácil ver que el vector nulo, $\mathbf{v} = \mathbf{0}$, siempre cumple con $\mathbf{F}\mathbf{v} = \mathbf{0}$. Es decir, dos iteraciones sin duda se refieren al mismo elemento del arreglo si su diferencia es $\mathbf{0}$; en otras palabras, si son en realidad la misma iteración. Además, el espacio nulo es sin duda un espacio vectorial. Es decir, si $\mathbf{F}\mathbf{v}_1 = \mathbf{0}$ y $\mathbf{F}\mathbf{v}_2 = \mathbf{0}$, entonces $\mathbf{F}(\mathbf{v}_1 + \mathbf{v}_2) = \mathbf{0}$ y $\mathbf{F}(c\mathbf{v}_1) = \mathbf{0}$.

Si la matriz \mathbf{F} tiene un *rango completo*; es decir, si su rango es d , entonces el espacio nulo de \mathbf{F} consiste sólo en el vector nulo. En ese caso, todas las iteraciones en un anidamiento de ciclos se refieren a datos distintos. En general, la dimensión del espacio nulo, también conocida como la *nulidad*, es $d - r$. Si $d > r$, entonces para cada elemento hay un espacio $(d - r)$ -dimensional de iteraciones que acceden a ese elemento.

El espacio nulo puede representarse mediante sus vectores básicos. Un espacio nulo k -dimensional se representa mediante k vectores independientes; cualquier vector que pueda expresarse como una combinación lineal de los vectores básicos pertenece al espacio nulo.

Ejemplo 11.23: Vamos a reconsiderar la matriz del ejemplo 11.21:

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 4 & 5 & 6 \\ 2 & 1 & 0 \end{bmatrix}$$

En ese ejemplo determinamos que el rango de la matriz es 2; por ende, la nulidad es $3 - 2 = 1$. Para encontrar una base para el espacio nulo, que en este caso debe ser un solo vector distinto de cero, de longitud 3, podemos suponer que un vector en el espacio nulo es $[x, y, z]$ y tratar de resolver la siguiente ecuación:

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 4 & 5 & 6 \\ 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

Si multiplicamos las primeras dos filas por el vector de valores desconocidos, obtenemos las siguientes dos ecuaciones:

$$\begin{aligned} x + 2y + 3z &= 0 \\ 5x + 7y + 9z &= 0 \end{aligned}$$

Podríamos escribir también las ecuaciones que provienen de las tercera y cuarta filas, pero como no hay tres filas linealmente independientes, sabemos que las ecuaciones adicionales no agregan nuevas restricciones sobre x , y y z . Por ejemplo, la ecuación que obtenemos de la tercera fila, $4x + 5y + 6z = 0$, se puede obtener si restamos la primera ecuación de la segunda.

Debemos eliminar todas las variables que podamos de las ecuaciones anteriores. Empiece por usar la primera ecuación para resolver para x ; es decir, $x = -2y - 3z$. Despues sustituya para x en la segunda ecuación, para obtener $-3y = 6z$, o $y = -2z$. Como $x = -2y - 3z$, y $y = -2z$, resulta que $x = z$. Por ende, el vector $[x, y, z]$ es en realidad $[z, -2z, z]$. Podemos elegir de z cualquier valor distinto de cero, para formar el único vector básico para el espacio nulo. Por ejemplo, podemos elegir $z = 1$ y usar $[1, -2, 1]$ como la base del espacio nulo. \square

Ejemplo 11.24: El rango, la nulidad y el espacio nulo para cada una de las referencias en el ejemplo 11.7 se muestran en la figura 11.19. Observe que la suma del rango y la nulidad en todos los casos es la profundidad del anidamiento de ciclos, 2. Como los accesos $Y[i, j]$ y $Z[1, i, 2 * i + j]$ tienen un rango de 2, todas las iteraciones hacen referencia a distintas ubicaciones.

Los accesos $X[i - 1]$ y $Y[j, j + 1]$ tienen matrices de rango 1, por lo que hay $O(n)$ iteraciones que hacen referencia a la misma ubicación. En el caso anterior, hay filas enteras en el espacio de iteraciones que se refieren a la misma ubicación. En otras palabras, las iteraciones que sólo difieren en la dimensión j comparten la misma ubicación, que se representa en forma sintetizada mediante la base del espacio nulo, $[0, 1]$. Para $Y[j, j + 1]$, hay columnas enteras en el espacio de iteraciones que se refieren a la misma ubicación, y este hecho se representa en forma sintetizada mediante la base del espacio nulo, $[1, 0]$.

Por último, el acceso $Y[1, 2]$ se refiere a la misma ubicación en todas las iteraciones. El espacio nulo correspondiente tiene 2 vectores básicos, $[0, 1]$, $[1, 0]$, lo cual significa que todos los pares de iteraciones en el anidamiento de ciclos se refieren exactamente a la misma ubicación. \square

11.5.3 Reutilización espacial propia

El análisis de la reutilización espacial depende de la distribución de los datos de la matriz. En C, las matrices se distribuyen en orden por filas y en Fortran se distribuyen en orden por columnas.

ACCESO	EXPRESIÓN AFÍN	RANGO	NULIDAD	BASE DEL ESPACIO NULO
$x[i-1]$	$[1 \ 0] \begin{bmatrix} i \\ j \end{bmatrix} + [-1]$	1	1	$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$
$y[i, j]$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$	2	0	
$y[j, j+1]$	$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$	1	1	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$
$y[1, 2]$	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix}$	0	2	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
$z[1, i, 2*i+j]$	$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$	2	0	

Figura 11.19: Rango y nulidad de los accesos afines

En otras palabras, los elementos de un arreglo $X[i, j]$ y $X[i, j + 1]$ son contiguos en C; $X[i, j]$ y $X[i + 1, j]$ son contiguos en Fortran. Sin perder la generalidad, en el resto del capítulo adoptaremos la distribución de los arreglos en C (orden por filas).

Como una primera aproximación, vamos a considerar dos elementos de un arreglo para compartir la misma línea de caché, si y sólo si comparten la misma fila en un arreglo bidimensional. En forma más general, en un arreglo de d dimensiones tomamos elementos del arreglo para compartir una línea de caché, si sólo difieren en la última dimensión. Como para un arreglo y caché comunes, muchos elementos del arreglo pueden caber en una línea de caché, debe haber un aumento considerable en la velocidad al acceder a toda una fila completa en orden, aun cuando, hablando en sentido estricto, en ocasiones tenemos que esperar para cargar una nueva línea de caché.

El truco para descubrir y aprovechar la reutilización espacial propia es retirar la última fila de la matriz de coeficientes \mathbf{F} . Si la matriz *trunca* resultante tiene un rango menor que la profundidad del anidamiento de ciclos, entonces podemos asegurar la localidad espacial, al verificar que el ciclo más interno sólo varíe la última coordenada del arreglo.

Ejemplo 11.25: Considere el último acceso, $Z[1, i, 2 * i + j]$, en la figura 11.19. Si eliminamos la última fila, nos quedamos con la siguiente matriz trunca:

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

Es evidente que el rango de esta matriz es 1, y como el anidamiento de ciclos tiene profundidad 2, existe la oportunidad de la reutilización espacial. En este caso, como j es el índice del ciclo interno, éste visita los elementos contiguos del arreglo Z , almacenados en orden por filas. Al hacer a i el índice del ciclo interior no se producirá una localidad espacial, ya que a medida que cambia i , cambian tanto la segunda como la tercera dimensión. \square

La regla general para determinar si hay reutilización espacial propia es la siguiente. Como siempre, suponemos que los índices de ciclo corresponden a las columnas de la matriz de coeficientes en orden, con el ciclo más externo primero, y el ciclo más interno al último. Así, para que pueda haber reutilización espacial, el vector $[0, 0, \dots, 0, 1]$ debe estar en el espacio nulo de la matriz trunca. La razón es que si este vector está en el espacio nulo, entonces al corregir todos los índices de ciclo excepto el más interno, sabemos que todos los accesos dinámicos durante una ejecución a través del ciclo interno varían sólo en el último índice del arreglo. Si el arreglo se almacena en orden por filas, entonces estos elementos están cerca unos de otros, tal vez en la misma línea de caché.

Ejemplo 11.26: Observe que $[0, 1]$ (transpuesto como un vector columna) está en el espacio nulo de la matriz trunca del ejemplo 11.25. Por ende, como se mencionó ahí, esperamos que con j como el índice del ciclo interno, haya localidad espacial. Por otro lado, si invertimos el orden de los ciclos, de manera que i sea el ciclo interno, entonces la matriz de coeficientes se convierte en:

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

Ahora, $[0, 1]$ no está en el espacio nulo de esta matriz. En vez de ello, el espacio nulo se genera mediante el vector básico $[1, 0]$. Entonces, como sugerimos en el ejemplo 11.25, no esperamos localidad espacial si i es el ciclo interno.

No obstante, hay que observar que la prueba para $[0, 0, \dots, 0, 1]$, estando en el espacio nulo, no es suficiente para asegurar la localidad espacial. Por ejemplo, suponga que el acceso no fuera $Z[1, i, 2 * i + j]$ sino $Z[1, i, 2 * i + 50 * j]$. Entonces, sólo se accedería a cada quincuagésimo elemento de Z durante una ejecución del ciclo interno, y no reutilizaríamos una línea de caché sino hasta que fuera lo bastante larga como para contener más de 50 elementos. \square

11.5.4 Reutilización de grupo

Calculamos la reutilización de grupo sólo entre los accesos en un ciclo que comparte la misma matriz de coeficientes. Dados dos accesos dinámicos $\mathbf{F}_i + \mathbf{f}_1$ y $\mathbf{F}_i + \mathbf{f}_2$, la reutilización de los mismos datos requiere que:

$$\mathbf{F}_i + \mathbf{f}_1 = \mathbf{F}_i + \mathbf{f}_2$$

o

$$\mathbf{F}(\mathbf{i}_1 - \mathbf{i}_2) = (\mathbf{f}_2 - \mathbf{f}_1).$$

Suponga que \mathbf{v} es una solución a esta ecuación. Entonces, si \mathbf{w} es cualquier vector en el espacio nulo de \mathbf{F}_1 , $\mathbf{w} + \mathbf{v}$ es también una solución, y de hecho ésas son todas las soluciones a la ecuación.

Ejemplo 11.27: El siguiente anidamiento de ciclos de 2 niveles:

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        Z[i,j] = Z[i-1,j];
```

tiene dos accesos al arreglo, $Z[i, j]$ y $Z[i - 1, j]$. Observe que estos dos accesos se caracterizan por la siguiente matriz de coeficientes:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

como el segundo acceso, $Z[i, j]$ en la figura 11.19. Esta matriz tiene un rango de 2, por lo que no hay reutilización temporal propia.

No obstante, cada acceso exhibe una reutilización espacial propia. Como se describe en la sección 11.5.3, cuando eliminamos la fila inferior de la matriz, nos quedamos sólo con la fila superior, $[1, 0]$, que tiene un rango de 1. Como $[0, 1]$ está en el espacio nulo de esta matriz trunca, esperamos la reutilización espacial. A medida que cada incremento del índice j del ciclo interno incrementa el segundo índice del arreglo por uno, de hecho accedemos a los elementos adyacentes del arreglo, y utilizamos al máximo cada línea de caché.

Aunque no hay una reutilización temporal propia para cualquiera de los accesos, observe que las dos referencias $Z[i, j]$ y $Z[i - 1, j]$ acceden casi al mismo conjunto de elementos del arreglo. Es decir, hay una reutilización temporal de grupo, ya que los datos leídos por el acceso $Z[i - 1, j]$ son los mismos que escribe el acceso $Z[i, j]$, excepto para el caso en el que $i = 1$. Este patrón simple se aplica a todo el espacio de iteraciones completo y puede explotarse para mejorar la localidad de los datos en el código. De manera formal, si descontamos los límites de ciclo, los dos accesos $Z[i, j]$ y $Z[i - 1, j]$ se refieren a la misma ubicación en las iteraciones (i_1, j_1) y (i_2, j_2) , respectivamente, siempre y cuando:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix}.$$

Si escribimos los términos, obtenemos lo siguiente:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 - i_2 \\ j_1 - j_2 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}.$$

Es decir, $j_1 = j_2$ e $i_2 = i_1 + 1$.

Observe que la reutilización ocurre a lo largo del eje i del espacio de iteraciones. Es decir, la iteración (i_2, j_2) ocurre n iteraciones (del ciclo interior) después de la iteración (i_1, j_1) . Por

ende, se ejecutan muchas iteraciones antes de reutilizar los datos escritos. Estos datos pueden seguir o no en la caché. Si la caché puede guardar dos filas consecutivas de la matriz Z , entonces el acceso $Z[i-1, j]$ no falla en la caché, y el número total de fallos de caché para todo el anidamiento de ciclos completo es n^2/c , en donde c es el número de elementos por línea de caché. En caso contrario habrá el doble de fallos, ya que ambos accesos estáticos requieren una nueva línea de caché para cada c accesos dinámicos. \square

Ejemplo 11.28: Suponga que hay dos accesos:

$$A[i, j, i+j] \text{ y } A[i+1, j-1, i+j]$$

en un anidamiento de ciclos de 3 niveles, con los índices i, j y k del ciclo externo al interno. Entonces, dos accesos $\mathbf{i}_1 = [i_1, j_1, k_1]$ e $\mathbf{i}_2 = [i_2, j_2, k_2]$ reutilizan el mismo elemento cada vez que

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \\ k_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \\ k_2 \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}.$$

Una solución a esta ecuación para un vector $\mathbf{v} = [i_1 - i_2, j_1 - j_2, k_1 - k_2]$ es $\mathbf{v} = [1, -1, 0]$; es decir, $i_1 = i_2 + 1, j_1 = j_2 - 1$ y $k_1 = k_2$.⁴ Sin embargo, el espacio nulo de la matriz

$$\mathbf{F} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

se genera mediante el vector básico $[0, 0, 1]$; es decir, el tercer índice de ciclo, k , puede ser arbitrario. Por ende, \mathbf{v} , la solución a la anterior ecuación, es cualquier vector $[1, -1, m]$ para cierta m . Dicho de otra forma, un acceso dinámico a $A[i, j, i+j]$, en un anidamiento de ciclos con los índices i, j y k , se reutiliza no sólo por otros accesos dinámicos $A[i, j, i+j]$ con los mismos valores de i y j y un valor distinto de k , sino también por accesos dinámicos $A[i+1, j-1, i+j]$ con los valores de índice de ciclo $i+1, j-1$ y cualquier valor de k . \square

Aunque no lo haremos aquí, podemos razonar acerca de la reutilización espacial de grupo en forma parecida. Al igual que con la explicación de la reutilización espacial propia, sólo eliminamos la última dimensión de toda consideración.

La extensión de la reutilización es diferente para las distintas categorías de reutilización. La reutilización temporal propia proporciona el mayor beneficio: una referencia con un espacio nulo k -dimensional reutiliza los mismos datos $O(n)^k$ veces. El grado de reutilización espacial propia se limita en base a la longitud de la línea de caché. Por último, el grado de reutilización de grupo se limita en base al número de referencias en un grupo que comparte la reutilización.

⁴Es interesante observar que, aunque hay una solución en este caso, no habría solución si modificáramos uno de los terceros componentes de $i+j$ a $i+j+1$. Es decir, en el ejemplo que se da, ambos accesos entran en contacto con esos elementos del arreglo que se encuentran en el subespacio S bidimensional definido por “el tercer componente es la suma de los primeros dos componentes”. Si cambiáramos $i+j$ a $i+j+1$, ninguno de los elementos que hicieran contacto con el segundo acceso se encontrarían en S , y no habría ningún tipo de reutilización.

11.5.5 Ejercicios para la sección 11.5

Ejercicio 11.5.1: Calcule los rangos de cada una de las matrices en la figura 11.20. Proporcione un conjunto máximo de columnas linealmente dependientes y un conjunto máximo de filas linealmente dependientes.

$$(a) \begin{bmatrix} 0 & 1 & 5 \\ 1 & 2 & 6 \\ 2 & 3 & 7 \\ 3 & 4 & 8 \end{bmatrix} \quad (b) \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 12 & 15 \\ 3 & 2 & 2 & 3 \end{bmatrix} \quad (c) \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 5 & 6 & 3 \end{bmatrix}$$

(a)

(b)

(c)

Figura 11.20: Calcule los rangos y espacios nulos de estas matrices

Ejercicio 11.5.2: Encuentre una base para el espacio nulo de cada matriz de la figura 11.20.

Ejercicio 11.5.3: Suponga que el espacio de iteraciones tiene las dimensiones (variables) i , j y k . Para cada uno de los siguientes accesos, describa los subespacios que hacen referencia a los siguientes elementos individuales del arreglo:

- a) $A[i, j, i + j]$
- b) $A[i, i + 1, i + 2]$
- ! c) $A[i, i, j + k]$
- ! d) $A[i - j, j - k, k - i]$

! Ejercicio 11.5.4: Suponga que el arreglo A se almacena en orden por filas y que se accede al mismo dentro del siguiente anidamiento de ciclos:

```
for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
        for (k = 0; k < 100; k++)
            <cierto acceso a A>
```

Indique para cada uno de los siguientes accesos si es posible rescribir los ciclos, de manera que el acceso a A exhiba una reutilización espacial propia; es decir, que se utilicen todas las líneas de la caché en forma consecutiva. Muestre cómo rescribir los ciclos, en caso de ser así. Nota: la rescritura de los ciclos puede implicar tanto el reordenamiento como la introducción de nuevos índices de ciclo. Sin embargo, no puede modificar la distribución del arreglo; por ejemplo, cambiándolo al orden por columnas. Además, tenga en cuenta lo siguiente: en general, el reordenamiento de los índices de ciclo puede ser válido o inválido, dependiendo de los criterios que desarrollemos en la siguiente sección. Sin embargo, en este caso, en donde el efecto de cada acceso es tan sólo establecer un elemento del arreglo a 0, no tenemos que preocuparnos por el efecto de reordenar los ciclos, en lo que a la semántica del programa respecta.

- a) $A[i+1, i+k, j] = 0$.
- !! b) $A[j+k, i, i] = 0$.
- c) $A[i, j, k, i+j+k] = 0$.
- !! d) $A[i, j-k, i+j, i+k] = 0$.

Ejercicio 11.5.5: En la sección 11.5.3 comentamos que obtenemos una localidad espacial si el ciclo más interno varía sólo como la última coordenada de un acceso al arreglo. Sin embargo, esa afirmación depende de nuestra suposición de que el arreglo se almacenó en orden por filas. ¿Qué condición aseguraría una localidad espacial, si el arreglo se almacenara en orden por columnas?

! **Ejercicio 11.5.6:** En el ejemplo 11.28 observamos que la existencia de la reutilización entre dos accesos similares dependía en gran parte de las expresiones particulares para las coordenadas del arreglo. Generalice nuestra observación sobre eso para determinar para qué funciones $f(i, j)$ hay reutilización entre los accesos $A[i, j, i + j]$ y $A[i + 1, j - 1, f(i, j)]$.

! **Ejercicio 11.5.7:** En el ejemplo 11.27 sugerimos que habrá más fallos de caché de los necesarios, si las filas de la matriz Z son tan largas que no puedan caber en la caché. Si ése es el caso, ¿cómo podríamos rescribir el anidamiento de ciclos para poder garantizar la reutilización espacial de grupo?

11.6 Análisis de dependencias de datos de arreglos

La paralelización o las optimizaciones locales ordenan con frecuencia las operaciones ejecutadas en el programa original. Al igual que con todas las optimizaciones, las operaciones sólo pueden reordenarse si este reordenamiento no modifica los resultados del programa. Como en general no podemos comprender con detalle lo que hace un programa, la optimización de código adopta una prueba conservadora más simple para cuando podemos estar seguros que no se ven afectados los resultados del programa: comprobamos que las operaciones en cualquier ubicación de memoria se realicen en el mismo orden en el programa original y en el modificado. En el presente estudio, nos enfocamos en los accesos a los arreglos, por lo que los elementos de un arreglo son las ubicaciones de memoria de interés.

Es evidente que dos accesos, ya sean de lectura o de escritura, son *independientes* (pueden reordenarse) si hacen referencia a dos ubicaciones distintas. Además, las operaciones de lectura no modifican el estado de la memoria y, por lo tanto, también son independientes. En base a la sección 11.5, decimos que dos accesos son *dependientes de datos* si hacen referencia a la misma ubicación de memoria y, por lo menos, uno de ellos es una operación de escritura. Para asegurarnos de que el programa modificado hace lo mismo que el original, el nuevo programa debe preservar el ordenamiento de ejecución relativo entre cada par de operaciones dependientes de datos en el programa original.

En la sección 10.2.1 vimos que hay tres tipos de dependencias de datos:

1. La *dependencia verdadera*, en donde una escritura va seguida de una lectura de la misma ubicación.

2. La *antidependencia*, en donde una lectura va seguida de una escritura a la misma ubicación.
3. La *dependencia de salida*, que consiste en dos escrituras a la misma ubicación.

En la explicación anterior, la dependencia de datos se define para los accesos dinámicos. Decimos que un acceso estático en un programa depende de otro, siempre y cuando exista una instancia dinámica del primer acceso que dependa de cierta instancia del segundo.⁵

Es fácil ver cómo puede usarse la dependencia de datos en la paralelización. Por ejemplo, si no se encuentran dependencias de datos en los accesos de un ciclo, podemos asignar con facilidad cada iteración a un procesador distinto. La sección 11.7 habla acerca de cómo podemos usar esta información de manera sistemática en la paralelización.

11.6.1 Definición de la dependencia de datos de los accesos a arreglos

Vamos a considerar dos accesos estáticos al mismo arreglo en ciclos que tal vez son diferentes. El primero se representa mediante la función de acceso y los límites $\mathcal{F} = \langle \mathbf{F}, \mathbf{f}, \mathbf{B}, \mathbf{b} \rangle$ y es un anidamiento de ciclos con d niveles; el segundo se representa mediante $\mathcal{F}' = \langle \mathbf{F}', \mathbf{f}', \mathbf{B}', \mathbf{b}' \rangle$ y es un anidamiento de ciclos con d' niveles. Estos accesos son dependientes de datos si:

1. Por lo menos uno de ellos es una referencia de escritura.
2. Existen vectores \mathbf{i} en Z^d , e \mathbf{i}' en $Z^{d'}$ de tal forma que:
 - (a) $\mathbf{Bi} \geq \mathbf{0}$,
 - (b) $\mathbf{B}'\mathbf{i}' \geq \mathbf{0}$, y
 - (c) $\mathbf{Fi} + \mathbf{f} = \mathbf{F}'\mathbf{i}' + \mathbf{f}'$.

Como, por lo general, un acceso estático abarca muchos accesos dinámicos, también es importante preguntarnos si sus accesos dinámicos pueden hacer referencia a la misma ubicación de memoria. Para buscar dependencias entre instancias del mismo acceso estático, suponemos que $\mathcal{F} = \mathcal{F}'$ y aumentamos la definición anterior con la restricción adicional de que $\mathbf{i} \neq \mathbf{i}'$ para descartar la solución trivial.

Ejemplo 11.29: Considere el siguiente anidamiento de ciclos con 1 nivel:

```
for (i = 1; i < 10; i++) {
    Z[i] = Z[i-1];
}
```

Este ciclo tiene dos accesos: $Z[i - 1]$ y $Z[i]$; el primero es una referencia de lectura y el segundo una escritura. Para buscar todas las dependencias de datos en este programa, debemos verificar si la referencia de escritura comparte una dependencia consigo misma y con la referencia de lectura:

⁵ Recuerde la diferencia entre los accesos estáticos y dinámicos. Un acceso estático es una referencia a un arreglo en una ubicación específica en un programa, mientras que un acceso dinámico es una ejecución de esa referencia.

1. *Dependencia de datos entre $Z[i - 1]$ y $Z[i]$.* Excepto por la primera iteración, cada iteración lee el valor escrito en la iteración anterior. En sentido matemático, sabemos que hay una dependencia debido a que existen enteros i e i' tales que:

$$1 \leq i \leq 10, 1 \leq i' \leq 10 \text{ e } i - 1 = i'.$$

Hay nueve soluciones para el sistema anterior de restricciones: $(i = 2, i' = 1)$, $(i = 3, i' = 2)$, y así sucesivamente.

2. *Dependencia de datos entre $Z[i]$ y consigo misma.* Es fácil ver que las distintas iteraciones en el ciclo escriben en distintas ubicaciones; es decir, no hay dependencias de datos entre las instancias de la referencia de escritura $Z[i]$. En sentido matemático, sabemos que no existe una dependencia, ya que no existen enteros i e i' que cumplan con:

$$1 \leq i \leq 10, 1 \leq i' \leq 10 \text{ e } i = i' \text{ e } i \neq i'.$$

Observe que la tercera condición, $i = i'$, se debe al requerimiento que establece que $Z[i]$ y $Z[i']$ son la misma ubicación de memoria. La cuarta condición contradictoria, $i \neq i'$, se debe al requerimiento que establece que la dependencia no sea trivial (entre los distintos accesos dinámicos).

No es necesario considerar las dependencias de datos entre la referencia de lectura $Z[i - 1]$ y ella misma, ya que dos accesos de lectura cualesquiera son independientes. \square

11.6.2 Programación lineal entera

La dependencia de datos requiere buscar si existen enteros que cumplan con un sistema consistente en igualdades y desigualdades. Las igualdades se derivan de las matrices y los vectores que representan a los accesos; las desigualdades se derivan de los límites de ciclo. Las igualdades pueden expresarse como desigualdades: una igualdad $x = y$ puede sustituirse por dos desigualdades, $x \geq y$ y $y \geq x$.

Por ende, la dependencia de datos puede parafrasearse como una búsqueda de soluciones enteras que cumplan con un conjunto de desigualdades lineales, lo cual es precisamente el reconocido problema de la *programación lineal entera*. La programación lineal entera es un problema NP-completo. Aunque no se conoce un algoritmo polinomial, se ha desarrollado la heurística para resolver programas lineales que involucran muchas variables, los cuales pueden ser bastante rápidos en muchos casos. Por desgracia, dicha heurística estándar es inapropiada para el análisis de la dependencia de datos, en donde el reto es resolver muchos programas lineales enteros pequeños y simples, en vez de programas lineales enteros grandes y complicados.

El algoritmo del análisis de dependencias de datos consiste en tres partes:

1. Aplicar la prueba del GCD (Greatest Common Divisor, Máximo común divisor), que comprueba si hay una solución entera a las igualdades, usando la teoría de las ecuaciones diofantinas lineales. Si no hay soluciones enteras, entonces no hay dependencias de datos. En cualquier otro caso, usamos las igualdades para sustituir para algunas de las variables, con lo cual obtenemos desigualdades más simples.
2. Usar un conjunto de heurística simple para manejar los números grandes de desigualdades comunes.
3. En el caso extraño en el que no funcione la heurística, utilizamos un solucionador de programación entera lineal, el cual usa un método de bifurcar y delimitar, basado en la eliminación de Fourier-Motzkin.

11.6.3 La prueba del GCD

El primer subproblema es comprobar la existencia de soluciones enteras a las igualdades. Las ecuaciones con la estipulación de que las soluciones deben ser enteras se conocen como *ecuaciones diofantinas*. El siguiente ejemplo muestra cómo surge la cuestión de las soluciones enteras; también demuestra que, aun cuando muchos de nuestros ejemplos involucran un solo anidamiento de ciclos a la vez, la formulación de las dependencias de datos se aplica a los accesos que tal vez se encuentren en distintos ciclos.

Ejemplo 11.30: Considere el siguiente fragmento de código:

```
for (i = 1; i < 10; i++) {
    Z[2*i] = ...;
}
for (j = 1; j < 10; j++) {
    Z[2*j+1] = ...;
}
```

El acceso $Z[2 * i]$ sólo entra en contacto con los elementos pares de Z , mientras que el acceso $Z[2 * j + 1]$ sólo entra en contacto con los elementos impares. Es evidente que estos dos accesos no comparten dependencia de datos, sin importar los límites de los ciclos. Podemos ejecutar iteraciones en el segundo ciclo antes que el primero, o intercalar las iteraciones. Este ejemplo no es tan artificial como parece. Un ejemplo en el que los números pares e impares se tratan en forma distinta es un arreglo de números complejos, en donde los componentes real e imaginario se distribuyen uno al lado de otro.

Para demostrar la ausencia de las dependencias de datos en este ejemplo, razonamos de la siguiente manera. Suponga que hay enteros i y j tales que $Z[2 * i]$ y $Z[2 * j + 1]$ sean el mismo elemento del arreglo. Obtenemos la siguiente ecuación diofantina:

$$2i = 2j + 1.$$

No hay enteros i y j que puedan cumplir la ecuación anterior. La prueba es que si i es un entero, entonces $2i$ es par. Si j es un entero, entonces $2j$ es par, por lo que $2j + 1$ es impar. Ningún

número par es también un número impar. Por lo tanto, la ecuación no tiene soluciones enteras, y por ende no hay dependencia entre los accesos de lectura y de escritura. \square

Para describir cuándo hay una solución a una ecuación diofantina lineal, necesitamos el concepto del *máximo común divisor* de dos o más enteros. El GCD de los enteros a_1, a_2, \dots, a_n , que se denota como $\gcd(a_1, a_2, \dots, a_n)$, es el entero más grande que divide de manera uniforme a todos estos enteros. Los GCDs pueden calcularse con eficiencia mediante el reconocido algoritmo de euclides (vea el recuadro que habla sobre ese tema).

Ejemplo 11.31: $\gcd(24, 36, 54) = 6$, ya que $24/6, 36/6$ y $54/6$ tienen un residuo de 0, y además cualquier entero mayor que 6 debe dejar un residuo distinto de cero en por lo menos una de las divisiones con 24, 36 y 54. Por ejemplo, 12 divide a 24 y 36 de manera uniforme, pero no a 54. \square

La importancia del GCD se explica en el siguiente teorema.

Teorema 11.32: La siguiente ecuación diofantina lineal:

$$a_1x_1 + a_2x_2 + \dots + a_nx_n = c$$

tiene una solución entera para x_1, x_2, \dots, x_n , si y solo si $\gcd(a_1, a_2, \dots, a_n)$ divide a c . \square

Ejemplo 11.33: En el ejemplo 11.30 observamos que la ecuación diofantina lineal $2i = 2j + 1$ no tiene solución. Podemos escribir esta ecuación así:

$$2i - 2j = 1.$$

Ahora, $\gcd(2, -2) = 2$, y 2 no divide a 1 de manera uniforme. Por ende, no hay solución.

Para otro ejemplo, considere la siguiente ecuación:

$$24x + 36y + 54z = 30.$$

Como $\gcd(24, 36, 54) = 6$, y $30/6 = 5$, hay una solución en los enteros para x, y y z . Una solución es $x = -1, y = 0$ y $z = 1$, pero hay una infinidad de soluciones más. \square

El primer paso para el problema de la dependencia de datos es utilizar un método estándar, como la eliminación gaussiana, para resolver las igualdades dadas. Cada vez que se construye una ecuación lineal, se aplica el Teorema 11.32 para descartar, si es posible, la existencia de una solución entera. Si podemos descartar dichas soluciones, entonces hay que responder “no”. En caso contrario, utilizamos la solución de las ecuaciones para reducir el número de variables en las desigualdades.

Ejemplo 11.34: Considere las siguientes dos igualdades:

$$\begin{aligned} x - 2y + z &= 0 \\ 3x + 2y + z &= 5 \end{aligned}$$

El algoritmo de Euclides

El *algoritmo euclidian*o para encontrar el $\gcd(a, b)$ funciona de la siguiente manera. Primero hay que suponer que a y b son enteros positivos, y que $a \geq b$. Observe que el GCD de números negativos, o el GCD de un número negativo y uno positivo es el mismo que el GCD de sus valores absolutos, por lo que podemos suponer que todos los enteros son positivos.

Si $a = b$, entonces $\gcd(a, b) = a$. Si $a > b$, suponga que c es el residuo de a/b . Si $c = 0$, entonces b divide a a de manera uniforme, por lo que $\gcd(a, b) = b$. En caso contrario, hay que calcular $\gcd(b, c)$; este resultado también será $\gcd(a, b)$.

Para calcular $\gcd(a_1, a_2, \dots, a_n)$, para $n > 2$, utilizamos el algoritmo de Euclides para calcular $\gcd(a_1, a_2) = c$. Despues calculamos en forma recursiva $\gcd(c, a_3, a_4, \dots, a_n)$.

Si analizamos cada igualdad por separado, quizá habría una solución. Para la primera igualdad, $\gcd(1, -2, 1) = 1$ divide a 0, y para la segunda igualdad, $\gcd(3, 2, 1) = 1$ divide a 5. No obstante, si utilizamos la primera igualdad para resolver para $z = 2y - x$ y sustituimos para z en la segunda igualdad, obtenemos $2x + 4y = 5$. Esta ecuación diofantina no tiene solución, ya que $\gcd(2, 4) = 2$ no divide a 5 de manera uniforme. \square

11.6.4 Heurística para resolver programas lineales enteros

El problema de la dependencia de datos requiere resolver muchos programas lineales enteros. Ahora veremos varias técnicas para manejar las desigualdades simples y una técnica para aprovechar la similitud encontrada en el análisis de dependencias de datos.

Prueba de variables independientes

Muchos de los programas lineales enteros de la dependencia de datos consisten en desigualdades que involucran sólo un valor desconocido. Los programas pueden resolverse en forma simple, probando si hay enteros entre los límites superior e inferior de la constante, de manera independiente.

Ejemplo 11.35: Considere el siguiente ciclo anidado:

```
for (i = 0; i <= 10; i++)
    for (j = 0; j <= 10; j++)
        Z[i,j] = Z[j+10,i+9];
```

Para averiguar si hay una dependencia de datos entre $Z[i, j]$ y $Z[j + 10, i + 9]$, preguntamos si existen enteros i, j, i' y j' de tal forma que:

$$\begin{aligned} 0 &\leq i, j, i', j' \leq 10 \\ i &= j' + 10 \\ j &= i' + 11 \end{aligned}$$

La prueba del GCD, que se aplica a las dos igualdades anteriores, determinará que *puede* haber una solución entera. Las soluciones enteras a las igualdades se expresan mediante:

$$i = t_1, j = t_2, i' = t_2 - 11, j' = t_1 - 10.$$

para cualquier entero t_1 y t_2 . Si sustituimos las variables t_1 y t_2 en las desigualdades lineales, obtenemos:

$$\begin{aligned} 0 &\leq t_1 & \leq 10 \\ 0 &\leq t_2 & \leq 10 \\ 0 &\leq t_2 - 11 & \leq 10 \\ 0 &\leq t_1 - 10 & \leq 10 \end{aligned}$$

Por ende, al combinar los límites inferiores de las últimas dos desigualdades con los límites superiores de las primeras dos, deducimos lo siguiente:

$$\begin{aligned} 10 &\leq t_1 \leq 10 \\ 11 &\leq t_2 \leq 10 \end{aligned}$$

Como el límite inferior sobre t_2 es mayor que su límite superior, no hay una solución entera y, en consecuencia, tampoco hay dependencia de datos. Este ejemplo muestra que, aun cuando hay igualdades que involucren varias variables, la prueba del GCD puede de todas formas crear desigualdades lineales que involucran a una variable a la vez. \square

Prueba acíclica

Otra heurística simple es averiguar si existe una variable que esté delimitada hacia abajo o hacia arriba por una constante. En ciertas circunstancias, podemos sustituir sin peligro la variable por la constante; las desigualdades simplificadas tienen una solución, si y sólo si las desigualdades originales tienen una solución. Dicho en forma específica, suponga que cada límite inferior sobre v_i es de la siguiente forma:

$$c_0 \leq c_i v_i \text{ para cierta } c_i > 0.$$

mientras todos los límites superiores sobre v_i son de la forma:

$$c_i v_i \leq c_0 + c_1 v_1 + \dots + c_{i-1} v_{i-1} + c_{i+1} v_{i+1} + \dots + c_n + v_n.$$

en donde c_1, c_2, \dots, c_i son todos no negativos. Entonces, podemos sustituir la variable v_i por su menor valor entero posible. Si no hay dicho límite inferior, sólo sustituimos v_i con $-\infty$. De

manera similar, si cada restricción que involucra a v_i se puede expresar en las dos formas anteriores, pero con las direcciones de las desigualdades invertidas, entonces podemos sustituir la variable v_i con el mayor valor entero posible, o por ∞ si no hay un límite superior constante. Podemos repetir este paso para simplificar las desigualdades y, en algunos casos, determinar si hay una solución.

Ejemplo 11.36: Considere las siguientes desigualdades:

$$\begin{array}{lclclcl} 1 & \leq & v_1, v_2 & \leq & 10 \\ 0 & \leq & v_3 & \leq & 4 \\ v_2 & \leq & v_1 & & \\ & & v_1 & \leq & v_3 + 4 \end{array}$$

La variable v_1 está delimitada de abajo por v_2 y de arriba por v_3 . Sin embargo, v_2 está delimitada de abajo sólo por la constante 1, y v_3 está delimitada de arriba sólo por la constante 4. Por ende, si sustituimos v_2 por 1 y v_3 por 4 en las desigualdades, obtenemos lo siguiente:

$$\begin{array}{lclcl} 1 & \leq & v_1 & \leq & 10 \\ 1 & \leq & v_1 & & \\ v_1 & \leq & 8 & & \end{array}$$

que ahora pueden resolverse fácilmente con la prueba de las variables independientes. \square

La prueba del residuo de ciclo

Ahora vamos a considerar el caso en el que cada variable está delimitada de abajo y de arriba por otras variables. Es muy común en el análisis de dependencias de datos que se dé el caso en el que las restricciones tienen la forma $v_i \leq v_j + c$, lo cual puede resolverse mediante el uso de una versión simplificada de la *prueba de residuo de ciclo*, ideada por Shostack. Un conjunto de estas restricciones puede representarse mediante un grafo dirigido, cuyos nodos están etiquetados con variables. Hay un flanco de v_i a v_j etiquetado como c , cada vez que hay una restricción $v_i \leq v_j + c$.

Definimos el *peso* de una ruta como la suma de las etiquetas de todos los flancos a lo largo de la ruta. Cada ruta en el grafo representa una combinación de las restricciones en el sistema. Es decir, podemos inferir que $v \leq v' + c$ cada vez que exista una ruta de v a v' con el peso c . Un ciclo en el grafo con el peso c representa a la restricción $v \leq v + c$ para cada nodo v en el ciclo. Si podemos encontrar un ciclo con peso negativo en el grafo, entonces podemos inferir que $v < v$, lo cual es imposible. En este caso, podemos concluir que no hay solución y, por ende, no hay dependencia.

También podemos incorporar a la prueba de residuo de ciclo restricciones de la forma $c \leq v$ y $v \leq c$ para la variable v y la constante c . Introducimos al sistema de desigualdades una nueva variable basura v_0 , la cual se agrega a cada límite constante superior e inferior. Desde luego que v_0 debe tener el valor 0, pero como la prueba de residuo de ciclo sólo busca ciclos, los valores actuales de las variables nunca se vuelven importantes. Para manejar los límites constantes, sustituimos:

$$\begin{aligned} v &\leq c \text{ por } v \leq v_0 + c \\ c &\leq v \text{ por } v_0 \leq v - c. \end{aligned}$$

Ejemplo 11.37: Considere las siguientes desigualdades:

$$\begin{array}{lcl} 1 & \leq & v_1, v_2 \leq 10 \\ 0 & \leq & v_3 \leq 4 \\ v_2 & \leq & v_1 \\ & & 2v_1 \leq 2v_3 - 7 \end{array}$$

Los límites constantes superior e inferior sobre v_1 se convierten en $v_0 \leq v_1 - 1$ y $v_1 \leq v_0 + 10$; los límites constantes sobre v_2 y v_3 se manejan de manera similar. Después, al convertir la última restricción a $v_1 \leq v_3 - 4$, podemos crear el grafo que se muestra en la figura 11.21. El ciclo v_1, v_3, v_0, v_1 tiene un peso de -1 , por lo que no hay solución a este conjunto de desigualdades. \square

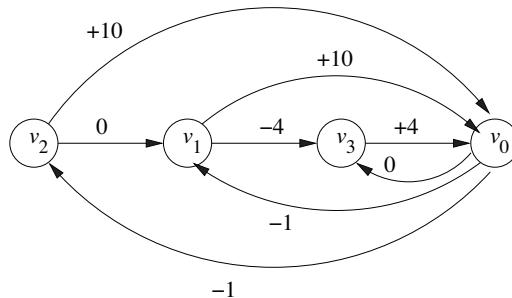


Figura 11.21: Grafo para las restricciones del ejemplo 11.37

Memorización

A menudo, los problemas de dependencias de datos similares se resuelven en forma repetitiva, ya que los patrones de acceso simple se repiten a lo largo del programa. Una técnica importante para agilizar el procesamiento de dependencias de datos es el uso de la *memorización*. La memorización tabula los resultados a los problemas a medida que se generan. La tabla de soluciones almacenadas se consulta a medida que se presenta cada problema; el problema debe resolverse sólo si no se puede encontrar el resultado para el problema en la tabla.

11.6.5 Solución de programas lineales enteros generales

Ahora describiremos un método general para resolver el problema de programación lineal entera. El problema es NP-completo; nuestro algoritmo utiliza un método de bifurcar y delimitar que puede requerir una cantidad de tiempo exponencial, en el peor de los casos. Sin embargo, es raro que la heurística de la sección 11.6.4 no pueda resolver el problema, e incluso si no necesitamos aplicar el algoritmo de esta sección, raras veces debe realizar el paso divide y vencerás.

El método es primero comprobar la existencia de soluciones racionales para las desigualdades. Éste es el clásico problema de la programación lineal. Si no hay solución racional para las desigualdades, entonces las regiones de datos que entran en contacto con los accesos en cuestión no se traslanan, y sin duda no hay dependencia de datos. Si hay una solución racional, primero tratamos de probar que hay una solución entera, que casi siempre viene siendo el caso. Si eso falla, entonces dividimos el poliedro delimitado por las desigualdades en dos problemas más pequeños y utilizamos la recursividad.

Ejemplo 11.38: Considere el siguiente ciclo simple:

```
for (i = 1; i < 10; i++)
    Z[i] = Z[i+10];
```

Los elementos que entran en contacto con el acceso $Z[i]$ son $Z[1], \dots, Z[9]$, mientras que los elementos que entran en contacto con $Z[i+10]$ son $Z[11], \dots, Z[19]$. Los rangos no se traslanan y, por lo tanto, no hay dependencias de datos. Dicho de manera más formal, debemos mostrar que no hay dos accesos dinámicos i e i' , con $1 \leq i \leq 9$, $1 \leq i' \leq 9$ e $i = i' + 10$. Si hubiera tales enteros i e i' , entonces podríamos sustituir $i' + 10$ para i y obtener las cuatro restricciones sobre i' : $1 \leq i' \leq 9$ y $1 \leq i' + 10 \leq 9$. Sin embargo, $i' + 10 \leq 9$ implica que $i' \leq -1$, lo cual contradice a $1 \leq i'$. Por ende, no existen tales enteros i e i' . \square

El Algoritmo 11.39 describe cómo determinar si puede encontrarse una solución entera para un conjunto de desigualdades lineales basadas en el algoritmo de eliminación de Fourier-Motzkin.

Algoritmo 11.39: Solución divide y vencerás para los problemas de programación lineal entera.

ENTRADA: Un poliedro convexo S_n sobre las variables v_1, \dots, v_n .

SALIDA: “sí” si S_n tiene una solución entera, “no” en cualquier otro caso.

MÉTODO: El algoritmo se muestra en la figura 11.22. \square

Las líneas (1) a (3) tratan de encontrar una solución racional a las desigualdades. Si no hay solución racional, entonces no hay solución entera. Si se encuentra una solución racional, esto significa que las desigualdades definen un poliedro no vacío. Es muy raro que dicho poliedro no incluya soluciones enteras; para que eso ocurra, el poliedro debe ser bastante delgado a lo largo de cierta dimensión, y debe caber entre puntos enteros.

Por ende, las líneas (4) a (9) tratan de comprobar con rapidez si hay una solución entera. Cada paso del algoritmo de eliminación de Fourier-Motzkin produce un poliedro con una dimensión menor que la anterior. Consideramos a los poliedros en orden inverso. Empezamos con el poliedro con una variable, y a esa variable le asignamos una solución entera, aproximadamente en medio del rango de valores posibles, si es viable. Después sustituimos el valor por la variable en todos los demás poliedros, reduciendo el número de sus variables desconocidas en uno. Repetimos el mismo proceso hasta haber procesado todos los poliedros, en cuyo caso

- 1) aplicar el Algoritmo 11.13 a S_n para proyectar las variables hacia fuera v_n, v_{n-1}, \dots, v_1 en ese orden;
- 2) hacer que S_i sea el poliedro después de proyectar v_{i+1} hacia fuera, para $i = n - 1, n - 2, \dots, 0$;
- 3) **if** S_0 es falsa **return** “no”;
/* No hay solución racional si S_0 , que involucra sólo constantes, tiene restricciones que no se pueden cumplir */
- 4) **for** ($i = 1; i \leq n; i++$) {
- 5) **if** (S_i no incluye un valor entero) **break**;
- 6) elegir c_i , un entero en medio del rango para v_i en S_i ;
- 7) modificar S_i , sustituyendo v_i por c_i ;
- 8) }
- 9) **if** ($i == n + 1$) **return** “sí”;
- 10) **if** ($i == 1$) **return** “no”;
- 11) hacer que los límites inferior y superior sobre v_i en S_i sean l_i y u_i , respectivamente;
- 12) aplicar este algoritmo en forma recursiva a $S_n \cup \{v_i \leq [l_i]\}$ y $S_n \cup \{v_i \geq [u_i]\}$;
- 13) **if** (devuelve “sí”) **return** “sí” **else return** “no”;

Figura 11.22: Búsqueda de una solución entera en las desigualdades

se encuentra una solución entera, o encontramos una variable para la cual no hay solución entera.

Si no podemos encontrar un valor entero incluso ni para la primera variable, entonces no hay solución entera (línea 10). En caso contrario, todo lo que sabemos es que no hay una solución entera que incluya la combinación de enteros específicos que hemos elegido hasta ahora, y el resultado no es concluyente. Las líneas (11) a (13) representan el paso divide y vencerás. Si encontramos que la variable v_i tiene una solución racional, pero no entera, dividimos el poliedro en dos, en donde el primero requiere que v_i debe ser un entero más pequeño que la solución racional encontrada, y el segundo requiere que v_i sea un entero mayor que la solución racional encontrada. Si ninguno de los dos tiene una solución, entonces no hay dependencia.

11.6.6 Resumen

Hemos mostrado que las piezas esenciales de información que un compilador puede deducir de las referencias a los arreglos son equivalentes a ciertos conceptos matemáticos estándar. Dada una función de acceso $\mathcal{F} = \langle \mathbf{F}, \mathbf{f}, \mathbf{B}, \mathbf{b} \rangle$:

1. La dimensión de la región de datos a la que se accedió se proporciona mediante el rango de la matriz \mathbf{F} . La dimensión del espacio de accesos a la misma ubicación se proporciona mediante la nulidad de \mathbf{F} . Las iteraciones cuyas diferencias pertenecen al espacio nulo de \mathbf{F} hacen referencia a los mismos elementos del arreglo.

2. Las iteraciones que comparten la reutilización temporal propia de un acceso se separan mediante vectores en el espacio nulo de \mathbf{F} . La reutilización espacial propia puede calcularse de manera similar, preguntando cuando dos iteraciones utilizan la misma fila, en vez del mismo elemento. Dos accesos $\mathbf{F}\mathbf{i}_1 + \mathbf{f}_1$ y $\mathbf{F}\mathbf{i}_2 + \mathbf{f}_2$ comparten una localidad que puede explotarse con facilidad a lo largo de la dirección \mathbf{d} , si \mathbf{d} es la solución específica para la ecuación $\mathbf{F}\mathbf{d} = (\mathbf{f}_1 - \mathbf{f}_2)$. En especial, si \mathbf{d} es la dirección correspondiente al ciclo más interno, es decir, el vector $[0, 0, \dots, 0, 1]$, entonces hay localidad espacial si el arreglo se almacena en formato de orden por fila.
3. El problema de dependencia de datos (si dos referencias pueden referirse a la misma ubicación) es equivalente a la programación lineal entera. Dos funciones de acceso comparten una dependencia de datos si hay vectores con valor de entero \mathbf{i} e \mathbf{i}' , de tal forma que $\mathbf{B}\mathbf{i} \geq \mathbf{0}$, $\mathbf{B}'\mathbf{i}' \geq \mathbf{0}$ y $\mathbf{F}\mathbf{i} + \mathbf{f} = \mathbf{F}'\mathbf{i}' + \mathbf{f}'$.

11.6.7 Ejercicios para la sección 11.6

Ejercicio 11.6.1: Busque los GCDs para los siguientes conjuntos de enteros:

- a) $\{16, 24, 56\}$.
- b) $\{-45, 105, 240\}$.
- c) $\{84, 105, 180, 315, 350\}$.

Ejercicio 11.6.2: Para el siguiente ciclo:

```
for (i = 0; i < 10; i++)
    A[i] = A[10-i];
```

indique todas las

- a) Dependencias verdaderas (escritura seguida de una lectura de la misma ubicación).
- b) Antidependencias (lectura seguida de una escritura a la misma ubicación).
- c) Dependencias de salida (escritura seguida de otra escritura a la misma ubicación).

Ejercicio 11.6.3: En el recuadro del algoritmo de Euclides, hicimos varias afirmaciones sin prueba. Demuestre cada una de las siguientes aseveraciones:

- a) El algoritmo de Euclides antes definido siempre funciona. En especial, $\gcd(b, c) = \gcd(a, b)$, en donde c es el residuo distinto de cero de a/b .
- b) $\gcd(a, b) = \gcd(a, -b)$.
- c) $\gcd(a_1, a_2, \dots, a_n) = \gcd(\gcd(a_1, a_2), a_3, a_4, \dots, a_n)$ para $n > 2$.

- d) El GCD es en realidad una función sobre conjuntos de enteros; es decir, el orden no importa. Muestre la *ley conmutativa* para el GCD: $\gcd(a, b) = \gcd(b, a)$. Después, muestre la instrucción más difícil, la *ley asociativa* para el GCD: $\gcd(\gcd(a, b), c) = \gcd(a, \gcd(b, c))$. Por último, muestre que en conjunto, estas leyes implican que el GCD de un conjunto de enteros es el mismo, sin importar el orden en el que se calculen los GCDs de pares de enteros.
- e) Si S y T son conjuntos de enteros, entonces $\gcd(S \cup T) = \gcd(\gcd(S), \gcd(T))$.

Ejercicio 11.6.4: Busque otra solución para la segunda ecuación diofantina en el ejemplo 11.33.

Ejercicio 11.6.5: Aplique la prueba de variables independientes en la siguiente situación. El anidamiento de ciclos es:

```
for (i=0; i<100; i++)
    for (j=0; j<100; j++)
        for (k=0; k<100; k++)
```

y dentro del anidamiento hay una asignación que involucra accesos a arreglos. Determine si hay dependencias de datos debido a cada una de las siguientes instrucciones:

- a) $A[i, j, k] = A[i+100, j+100, k+100]$.
- b) $A[i, j, k] = A[j+100, k+100, i+100]$.
- c) $A[i, j, k] = A[j-50, k-50, i-50]$.
- d) $A[i, j, k] = A[i+99, k+100, j]$.

Ejercicio 11.6.6: En las siguientes dos restricciones:

$$\begin{array}{rcl} 1 & \leq & x & \leq y - 100 \\ 3 & \leq & x & \leq 2y - 50 \end{array}$$

elimine x , sustituyéndola por un límite inferior constante sobre y .

Ejercicio 11.6.7: Aplique la prueba de residuo de ciclo al siguiente conjunto de restricciones:

$$\begin{array}{ll} 0 \leq x \leq 99 & y \leq x - 50 \\ 0 \leq y \leq 99 & z \leq y - 60 \\ 0 \leq z \leq 99 & \end{array}$$

Ejercicio 11.6.8: Aplique la prueba de residuo de ciclo al siguiente conjunto de restricciones:

$$\begin{array}{ll} 0 \leq x \leq 99 & y \leq x - 50 \\ 0 \leq y \leq 99 & z \leq y + 40 \\ 0 \leq z \leq 99 & x \leq z + 20 \end{array}$$

Ejercicio 11.6.9: Aplique la prueba de residuo de ciclo al siguiente conjunto de restricciones:

$$\begin{aligned} 0 \leq x \leq 99 \quad & y \leq x - 100 \\ 0 \leq y \leq 99 \quad & z \leq y + 60 \\ 0 \leq z \leq 99 \quad & x \leq z + 50 \end{aligned}$$

11.7 Búsqueda del paralelismo sin sincronización

Habiendo desarrollado la teoría de los accesos afines a un arreglo, su reutilización de los datos y las dependencias entre ellos, ahora comenzaremos a aplicar esta teoría a la paralelización y optimización de programas reales. Como vimos en la sección 11.1.4, es importante que busquemos el paralelismo, disminuyendo al mínimo al mismo tiempo la comunicación entre los procesadores. Vamos a empezar por estudiar el problema de parallelizar una aplicación sin permitir ningún tipo de comunicación o sincronización entre los procesadores. Esta restricción puede parecer un ejercicio puramente académico; ¿con qué frecuencia podemos encontrar programas y rutinas que tengan tal forma de paralelismo? De hecho, existen muchos de esos programas en la vida real, y el algoritmo para resolver este problema es útil por derecho propio. Además, los conceptos utilizados para resolver este problema se pueden extender para manejar la sincronización y la comunicación.

11.7.1 Un ejemplo introductorio

En la figura 11.23 se muestra un extracto de una traducción en C (con accesos a arreglos estilo Fortran que se retienen por claridad) a partir de un algoritmo multirrejillas de 5000 líneas en Fortran para resolver ecuaciones de Euler tridimensionales. El programa invierte la mayor parte de su tiempo en un pequeño número de rutinas como la que se muestra en la figura. Es típico de muchos programas numéricos. A menudo, éstos consisten en numerosos ciclos for, con distintos niveles de anidamiento y tienen muchos accesos a arreglos, todos los cuales son expresiones afines de índices de ciclos circundantes. Para mantener el ejemplo corto, hemos omitido las líneas del programa original con características similares.

El código de la figura 11.23 opera sobre la variable escalar T y un número de arreglos distintos, con distintas dimensiones. Primero vamos a examinar el uso de la variable T . Como cada iteración en el ciclo utiliza la misma variable T , no podemos ejecutar las iteraciones en paralelo. Sin embargo, T se utiliza sólo como una forma de contener una subexpresión común, que se utiliza dos veces en la misma iteración. En los primeros dos de los tres anidamientos de ciclos en la figura 11.23, cada iteración del ciclo más interno escribe un valor en T y lo utiliza justo después dos veces, en la misma iteración. Podemos eliminar las dependencias sustituyendo cada uso de T por la expresión del lado derecho en la asignación anterior de T , sin modificar la semántica del programa. O, podemos sustituir la escalar de T por un arreglo. Así, cada iteración (j, i) utiliza su propio elemento del arreglo $T[j, i]$.

Con esta modificación, el cálculo de un elemento del arreglo en cada instrucción de asignación sólo depende de otros elementos del arreglo con los mismos valores para los últimos dos componentes (j e i , respectivamente). Por lo tanto, podemos agrupar todas las operaciones que

```

for (j = 2; j <= jl; j++)
    for (i = 2, i <= il, i++) {
        AP[j,i]      = ...;
        T            = 1.0/(1.0 + AP[j,i]);
        D[2,j,i]    = T*AP[j,i];
        DW[1,2,j,i] = T*DW[1,2,j,i];
    }
    for (k = 3; k <= kl-1; k++)
        for (j = 2; j <= jl; j++)
            for (i = 2; i <= il; i++) {
                AM[j,i]      = AP[j,i];
                AP[j,i]      = ...;
                T            = ...AP[j,i] - AM[j,i]*D[k-1,j,i]...
                D[k,j,i]    = T*AP[j,i];
                DW[1,k,j,i] = T*(DW[1,k,j,i] + DW[1,k-1,j,i])...
            }
        ...
    for (k = kl-1; k >= 2; k--)
        for (j = 2; j <= jl; j++)
            for (i = 2; i <= il; i++)
                DW[1,k,j,i] = DW[1,k,j,i] + D[k,j,i]*DW[1,k+1,j,i];

```

Figura 11.23: Extracto de código de un algoritmo multirrejillas

trabajan sobre el (j, i) -ésimo elemento de todos los arreglos en una unidad de cómputo, y ejecutarlos en el orden secuencial original. Esta modificación produce $(jl - 1) \times (il - 1)$ unidades de cómputo, las cuales son todas independientes. Observe que los anidamientos segundo y tercero en el programa fuente involucran a un tercer ciclo, con el índice k . Sin embargo, como no hay dependencia entre los accesos dinámicos con los mismos valores para j e i , podemos ejecutar sin problemas los ciclos en k dentro de los ciclos en j e i ; es decir, dentro de una unidad de cómputo.

Al saber que estas unidades de cómputo son independientes se permiten varias transformaciones válidas sobre este código. Por ejemplo, en vez de ejecutar el código como estaba originalmente escrito, un uniprocesador puede realizar el mismo cálculo ejecutando las unidades de operación independiente, una unidad a la vez. El código resultante, que se muestra en la figura 11.24, ha mejorado la localidad temporal, ya que los resultados producidos se consumen de inmediato.

Las unidades independientes de cómputo también pueden asignarse a distintos procesadores y ejecutarse en paralelo, sin requerir ningún tipo de sincronización o comunicación. Como hay $(jl - 1) \times (il - 1)$ unidades independientes de cómputo, podemos utilizar a lo más $(jl - 1) \times (il - 1)$ procesadores. Al organizar los procesadores como si estuvieran en un arreglo bidimensional, con IDs (j, i) , en donde $2 \leq j < jl$ y $2 \leq i < il$, el programa SPMD que va a ejecutar cada procesador es sólo el cuerpo en el ciclo interno de la figura 11.24.

```

for (j = 2; j <= jl; j++) {
    for (i = 2; i <= il; i++) {
        AP[j,i]      = ...;
        T[j,i]        = 1.0/(1.0 + AP[j,i]);
        D[2,j,i]      = T[j,i]*AP[j,i];
        DW[1,2,j,i]   = T[j,i]*DW[1,2,j,i];
        for (k = 3; k <= kl-1; k++) {
            AM[j,i]    = AP[j,i];
            AP[j,i]    = ...;
            T[j,i]      = ...AP[j,i] - AM[j,i]*D[k-1,j,i]...
            D[k,j,i]    = T[j,i]*AP[j,i];
            DW[1,k,j,i] = T[j,i]*(DW[1,k,j,i] + DW[1,k-1,j,i])...
        }
        ...
        for (k = kl-1; k >= 2; k--)
            DW[1,k,j,i] = DW[1,k,j,i] + D[k,j,i]*DW[1,k+1,j,i];
    }
}

```

Figura 11.24: Código de la figura 11.23, transformado para ejecutar los ciclos paralelos más externos

El ejemplo anterior ilustra el método básico para buscar el paralelismo sin sincronización. Primero dividimos el cómputo en todas las unidades independientes que sea posible. Este particionamiento expone las opciones de programación disponibles. Despues asignamos unidades de cómputo a los procesadores, dependiendo del número de procesadores que tengamos. Por último, generamos un programa SPMD para ejecutarlo en cada procesador.

11.7.2 Particionamientos de espacio afín

Se dice que un anidamiento de ciclos tiene k grados de paralelismo si tiene, dentro del anidamiento, k ciclos paralelizables; es decir, ciclos tales que no haya dependencias de datos entre las distintas iteraciones de los ciclos. Por ejemplo, el código en la figura 11.24 tiene 2 grados de paralelismo. Es conveniente asignar las operaciones en un cómputo con k grados de paralelismo a un arreglo de procesadores con k dimensiones.

Vamos a suponer al principio que cada dimensión del arreglo de procesadores tiene tanto procesadores como iteraciones del ciclo correspondiente. Una vez que se han encontrado todas las unidades de cómputo independientes, debemos asignar estos procesadores “virtuales” a los procesadores actuales. En la práctica, cada procesador debe ser responsable de un buen número de iteraciones, ya que de lo contrario no hay suficiente trabajo para amortizar la sobrecarga de la paralelización.

Descomponemos el programa para paralelizarlo en instrucciones elementales, como las instrucciones de 3 direcciones. Para cada instrucción, buscamos una *partición del espacio afín* que asigne cada instancia dinámica de la instrucción, según como la identifican sus índices de ciclo, a un ID de procesador.

Ejemplo 11.40: Como vimos antes, el código de la figura 11.23 tiene dos grados de paralelismo. Vemos el arreglo de procesadores como un espacio bidimensional. Suponga que (p_1, p_2) es el ID de un procesador en el arreglo. El esquema de parallelización descrito en la sección 11.7.1 puede describirse mediante funciones simples de partición afín. Todas las instrucciones en el primer anidamiento de ciclos tienen esta misma partición afín:

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Todas las instrucciones en el segundo y tercer anidamiento de ciclos tienen la misma partición afín:

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} k \\ j \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

□

El algoritmo para buscar el paralelismo sin sincronización consiste en tres pasos:

1. Buscar, para cada instrucción en el programa, una partición afín que incremente al máximo el grado de paralelismo. Observe que, por lo general, tratamos a la instrucción, en vez del acceso individual, como la unidad de cómputo. La misma partición afín debe aplicarse a cada acceso en la instrucción. Este agrupamiento de accesos tiene sentido, ya que casi siempre hay dependencia entre los accesos de la misma instrucción, de todas formas.
2. Asignar las unidades de cómputo independientes resultantes entre los procesadores y elegir una intercalación de los pasos en cada procesador. Esta asignación se controla mediante las consideraciones de localidad.
3. Generar un programa SMPD para ejecutarlo en cada procesador.

A continuación vamos a ver cómo buscar las funciones de partición afín, cómo generar un programa secuencial que ejecute las particiones en serie, y cómo generar un programa SPMD que ejecute cada partición en un procesador distinto. Después de ver cómo se maneja el paralelismo con sincronizaciones en las secciones 11.8 a 11.9.9, regresaremos al paso 2 anterior en la sección 11.10 y hablaremos sobre la optimización de la localidad para uniprocesadores y multiprocesadores.

11.7.3 Restricciones de partición de espacio

Para no requerir comunicación, a cada par de operaciones que comparten una dependencia de datos se les debe asignar el mismo procesador. Nos referimos a estas restricciones como “restricciones de partición de espacio”. Cualquier asignación que cumpla con estas restricciones crea particiones independientes unas de otras. Observe que dichas restricciones pueden cumplirse si colocamos todas las operaciones en una sola partición. Por desgracia, esa “solución” no

produce ningún paralelismo. Nuestra meta es crear todas las particiones independientes que sea posible, al mismo tiempo que se cumpla con las restricciones de partición de espacio; es decir, las operaciones no deben colocarse en el mismo procesador, a menos que sea necesario.

Cuando nos limitamos a las particiones afines, en vez de maximizar el número de unidades independientes, podemos maximizar el grado (número de dimensiones) de paralelismo. Algunas veces es posible crear unidades más independientes, si podemos usar particiones afines a *nivel de pieza*. Una partición afín a nivel de pieza divide las instancias de un acceso individual en conjuntos diferentes, y permite una partición afín distinta para cada conjunto. Sin embargo, no consideraremos dicha opción aquí.

De manera formal, una partición afín de un programa *no está sincronizada* si, y sólo si por cada dos accesos (no necesariamente distintos) que comparten una dependencia, $\mathcal{F}_1 = \langle \mathbf{F}_1, \mathbf{f}_1, \mathbf{B}_1, \mathbf{b}_1 \rangle$ en la instrucción s_1 anidada en d_1 ciclos, y $\mathcal{F}_2 = \langle \mathbf{F}_2, \mathbf{f}_2, \mathbf{B}_2, \mathbf{b}_2 \rangle$ en la instrucción s_2 anidada en d_2 ciclos, las particiones $\langle \mathbf{C}_1, \mathbf{c}_1 \rangle$ y $\langle \mathbf{C}_2, \mathbf{c}_2 \rangle$ para las instrucciones s_1 y s_2 , respectivamente, cumplen con las siguientes *restricciones de partición de espacio*:

- Para todas las \mathbf{i}_1 en Z^{d_1} e \mathbf{i}_2 en Z^{d_2} tales que
 - $\mathbf{B}_1\mathbf{i}_1 + \mathbf{b}_1 \geq 0$,
 - $\mathbf{B}_2\mathbf{i}_2 + \mathbf{b}_2 \geq 0$, y
 - $\mathbf{F}_1\mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2\mathbf{i}_2 + \mathbf{f}_2$,
 se da el caso de que $\mathbf{C}_1\mathbf{i}_1 + \mathbf{c}_1 = \mathbf{C}_2\mathbf{i}_2 + \mathbf{c}_2$.

El objetivo del algoritmo de paralelización es encontrar, para cada instrucción, la partición con el rango más alto que cumpla con estas restricciones.

En la figura 11.25 se muestra un diagrama que ilustra la esencia de las restricciones de partición de espacio. Suponga que hay dos accesos estáticos en dos anidamientos de ciclo con los vectores índice \mathbf{i}_1 e \mathbf{i}_2 . Estos accesos son dependientes en el sentido en que acceden por lo menos a un elemento del arreglo en común y, por lo menos, uno de ellos es una escritura. La figura muestra accesos dinámicos específicos en los dos ciclos que por casualidad acceden al mismo elemento del arreglo, de acuerdo con las funciones de acceso afines $\mathbf{F}_1\mathbf{i}_1 + \mathbf{f}_1$ y $\mathbf{F}_2\mathbf{i}_2 + \mathbf{f}_2$. La sincronización es necesaria, a menos que las particiones afines para los dos accesos estáticos, $\mathbf{C}_1\mathbf{i}_1 + \mathbf{c}_1$ y $\mathbf{C}_2\mathbf{i}_2 + \mathbf{c}_2$, asignen los accesos dinámicos al mismo procesador.

Si elegimos una partición afín cuyo rango sea el máximo de los rangos de todas las instrucciones, obtenemos el máximo paralelismo posible. Sin embargo, bajo este particionamiento algunos procesadores pueden estar inactivos en ciertas ocasiones, mientras que otros procesadores ejecutan instrucciones cuyas particiones afines tienen un rango más pequeño. Esta situación puede ser aceptable si el tiempo requerido para ejecutar esas instrucciones es relativamente corto. En caso contrario, podemos elegir una partición afín cuyo rango sea más pequeño que el máximo posible, siempre y cuando ese rango sea mayor que 0.

En el ejemplo 11.41 mostramos un pequeño programa, diseñado para ilustrar el poder de esta técnica. Por lo general, las aplicaciones reales son mucho más simples que ésta, pero pueden tener condiciones delimitadoras que se asemejen a algunas de las cuestiones que se muestran aquí. Vamos a utilizar este ejemplo a lo largo de este capítulo para demostrar que los programas con accesos afines tienen restricciones de partición de espacio relativamente simples, que

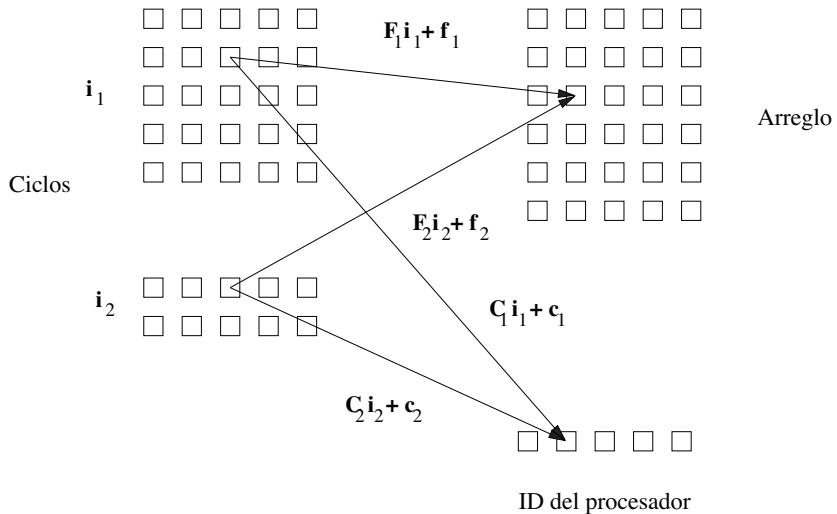


Figura 11.25: Restricciones de partición de espacio

estas restricciones pueden resolverse utilizando las técnicas estándar del álgebra lineal, y que el programa SPMD deseado se puede generar de manera mecánica a partir de las particiones afines.

Ejemplo 11.41: Este ejemplo muestra cómo formulamos las restricciones de partición de espacio para el programa que consiste en el anidamiento de ciclos pequeño con dos instrucciones, s_1 y s_2 , que se muestra en la figura 11.26.

```
for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++) {
        X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
        Y[i,j] = Y[i,j] + X[i,j-1]; /* (s2) */
    }
```

Figura 11.26: Un anidamiento de ciclos que exhibe largas cadenas de operaciones dependientes

En la figura 11.27 mostramos las dependencias de datos en el programa. Es decir, cada punto negro representa una instancia de la instrucción s_1 , y cada punto blanco representa una instancia de la instrucción s_2 . El punto ubicado en las coordenadas (i, j) representa la instancia de la instrucción que se ejecuta para esos valores de los índices de ciclo. Sin embargo, tenga en cuenta que la instancia de s_2 se encuentra justo debajo de la instancia de s_1 para el mismo par (i, j) , por lo que la escala vertical de j es mayor que la escala horizontal de i .

Observe que $X[i, j]$ se escribe mediante $s_1(i, j)$; es decir, mediante la instancia de la instrucción s_1 con los valores de índice i y j . Después se lee por $s_2(i, j + 1)$, de manera que $s_1(i, j)$

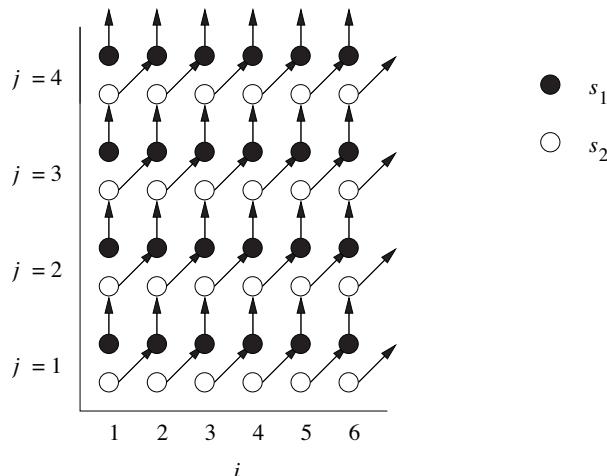


Figura 11.27: Dependencias del código en el ejemplo 11.41

debe preceder a $s_2(i, j + 1)$. Esta observación explica las flechas verticales que van de los puntos negros a los puntos blancos. De manera similar, $Y[i, j]$ se escribe mediante $s_2(i, j)$ y se lee después por $s_1(i + 1, j)$. Por ende, $s_2(i, j)$ debe preceder a $s_1(i + 1, j)$, lo cual explica las flechas que van de los puntos blancos a los negros.

Mediante este diagrama podemos ver con facilidad que este código puede parallelizarse sin sincronización, al asignar cada cadena de operaciones dependientes al mismo procesador. Sin embargo, no es fácil escribir el programa SPMD que implementa este esquema de asignación. Mientras que los ciclos en el programa original tienen 100 iteraciones cada uno, hay 200 cadenas, en las que la mitad empieza y termina con la instrucción s_1 , y la otra mitad empieza y termina con s_2 . Las longitudes de las cadenas varían de 1 a 100 iteraciones.

Como hay dos instrucciones, estamos buscando dos particiones afines, una para cada instrucción. Sólo tenemos que expresar las restricciones de partición de espacio para las particiones afines unidimensionales. Estas restricciones se utilizarán más adelante con el método de solución que trata de encontrar todas las particiones afines unidimensionales independientes, para combinarlas y obtener particiones afines multidimensionales. Podemos entonces representar la partición afín para cada instrucción mediante una matriz de 1×2 y un vector de 1×1 , para traducir el vector de índices $[i, j]$ en un solo número de procesador. Suponga que $\langle [C_{11} C_{12}], [c_1] \rangle, \langle [C_{21} C_{22}], [c_2] \rangle$ son las particiones afines unidimensionales para las instrucciones s_1 y s_2 , respectivamente.

Aplicamos seis pruebas de dependencia de datos:

1. El acceso de escritura $X[i, j]$ y él mismo en la instrucción s_1 ,
2. El acceso de escritura $X[i, j]$ con el acceso de lectura $X[i, j]$ en la instrucción s_1 ,
3. El acceso de escritura $X[i, j]$ en la instrucción s_1 con el acceso de lectura $X[i, j - 1]$ en la instrucción s_2 ,
4. El acceso de escritura $Y[i, j]$ y él mismo en la instrucción s_2 ,

5. El acceso de escritura $Y[i, j]$ con el acceso de lectura $Y[i, j]$ en la instrucción s_2 ,
6. El acceso de escritura $Y[i, j]$ en la instrucción s_2 con el acceso de lectura $Y[i - 1, j]$ en la instrucción s_1 .

Podemos ver que todas las pruebas de dependencia son simples y muy repetitivas. Las únicas dependencias presentes en este código ocurren en el caso (3), entre las instancias de los accesos $X[i, j]$ y $X[i, j - 1]$ y en el caso (6), entre $Y[i, j]$ y $Y[i - 1, j]$.

Las restricciones de partición de espacio impuestas por la dependencia de datos entre $X[i, j]$ en la instrucción s_1 y $X[i, j - 1]$ en la instrucción s_2 pueden expresarse en los siguientes términos:

Para todas las (i, j) y (i', j') tales que:

$$\begin{array}{ll} 1 \leq i \leq 100 & 1 \leq j \leq 100 \\ 1 \leq i' \leq 100 & 1 \leq j' \leq 100 \\ i = i' & j = j' - 1 \end{array}$$

tenemos que:

$$\begin{bmatrix} C_{11} & C_{12} \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} c_1 \end{bmatrix} = \begin{bmatrix} C_{21} & C_{22} \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} + \begin{bmatrix} c_2 \end{bmatrix}.$$

Es decir, las primeras cuatro condiciones establecen que (i, j) y (i', j') se encuentran dentro del espacio de iteraciones del anidamiento de ciclos, y las últimas dos condiciones establecen que los accesos dinámicos $X[i, j]$ y $X[i, j - 1]$ entran en contacto con el mismo elemento del arreglo. Podemos derivar la restricción de partición de espacio para los accesos $Y[i - 1, j]$ en la instrucción s_2 y $Y[i, j]$ en la instrucción s_1 de una forma similar. \square

11.7.4 Resolución de restricciones de partición de espacio

Una vez que se han extraído las restricciones de partición de espacio, podemos utilizar las técnicas estándar del álgebra lineal para buscar las particiones afines que cumplan con las restricciones. Primero vamos a mostrar cómo encontrar la solución para el ejemplo 11.41.

Ejemplo 11.42: Podemos encontrar las particiones afines para el ejemplo 11.41 con los siguientes pasos:

1. Crear las restricciones de partición de espacio que se muestran en el ejemplo 11.41. Utilizamos los límites de ciclo para determinar las dependencias de datos, pero no se utilizan en el resto del algoritmo para ninguna otra cosa.
2. Las variables desconocidas en las igualdades son $i, i', j, j', C_{11}, C_{12}, c_1, C_{21}, C_{22}$ y c_2 . Hay que reducir el número de variables desconocidas mediante el uso de las igualdades debido a las funciones de acceso: $i = i'$ y $j = j' - 1$. Para ello, utilizamos la eliminación gaussiana, la cual reduce las cuatro variables a dos: por decir $t_1 = i = i'$, y $t_2 = j + 1 = j'$. La igualdad para la partición se convierte en:

$$\begin{bmatrix} C_{11} - C_{21} & C_{12} - C_{22} \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} + \begin{bmatrix} c_1 - c_2 - C_{22} \end{bmatrix} = 0.$$

3. La ecuación anterior es válida para todas las combinaciones de t_1 y t_2 . Por ende, debe cumplirse que:

$$\begin{aligned} C_{11} - C_{21} &= 0 \\ C_{12} - C_{22} &= 0 \\ c_1 - c_2 - C_{22} &= 0. \end{aligned}$$

Si realizamos los mismos pasos con la restricción entre los accesos $Y[i-1, j]$ y $Y[i, j]$, obtenemos lo siguiente:

$$\begin{aligned} C_{11} - C_{21} &= 0 \\ C_{12} - C_{22} &= 0 \\ c_1 - c_2 + C_{21} &= 0. \end{aligned}$$

Al simplificar todas las restricciones en conjunto, obtenemos las siguientes relaciones:

$$C_{11} = C_{21} = -C_{22} = -C_{12} = c_2 - c_1.$$

4. Buscar todas las soluciones independientes para las ecuaciones que involucran sólo variables desconocidas en la matriz de coeficientes, ignorando las variables desconocidas en los vectores constantes de este paso. Sólo hay una elección independiente en la matriz de coeficientes, por lo que las particiones afines que buscamos pueden tener como máximo un rango de uno. Mantenemos la partición tan simple como sea posible, para lo cual establecemos $C_{11} = 1$. No podemos asignar 0 a C_{11} , ya que esto creará una matriz de coeficientes con rango 0, la cual asigna todas las iteraciones al mismo procesador. Después como resultado $C_{21} = 1$, $C_{22} = -1$, $C_{12} = -1$.
5. Buscar los términos constantes. Sabemos que la diferencia entre los términos constantes, $c_2 - c_1$, debe ser -1 . No obstante, podemos elegir los valores actuales. Para mantener las particiones simples, elegimos $c_2 = 0$; por lo tanto, $c_1 = -1$.

Hagamos que p sea el ID del procesador que ejecuta la iteración (i, j) . En términos de p , la partición afín es:

$$\begin{aligned} s_1 : [p] &= [1 \ -1] \begin{bmatrix} i \\ j \end{bmatrix} + [-1] \\ s_2 : [p] &= [1 \ -1] \begin{bmatrix} i \\ j \end{bmatrix} + [0] \end{aligned}$$

Es decir, la (i, j) -ésima iteración de s_1 se asigna al procesador $p = i - j - 1$, y la (i, j) -ésima iteración de s_2 se asigna al procesador $p = i - j$. \square

Algoritmo 11.43: Buscar una partición afín sin sincronización con el rango más alto para un programa.

ENTRADA: Un programa con accesos a arreglos afines.

SALIDA: Una partición.

MÉTODO: Haga lo siguiente:

1. Busque todos los pares dependientes de datos de accesos en un programa para cada par de accesos dependientes de datos, $\mathcal{F}_1 = \langle \mathbf{F}_1, \mathbf{f}_1, \mathbf{B}_1, \mathbf{b}_1 \rangle$ en la instrucción s_1 anidada en los ciclos d_1 , y $\mathcal{F}_2 = \langle \mathbf{F}_2, \mathbf{f}_2, \mathbf{B}_2, \mathbf{b}_2 \rangle$ en la instrucción s_2 anidada en los ciclos d_2 . Hagamos que $\langle \mathbf{C}_1, \mathbf{c}_1 \rangle$ y $\langle \mathbf{C}_2, \mathbf{c}_2 \rangle$ representen las particiones (actualmente desconocidas) de las instrucciones s_1 y s_2 , respectivamente. Las restricciones de partición de espacio establecen que si:

$$\mathbf{F}_1 \mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2 \mathbf{i}_2 + \mathbf{f}_2$$

entonces

$$\mathbf{C}_1 \mathbf{i}_1 + \mathbf{c}_1 = \mathbf{C}_2 \mathbf{i}_2 + \mathbf{c}_2$$

para todas las \mathbf{i}_1 e \mathbf{i}_2 , dentro de sus límites de ciclo respectivos. Vamos a generalizar el dominio de iteraciones para incluir todas las \mathbf{i}_1 en Z^{d_1} y todas las \mathbf{i}_2 en Z^{d_2} ; es decir, se asume que los límites son de infinito negativo a infinito. Esta suposición tiene sentido, ya que una partición afín no puede hacer uso del hecho de que una variable índice sólo puede asumir un conjunto limitado de valores enteros.

2. Para cada par de accesos dependientes, reducimos el número de variables desconocidas en los vectores índice.
- (a) Observe que $\mathbf{F}\mathbf{i} + \mathbf{f}$ es el mismo vector que:

$$[\mathbf{F} \quad \mathbf{f}] \begin{bmatrix} \mathbf{i} \\ 1 \end{bmatrix}.$$

Es decir, al agregar un componente 1 adicional al final del vector columna \mathbf{i} , podemos hacer que el vector columna \mathbf{f} sea una última columna adicional de la matriz \mathbf{F} . Por lo tanto, podemos rescribir la igualdad de las funciones de acceso $\mathbf{F}_1 \mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2 \mathbf{i}_2 + \mathbf{f}_2$ de la siguiente forma:

$$[\mathbf{F}_1 \quad -\mathbf{F}_2 \quad (\mathbf{f}_1 - \mathbf{f}_2)] \begin{bmatrix} \mathbf{i}_1 \\ \mathbf{i}_2 \\ 1 \end{bmatrix} = 0.$$

- (b) Las ecuaciones anteriores tendrán en general más de una solución. Sin embargo, aún podemos usar la eliminación gaussiana en la resolución de las ecuaciones para los componentes de \mathbf{i}_1 e \mathbf{i}_2 de la mejor forma posible. Es decir, tratamos de eliminar todas las variables hasta quedarnos sólo con las que no podemos eliminar. La solución resultante para \mathbf{i}_1 e \mathbf{i}_2 tendrá la siguiente forma:

$$\begin{bmatrix} \mathbf{i}_1 \\ \mathbf{i}_2 \\ 1 \end{bmatrix} = \mathbf{U} \begin{bmatrix} \mathbf{t} \\ 1 \end{bmatrix}.$$

en donde \mathbf{U} es una matriz triangular superior y \mathbf{t} es un vector de variables libres, cuyo rango comprende a todos los números enteros.

- (c) Podemos usar el mismo truco que en el paso (2a) para rescribir la igualdad de las particiones. Al sustituir el vector $(\mathbf{i}_1, \mathbf{i}_2, 1)$ con el resultado del paso (2b), podemos escribir las restricciones sobre las particiones de la siguiente manera:

$$[\mathbf{C}_1 \quad -\mathbf{C}_2 \quad (\mathbf{c}_1 - \mathbf{c}_2)] \mathbf{U} \begin{bmatrix} \mathbf{t} \\ 1 \end{bmatrix} = \mathbf{0}.$$

3. Eliminamos las variables que no son de partición. Las ecuaciones anteriores son válidas para todas las combinaciones de \mathbf{t} si:

$$[\mathbf{C}_1 \quad -\mathbf{C}_2 \quad (\mathbf{c}_1 - \mathbf{c}_2)] \mathbf{U} = \mathbf{0}.$$

Rescriba estas ecuaciones en la forma $\mathbf{Ax} = \mathbf{0}$, en donde \mathbf{x} es un vector de todos los coeficientes desconocidos de las particiones afines.

4. Buscamos el rango de la partición afín y resolvemos para las matrices de coeficientes. Como el rango de una partición afín es independiente del valor de los términos constantes en la partición, eliminamos todas las variables desconocidas que provienen de los vectores constantes, como \mathbf{c}_1 o \mathbf{c}_2 , con lo cual sustituimos $\mathbf{Ax} = \mathbf{0}$ por las restricciones simplificadas $\mathbf{A}'\mathbf{x}' = \mathbf{0}$. Buscamos las soluciones a $\mathbf{A}'\mathbf{x}' = \mathbf{0}$, expresándolas como \mathbf{B} , un conjunto de vectores básicos que abarcan el espacio nulo de \mathbf{A}' .
5. Buscamos los términos constantes. Derivamos una fila de la partición afín deseada de cada vector básico en \mathbf{B} , y derivamos los términos constantes usando $\mathbf{Ax} = \mathbf{0}$.

□

Observe que el paso 3 ignora las restricciones impuestas por los límites de ciclo en las variables \mathbf{t} . Como resultado, las restricciones son sólo más estrictas y el algoritmo debe, por lo tanto, ser seguro. Es decir, colocamos restricciones sobre las \mathbf{Cs} y las \mathbf{cs} , asumiendo que el valor de \mathbf{t} es arbitrario. Es posible que pueda haber otras soluciones para las \mathbf{Cs} y \mathbf{cs} que sean válidas sólo porque algunos valores de \mathbf{t} son imposibles. Si no buscamos estas otras soluciones podríamos perder una optimización, pero esto no puede ocasionar que el programa se cambie a un programa que haga algo distinto de lo que hace el programa original.

11.7.5 Un algoritmo simple de generación de código

El Algoritmo 11.43 genera particiones afines que dividen los cálculos en particiones independientes. Las particiones pueden asignarse de manera arbitraria a distintos procesadores, ya que son independientes unos de otros. Un procesador puede asignarse a más de una partición, y puede intercalar la ejecución de sus particiones, siempre y cuando las operaciones dentro de cada partición que, por lo general, tienen dependencias de datos, se ejecuten en forma secuencial.

Es muy fácil generar un programa correcto, dada una partición afín. Primero vamos a introducir el Algoritmo 11.45, un método simple que genera código para un solo procesador que ejecuta cada una de las particiones independientes en forma secuencial. Dicho código optimiza la localidad temporal, ya que los accesos a arreglos que tienen varios usos están muy cercanos en el tiempo. Además, el código puede convertirse con facilidad en un programa SPMD que ejecute cada partición en un procesador distinto. Por desgracia, el código generado es ineficiente; a continuación hablaremos sobre las optimizaciones para hacer que el código se ejecute con eficiencia.

La idea esencial es la siguiente. Recibimos límites para las variables índice de un anidamiento de ciclos, y hemos determinado, en el Algoritmo 11.43, una partición para los accesos de una instrucción específica s . Suponga que deseamos generar código secuencial para realizar la acción de cada procesador en forma secuencial. Creamos un ciclo externo para iterar a través de los IDs de los procesadores. Es decir, cada iteración de este ciclo realiza las operaciones asignadas a un ID de procesador específico. Este programa original se inserta como el cuerpo de este ciclo; además se agrega una prueba para proteger cada operación en el código y asegurar que cada procesador sólo ejecute las operaciones que tenga asignadas. De esta forma, garantizamos que el procesador ejecutará todas las instrucciones que tenga asignadas, en el orden secuencial original.

Ejemplo 11.44: Vamos a generar código para ejecutar las particiones independientes del ejemplo 11.41 en forma secuencial. El programa secuencial original de la figura 11.26 se repite aquí, como la figura 11.28.

```

for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++) {
        X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
        Y[i,j] = Y[i,j] + X[i,j-1]; /* (s2) */
    }

```

Figura 11.28: Repetición de la figura 11.26

En el ejemplo 11.41, el algoritmo de particionamiento afín encontró un grado de paralelismo. Por ende, el espacio del procesador se puede representar mediante una sola variable p . En ese ejemplo también seleccionamos una partición afín que, para todos los valores de las variables índice i y j con $1 \leq i \leq 100$ y $1 \leq j \leq 100$, asignó:

1. La instancia (i, j) de la instrucción s_1 al procesador $p = i - j - 1$.
2. La instancia (i, j) de la instrucción s_2 al procesador $p = i - j$.

Podemos generar el código en tres pasos:

1. Para cada instrucción, se buscan todos los IDs de los procesadores que participen en el cálculo. Combinamos las restricciones $1 \leq i \leq 100$ y $1 \leq j \leq 100$ con una de las ecuaciones $p = i - j - 1$ o $p = i - j$, y proyectamos i y j hacia fuera para obtener las nuevas restricciones:

- (a) $-100 \leq p \leq 98$, si utilizamos la función $p = i - j - 1$ que obtenemos para la instrucción s_1 .
- (b) $-99 \leq p \leq 99$ si utilizamos $p = i - j$ de la instrucción s_2 .
2. Se buscan todos los IDs de los procesadores que participen en cualquiera de las instrucciones. Al tomar la unión de estos rangos, obtenemos $-100 \leq p \leq 99$; estos límites son suficientes para cubrir todas las instancias de ambas instrucciones s_1 y s_2 .
3. Se genera el código para iterar a través de los cálculos en cada partición, en forma secuencial. El código, que se muestra en la figura 11.29, tiene un ciclo externo que itera a través de todos los IDs de las particiones que participan en el cálculo (línea (1)). Cada partición pasa a través del movimiento de generar los índices de todas las iteraciones en el programa secuencial original en las líneas (2) y (3), de manera que pueda elegir las iteraciones que se supone debe ejecutar el procesador p . Las pruebas de las líneas (4) y (6) se aseguran de que las instrucciones s_1 y s_2 se ejecuten sólo cuando el procesador p las vaya a ejecutar.

El código generado, aunque correcto, es en extremo ineficiente. En primer lugar, aun cuando cada procesador ejecuta cálculos de 99 iteraciones como máximo, genera índices de ciclo para 100×100 iteraciones, un orden de magnitud más de lo necesario. En segundo lugar, cada suma en el ciclo más interno está protegida por una prueba que crea otro factor constante de sobre-carga. En las secciones 11.7.6 y 11.7.7, respectivamente, trataremos estos dos tipos de ineficiencias. \square

```

1)   for (p = -100; p <= 99; p++)
2)     for (i = 1; i <= 100; i++)
3)       for (j = 1; j <= 100; j++) {
4)         if (p == i-j-1)
5)           X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
6)         if (p == i-j)
7)           Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
8)       }

```

Figura 11.29: Una simple rescritura de la figura 11.28 que itera sobre el espacio de procesadores

Aunque el código de la figura 11.29 parece estar diseñado para ejecutarse en un uniprocesador, podríamos tomar los ciclos internos, de las líneas (2) a (8), y ejecutarlos en 200 procesadores distintos, cada uno de los cuales tendría un valor distinto para p , de -100 a 99 . O, podríamos particionar la responsabilidad por los ciclos internos entre cualquier número de procesadores menor que 200, siempre y cuando arregláramos que cada procesador supiera de qué valores de p es responsable, y ejecutara las líneas (2) a (8) sólo para esos valores de p .

Algoritmo 11.45: Generación de código que ejecuta particiones de un programa en forma secuencial.

ENTRADA: Un programa P con accesos a arreglos afines. Cada instrucción s en el programa tiene límites asociados de la forma $\mathbf{B}_s \mathbf{i} + \mathbf{b}_s \geq 0$, en donde \mathbf{i} es el vector de los índices de ciclo para el anidamiento de ciclos en el que aparece la instrucción s . También asociada con la instrucción s , hay una partición $\mathbf{C}_s \mathbf{i} + \mathbf{c}_s = \mathbf{p}$, en donde \mathbf{p} es un vector m -dimensional de variables que representa a un ID de procesador; m es el máximo, sobre todas las instrucciones en el programa P , del rango de la partición para esa instrucción.

SALIDA: Un programa equivalente a P , pero que itera sobre el espacio de procesadores, en vez de hacerlo sobre los índices de ciclo.

MÉTODO: Haga lo siguiente:

1. Para cada instrucción, utilice la eliminación de Fourier-Motzkin para proyectar hacia fuera todas las variables índices de ciclo de los límites.
2. Use el Algoritmo 11.13 para determinar los límites en los IDs de las particiones.
3. Genere ciclos, uno para cada una de las m dimensiones del espacio de procesadores. Haga que $\mathbf{p} = [p_1, p_2, \dots, p_m]$ sea el vector de variables para estos ciclos; es decir, debe haber una variable para cada dimensión del espacio de procesadores. Cada variable p_i del ciclo varía a través de la unión de los espacios de particiones para todas las instrucciones en el programa P .

Observe que la unión de los espacios de particiones no es necesariamente convexa. Para mantener el algoritmo simple, en vez de enumerar sólo aquellas particiones que tienen un cálculo no vacío que realizar, establezca el límite inferior de cada p_i al mínimo de todos los límites inferiores impuestos por todas las instrucciones, y el límite superior de cada p_i al máximo de todos los límites superiores impuestos por todas las instrucciones. Por lo tanto, algunos valores de \mathbf{p} pueden no tener operaciones.

El código a ejecutar por cada partición es el programa secuencial original. Sin embargo, cada instrucción está protegida por un predicado, de manera que sólo se ejecuten las operaciones que pertenecen a la partición. \square

En breve veremos un ejemplo del Algoritmo 11.45. Sin embargo, tenga en cuenta que todavía estamos lejos del código óptimo para los ejemplos típicos.

11.7.6 Eliminación de iteraciones vacías

Ahora veremos la primera de las dos transformaciones necesarias para generar un código SPMD eficiente. El código ejecutado por cada procesador recorre todas las iteraciones en el programa original y elige las operaciones que se supone debe ejecutar. Si el código tiene k grados de paralelismo, el efecto es que cada procesador realiza k órdenes de magnitud más de trabajo. El propósito de la primera transformación es estrechar los límites de los ciclos para eliminar todas las iteraciones vacías.

Empezaremos por considerar las instrucciones en el programa, una a la vez. El espacio de iteraciones de una instrucción que debe ejecutar cada partición es el espacio de iteraciones original, más la restricción que impone la partición afín. Podemos generar límites estrechos para

cada instrucción mediante la aplicación del Algoritmo 11.13 al nuevo espacio de iteraciones; el nuevo vector de índices es como el vector de índices secuencial original, pero se le agregan los IDs de los procesadores como índices externos. Recuerde que el algoritmo generará límites estrechos para cada índice, en términos de los índices de ciclo circundantes.

Después de buscar los espacios de iteraciones de las distintas instrucciones, los combinamos, ciclo por ciclo, haciendo a los límites la unión de ellos para cada instrucción. Algunos ciclos terminan con una sola iteración, como se muestra a continuación en el ejemplo 11.46, y podemos simplemente eliminar el ciclo y establecer el índice de ciclo al valor para esa iteración.

Ejemplo 11.46: Para el ciclo de la figura 11.30(a), el Algoritmo 11.43 creará la siguiente partición afín:

$$\begin{aligned}s_1 : p &= i \\ s_2 : p &= j\end{aligned}$$

El Algoritmo 11.45 creará el código de la figura 11.30(b). Al aplicar el Algoritmo 11.13 a la instrucción s_1 se produce el siguiente límite: $p \leq i \leq p$, o implemente $i = p$. De manera similar, el algoritmo determina que $j = p$ para la instrucción s_2 . Por ende, obtenemos el código de la figura 11.30(c). La propagación de copia de las variables i y j eliminará la prueba innecesaria y producirá el código de la figura 11.30(d). \square

Ahora regresaremos al ejemplo 11.44 e ilustraremos el paso para combinar varios espacios de iteraciones provenientes de distintas instrucciones en uno solo.

Ejemplo 11.47: Ahora vamos a estrechar los límites de ciclo del código en el ejemplo 11.44. El espacio de iteraciones ejecutado por la partición p para la instrucción s_1 se define mediante las siguientes igualdades y desigualdades:

$$\begin{array}{rcl} -100 & \leq & p & \leq & 99 \\ 1 & \leq & i & \leq & 100 \\ 1 & \leq & j & \leq & 100 \end{array}$$

$$i - p - 1 = j$$

Al aplicar el Algoritmo 11.13 a lo anterior se crean las restricciones que se muestran en la figura 11.31(a). El Algoritmo 11.13 genera la restricción $p + 2 \leq i \leq 100 + p + 1$ de $i - p - 1 = j$ y $1 \leq j \leq 100$, y estrecha el límite superior de p a 98. De igual forma, los límites para cada una de las variables para la instrucción s_2 se muestran en la figura 11.31(b).

Los espacios de iteraciones para s_1 y s_2 en la figura 11.31 son similares, pero como se espera de la figura 11.27, ciertos límites difieren por 1 entre los dos. El código en la figura 11.32 se ejecuta a través de esta unión de espacios de iteraciones. Por ejemplo, para i se utiliza $\min(1, p + 1)$ como el límite inferior, y $\max(100, 100 + p + 1)$ como el límite superior. Observe que el ciclo más interno tiene 2 iteraciones, excepto que sólo tiene una la primera y última vez que se ejecuta. Por lo tanto, la sobrecarga al generar índices de ciclo se reduce por un orden de magnitud. Como el espacio de iteraciones ejecutado es más grande que el de s_1 y s_2 , aún son necesarias las condicionales para seleccionar cuándo deben ejecutarse estas instrucciones. \square

```

for (i=1; i<=N; i++)
    Y[i] = Z[i]; /* (s1) */
for (j=1; j<=N; j++)
    X[j] = Y[j]; /* (s2) */

```

(a) Código inicial.

```

for (p=1; p<=N; p++) {
    for (i=1; i<=N; i++)
        if (p == i)
            Y[i] = Z[i]; /* (s1) */
    for (j=1; j<=N; j++)
        if (p == j)
            X[j] = Y[j]; /* (s2) */
}

```

(b) Resultado de aplicar el Algoritmo 11.45.

```

for (p=1; p<=N; p++) {
    i = p;
    if (p == i)
        Y[i] = Z[i]; /* (s1) */
    j = p;
    if (p == j)
        X[j] = Y[j]; /* (s2) */
}

```

(c) Despues de aplicar el Algoritmo 11.13.

```

for (p=1; p<=N; p++) {
    Y[p] = Z[p]; /* (s1) */
    X[p] = Y[p]; /* (s2) */
}

```

(d) Código final.

Figura 11.30: Código para el ejemplo 11.46

$$j : \begin{array}{l} i - p - 1 \leq j \leq i - p - 1 \\ 1 \leq j \leq 100 \end{array}$$

$$i : \begin{array}{l} p + 2 \leq i \leq 100 + p + 1 \\ 1 \leq i \leq 100 \end{array}$$

$$p : -100 \leq p \leq 98$$

(a) Límites para la instrucción s_1 .

$$j : \begin{array}{l} i - p \leq j \leq i - p \\ 1 \leq j \leq 100 \end{array}$$

$$i : \begin{array}{l} p + 1 \leq i \leq 100 + p \\ 1 \leq i \leq 100 \end{array}$$

$$p : -99 \leq p \leq 99$$

(b) Límites para la instrucción s_2 .

Figura 11.31: Límites más estrechos en p , i y j para la figura 11.29

11.7.7 Eliminación de las pruebas de los ciclos más internos

La segunda transformación es la eliminación de las pruebas condicionales de los ciclos internos. Como se puede ver en los ejemplos anteriores, las pruebas condicionales permanecen si los espacios de iteraciones de las instrucciones en el ciclo se cruzan, pero no por completo. Para evitar la necesidad de pruebas adicionales, dividimos el espacio de iteraciones en subespacios, cada uno de los cuales ejecuta el mismo conjunto de instrucciones. Esta optimización requiere la duplicación de código, por lo cual sólo debe usarse para eliminar condicionales en los ciclos internos.

Para dividir un espacio de iteraciones de manera que se eliminan las pruebas en los ciclos internos, aplicamos los siguientes pasos repetidas veces hasta eliminar todas las pruebas en los ciclos internos:

1. Seleccionar un ciclo que consista de instrucciones con distintos límites.
2. Dividir el ciclo que utilice una condición tal que cierta instrucción se excluya de, por lo menos, uno de sus componentes. Elegimos la condición de entre los límites de los distintos poliedros que se traslanan. Si alguna instrucción tiene todas sus iteraciones sólo en uno de los medios planos de la condición, entonces dicha condición es útil.
3. Generar código para cada uno de estos espacios de iteraciones por separado.

Ejemplo 11.48: Vamos a eliminar las condicionales del código de la figura 11.32. Las instrucciones s_1 y s_2 se asignan al mismo conjunto de IDs de particiones, excepto para las particiones

```

for (p = -100; p <= 99; p++)
    for (i = max(1,p+1); i <= min(100,101+p); i++)
        for (j = max(1,i-p-1); j <= min(100,i-p); j++) {
            if (p == i-j-1)
                X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
            if (p == i-j)
                Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
        }
    }
}

```

Figura 11.32: Código de la figura 11.29 mejorado mediante límites de ciclo más estrechos

delimitadoras en ambos extremos. Por ende, sepáramos el espacio de particiones en tres subespacios:

1. $p = -100$,
2. $-99 \leq p \leq 98$, y
3. $p = 99$.

El código para cada subespacio puede entonces especializarse para el (los) valor(es) de p contenidos. La figura 11.33 muestra el código resultante para cada uno de los tres espacios de iteraciones.

Observe que los espacios primero y tercero no necesitan ciclos en i o j , ya que para el valor específico de p que define a cada espacio, estos ciclos son degenerados; sólo tienen una iteración. Por ejemplo, en el espacio (1), al sustituir $p = -100$ en los límites de ciclo se restringe i a 1, y por consecuencia j a 100. Las asignaciones a p en los espacios (1) y (3) son sin duda código muerto, y pueden eliminarse.

A continuación, dividimos el ciclo con el índice i en el espacio (2). De nuevo, las iteraciones primera y última del índice de ciclo i son distintas. Por ende, dividimos el ciclo en tres subespacios:

- a) $\max(1, p + 1) \leq i < p + 2$, en donde sólo se ejecuta s_2 .
- b) $\max(1, p + 2) \leq i \leq \min(100, 100 + p)$, en donde se ejecutan s_1 y s_2 .
- c) $101 + p < i \leq \min(101 + p, 100)$, en donde sólo se ejecuta s_1 .

Por lo tanto, el anidamiento de ciclos para el espacio (2) en la figura 11.33 puede escribirse como en la figura 11.34(a).

La figura 11.34(b) muestra el programa optimizado. Hemos sustituido la figura 11.34(a) por el anidamiento de ciclos en la figura 11.33. También propagamos las asignaciones a p , i y j hacia los accesos al arreglo. Al optimizar en el nivel de código intermedio, algunas de estas asignaciones se identificarán como subexpresiones comunes y se volverán a extraer del código de acceso al arreglo. \square

```

/* espacio (1) */
p = -100;
i = 1;
j = 100;
X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */

/* espacio (2) */
for (p = -99; p <= 98; p++)
    for (i = max(1,p+1); i <= min(100,101+p); i++)
        for (j = max(1,i-p-1); j <= min(100,i-p); j++) {
            if (p == i-j-1)
                X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
            if (p == i-j)
                Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
        }
}

/* espacio (3) */
p = 99;
i = 100;
j = 1;
Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */

```

Figura 11.33: Dividir el espacio de iteraciones sobre el valor de p

11.7.8 Transformaciones del código fuente

Hemos visto cómo podemos derivar, de simples particiones afines para cada instrucción, programas que son bastante distintos del código fuente original. En los ejemplos que hemos visto hasta ahora no es aparente de qué manera se correlacionan las particiones afines con los cambios en el nivel de código fuente. Esta sección muestra que podemos razonar acerca de los cambios al código fuente con relativa facilidad, al descomponer las particiones afines en una serie de transformaciones primitivas.

Siete transformaciones afines primitivas

Cada partición afín puede expresarse como una serie de transformaciones afines primitivas, cada una de las cuales corresponde a una simple modificación en el nivel de código fuente. Hay siete tipos de transformaciones primitivas: las primeras cuatro primitivas se ilustran en la figura 11.35, las últimas tres, también conocidas como *transformaciones unimodulares*, se ilustran en la figura 11.36.

La figura muestra un ejemplo para cada primitiva: un código fuente, una partición afín, y el código resultante. También dibujamos las dependencias de datos para el código, antes y después de las transformaciones. De los diagramas de dependencias de datos, podemos ver que cada primitiva corresponde a una transformación geométrica simple, e induce una transformación de código bastante simple. Las siete primitivas son:

```

/* espacio (2u *)
for (p = -99; p <= 98; p++) {
    /* espacio (2au *)
    if (p >= 0) {
        i = p+1;
        j = 1;
        Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
    }
    /* espacio (2bu *)
    for (i = max(1,p+2); i <= min(100,100+p); i++) {
        j = i-p-1;
        X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
        j = i-p;
        Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
    }
    /* espacio (2cu *)
    if (p <= -1) {
        i = 101+p;
        j = 100;
        X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
    }
}

```

(a) División del espacio (2) sobre el valor de i .

```

/* espacio (1u; p = -100 */
X[1,100] = X[1,100] + Y[0,100]; /* (s1) */

/* espacio (2u *)
for (p = -99; p <= 98; p++) {
    if (p >= 0)
        Y[p+1,1] = X[p+1,0] + Y[p+1,1]; /* (s2) */
    for (i = max(1,p+2); i <= min(100,100+p); i++) {
        X[i,i-p-1] = X[i,i-p-1] + Y[i-1,i-p-1]; /* (s1) */
        Y[i,i-p] = X[i,i-p-1] + Y[i,i-p]; /* (s2) */
    }
    if (p <= -1)
        X[101+p,100] = X[101+p,100] + Y[101+p-1,100]; /* (s1) */
}
/* espacio (3u; p = 99 */
Y[100,1] = X[100,0] + Y[100,1]; /* (s2) */

```

(b) Código optimizado equivalente a la figura 11.28.

Figura 11.34: Código para el ejemplo 11.48

CÓDIGO FUENTE	PARTICIÓN	CÓDIGO TRANSFORMADO
<pre>for (i=1; i<=N; i++) Y[i] = Z[i]; /*s1*/ for (j=1; j<=N; j++) X[j] = Y[j]; /*s2*/</pre>	Fusión $s_1 : p = i$ $s_2 : p = j$	<pre>for (p=1; p<=N; p++){ Y[p] = Z[p]; X[p] = Y[p]; }</pre>
<pre>for (p=1; p<=N; p++){ Y[p] = Z[p]; X[p] = Y[p]; }</pre>	Fisión $s_1 : i = p$ $s_2 : j = p$	<pre>for (i=1; i<=N; i++) Y[i] = Z[i]; /*s1*/ for (j=1; j<=N; j++) X[j] = Y[j]; /*s2*/</pre>
<pre>for (i=1; i<=N; i++) { Y[i] = Z[i]; /*s1*/ X[i] = Y[i-1]; /*s2*/ }</pre>	Reindexado $s_1 : p = i$ $s_2 : p = i - 1$	<pre>if (N>=1) X[1]=Y[0]; for (p=1; p<=N-1; p++){ Y[p]=Z[p]; X[p+1]=Y[p]; } if (N>=1) Y[N]=Z[N];</pre>
<pre>for (i=1; i<=N; i++) Y[2*i] = Z[2*i]; /*s1*/ for (j=1; j<=2N; j++) X[j]=Y[j]; /*s2*/</pre>	Escalado $s_1 : p = 2 * i$ $(s_2 : p = j)$	<pre>for (p=1; p<=2*N; p++){ if (p mod 2 == 0) Y[p] = Z[p]; X[p] = Y[p]; }</pre>

Figura 11.35: Transformaciones afines primitivas (I)

CÓDIGO FUENTE	PARTICIÓN	CÓDIGO TRANSFORMADO
<pre>for (i=0; i>=N; i++) Y[N-i] = Z[i]; /*s1*/ for (j=0; j<=N; j++) X[j] = Y[j]; /*s2*/ </pre>	<p>Inversión $s_1 : p = N - i$ $(s_2 : p = j)$</p>	<pre>for (p=0; p<=N; p++) Y[p] = Z[N-p]; X[p] = Y[p]; }</pre>
<pre>for (i=1; i<=N; i++) for (j=0; j<=M; j++) Z[i,j] = Z[i-1,j]; </pre>	<p>Permutación</p> $\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$	<pre>for (p=0; p<=M; p++) for (q=1; q<=N; i++) Z[q,p] = Z[q-1,p]; </pre>
<pre>for (i=1; i<=N+M-1; i++) for (j=max(1,i+N); j<=min(i,M); j++) Z[i,j] = Z[i-1,j-1]; </pre>	<p>Desplazamiento</p> $\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$	<pre>for (p=1; p<=N; p++) for (q=1; q<=M; q++) Z[p,q-p] = Z[p-1,q-p-1]; </pre>

Figura 11.36: Transformaciones afines primitivas (II)

Transformaciones unimodulares

Una transformación unimodular se representa mediante sólo una matriz de coeficientes unimodular, sin un vector constante. Una *matriz unimodular* es una matriz cuadrada, cuyo determinante es ± 1 . La importancia de una transformación unimodular es que asigna un espacio de iteraciones n -dimensional a otro poliedro n -dimensional, en donde hay una correspondencia de uno a uno entre las iteraciones de los dos espacios.

1. *Fusión.* La transformación de fusión se caracteriza por la asignación de múltiples índices de ciclo en el programa original al mismo índice de ciclo. El nuevo ciclo fusiona las instrucciones de distintos ciclos.
2. *Fisión.* La fisión es el inverso de la fusión. Asigna el mismo índice de ciclo para distintas instrucciones a distintos índices de ciclo en el código transformado. Esto divide el ciclo original en varios ciclos.
3. *Reindexado.* El reindexado desplaza las ejecuciones dinámicas de una instrucción en base a un número constante de iteraciones. La transformación afín tiene un término constante.
4. *Escalado.* Las iteraciones consecutivas en el programa de código fuente se separan mediante un factor constante. La transformación afín tiene un coeficiente no unitario positivo.
5. *Inversión.* Ejecuta las iteraciones en un ciclo en orden inverso. La inversión se caracteriza por tener -1 como coeficiente.
6. *Permutación.* Permuta los ciclos internos y externos. La transformación afín consiste en filas permutadas de la matriz de identidad.
7. *Desplazamiento.* Itera a través del espacio de iteraciones en los ciclos a cierto ángulo. La transformación afín es una matriz unimodular, con 1s en la diagonal.

Una interpretación geométrica de la paralelización

Las transformaciones afines mostradas en todos los ejemplos, excepto el de la fisión, se derivan mediante la aplicación del algoritmo de partición afín sin sincronización a los códigos fuente respectivos (en la siguiente sección veremos cómo la fisión puede paralelizar el código con sincronización). En cada uno de los ejemplos, el código generado tiene un ciclo paralelizable (más externo), cuyas iteraciones pueden asignarse a distintos procesadores y no se requiere sincronización.

Estos ejemplos muestran que hay una interpretación geométrica simple de cómo funciona la paralelización. Las aristas de dependencia siempre apuntan de una instancia anterior a una posterior. Así, las dependencias entre las instrucciones separadas que no estén anidadas en un ciclo común siguen el orden léxico; las dependencias entre las instrucciones anidadas en el

mismo ciclo siguen el orden lexicográfico. En sentido geométrico, las dependencias de un ciclo bidimensional siempre apuntan dentro del rango $[0^\circ, 180^\circ]$, lo cual significa que el ángulo de la dependencia debe estar por debajo de 180° , pero no debe ser menor que 0° .

Las transformaciones afines modifican el orden de las iteraciones de tal forma que todas las dependencias se encuentran sólo entre las operaciones anidadas dentro de la misma iteración del ciclo más externo. En otras palabras, no hay aristas de dependencia en los límites de las iteraciones en el ciclo más externo. Podemos parallelizar los códigos fuente simples al dibujar sus dependencias y buscar dichas transformaciones en forma geométrica.

11.7.9 Ejercicios para la sección 11.7

Ejercicio 11.7.1: Para el siguiente ciclo:

```
for (i = 2; i < 100; i++)
    A[i] = A[i-2];
```

- ¿Cuál es el mayor número de procesadores que pueden usarse en forma efectiva para ejecutar este ciclo?
- Rescriba el código con el procesador p como parámetro.
- Establezca y encuentre una solución a las restricciones de partición de espacio para este ciclo.
- ¿Cuál es la partición afín del rango más alto para este ciclo?

Ejercicio 11.7.2: Repita el ejercicio 11.7.1 para los anidamientos de ciclos en la figura 11.37.

Ejercicio 11.7.3: Rescriba el siguiente código:

```
for (i = 0; i < 100; i++)
    A[i] = 2*A[i];
for (j = 0; j < 100; j++)
    A[j] = A[j] + 1;
```

de manera que consista en un solo ciclo. Rescriba el ciclo en términos de un número de procesador p , de forma que el código pueda particionarse entre 100 procesadores, en donde el procesador p ejecute la iteración p .

Ejercicio 11.7.4: En el siguiente código:

```
for (i = 1; i < 100; i++)
    for (j = 1; j < 100; j++)
/* (s) */    A[i,j] =
                    (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1])/4;
```

```
for (i = 0; i <= 97; i++)
  A[i] = A[i+2];
```

(a)

```
for (i = 1; i <= 100; i++)
  for (j = 1; j <= 100; j++)
    for (k = 1; k <= 100; k++) {
      A[i,j,k] = A[i,j,k] + B[i-1,j,k];
      B[i,j,k] = B[i,j,k] + C[i,j-1,k];
      C[i,j,k] = C[i,j,k] + A[i,j,k-1];
    }
```

!(b)

```
for (i = 1; i <= 100; i++)
  for (j = 1; j <= 100; j++)
    for (k = 1; k <= 100; k++) {
      A[i,j,k] = A[i,j,k] + B[i-1,j,k];
      B[i,j,k] = B[i,j,k] + A[i,j-1,k];
      C[i,j,k] = C[i,j,k] + A[i,j,k-1] + B[i,j,k];
    }
```

!(c)

Figura 11.37: Código para el ejercicio 11.7.2

las únicas restricciones son que la instrucción s que forma el cuerpo del anidamiento de ciclos debe ejecutar las iteraciones $s(i-1, j)$ y $s(i, j-1)$ antes de ejecutar la iteración $s(i, j)$. Verifique que éstas sean las únicas restricciones necesarias. Después rescriba el código, de forma que el ciclo externo tenga la variable índice p , y en la p -ésima iteración del ciclo externo se ejecuten todas las instancias de $s(i, j)$ tales que $i + j = p$.

Ejercicio 11.7.5: Repita el ejercicio 11.7.4, pero haga que en la p -ésima iteración del ciclo externo, se ejecuten las instancias de s tales que $i - j = p$.

! Ejercicio 11.7.6: Combine los siguientes ciclos:

```
for (i = 0; i < 100; i++)
  A[i] = B[i];
for (j = 98; j >= 0; j = j-2)
  B[i] = i;
```

en un solo ciclo, preservando todas las dependencias.

Ejercicio 11.7.7: Muestre que la siguiente matriz:

$$\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

es unimodular. Describa la transformación que realiza sobre un anidamiento de ciclos bidimensional.

Ejercicio 11.7.8: Repita el ejercicio 11.7.7 con la siguiente matriz:

$$\begin{bmatrix} 1 & 3 \\ 2 & 5 \end{bmatrix}$$

11.8 Sincronización entre ciclos paralelos

La mayoría de los programas no tienen paralelismo si no permitimos que los procesadores realicen sincronizaciones. Pero al agregar hasta un pequeño número constante de operaciones de sincronización a un programa, se puede exponer más paralelismo. En esta sección veremos primero el paralelismo que se logra mediante un número constante de sincronizaciones, y en la siguiente sección veremos el caso general, en donde incrustamos las operaciones de sincronización en los ciclos.

11.8.1 Un número constante de sincronizaciones

Los programas que no tienen paralelismo sin sincronización pueden contener una secuencia de ciclos, algunos de los cuales son paralelizables si se consideran en forma independiente. Podemos paralelizar dichos ciclos mediante la introducción de barreras de sincronización, antes y después de su ejecución. El ejemplo 11.49 ilustra este punto.

```

for (i = 1; i < n; i++)
    for (j = 0; j < n; j++)
        X[i,j] = f(X[i,j] + X[i-1,j]);
    for (i = 0; i < n; i++)
        for (j = 1; j < n; j++)
            X[i,j] = g(X[i,j] + X[i,j-1]);

```

Figura 11.38: Dos anidamientos de ciclos secuenciales

Ejemplo 11.49: En la figura 11.38 hay un programa que representa a un algoritmo de integración ADI (Alternating Direction Implicit, implícito de dirección alternante). No hay paralelismo sin sincronización. Las dependencias en el primer anidamiento de ciclos requieren que cada procesador trabaje en una columna del arreglo X ; sin embargo, las dependencias en el segundo anidamiento de ciclos requieren que cada procesador trabaje en una fila del arreglo X . Para que no haya comunicación, todo el arreglo tiene que residir en el mismo procesador, por

lo cual no hay paralelismo. Sin embargo, observamos que ambos ciclos pueden paralelizarse en forma independiente.

Una manera de paralelizar el código es hacer que distintos procesadores trabajen en distintas columnas del arreglo en el primer ciclo, que sincronicen y esperen a que todos los procesadores terminen, y después operen sobre las filas individuales. De esta forma, todos los cálculos en el algoritmo pueden paralelizarse con la introducción de sólo una operación de sincronización. Sin embargo, observamos que aunque sólo se realiza una sincronización, esta paralelización requiere que se transfieran casi todos los datos en la matriz X entre los procesadores. Es posible reducir la cantidad de comunicación al introducir más sincronizaciones, lo cual veremos en la sección 11.9.9. \square

Puede parecer que este método sólo puede aplicarse a los programas que consisten en una secuencia de anidamientos de ciclos. Sin embargo, podemos crear oportunidades adicionales para la optimización, a través de las transformaciones de código. Podemos aplicar la fisión de ciclos para descomponer los ciclos en el programa original en varios ciclos más pequeños, que después se pueden paralelizar de manera individual al separarlos con barreras. Ilustraremos esta técnica con el ejemplo 11.50.

Ejemplo 11.50: Considere el siguiente ciclo:

```
for (i=1; i<=n; i++) {
    X[i] = Y[i] + Z[i];      /* (s1) */
    W[A[i]] = X[i];          /* (s2) */
}
```

Si no conocemos los valores en el arreglo A , debemos asumir que el acceso en la instrucción s_2 puede escribir a cualquiera de los elementos de W . Por ende, las instancias de s_2 deben ejecutarse en forma secuencial, en el orden en el que se ejecutan en el programa original.

No hay paralelismo sin sincronización, y el Algoritmo 11.43 sólo asigna todos los cálculos al mismo procesador. Sin embargo, como mínimo, las instancias de la instrucción s_1 pueden ejecutarse en paralelo. Podemos paralelizar parte de este código, al hacer que distintos procesadores ejecuten distintas instancias de la instrucción s_1 . Después, en un ciclo secuencial separado, un procesador (por decir, el número 0) ejecuta a s_2 , como en el código SPMD que se muestra en la figura 11.39. \square

11.8.2 Grafos de dependencias del programa

Para encontrar todo el paralelismo posible mediante un número constante de sincronizaciones, podemos aplicar la fisión al programa original masivamente. Se descomponen los ciclos en todos los ciclos separados que sea posible, y después se paraleliza cada ciclo de manera independiente.

Para exponer todas las oportunidades para la fisión de ciclos, utilizamos la abstracción de un *grafo de dependencias de programa* (Program-Dependence Graph, PDG). Un grafo de depen-

```

X[p] = Y[p] + Z[p]; /* (s1) */
/* barrera de sincronización */
if (p == 0)
    for (i=1; i<=n; i++)
        W[A[i]] = X[i]; /* (s2) */

```

Figura 11.39: Código SPMD para el ciclo en el ejemplo 11.50, en donde p es una variable que contiene el ID del procesador

dencias de un programa es un grafo cuyos nodos son las instrucciones de asignación del programa, y cuyas aristas capturan las dependencias de datos, y las direcciones de la dependencia de datos, entre las instrucciones. Una arista de la instrucción s_1 a la instrucción s_2 existe cada vez que cierta instancia dinámica de s_1 comparte una dependencia de datos con una instancia dinámica *posterior* de s_2 .

Para construir el PDG para un programa, primero buscamos las dependencias de datos entre cada par de accesos estáticos (no necesariamente distintos) en cada par de instrucciones (no necesariamente distintas). Suponga que determinamos que hay una dependencia entre el acceso \mathcal{F}_1 en la instrucción s_1 y el acceso \mathcal{F}_2 en la instrucción s_2 . Recuerde que una instancia de una instrucción se especifica mediante un vector de índices $\mathbf{i} = [i_1, i_2, \dots, i_m]$, en donde i_k es el índice del k -ésimo ciclo más externo en el cual está incrustada la instrucción.

1. Si existe un par de instancias dependientes de datos, \mathbf{i}_1 de s_1 e \mathbf{i}_2 de s_2 , y si \mathbf{i}_1 se ejecuta antes de \mathbf{i}_2 en el programa original, lo cual se escribe como $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$, entonces hay una arista de s_1 a s_2 .
2. De manera similar, si existe un par de instancias dependientes de datos, \mathbf{i}_1 de s_1 e \mathbf{i}_2 de s_2 , y si $\mathbf{i}_2 \prec_{s_1 s_2} \mathbf{i}_1$, entonces hay una arista de s_2 a s_1 .

Observe que es posible para una dependencia de datos entre dos instrucciones s_1 y s_2 generar tanto una arista de s_1 a s_2 , como una arista de s_2 que regrese a s_1 .

En el caso especial en el que las instrucciones s_1 y s_2 no son distintas, $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$ si, y sólo si $\mathbf{i}_1 \prec \mathbf{i}_2$ (\mathbf{i}_1 es menor que \mathbf{i}_2 , en sentido lexicográfico). En el caso general, s_1 y s_2 pueden ser distintas instrucciones, que quizás pertenezcan a distintos anidamientos de ciclos.

Ejemplo 11.51: Para el programa del ejemplo 11.50, no hay dependencias entre las instancias de la instrucción s_1 . Sin embargo, la i -ésima instancia de la instrucción s_2 debe seguir a la i -ésima instancia de la instrucción s_1 . Peor aún, como la referencia $W[A[i]]$ puede escribir en cualquier elemento del arreglo W , la i -ésima instancia de s_2 depende de todas las instancias anteriores de s_2 . Es decir, la instrucción s_2 depende de sí misma. El PDG para el programa del ejemplo 11.50 se muestra en la figura 11.40. Observe que hay un ciclo en el grafo, el cual contiene sólo a s_2 . \square

El grafo de dependencias del programa ayuda a determinar si podemos dividir las instrucciones en un ciclo. Las instrucciones conectadas en un ciclo en un PDG no pueden dividirse.



Figura 11.40: Grafo de dependencias del programa del ejemplo 11.50

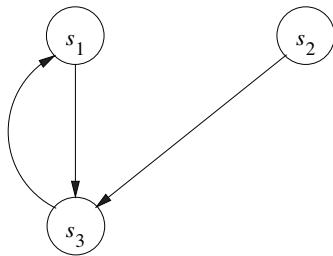
Si $s_1 \rightarrow s_2$ es una dependencia entre dos instrucciones en un ciclo, entonces cierta instancia de s_1 debe ejecutarse antes de cierta instancia de s_2 , y viceversa. Observe que esta dependencia mutua ocurre sólo si s_1 y s_2 están incrustadas en cierto ciclo común. Debido a la dependencia mutua, no podemos ejecutar todas las instancias de una instrucción antes de la otra y, por lo tanto, no se permite la fisión de ciclos. Por otro lado, si la dependencia $s_1 \rightarrow s_2$ es unidireccional, podemos dividir el ciclo y ejecutar todas las instancias de s_1 primero, y después todas las de s_2 .

```

for (i = 0; i < n; i++) {
    Z[i] = Z[i] / W[i];           /* (s1) */
    for (j = i; j < n; j++) {
        X[i,j] = Y[i,j]*Y[i,j];  /* (s2) */
        Z[j] = Z[j] + X[i,j];    /* (s3) */
    }
}

```

(a) Un programa.



(b) Su grafo de dependencias.

Figura 11.41: Programa y grafo de dependencias para el ejemplo 11.52

Ejemplo 11.52: La figura 11.41(b) muestra el grafo de dependencias para el programa de la figura 11.41(a). Las instrucciones s_1 y s_3 pertenecen a un ciclo en el grafo y, por lo tanto, no pueden colocarse en ciclos separados. Sin embargo, podemos dividir la instrucción s_2 y ejecutar todas sus instancias antes de ejecutar el resto de los cálculos, como en la figura 11.42. El primer ciclo es paralelizable, pero el segundo no. Podemos paralelizar el primer ciclo colocando barreras antes y después de su ejecución en paralelo. \square

```

for (i = 0; i < n; i++)
    for (j = i; j < n; j++)
        X[i,j] = Y[i,j]*Y[i,j]; /* (s2) */
for (i = 0; i < n; i++) {
    Z[i] = Z[i] / W[i];           /* (s1) */
    for (j = i; j < n; j++)
        Z[j] = Z[j] + X[i,j];    /* (s3) */
}

```

Figura 11.42: Agrupamiento de los componentes de un anidamiento de ciclos fuertemente conectados

11.8.3 Tiempo jerárquico

Aunque la relación $\prec_{s_1s_2}$ puede ser muy difícil de calcular en general, hay una familia de programas para la cual las optimizaciones de esta sección se aplican comúnmente, y para la cual hay una manera directa de calcular las dependencias. Suponga que el programa está estructurado por bloques, que consiste en ciclos y operaciones aritméticas simples, y ninguna otra construcción de control. Una instrucción en el programa es una instrucción de asignación, una secuencia de instrucciones, o una construcción de ciclo, cuyo cuerpo es una instrucción. Por ende, la estructura de control representa una jerarquía. En la parte superior de ésta se encuentra el nodo que representa a la instrucción de todo el programa. Una instrucción de asignación es un nodo hoja. Si una instrucción es una secuencia, entonces sus hijos son las instrucciones dentro de la secuencia, distribuidos de izquierda a derecha, de acuerdo con su orden léxico. Si una instrucción es un ciclo, entonces sus hijos son los componentes del cuerpo del ciclo que, por lo general, es una secuencia de una o más instrucciones.

```

s0;
L1: for (i = 0; ...) {
    s1;
    L2: for (j = 0; ...) {
        s2;
        s3;
    }
    L3: for (k = 0; ... )
        s4;
    s5;
}

```

Figura 11.43: Un programa estructurado en forma jerárquica

Ejemplo 11.53: La estructura jerárquica del programa en la figura 11.43 se muestra en la figura 11.44. La naturaleza jerárquica de la secuencia de ejecución se destaca en la figura 11.45.

La instancia individual de s_0 precede a todas las demás operaciones, ya que es la primera instrucción que se ejecuta. A continuación, ejecutamos todas las instrucciones a partir de la primera iteración del ciclo externo, antes de las que se encuentran en la segunda iteración, y así sucesivamente. Para todas las instancias dinámicas cuyo índice de ciclo i tiene el valor 0, las instrucciones s_1 , L_2 , L_3 y s_5 se ejecutan en orden léxico. Podemos repetir el mismo argumento para generar el resto del orden de ejecución. \square

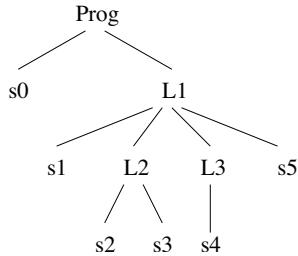


Figura 11.44: Estructura jerárquica del programa en el ejemplo 11.53

1:	s_0						
2:	L_1	$i = 0$	s_1				
3:			L_2	$j = 0$	s_2		
4:					s_3		
5:				$j = 1$	s_2		
6:					s_3		
7:					...		
8:			L_3	$k = 0$	s_4		
9:				$k = 1$	s_4		
10:					...		
11:				s_5			
12:		$i = 1$	s_1				
13:			...				

Figura 11.45: Orden de ejecución del programa en el ejemplo 11.53

Podemos resolver el ordenamiento de dos instancias provenientes de dos instrucciones distintas en forma jerárquica. Si las instrucciones comparten ciclos comunes, comparamos los valores de sus índices de ciclo comunes, empezando con el ciclo más externo. Tan pronto como encontramos una diferencia entre sus valores índice, ésta determina el ordenamiento. Sólo si los valores índice para los ciclos externos son iguales, tenemos que comparar los índices del siguiente ciclo interno. Este proceso es similar a la forma en que podríamos comparar el tiempo expresado en términos de horas, minutos y segundos. Para comparar dos tiempos, primero comparamos las horas, y sólo si se refieren a la misma hora compararíamos entonces los minutos,

y así sucesivamente. Si los valores índice son iguales para todos los ciclos comunes, entonces resolvemos el orden con base en su colocación léxica relativa. Así, al orden de ejecución para los programas con ciclos anidados simples que hemos estado viendo se le conoce como “tiempo jerárquico”.

Hagamos que s_1 sea una instrucción anidada en un ciclo con d_1 niveles, y s_2 en un ciclo con d_2 niveles, compartiendo d ciclos comunes (externos); observe que, sin duda, $d \leq d_1$ y $d \leq d_2$. Suponga que $\mathbf{i} = [i_1, i_2, \dots, i_{d_1}]$ es una instancia de s_1 y que $\mathbf{j} = [j_1, j_2, \dots, j_{d_2}]$ es una instancia de s_2 .

$\mathbf{i} \prec_{s_1 s_2} \mathbf{j}$ si y sólo si se cumple una de las siguientes dos condiciones:

1. $[i_1, i_2, \dots, i_d] \prec [j_1, j_2, \dots, j_d]$, o
2. $[i_1, i_2, \dots, i_d] = [j_1, j_2, \dots, j_d]$, y s_1 aparece antes que s_2 , en sentido léxico.

El predicado $[i_1, i_2, \dots, i_d] \prec [j_1, j_2, \dots, j_d]$ puede escribirse como una desunión de desigualdades lineales:

$$(i_1 < j_1) \vee (i_1 = j_1 \wedge i_2 < j_2) \vee \dots \vee (i_1 = j_1 \wedge \dots \wedge i_{d-1} = j_{d-1} \wedge i_d < j_d)$$

La arista de un PDG de s_1 a s_2 existe mientras que la condición de dependencia de datos y una de las cláusulas de desunión puedan volverse verdaderas al mismo tiempo. Por ende, tal vez tengamos que resolver hasta d o $d + 1$ programas enteros lineales, dependiendo de si s_1 aparece antes que s_2 en sentido léxico, para determinar la existencia de una arista.

11.8.4 El algoritmo de paralelización

Ahora presentaremos un algoritmo simple que primero divide el cálculo en todos los ciclos distintos que sea posible, y después los paraleliza en forma independiente.

Algoritmo 11.54: Maximizar el grado de paralelismo permitido por $O(1)$ sincronizaciones.

ENTRADA: Un programa con accesos a un arreglo.

SALIDA: Código SPMD con un número constante de barreras de sincronización.

MÉTODO:

1. Construya el grafo de dependencias del programa y particione las instrucciones en componentes fuertemente conectados (SCCs). En la sección 10.5.8 vimos que un componente fuertemente conectado es un subgrafo máximo del original, en el cual todos los nodos en el subgrafo pueden llegar a cualquier otro nodo.
2. Transforme el código para ejecutar SCCs en un orden topológico, aplicando la fisión si es necesario.
3. Aplique el Algoritmo 11.43 a cada SCC para encontrar todo su paralelismo sin sincronización. Se insertan barreras antes y después de cada SCC paralelizado.

□

Aunque el Algoritmo 11.54 encuentra todos los grados de paralelismo con $O(1)$ sincronizaciones, tiene varias debilidades. En primer lugar, puede introducir sincronizaciones innecesarias. De hecho, si aplicamos este algoritmo a un programa que pueda paralelizarse sin sincronización, el algoritmo paralelizará cada instrucción de manera independiente, e introducirá una barrera de sincronización entre los ciclos paralelos que ejecutan cada instrucción. En segundo lugar, aunque puede haber sólo un número constante de sincronizaciones, el esquema de paralelización puede transferir muchos datos entre los procesadores con cada sincronización. En algunos casos, el costo de la comunicación hace que el paralelismo sea demasiado costoso, e incluso hasta podemos estar mejor si ejecutamos el programa en forma secuencial, en un uniprocesador. En las siguientes secciones, veremos formas de incrementar la localidad de los datos, y en consecuencia reducir la cantidad de comunicación.

11.8.5 Ejercicios para la sección 11.8

Ejercicio 11.8.1: Aplique el Algoritmo 11.54 al código de la figura 11.46.

```

for (i=0; i<100; i++)
    A[i] = A[i] + X[i]; /* (s1) */
for (i=0; i<100; i++)
    for (j=0; j<100; j++)
        B[i,j] = Y[i,j] + A[i] + A[j]; /* (s2) */

```

Figura 11.46: Código para el ejercicio 11.8.1

Ejercicio 11.8.2: Aplique el Algoritmo 11.54 al código de la figura 11.47.

```

for (i=0; i<100; i++)
    A[i] = A[i] + X[i]; /* (s1) */
for (i=0; i<100; i++) {
    B[i] = B[i] + A[i]; /* (s2) */
    for (j=0; j<100; j++)
        C[j] = Y[j] + B[j]; /* (s3) */
}

```

Figura 11.47: Código para el ejercicio 11.8.2

Ejercicio 11.8.3: Aplique el Algoritmo 11.54 al código de la figura 11.48.

```

for (i=0; i<100; i++)
    A[i] = A[i] + X[i]; /* (s1) */
for (i=0; i<100; i++) {
    for (j=0; j<100; j++)
        B[j] = A[i] + Y[j]; /* (s2) */
    C[i] = B[i] + Z[i]; /* (s3) */
    for (j=0; j<100; j++)
        D[i,j] = A[i] + B[j]; /* (s4) */
}

```

Figura 11.48: Código para el ejercicio 11.8.3

11.9 Canalizaciones

En las canalizaciones, una tarea se descompone en varias etapas a realizar en distintos procesadores. Por ejemplo, una tarea que se calcula mediante un ciclo de n iteraciones puede estructurarse como una canalización de n etapas. Cada etapa se asigna a un procesador distinto; cuando un procesador termina con su etapa, los resultados se pasan como entrada para el siguiente procesador en la canalización.

A continuación, empezaremos por explicar el concepto de las canalizaciones con más detalle. Despu s mostraremos un algoritmo num rico real, conocido como sobrerelajaci n sucesiva, para ilustrar las condiciones bajo las cuales se puede aplicar la canalizaci n, en la secci n 11.9.2. M s adelante, en la secci n 11.9.6, definimos de manera formal las restricciones que deben resolverse, y en la secci n 11.9.7 describimos un algoritmo para resolverlas. A los programas que tienen varias soluciones independientes para las restricciones de partici n de tiempo se les considera que sus ciclos m s externos son *ciclos completamente permutables*; dichos ciclos pueden canalizarse con facilidad, como vimos en la secci n 11.9.8.

11.9.1 ¿Qu  es la canalizaci n?

Nuestros primeros intentos por paralelizar los ciclos partieron las iteraciones de un anidamiento de ciclos, de tal forma que dos iteraciones que compart n datos se asignaban al mismo procesador. La canalizaci n permite a los procesadores compartir datos, pero por lo general s lo en forma “local”, en donde los datos se pasan de un procesador a otro que est  adyacente en el espacio de procesadores. He aqu  un ejemplo simple.

Ejemplo 11.55: Considere el siguiente ciclo:

```

for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++)
        X[i] = X[i] + Y[i,j];

```

Este c digo suma la i - sima fila de Y y la agrega al i - simo elemento de X . El ciclo interior, que corresponde a la suma, debe ejecutarse en forma secuencial debido a la dependencia de

Tiempo	Procesadores		
	1	2	3
1	$X[1] += Y[1,1]$		
2	$X[2] += Y[2,1]$	$X[1] += Y[1,2]$	
3	$X[3] += Y[3,1]$	$X[2] += Y[2,2]$	$X[1] += Y[1,3]$
4	$X[4] += Y[4,1]$	$X[3] += Y[3,2]$	$X[2] += Y[2,3]$
5		$X[4] += Y[4,2]$	$X[3] += Y[3,3]$
6			$X[4] += Y[4,3]$

Figura 11.49: Ejecución canalizada del ejemplo 11.55 con $m = 4$ y $n = 3$

datos;⁶ sin embargo, las distintas tareas de suma son independientes. Podemos paralelizar este código al hacer que cada procesador ejecute una suma por separado. El procesador i accede a la fila i de Y y actualiza el i -ésimo elemento de X .

De manera alternativa, podemos estructurar los procesadores para ejecutar la suma en una canalización, y derivar el paralelismo al traslapar la ejecución de las sumas, como se muestra en la figura 11.49. Dicho en forma más específica, cada iteración del ciclo interno puede tratarse como una etapa de una canalización: la etapa j toma un elemento de X que se generó en la etapa anterior, lo suma a un elemento de Y y pasa el resultado a la siguiente etapa. Observe que en este caso, cada procesador accede a una columna, en vez de una fila de Y . Si Y se almacena en formato ordenado por columnas, hay una ganancia en la localidad al particionar de acuerdo a las columnas, en vez de hacerlo en base a las filas.

Podemos iniciar una nueva tarea tan pronto como el primer procesador termina con la primera etapa de la tarea anterior. Al principio, la canalización está vacía y sólo el primer procesador está ejecutando la primera etapa. Una vez que termina, los resultados se pasan al segundo procesador, mientras que el primer procesador empieza con la segunda tarea, y así sucesivamente. De esta forma, la canalización se llena en forma gradual hasta que todos los procesadores están ocupados. Cuando el primer procesador termina con la última tarea, la canalización empieza a drenarse, y cada vez más procesadores se vuelven inactivos hasta que el último procesador termina la última tarea. En el estado estable, pueden ejecutarse n tareas en forma concurrente, en una canalización de n procesadores. \square

Es interesante contrastar la canalización con el paralelismo simple, en donde distintos procesadores ejecutan distintas tareas:

- La canalización sólo puede aplicarse a anidamientos con una profundidad de dos, por lo menos. Podemos tratar a cada iteración del ciclo externo como una tarea, y a las iteraciones en el ciclo interno como etapas de esa tarea.
- Las tareas ejecutadas sobre una canalización pueden compartir dependencias. La información que pertenece a la misma etapa de cada tarea se guarda en el mismo procesador; así, los resultados generados por la i -ésima etapa de una tarea pueden usarse por la i -ésima etapa de las tareas subsiguientes, sin que esto afecte a la comunicación. De manera similar,

⁶Recuerde que no aprovechamos la supuesta propiedad conmutativa y asociativa de la suma.

cada elemento de datos de entrada utilizado por una etapa individual de distintas tareas debe residir sólo en un procesador, como lo ilustra el ejemplo 11.55.

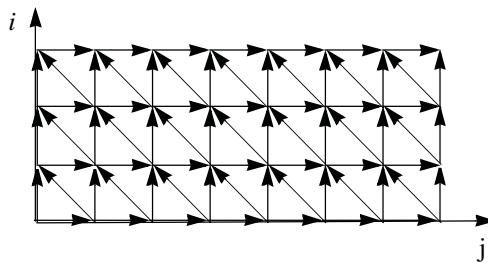
- Si las tareas son independientes, entonces la paralelización simple tiene un mejor uso de los procesadores, ya que éstos pueden ejecutar instrucciones al mismo tiempo, sin tener que pagar por la sobrecarga de llenar y vaciar la canalización. Sin embargo, como se muestra en el ejemplo 11.55, el patrón de accesos de datos en un esquema canalizado es distinto del de la paralelización simple. Puede ser preferible la canalización, si ésta reduce la comunicación.

11.9.2 Sobrerelajación sucesiva (Successive Over-Relaxation, SOR): un ejemplo

La *sobrerelajación sucesiva* (Successive Over-Relaxation, SOR) es una técnica para acelerar la convergencia de los métodos de relajación para resolver conjuntos de ecuaciones lineales simultáneas. En la figura 11.50(a) se muestra una plantilla bastante simple, que ilustra su patrón de acceso de datos. Aquí, el nuevo valor de un elemento en el arreglo depende de los valores de los elementos en su alrededor. Dicha operación se realiza repetidas veces, hasta cumplir con cierto criterio de convergencia.

```
for (i = 0; i <= m; i++)
    for (j = 0; j <= n; j++)
        X[j+1] = 1/3 * (X[j] + X[j+1] + X[j+2])
```

(a) Código fuente original.



(b) Dependencias de datos en el código.

Figura 11.50: Un ejemplo de sobrerelajación sucesiva (SOR)

En la figura 11.50(b) se muestra una imagen de las dependencias de datos clave. No mostramos las dependencias que pueden inferirse mediante las dependencias ya incluidas en la figura. Por ejemplo, la iteración $[i, j]$ depende de las iteraciones $[i, j - 1]$, $[i, j - 2]$, y así sucesivamente. De las dependencias queda claro que no hay paralelismo sin sincronización. Como la cadena más larga de dependencias consiste en $O(m + n)$ aristas, al introducir la sincronización debemos tener la capacidad de encontrar un grado de paralelismo y ejecutar las $O(mn)$ operaciones en un tiempo unitario representado por $O(m + n)$.

En especial, observamos que las iteraciones que se encuentran a lo largo de las diagonales ⁷ de 150° en la figura 11.50(b) no comparten dependencias. Sólo dependen de las iteraciones que se encuentran a lo largo de las diagonales cercanas al origen. Por lo tanto, podemos parallelizar este código al ejecutar las iteraciones sobre cada diagonal en orden, empezando en el origen y procediendo hacia fuera. A las iteraciones a lo largo de cada diagonal se les conoce como *frente de onda (wavefront)*, y a un esquema de parallelización de este tipo se le denomina *formación de frentes de ondas (wavefronting)*.

11.9.3 Ciclos completamente permutables

Primero introduciremos la noción de la *permutabilidad completa*, un concepto útil para la canalización y otras optimizaciones. Los ciclos son *completamente permutables* si pueden permutarse en forma arbitraria, sin cambiar la semántica del programa original. Una vez que los ciclos se colocan en una forma completamente permutable, podemos canalizar el código con facilidad y aplicar transformaciones como el uso de bloques, para mejorar la localidad de los datos.

El código de la SOR, como está escrito en la figura 11.50(a), no es completamente permutable. Como se muestra en la sección 11.7.8, la permutación de dos ciclos significa que las iteraciones en el espacio de iteraciones original se ejecutan columna por columna, en vez de hacerlo fila por fila. Por ejemplo, el cómputo original en la iteración [2, 3] se ejecutaría antes del de la iteración [1, 4], violando las dependencias que se muestran en la figura 11.50(b).

Sin embargo, podemos transformar el código para hacerlo completamente permutable. Al aplicar al código la siguiente transformación afín:

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

se produce el código que se muestra en la figura 11.51(a). Este código transformado es completamente permutable, y su versión permutada se muestra en la figura 11.51(c). También mostramos el espacio de iteraciones y las dependencias de datos de estos dos programas en las figuras 11.51(b) y (d), respectivamente. De la figura podemos ver con facilidad que este orden preserva el orden relativo entre cada par de accesos dependiente de datos.

Al permutar ciclos, modificamos en forma drástica el conjunto de las operaciones que se ejecutan en cada iteración del ciclo más externo. El hecho de tener este grado de libertad en la programación significa que hay mucho descuido en el orden de las operaciones en el programa. El descuido en la programación se traduce en oportunidades para la parallelización. Más adelante en esta sección mostraremos que si un ciclo tiene k ciclos externos completamente permutables, al introducir sólo $O(n)$ sincronizaciones podemos obtener $O(k - 1)$ grados de paralelismo (n es el número de iteraciones en un ciclo).

11.9.4 Canalización de ciclos completamente permutables

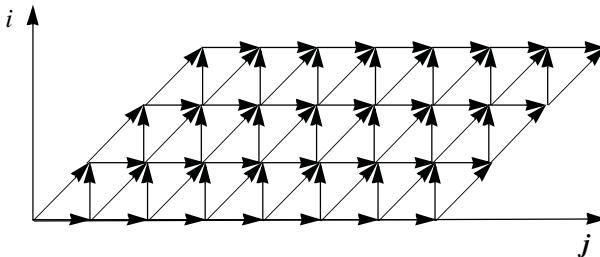
Un ciclo con k ciclos externos completamente permutables puede estructurarse como una canalización con $O(k - 1)$ dimensiones. En el ejemplo de la SOR, $k = 2$, por lo que podemos estructurar los procesadores como una canalización lineal.

⁷Es decir, las secuencias de puntos que se forman al moverse 1 hacia abajo y 2 a la derecha, en forma repetida.

```

for (i = 0; i <= m; i++)
    for (j = i; j <= i+n; j++)
        X[j-i+1] = 1/3 * (X[j-i] + X[j-i+1] + X[j-i+2])
    
```

(a) El código de la figura 11.50, transformado por $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$.

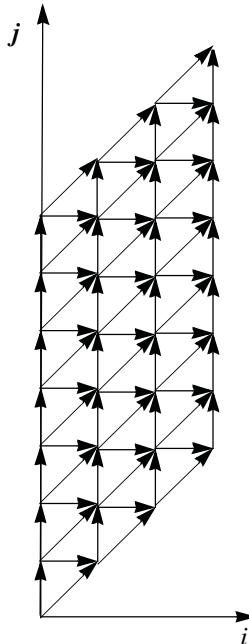


(b) Dependencias de datos del código en (a).

```

for (j = 0; j <= m+n; j++)
    for (i = max(0,j); i <= min(m,j), i++)
        X[j-i+1] = 1/3 * (X[j-i] + X[j-i+1] + X[j-i+2])
    
```

(c) Una permutación de los ciclos en (a).



(d) Dependencias de datos del código en (b).

Figura 11.51: Versión completamente permutable del código de la figura 11.50

Podemos canalizar el código de la SOR en dos formas distintas, las cuales se muestran en las figuras 11.52(a) y (b), y corresponden a las dos permutaciones posibles que se muestran en las figuras 11.51(a) y (c), respectivamente. En cada caso, cada columna del espacio de iteraciones constituye una tarea, y cada fila constituye una etapa. Asignamos la etapa i al procesador i , por lo cual cada procesador ejecuta el ciclo interno del código. Si ignoramos las condiciones delimitadoras, un procesador puede ejecutar la iteración i sólo hasta después de que el procesador $p - 1$ ha ejecutado la iteración $i - 1$.

```
/* 0 <= p <= m */
for (j = p; j <= p+n; j++) {
    if (p > 0) wait (p-1);
    X[j-p+1] = 1/3 * (X[j-p] + X[j-p+1] + X[j-p+2]);
    if (p < min (m,j)) signal (p+1);
}
```

(a) Procesadores asignados a las filas.

```
/* 0 <= p <= m+n */
for (i = max(0,p); i <= min(m,p); i++) {
    if (p > max(0,i)) wait (p-1);
    X[p-i+1] = 1/3 * (X[p-i] + X[p-i+1] + X[p-i+2])
    if (p < m+n) & (p > i) signal (p+1);
}
```

(b) Procesadores asignados a las columnas.

Figura 11.52: Dos implementaciones de canalización del código de la figura 11.51

Suponga que cada procesador requiere exactamente la misma cantidad de tiempo para ejecutar una iteración, y que la sincronización ocurre en forma instantánea. Ambos esquemas canalizados ejecutarían las mismas iteraciones en paralelo; la única diferencia es que tienen distintas asignaciones de procesadores. Todas las iteraciones ejecutadas en paralelo se encuentran a lo largo de las diagonales de 135° en el espacio de iteraciones de la figura 11.51(b), que corresponde a las diagonales de 150° en el espacio de iteraciones del código original; vea la figura 11.50(b).

No obstante, en la práctica los procesadores con cachés no siempre ejecutan el mismo código en la misma cantidad de tiempo, y el tiempo para la sincronización también varía. A diferencia del uso de barreras de sincronización, lo cual obliga a todos los procesadores a operar en pasos sincronizados, la canalización requiere que los procesadores se sincronicen y se comuniquen como máximo dos procesadores más. Por ende, la canalización tiene frentes de onda relajados, lo cual permite que ciertos procesadores se adelanten impulsivamente, al tiempo que otros se retrasen por unos momentos. Esta flexibilidad reduce el tiempo que invierten los procesadores esperando a los demás procesadores, y mejora el rendimiento en paralelo.

Los esquemas de canalización antes mostrados son sólo dos de las muchas formas en las que puede canalizarse el cómputo. Como dijimos antes, una vez que un ciclo es completamente

permutable, tenemos mucha libertad en la forma en la que queramos paralelizar el código. El primer esquema de canalización asigna la iteración $[i, j]$ al procesador i ; el segundo asigna la iteración $[i, j]$ al procesador j . Podemos crear canalizaciones alternativas al asignar la iteración $[i, j]$ al procesador $c_0 i + c_1 j$, siempre y cuando c_0 y c_1 sean constantes positivas. Dicho esquema crearía canalizaciones con frentes de ondas relajados entre 90° y 180° , ambos exclusivos.

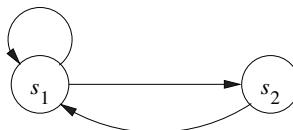
11.9.5 Teoría general

El ejemplo que acabamos de completar ilustra la siguiente teoría general de las canalizaciones: si podemos idear cuando menos dos ciclos más externos distintos para un anidamiento de ciclos y cumplir con todas las dependencias, entonces podemos canalizar el cómputo. Un ciclo con k ciclos más externos, completamente permutables, tiene $k - 1$ grados de paralelismo canalizado.

Los ciclos que no pueden canalizarse no tienen ciclos más externos alternativos. El ejemplo 11.56 muestra una instancia de ese tipo. Para respetar todas las dependencias, cada iteración en el ciclo más externo debe ejecutar precisamente el cómputo que se encuentra en el código original. Sin embargo, dicho código puede contener aún paralelismo en los ciclos internos, lo cual puede explotarse mediante la introducción de por lo menos n sincronizaciones, en donde n es el número de iteraciones en el ciclo más externo.

```
for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++)
        X[j] = X[j] + Y[i,j];    /* (s1) */
        Z[i] = X[A[i]];          /* (s2) */
}
```

(a)



(b)

Figura 11.53: Un ciclo externo secuencial (a) y su PDG (b)

Ejemplo 11.56: La figura 11.53 es una versión más compleja del problema que vimos en el ejemplo 11.50. Como se muestra en el grafo de dependencias del programa en la figura 11.53(b), las instrucciones s_1 y s_2 pertenecen al mismo componente fuertemente conectado. Como no conocemos el contenido de la matriz A , debemos suponer que el acceso en la instrucción s_2 puede leer de cualquiera de los elementos de X . Hay una dependencia verdadera de la instrucción s_1 a la instrucción s_2 , y una antidependencia de la instrucción s_2 a la instrucción s_1 .

No hay oportunidad para la canalización en ninguno de los casos, ya que todas las operaciones que pertenecen a la iteración i en el ciclo externo deben preceder a las que están en la iteración $i + 1$. Para encontrar más paralelismo, repetimos el proceso de paralelización en el ciclo interno. Las iteraciones en el segundo ciclo pueden parallelizarse sin sincronización. Por lo tanto, se requieren 200 barreras, con una antes y una después de cada ejecución del ciclo interno. \square

11.9.6 Restricciones de partición de tiempo

Ahora nos enfocaremos en el problema de buscar paralelismo canalizado. Nuestro objetivo es convertir un cómputo en un conjunto de tareas canalizables. Para buscar paralelismo canalizado, no resolvemos directamente lo que se va a ejecutar en cada procesador, como hicimos con la paralelización de ciclos. En vez de ello, hacemos la siguiente pregunta fundamental: ¿Cuáles son todas las posibles secuencias de ejecución que respetan a las dependencias de datos originales en el ciclo? Es obvio que la secuencia de ejecución original satisface todas las dependencias de datos. La pregunta es si hay transformaciones afines que puedan crear un programa alternativo, en donde las iteraciones del ciclo más externo ejecuten un conjunto distinto de operaciones del programa original, y aún así se cumpla con todas las dependencias. Si podemos encontrar dichas transformaciones, podemos canalizar el ciclo. El punto clave es que si hay libertad en la programación de las operaciones, hay paralelismo; más adelante explicaremos los detalles acerca de cómo derivar el paralelismo canalizado a partir de dichas transformaciones.

Para encontrar reordenamientos aceptables del ciclo externo, es conveniente encontrar transformaciones afines unidimensionales, una para cada instrucción, que asigan los valores índice de ciclo originales a un número de iteración en el ciclo más externo. Las transformaciones son válidas si la asignación puede cumplir con todas las dependencias de datos en el programa. Las “restricciones de partición de tiempo”, que se muestran a continuación, sólo indican que si una operación es dependiente de la otra, entonces a la primera se le debe asignar una iteración en el ciclo más externo, no antes que la de la segunda. Si se les asigna la misma iteración, entonces se comprende que la primera se ejecutará después de la segunda dentro de la iteración.

Una asignación de particiones afines de un programa es una *partición de tiempo válida* si, y sólo si para cada dos accesos (no necesariamente distintos) que comparten una dependencia, por decir:

$$\mathcal{F}_1 = \langle \mathbf{F}_1, \mathbf{f}_1, \mathbf{B}_1, \mathbf{b}_1 \rangle$$

en la instrucción s_1 , que está anidada en d_1 ciclos, y:

$$\mathcal{F}_2 = \langle \mathbf{F}_2, \mathbf{f}_2, \mathbf{B}_2, \mathbf{b}_2 \rangle$$

en la instrucción s_2 , anidada en d_2 ciclos, las asignaciones de particiones unidimensionales $\langle \mathbf{C}_1, \mathbf{c}_1 \rangle$ y $\langle \mathbf{C}_2, \mathbf{c}_2 \rangle$ para las instrucciones s_1 y s_2 , respectivamente, cumplen con las siguientes *restricciones de partición de tiempo*:

- Para todas las \mathbf{i}_1 en Z^{d_1} y todas las \mathbf{i}_2 en Z^{d_2} tales que:

- $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$,

- b) $\mathbf{B}_1\mathbf{i}_1 + \mathbf{b}_1 \geq 0,$
 c) $\mathbf{B}_2\mathbf{i}_2 + \mathbf{b}_2 \geq 0, \text{ y}$
 d) $\mathbf{F}_1\mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2\mathbf{i}_2 + \mathbf{f}_2,$

se da el caso en el que $\mathbf{C}_1\mathbf{i}_1 + \mathbf{c}_1 \leq \mathbf{C}_2\mathbf{i}_2 + \mathbf{c}_2$.

Esta restricción, que se ilustra en la figura 11.54, es bastante parecida a las restricciones de partición de espacio. Es una relajación de las restricciones de partición de espacio, en cuanto a que si dos iteraciones se refieren a la misma ubicación, no necesariamente tienen que asignarse a la misma partición; sólo requerimos que se preserve el orden de ejecución relativo original entre las dos iteraciones. Es decir, las restricciones aquí tienen \leq en donde las restricciones de partición de espacio tienen $=$.

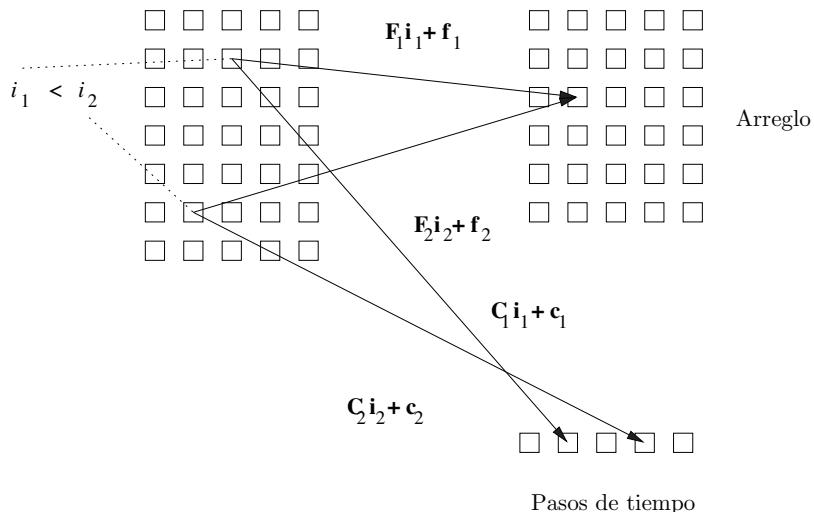


Figura 11.54: Restricciones de partición de tiempo

Sabemos que existe por lo menos una solución a las restricciones de partición de tiempo. Podemos asignar las operaciones en cada iteración del ciclo más externo de vuelta a la misma iteración, y se cumplirán todas las dependencias de datos. Esta solución es la única para las restricciones de partición de tiempo, para los programas que no pueden canalizarse. Por otro lado, si podemos encontrar varias soluciones independientes a las restricciones de partición de tiempo, el programa puede canalizarse. Cada solución independiente corresponde a un ciclo en el anidamiento más externo completamente permutable. Como es lógico, sólo hay una solución independiente para las restricciones de sincronización extraídas del programa del ejemplo 11.56, en donde no hay paralelismo canalizado, y que hay dos soluciones independientes para el ejemplo del código de la SOR.

Ejemplo 11.57: Vamos a considerar el ejemplo 11.56, y en especial las dependencias de datos de las referencias al arreglo X en las instrucciones s_1 y s_2 . Como el acceso no es afín en la ins-

trucción s_2 , para aproximar el acceso modelamos la matriz X simplemente como una variable escalar en el análisis de dependencias que involucra a la instrucción s_2 . Hagamos que (i, j) sea el valor índice de una instancia dinámica de s_1 y hagamos que i' sea el valor índice de una instancia dinámica de s_2 . Hagamos que las asignaciones de cómputo de las instrucciones s_1 y s_2 sean $\langle [C_{11}, C_{12}], c_1 \rangle$ y $\langle [C_{21}], c_2 \rangle$, respectivamente.

Primero vamos a considerar las restricciones de partición de tiempo impuestas por las dependencias de la instrucción s_1 a s_2 . Por ende, $i \leq i'$, la (i, j) -ésima iteración transformada de s_1 no debe ser posterior a la i' -ésima iteración transformada de s_2 ; es decir,

$$[C_{11} \ C_{12}] \begin{bmatrix} i \\ j \end{bmatrix} + c_1 \leq C_{21}i' + c_2.$$

Al expandir, obtenemos lo siguiente:

$$C_{11}i + C_{12}j + c_1 \leq C_{21}i' + c_2.$$

Como j puede ser arbitrariamente grande, independiente de i e i' , debe ser que $C_{12} = 0$. Por ende, una posible solución a las restricciones es:

$$C_{11} = C_{21} = 1 \quad \text{y} \quad C_{12} = c_1 = c_2 = 0.$$

Los argumentos similares acerca de la dependencia de datos de s_2 a s_1 y de s_2 de vuelta a sí misma producirán una respuesta similar. En esta solución específica, la i -ésima iteración del ciclo externo, que consiste en la instancia i de s_2 y en todas las instancias (i, j) de s_1 , se asignan todas al salto temporal i . Otras opciones válidas de C_{11} , C_{21} , c_1 y c_2 producen asignaciones similares, aunque podría haber saltos temporales en los que no ocurra nada. Es decir, todas las formas de programar el ciclo externo requieren que las iteraciones se ejecuten en el mismo orden que en el código original. Esta instrucción es válida ya sea si todas las 100 iteraciones se ejecutan en el mismo procesador, en 100 procesadores distintos, o en cualquier cosa intermedia. \square

Ejemplo 11.58: En el código de la SOR que se muestra en la figura 11.50(a), la referencia de escritura $X[j + 1]$ comparte una dependencia consigo misma y con las tres referencias de lectura en el código. Estamos buscando la asignación del cómputo $\langle [C_1, C_2], c \rangle$ para la instrucción de asignación tal que:

$$[C_1 \ C_2] \begin{bmatrix} i \\ j \end{bmatrix} + [c] \leq [C_1 \ C_2] \begin{bmatrix} i' \\ j' \end{bmatrix} + [c]$$

si hay una dependencia de (i, j) a (i', j') . Por definición, $(i, j) \prec (i', j')$; es decir, ya sea que $i < i'$ o $(i = i' \wedge j < j')$.

Vamos a considerar tres de los pares de dependencias de datos:

1. La dependencia verdadera del acceso de escritura $X[j + 1]$ al acceso de lectura $X[j + 2]$. Como las instancias deben acceder a la misma ubicación, $j + 1 = j' + 2$ o $j = j' + 1$. Al sustituir $j = j' + 1$ en las restricciones de sincronización, obtenemos:

$$C_1(i' - i) - C_2 \geq 0.$$

Como $j = j' + 1$, $j > j'$, las restricciones de precedencia se reducen a $i < i'$. Por lo tanto,

$$C_1 - C_2 \geq 0.$$

2. La antidependencia del acceso de lectura $X[j+2]$ al acceso de escritura $X[j+1]$. Aquí, $j+2 = j'+1$, o $j = j'-1$. Al sustituir $j = j'-1$ en las restricciones de sincronización, obtenemos:

$$C_1(i' - i) + C_2 \geq 0.$$

Cuando $i = i'$, obtenemos:

$$C_2 \geq 0.$$

Cuando $i < i'$, ya que $C_2 \geq 0$, obtenemos:

$$C_1 \geq 0.$$

3. La dependencia de salida del acceso $X[j+1]$ de vuelta a sí mismo. Aquí $j = j'$. Las restricciones de sincronización se reducen a:

$$C_1(i' - i) \geq 0.$$

Como sólo $i < i'$ es relevante, obtenemos de nuevo:

$$C_1 \geq 0.$$

El resto de las dependencias no producen nuevas restricciones. En total, hay tres restricciones:

$$\begin{aligned} C_1 &\geq 0 \\ C_2 &\geq 0 \\ C_1 - C_2 &\geq 0 \end{aligned}$$

He aquí dos soluciones independientes para estas restricciones:

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

La primera solución preserva el orden de ejecución de las iteraciones en el ciclo más externo. Tanto el código de la SOR original en la figura 11.50(a) como el código transformado que se muestra en la figura 11.51(a) son ejemplos de dicho arreglo. La segunda solución coloca las iteraciones a lo largo de las diagonales de 135° en el mismo ciclo externo. El código que se muestra en la figura 11.51(b) es un ejemplo de un código con esa composición de ciclo más externo.

Observe que existen muchos otros posibles pares de soluciones independientes. Por ejemplo,

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

también serían soluciones independientes para las mismas restricciones. Elegimos los vectores más simples para simplificar la transformación de código. \square

11.9.7 Resolución de restricciones de partición de tiempo mediante el Lema de Farkas

Como las restricciones de partición de tiempo son similares a las restricciones de partición de espacio, ¿podemos usar un algoritmo similar para resolverlas? Por desgracia, la ligera diferencia entre los dos problemas se traduce en una gran diferencia técnica entre los dos métodos de solución. El Algoritmo 11.43 simplemente resuelve para \mathbf{C}_1 , \mathbf{c}_1 , \mathbf{C}_2 y \mathbf{c}_2 , de tal forma que para todas las \mathbf{i}_1 en Z^{d_1} y todas las \mathbf{i}_2 en Z^{d_2} , si:

$$\mathbf{F}_1\mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2\mathbf{i}_2 + \mathbf{f}_2$$

entonces:

$$\mathbf{C}_1\mathbf{i}_1 + \mathbf{c}_1 = \mathbf{C}_2\mathbf{i}_2 + \mathbf{c}_2.$$

Las desigualdades lineales debido a los límites de ciclo se utilizan sólo para determinar si dos referencias comparten una dependencia de datos, y no se utilizan en ningún otro caso.

Para buscar soluciones a las restricciones de partición de tiempo, no podemos ignorar las desigualdades lineales $\mathbf{i} \prec \mathbf{i}'$; por lo general, ignorarlas sólo permitiría la solución trivial de colocar todas las iteraciones en la misma partición. Por ende, el algoritmo para encontrar soluciones a las restricciones de partición de tiempo debe manejar tanto las igualdades como las desigualdades.

El problema general que deseamos resolver es: dada una matriz \mathbf{A} , encontrar un vector \mathbf{c} de manera que, para todos los vectores \mathbf{x} tales que $\mathbf{Ax} \geq \mathbf{0}$, se dé el caso de que $\mathbf{c}^T \mathbf{x} \geq 0$. En otras palabras, estamos buscando \mathbf{c} de tal forma que el producto interno de \mathbf{c} y cualquier coordenada en el poliedro definida por las desigualdades $\mathbf{Ax} \geq \mathbf{0}$ siempre produzca una respuesta no negativa.

Para este problema utilizamos el *Lema de Farkas*. Hagamos que \mathbf{A} sea una matriz $m \times n$ de números reales, y que \mathbf{c} sea un n -vector real, distinto de cero. El lema de Farkas establece que el sistema *fundamental* de desigualdades:

$$\mathbf{Ax} \geq \mathbf{0}, \quad \mathbf{c}^T \mathbf{x} < 0$$

tiene una solución \mathbf{x} con valor real, o que el sistema *dual*:

$$\mathbf{A}^T \mathbf{y} = \mathbf{c}, \quad \mathbf{y} \geq \mathbf{0}$$

tiene una solución \mathbf{y} con valor real, pero nunca ambos.

El sistema dual puede manejarse mediante el uso de la eliminación de Fourier-Motzkin para proyectar a lo lejos las variables de \mathbf{y} . Para cada \mathbf{c} que tiene una solución en el sistema dual, el lema garantiza que no hay soluciones para el sistema fundamental. Dicho de otra forma, podemos probar la negación del sistema fundamental; es decir, podemos demostrar que $\mathbf{c}^T \mathbf{x} \geq 0$ para todas las \mathbf{x} tales que $\mathbf{Ax} \geq \mathbf{0}$, encontrando una solución \mathbf{y} al sistema dual: $\mathbf{A}^T \mathbf{y} = \mathbf{c}$ y $\mathbf{y} \geq 0$.

Algoritmo 11.59: Buscar un conjunto de asignaciones válidas de partición de tiempo afines, muy independientes, para un ciclo secuencial externo.

Acerca del Lema de Farkas

La prueba del lema puede encontrarse en muchos libros de texto estándar sobre programación lineal. El Lema de Farkas, que se demostró por primera vez en 1901, es uno de los *teoremas de la alternativa*. Estos teoremas son todos equivalentes, pero a pesar de los intentos a través de los años, no se ha encontrado una prueba simple e intuitiva de este lema, o de alguno de sus equivalentes.

ENTRADA: Un anidamiento de ciclos con accesos a arreglos.

SALIDA: Un conjunto máximo de asignaciones de partición de tiempo linealmente independientes.

MÉTODO: Los siguientes pasos constituyen el algoritmo:

1. Buscar todos los pares dependientes de datos de accesos en un programa.
2. Para cada par de accesos dependientes de datos, $\mathcal{F}_1 = \langle \mathbf{F}_1, \mathbf{f}_1, \mathbf{B}_1, \mathbf{b}_1 \rangle$ en la instrucción s_1 , anidados en d_1 ciclos, y $\mathcal{F}_2 = \langle \mathbf{F}_2, \mathbf{f}_2, \mathbf{B}_2, \mathbf{b}_2 \rangle$ en la instrucción s_2 , anidados en d_2 ciclos, hagamos que $\langle \mathbf{C}_1, \mathbf{c}_1 \rangle$ y $\langle \mathbf{C}_2, \mathbf{c}_2 \rangle$ sean las asignaciones de partición de tiempo (desconocidas) de las instrucciones s_1 y s_2 , respectivamente. Recuerde que las restricciones de partición de tiempo establecen que:

- Para todas las \mathbf{i}_1 en Z^{d_1} y todas las \mathbf{i}_2 en Z^{d_2} tales que:
 - $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$,
 - $\mathbf{B}_1 \mathbf{i}_1 + \mathbf{b}_1 \geq 0$,
 - $\mathbf{B}_2 \mathbf{i}_2 + \mathbf{b}_2 \geq 0$, y
 - $\mathbf{F}_1 \mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2 \mathbf{i}_2 + \mathbf{f}_2$,

se da el caso de que $\mathbf{C}_1 \mathbf{i}_1 + \mathbf{c}_1 \leq \mathbf{C}_2 \mathbf{i}_2 + \mathbf{c}_2$.

Como $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$ es una unión disyuntiva de un número de cláusulas, podemos crear un sistema de restricciones para cada cláusula y resolver cada una de ellas por separado, como se muestra a continuación:

- (a) De manera similar al paso (2a) en el Algoritmo 11.43, se aplica la eliminación gaussiana a las ecuaciones:

$$\mathbf{F}_1 \mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2 \mathbf{i}_2 + \mathbf{f}_2$$

para reducir el vector:

$$\begin{bmatrix} \mathbf{i}_1 \\ \mathbf{i}_2 \\ 1 \end{bmatrix}$$

a cierto vector de variables desconocidas, \mathbf{x} .

- (b) Hacemos que \mathbf{c} sea todas las variables desconocidas en las asignaciones de partición. Se expresan las restricciones de desigualdad lineales debido a las asignaciones de partición como:

$$\mathbf{c}^T \mathbf{D} \mathbf{x} \geq \mathbf{0}$$

para cierta matriz \mathbf{D} .

- (c) Se expresan las restricciones de precedencia en las variables índice de ciclo y en los límites de ciclo como:

$$\mathbf{A} \mathbf{x} \geq \mathbf{0}$$

para cierta matriz \mathbf{A} .

- (d) Se aplica el Lema de Farkas. Buscar \mathbf{x} para satisfacer las dos restricciones anteriores es equivalente a buscar \mathbf{y} de tal forma que:

$$\mathbf{A}^T \mathbf{y} = \mathbf{D}^T \mathbf{c} \quad \text{y} \quad \mathbf{y} \geq \mathbf{0}.$$

Observe que $\mathbf{c}^T \mathbf{D}$ aquí es \mathbf{c}^T en la instrucción del Lema de Farkas, y estamos usando la forma negada del lema.

- (e) De esta forma, se aplica la eliminación de Fourier-Motzkin para proyectar a lo lejos las variables \mathbf{y} , y se expresan las restricciones en los coeficientes \mathbf{c} como $\mathbf{E} \mathbf{c} \geq \mathbf{0}$.
- (f) Hacemos que $\mathbf{E}' \mathbf{c}' \geq \mathbf{0}$ sea el sistema sin los términos constantes.
3. Buscar un conjunto máximo de soluciones linealmente independientes a $\mathbf{E}' \mathbf{c}' \geq \mathbf{0}$, usando el Algoritmo B.1 del apéndice B. El método de ese algoritmo complejo es llevar la cuenta del conjunto actual de soluciones para cada una de las instrucciones, y después buscar cada vez más soluciones independientes, insertando restricciones que obliguen a que la solución sea linealmente independiente durante por lo menos una instrucción.
4. De cada solución de \mathbf{c}' que se haya encontrado, derivar una asignación de partición de tiempo afín. Los términos constantes se derivan usando $\mathbf{E} \mathbf{c} \geq \mathbf{0}$.

□

Ejemplo 11.60: Las restricciones para el ejemplo 11.57 pueden escribirse así:

$$\left[\begin{array}{cccc} -C_{11} & -C_{12} & C_{21} & (c_2 - c_1) \end{array} \right] \begin{bmatrix} i \\ j \\ i' \\ 1 \end{bmatrix} \geq \mathbf{0}$$

$$\left[\begin{array}{cccc} -1 & 0 & 1 & 0 \end{array} \right] \begin{bmatrix} i \\ j \\ i' \\ 1 \end{bmatrix} \geq \mathbf{0}$$

El lema de Farkas establece que estas restricciones son equivalentes a:

$$\begin{bmatrix} -1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} z \end{bmatrix} = \begin{bmatrix} -C_{11} \\ -C_{12} \\ C_{21} \\ c_2 - c_1 \end{bmatrix} \text{ y } z \geq 0.$$

Al resolver este sistema, obtenemos:

$$C_{11} = C_{21} \geq 0 \text{ y } C_{12} = c_2 - c_1 = 0.$$

Observe que estas restricciones se satisfacen mediante la solución específica que obtuvimos en el ejemplo 11.57. \square

11.9.8 Transformaciones de código

Si existen k soluciones independientes a las restricciones de partición de tiempo de un anidamiento de ciclos, entonces es posible transformar el anidamiento de ciclos para que tenga k ciclos más externos completamente permutables, los cuales pueden transformarse para crear $k - 1$ grados de canalización, o para crear $k - 1$ ciclos internos paralelizables. Además. Podemos aplicar el uso de bloques a los ciclos completamente permutables para mejorar la localidad de datos de los uniprocesadores, así como para reducir la sincronización entre los procesadores, en una ejecución en paralelo.

Explotación de ciclos completamente permutables

Podemos crear fácilmente un anidamiento de ciclos con k ciclos más externos completamente permutables, a partir de k soluciones independientes a las restricciones de partición de tiempo. Para ello, sólo debemos hacer que la k -ésima solución sea la k -ésima fila de la nueva transformación. Una vez que se crea la transformación afín, podemos usar el Algoritmo 11.45 para generar el código.

Ejemplo 11.61: Las soluciones encontradas en el ejemplo 11.58 para nuestro ejemplo de la SOR fueron:

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Si hacemos que la primera solución sea la primera fila y que la segunda solución sea la segunda fila, obtenemos la siguiente transformación:

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

lo cual produce el código de la figura 11.51(a).

Si en vez de eso hacemos que la segunda solución sea la primera fila, obtenemos la siguiente transformación:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

lo cual produce el código de la figura 11.51(c). \square

Es fácil ver que dichas transformaciones producen un programa secuencial válido. La primera fila partitiona todo el espacio de iteraciones de acuerdo con la primera solución. Las restricciones de sincronización garantizan que dicha descomposición no viole ninguna dependencia de datos. Después, partitionamos las iteraciones en cada uno de los ciclos más externos, de acuerdo con la segunda solución. De nuevo, esto debe ser válido, ya que estamos tratando sólo con subconjuntos del espacio de iteraciones original. Lo mismo se aplica para el resto de las filas en la matriz. Como podemos ordenar las soluciones en forma arbitraria, los ciclos son completamente permutables.

Explotación de las canalizaciones

Podemos transformar con facilidad un ciclo con k ciclos más externos completamente permutables, en un código con $k - 1$ grados de paralelismo de canalización.

Ejemplo 11.62: Vamos a regresar a nuestro ejemplo de la SOR. Una vez que los ciclos se transforman para ser completamente permutables, sabemos que la iteración $[i_1, i_2]$ puede ejecutarse, siempre y cuando se hayan ejecutado las iteraciones $[i_1, i_2 - 1]$ y $[i_1 - 1, i_2]$. Podemos garantizar este orden en una canalización de la siguiente manera. Asignamos la iteración i_1 al procesador p_1 . Cada procesador ejecuta iteraciones en el ciclo interno, en el orden secuencial original, con lo cual se garantiza que la iteración $[i_1, i_2]$ se ejecute después de $[i_1, i_2 - 1]$. Además, es necesario que el procesador p espere la señal del procesador $p - 1$ de que ha ejecutado la iteración $[p - 1, i_2]$ antes de que ejecute la iteración $[p, i_2]$. Esta técnica genera el código canalizado de las figuras 11.52(a) y (b), a partir de los ciclos completamente permutables de las figuras 11.51(a) y (c), respectivamente. \square

En general, dados k ciclos más externos completamente permutables, la iteración con los valores índice (i_1, \dots, i_k) pueden ejecutarse sin violar las restricciones de dependencia de datos, siempre y cuando se hayan ejecutado las siguientes iteraciones:

$$[i_1 - 1, i_2, \dots, i_k], [i_1, i_2 - 1, i_3, \dots, i_k], \dots, [i_1, \dots, i_{k-1}, i_k - 1]$$

Por lo tanto, podemos asignar las particiones de las primeras $k - 1$ dimensiones del espacio de iteraciones a $O(n^{k-1})$ procesadores, de la siguiente manera. Cada procesador es responsable de un conjunto de iteraciones cuyos índices concuerdan en las primeras $k - 1$ dimensiones, y varían sobre todos los valores del k -ésimo índice. Cada procesador ejecuta las iteraciones en el k -ésimo ciclo de manera secuencial. El procesador correspondiente a los valores $[p_1, p_2, \dots, p_{k-1}]$ para los primeros $k - 1$ índices de ciclo puede ejecutar la iteración i en el k -ésimo ciclo, siempre y cuando reciba una señal de los procesadores:

$$[p_1 - 1, p_2, \dots, p_{k-1}], \dots, [p_1, \dots, p_{k-2}, p_{k-1} - 1]$$

de que han ejecutado su i -ésima iteración en el k -ésimo ciclo.

Frentes de onda

También es fácil generar $k - 1$ ciclos internos paralelizables a partir de un ciclo con k ciclos más externos completamente permutables. Aunque es preferible la canalización, incluimos aquí esta información para tener una idea más completa.

Particionamos el cómputo de un ciclo con k ciclos más externos completamente permutables mediante el uso de una nueva variable índice i' , en donde i' se define como cierta combinación de todos los índices en el anidamiento de ciclos permutable k . Por ejemplo, $i' = i_1 + \dots + i_k$ es una de esas combinaciones.

Creamos un ciclo secuencial más externo que itere a través de las i' particiones en orden incremental; el cómputo anidado dentro de cada partición se ordena como antes. Se garantiza que los primeros $k - 1$ ciclos dentro de cada partición son paralelizables. Por intuición, si se proporciona un espacio de iteraciones bidimensional, esta transformación agrupa las iteraciones a lo largo de diagonales de 135° como una ejecución del ciclo más externo. Esta estrategia garantiza que las iteraciones dentro de cada iteración del ciclo más externo no tengan dependencia de datos.

Uso de bloques

Un ciclo con k niveles, completamente permutable, puede dividirse en bloques de k dimensiones. En vez de asignar las iteraciones a los procesadores con base en el valor de los índices externo o interno, podemos juntar bloques de iteraciones en una unidad. Los bloques son útiles para mejorar la localidad de los datos, así como para minimizar la sobrecarga de la canalización.

Suponga que tenemos un anidamiento de ciclos bidimensional, completamente permutable, como en la figura 11.55(a), y deseamos descomponer el cómputo en $b \times b$ bloques. El orden de ejecución del código en bloques se muestra en la figura 11.56, y el código equivalente está en la figura 11.55(b).

Si asignamos cada bloque a un procesador, entonces toda la acción de pasar datos de una iteración a otra que esté dentro de un bloque no requiere comunicación entre los procesadores. De manera alternativa, podemos aumentar la granularidad de la canalización al asignar una columna de bloques a un procesador. Observe que cada procesador se sincroniza con sus predecesores y sucesores sólo en los límites de los bloques. Así, otra ventaja del uso de bloques es que los programas sólo tienen que comunicar los datos a los que acceden en los límites del bloque con sus bloques vecinos. Los valores interiores para un bloque los maneja un solo procesador.

Ejemplo 11.63: Ahora vamos a usar un algoritmo numérico real (descomposición de Cholesky) para ilustrar cómo el Algoritmo 11.59 maneja los anidamientos de ciclos individuales sólo con paralelismo de canalización. El código, que se muestra en la figura 11.57, implementa a un algoritmo $O(n^3)$, que opera sobre un arreglo de datos bidimensional. El espacio de iteraciones ejecutado es una pirámide triangular, ya que j sólo itera hasta el valor del índice i del ciclo interno, y k sólo itera hasta el valor de j . El ciclo tiene cuatro instrucciones, todas anidadas en distintos ciclos.

Al aplicar el Algoritmo 11.59 a este programa encontramos tres dimensiones de tiempo válidas. Anida todas las operaciones, algunas de las cuales estaban anidadas originalmente en anidamientos de ciclos de 1 y 2 niveles, en un anidamiento de ciclos tridimensional, completamente permutable. En la figura 11.58 se muestra el código, junto con las asignaciones.

La rutina de generación de código protege la ejecución de las operaciones con los límites de ciclo originales, para asegurar que los nuevos programas sólo ejecuten operaciones que se encuentren en el código original. Podemos canalizar el código asignando la estructura tridimensional a un espacio de procesadores bidimensional. Las iteraciones $(i2, j2, k2)$ se asignan

```
for (i=0; i<n; i++)
  for (j=1; j<n; j++) {
    <S>
  }
```

(a) Un anidamiento de ciclos simple.

```
for (ii = 0; ii<n; ii+=b)
  for (jj = 0; jj<n; jj+=b)
    for (i = ii*b; i <= min(ii*b-1, n); i++)
      for (j = ii*b; j <= min(jj*b-1, n); j++) {
        <S>
      }
```

(b) Una versión en bloques de este anidamiento de ciclos.

Figura 11.55: Un anidamiento de ciclos bidimensional y su versión en bloques

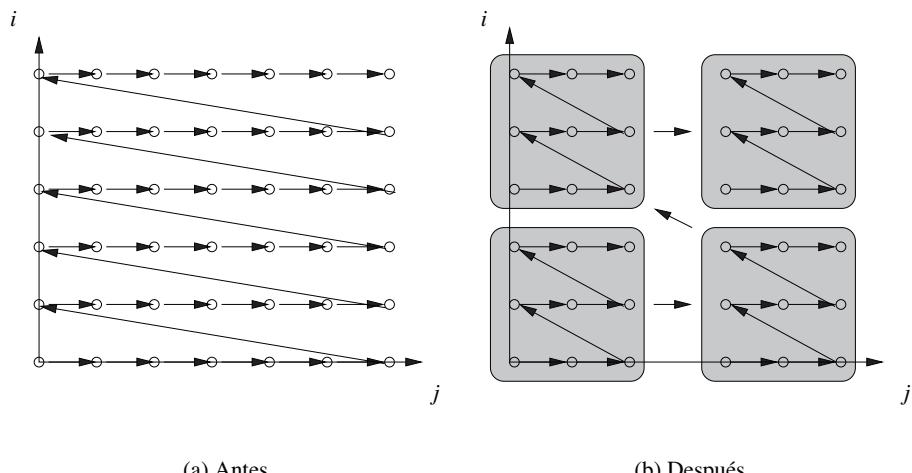


Figura 11.56: Orden de ejecución, antes y después de usar bloques en un anidamiento de ciclos de 2 niveles

```

for (i = 1; i <= N; i++) {
    for (j = 1; j <= i-1; j++) {
        for (k = 1; k <= j-1; k++)
            X[i,j] = X[i,j] - X[i,k] * X[j,k];
        X[i,j] = X[i,j] / X[j,j];
    }
    for (m = 1; m <= i-1; m++)
        X[i,i] = X[i,i] - X[i,m] * X[i,m];
    X[i,i] = sqrt(X[i,i]);
}

```

Figura 11.57: Descomposición de Cholesky

```

for (i2 = 1; i2 <= N; i2++)
    for (j2 = 1; j2 <= i2; j2++) {
        /* inicio del codigo para el procesador (i2,j2u */
        for (k2 = 1; k2 <= i2; k2++) {

            // Asignacion: i2 = i, j2 = j, k2 = k
            if (j2<i2 && k2<j2)
                X[i2,j2] = X[i2,j2] - X[i2,k2] * X[j2,k2];

            // Asignacion: i2 = i, j2 = j, k2 = j
            if (j2==k2 && j2<i2)
                X[i2,j2] = X[i2,j2] / X[j2,j2];

            // Asignacion: i2 = i, j2 = i, k2 = m
            if (i2==j2 && k2<i2)
                X[i2,i2] = X[i2,i2] - X[i2,k2] * X[i2,k2];

            // Asignacion: i2 = i, j2 = i, k2 = i
            if (i2==j2 && j2==k2)
                X[k2,k2] = sqrt(X[k2,k2]);
        }
        /* fin del codigo para el procesador (i2,j2u */
    }
}

```

Figura 11.58: La figura 11.57 escrita como un anidamiento de ciclos completamente permutable

al procesador con el ID $(i2, j2)$. Cada procesador ejecuta el ciclo más interno, el ciclo con el índice $k2$. Antes de ejecutar la k -ésima iteración, el procesador espera las señales de los procesadores con los IDs $(i2 - 1, j2)$ y $(i2, j2 - 1)$. Después de ejecutar su iteración, envía una señal a los procesadores $(i2 + 1, j2)$ y $(i2, j2 + 1)$. \square

11.9.9 Paralelismo con sincronización mínima

En las últimas tres secciones, hemos descrito tres poderosos algoritmos de paralelización: el Algoritmo 11.43 encuentra todo el paralelismo que no requiera sincronizaciones, el Algoritmo 11.54 encuentra todo el paralelismo que requiera sólo un número constante de sincronizaciones, y el Algoritmo 11.59 encuentra todo el paralelismo canalizable que requiera $O(n)$ sincronizaciones, en donde n es el número de iteraciones en el ciclo más externo. Como primera aproximación, nuestra meta es paralelizar todo el cómputo que sea posible, e introducir al mismo tiempo la menor cantidad de sincronización que sea necesaria.

El Algoritmo 11.64, que se muestra a continuación, encuentra todos los grados de paralelismo en un programa, empezando con la granularidad más grande del paralelismo. En la práctica, para paralelizar un código para un multiprocesador, no es necesario explotar todos los niveles de paralelismo, sólo los más externos que sea posible hasta paralelizar todo el cómputo, y hasta que se utilicen todos los procesadores por completo.

Algoritmo 11.64: Encontrar todos los grados de paralelismo en un programa, en donde todo el paralelismo debe tener la mayor granularidad posible.

ENTRADA: Un programa que se va a paralelizar.

SALIDA: Una versión paralelizada del mismo programa.

MÉTODO: Haga lo siguiente:

1. Encuentre el máximo grado de paralelismo que no requiera sincronización: aplique el Algoritmo 11.43 al programa.
2. Encuentre el máximo grado de paralelismo que requiere $O(1)$ sincronizaciones: aplique el Algoritmo 11.54 a cada una de las particiones de espacio descubiertas en el paso 1. Si no se encuentra ningún paralelismo sin sincronización, el cómputo completo se deja en una partición.
3. Encuentre el máximo grado de paralelismo que requiera $O(n)$ sincronizaciones. Aplique el Algoritmo 11.59 a cada una de las particiones descubiertas en el paso 2, para encontrar el paralelismo canalizado. Después aplique el Algoritmo 11.54 a cada una de las particiones asignadas a cada procesador, o al cuerpo del ciclo secuencial si no se encuentra canalización.
4. Encuentre el máximo grado de paralelismo con grados de sincronizaciones cada vez mayores: aplique en forma recursiva el paso 3 al cómputo que pertenezca a cada una de las particiones de espacio generadas por el mapa anterior. \square

Ejemplo 11.65: Vamos ahora a regresar al ejemplo 11.56. Los pasos 1 y 2 del Algoritmo 11.54 no encuentran paralelismo; es decir, necesitamos más que un número constante de sincronizaciones para parallelizar este código. En el paso 3, al aplicar el Algoritmo 11.59 determinamos que sólo hay un ciclo externo válido, que es el mismo del código original en la figura 11.53. Por lo tanto, el ciclo no tiene paralelismo canalizado. En la segunda parte del paso 3, aplicamos el Algoritmo 11.54 para parallelizar el ciclo interno. Tratamos el código dentro de una partición como un programa completo, en donde la única diferencia es que el número de partición se trata como una constante simbólica. En este caso, encontramos que el ciclo interno es parallelizable y, por lo tanto, el código puede parallelizarse con n barreras de sincronización. \square

El Algoritmo 11.64 encuentra todo el paralelismo en un programa, en cada nivel de sincronización. El algoritmo prefiere esquemas de parallelización que tienen menos sincronización, pero menos sincronización no significa que se minimiza la comunicación. Aquí veremos dos extensiones del algoritmo para lidiar con sus debilidades.

Consideración del costo de comunicación

El paso 2 del Algoritmo 11.64 paralleliza cada componente fuertemente conectado, sin importar que se encuentre paralelismo sin sincronización o no. Sin embargo, puede ser posible parallelizar varios componentes sin sincronización ni comunicación. Una solución es buscar masivamente el paralelismo sin sincronización entre los subconjuntos del grafo de dependencias del programa que comparten la mayor parte de los datos.

Si es necesaria la comunicación entre los componentes fuertemente conectados, observamos que cierta comunicación es más costosa que otras. Por ejemplo, el costo de transportar una matriz es mucho más alto que sólo tener que comunicarse entre los procesadores adyacentes. Suponga que s_1 y s_2 son instrucciones en dos componentes fuertemente conectados separados, que acceden a los mismos datos en las iteraciones \mathbf{i}_1 e \mathbf{i}_2 , respectivamente. Si no podemos encontrar las asignaciones de partición $\langle \mathbf{C}_1, \mathbf{c}_1 \rangle$ y $\langle \mathbf{C}_2, \mathbf{c}_2 \rangle$ para las instrucciones s_1 y s_2 , respectivamente, de forma que:

$$\mathbf{C}_1 \mathbf{i}_1 + \mathbf{c}_1 - \mathbf{C}_2 \mathbf{i}_2 + \mathbf{c}_2 = \mathbf{0},$$

en vez de ello tratamos de satisfacer la restricción:

$$\mathbf{C}_1 \mathbf{i}_1 + \mathbf{c}_1 - \mathbf{C}_2 \mathbf{i}_2 + \mathbf{c}_2 \leq \delta$$

en donde δ es una constante pequeña.

Intercambio de comunicación por sincronización

Algunas veces es mejor realizar más sincronización para minimizar la comunicación. El ejemplo 11.66 trata acerca de esto. Por lo tanto, si no podemos parallelizar un código sólo mediante

comunicación entre los componentes fuertemente conectados de la proximidad, debemos tratar de canalizar el cómputo en vez de parallelizar cada componente por separado. Como se muestra en el ejemplo 11.66, la canalización puede aplicarse a una secuencia de ciclos.

Ejemplo 11.66: Para el algoritmo de integración ADI en el ejemplo 11.49, hemos mostrado que al optimizar los anidamientos de ciclos primero y segundo en forma independiente, se encuentra paralelismo en cada uno de ellos. Sin embargo, dicho esquema requeriría transponer la matriz entre los ciclos, incurriendo en un tráfico de datos determinado por $O(n^2)$. Si utilizamos el Algoritmo 11.59 para buscar paralelismo canalizado, encontramos que podemos convertir todo el programa completo en un anidamiento de ciclos completamente permutable, como en la figura 11.59. Entonces podemos aplicar bloques para reducir la sobrecarga de comunicación. Este esquema incurriría en $O(n)$ sincronizaciones, pero requeriría de mucha menos comunicación. \square

```

for (j = 0; j < n; j++)
    for (i = 1; i < n+1; i++) {
        if (i < n) X[i,j] = f(X[i,j] + X[i-1,j])
        if (j > 0) X[i-1,j] = g(X[i-1,j],X[i-1,j-1]);
    }
}

```

Figura 11.59: Un anidamiento de ciclos completamente permutable para el código del ejemplo 11.49

11.9.10 Ejercicios para la sección 11.9

Ejercicio 11.9.1: En la sección 11.9.4, hablamos sobre la posibilidad de utilizar diagonales aparte de los ejes horizontal y vertical, para canalizar el código de la figura 11.51. Escriba código que sea análogo a los ciclos de la figura 11.52 para las diagonales: (a) 135° (b) 120° .

Ejercicio 11.9.2: La figura 11.55(b) puede simplificarse si b divide a n de manera uniforme. Rescriba el código bajo esa suposición.

```

for (i=0; i<100; i++) {
    P[i,0] = 1; /* s1 */
    P[i,i] = 1; /* s2 */
}
for (i=2; i<100; i++)
    for (j=1; j<i; j++)
        P[i,j] = P[i-1,j-1] + P[i-1,j]; /* s3 */

```

Figura 11.60: Cálculo del triángulo de Pascal

Ejercicio 11.9.3: En la figura 11.60 hay un programa para calcular las primeras 100 filas del triángulo de Pascal. Es decir, $P[i, j]$ se convertirá en el número de formas en que se pueden elegir j objetos de i , para $0 \leq j \leq i < 100$.

- Rescriba el código como un solo anidamiento de ciclos completamente permutable.
- Use 100 procesadores en una canalización para implementar este código. Escriba el código para cada procesador p , en términos de p , e indique la sincronización necesaria.
- Rescriba el código usando bloques cuadrados de 10 iteraciones en un lado. Como las iteraciones forman un triángulo, sólo habrá $1 + 2 + \dots + 10 = 55$ bloques. Muestre el código para un procesador (p_1, p_2) asignado al p_1 -ésimo bloque en la dirección i , y al p_2 -ésimo bloque en la dirección j , en términos de p_1 y p_2 .

```

for (i=0; i<100; i++) {
    A[i, 0,0] = B1[i]; /* s1 */
    A[i,99,0] = B2[i]; /* s2 */
}
for (j=1; j<99; j++) {
    A[ 0,j,0] = B3[j]; /* s3 */
    A[99,j,0] = B4[j]; /* s4 */
}
for (i=0; i<99; i++)
    for (j=0; j<99; j++)
        for (k=1; k<100; k++)
            A[i,j,k] = (4*A[i,j,k-1] + A[i-1,j,k-1] +
            A[i+1,j,k-1] + A[i,j-1,k-1] +
            A[i,j+1,k-1]; /* s5 */

```

Figura 11.61: Código para el ejercicio 11.9.4

! Ejercicio 11.9.4: Repita el ejercicio 11.9.2 para el código de la figura 11.61. Sin embargo, observe que las iteraciones para este problema forman un cubo tridimensional cuyo lado es igual a 100. Por ende, los bloques para la parte (c) deben ser $10 \times 10 \times 10$, y hay 1000 de ellos.

! Ejercicio 11.9.5: Vamos a aplicar el Algoritmo 11.59 a un ejemplo simple de las restricciones de partición de tiempo. En lo que sigue, suponga que el vector \mathbf{i}_1 es (i_1, j_1) , y que el vector \mathbf{i}_2 es (i_2, j_2) ; técnicamente, ambos vectores están transpuestos. La condición $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$ consiste en las siguientes desuniones:

- $i_1 < i_2$, o
- $i_1 = i_2$ y $j_1 < j_2$.

Las otras igualdades y desigualdades son:

$$\begin{array}{rcl}
 2i_1 + j_1 - 10 & \geq & 0 \\
 i_2 + 2j_2 - 20 & \geq & 0 \\
 i_1 & = & i_2 + j_2 - 50 \\
 j_1 & = & j_2 + 40
 \end{array}$$

Por último, la desigualdad de partición de tiempo, con las variables desconocidas c_1 , d_1 , e_1 , c_2 , d_2 y e_2 es:

$$c_1 i_1 + d_1 j_1 + e_1 \leq c_2 i_2 + d_2 j_2 + e_2.$$

- Resuelva las restricciones de partición de tiempo para el caso i ; es decir, en donde $i_1 < i_2$. En especial, elimine todas las i_1 , j_1 , i_2 y j_2 que pueda, y establezca las matrices D y A como en el Algoritmo 11.59. Después, aplique el Lema de Farkas a las desigualdades resultantes de las matrices.
- Repita la parte (a) para el caso ii , en donde $i_1 = i_2$ y $j_1 < j_2$.

11.10 Optimizaciones de localidad

El rendimiento de un procesador, ya sea que forme parte de un multiprocesador o no, es muy sensible al comportamiento de su caché. Los fallos en la caché pueden requerir decenas de ciclo del reloj, por lo que las proporciones altas de fallos de caché pueden provocar un bajo rendimiento del procesador. En el contexto de un multiprocesador con un bus común de memoria, la contención en el bus puede aumentar el efecto negativo de la mala localidad de los datos.

Como veremos más adelante, incluso si sólo deseamos mejorar la localidad de los uniprocesadores, el algoritmo de particionamiento afín para la parallelización es útil como un medio en la identificación de oportunidades para las transformaciones de ciclo. En esta sección describiremos tres técnicas para mejorar la localidad de los datos en uniprocesadores y multiprocesadores.

- Para mejorar la localidad temporal de los resultados calculados, tratamos de usar los resultados tan pronto como se generan. Para ello, dividimos un cálculo en particiones independientes y ejecutamos todas las operaciones dependientes en cada partición lo más cerca posible.
- La *contracción de arreglos* reduce las dimensiones de un arreglo y reduce el número de ubicaciones de memoria a las que se accede. Podemos aplicar la contracción de arreglos si sólo se utiliza una ubicación del arreglo en un momento dado.
- Además de mejorar la localidad temporal de los resultados calculados, también debemos optimizar la localidad espacial de los resultados calculados, y la localidad temporal y espacial de los datos de sólo lectura. En vez de ejecutar cada partición, una después de la otra, intercalamos varias de las particiones, de manera que las reutilizaciones entre particiones ocurran lo más cerca posible.

11.10.1 Localidad temporal de los datos calculados

El algoritmo de particionamiento afín junta todas las operaciones dependientes; al ejecutar estas operaciones en serie, mejoramos la localidad temporal de los datos calculados. Vamos a regresar al ejemplo con varias rejillas que vimos en la sección 11.7.1. Al aplicar el Algoritmo 11.43 para parallelizar el código de la figura 11.23, encontramos dos grados de paralelismo. El código en la figura 11.24 contiene dos ciclos externos que iteran a través de la partición independiente en serie. Este código transformado tiene una localidad temporal mejorada, ya que los resultados calculados se utilizan de inmediato en la misma iteración.

Por ende, incluso si nuestra meta es optimizar la ejecución secuencial, es beneficioso utilizar la parallelización para buscar estas operaciones relacionadas y juntarlas. El algoritmo que utilizamos aquí es similar al Algoritmo 11.64, el cual busca todas las granularidades del paralelismo, empezando con el ciclo más externo. Como vimos en la sección 11.9.9, el algoritmo paralleliza los componentes fuertemente conectados por separado, si no podemos encontrar paralelismo sin sincronización en cada nivel. Esta parallelización tiende a incrementar la comunicación. Por ende, combinamos masivamente los componentes fuertemente conectados, parallelizados por separado, si comparten la reutilización.

11.10.2 Contracción de arreglos

La optimización de la contracción de arreglos proporciona otra muestra de la concesión entre el almacenamiento y el paralelismo, que se presentó por primera vez en el contexto del paralelismo a nivel de instrucción, en la sección 10.2.3. Así como al usar más registros se permite un mayor paralelismo a nivel de instrucción, el uso de más memoria permite más paralelismo a nivel de ciclo. Como se muestra en el ejemplo con varias rejillas en la sección 11.7.1, al expandir una variable escalar temporal en un arreglo se permite que distintas iteraciones mantengan distintas instancias de las variables temporales, y que se ejecuten al mismo tiempo. Por el contrario, cuando tenemos una ejecución secuencial que opera sobre un elemento del arreglo en un momento dado en serie, podemos contraer el arreglo, sustituirlo con un escalar y hacer que cada iteración utilice la misma ubicación.

En el programa multirrejillas transformado que se muestra en la figura 11.24, cada iteración del ciclo interno produce y consume un elemento distinto de AP , AM , T y una fila de D . Si estos arreglos no se utilizan fuera del extracto de código, las iteraciones pueden utilizar en forma serial el mismo almacenamiento de datos, en vez de colocar los valores en distintos elementos y filas, respectivamente. La figura 11.62 muestra el resultado de reducir la dimensionalidad de los arreglos. Este código se ejecuta más rápido que el original, ya que lee y escribe menos datos. En especial, en el caso cuando un arreglo se reduce a una variable escalar, podemos asignar la variable a un registro y eliminar la necesidad de acceder a la memoria en conjunto.

A medida que se utiliza menos almacenamiento, hay menos paralelismo disponible. Las iteraciones en el código transformado en la figura 11.62 ahora comparten las dependencias de datos, y ya no pueden ejecutarse en paralelo. Para parallelizar el código en P procesadores, podemos expandir cada una de las variables escalares por un factor de P y hacer que cada procesador acceda a su propia copia privada. Así, la cantidad por la cual se expande el almacenamiento está directamente correlacionada a la cantidad de paralelismo explotado.

```

for (j = 2, j <= jl, j++) {
    for (i = 2, i <= il, i++) {
        AP      = ...;
        T       = 1.0/(1.0 +AP);
        D[2]    = T*AP;
        DW[1,2,j,i] = T*DW[1,2,j,i];
        for (k=3, k <= kl-1, k++) {
            AM      = AP;
            AP      = ...;
            T       = ...AP -AM*D[k-1]...
            D[k]    = T*AP;
            DW[1,k,j,i] = T*(DW[1,k,j,i]+DW[1,k-1,j,i])...
        }
        ...
        for (k=kl-1, k>=2, k--)
            DW[1,k,j,i] = DW[1,k,j,i] +D[k]*DW[1,k+1,j,i];
    }
}

```

Figura 11.62: Código de la figura 11.23 después de particionar (figura 11.24) y contracción de arreglos

Hay tres razones por las que es común buscar oportunidades para la contracción de arreglos:

1. Los lenguajes de programación de mayor nivel para aplicaciones científicas, como Matlab y Fortran 90, soportan operaciones a nivel de arreglo. Cada subexpresión de operaciones con arreglos produce un arreglo temporal. Como los arreglos pueden ser extensos, cada operación con arreglos como una multiplicación o suma requeriría leer y escribir en muchas ubicaciones de memoria, y a la vez requeriría muy pocas operaciones aritméticas. Es importante que reordenemos las operaciones, de manera que los datos se consuman a medida que se produzcan, y que contraigamos estos arreglos en variables escalares.
2. Las supercomputadoras construidas en las décadas de los 80 y 90 son todas máquinas de vectores, por lo que muchas aplicaciones científicas desarrolladas en esa época se han optimizado para tales máquinas. Aun cuando existen los compiladores de vectorización, muchos programadores todavía escriben su código para operar sobre vectores a la vez. El ejemplo de código multirrejillas de este capítulo es un ejemplo de este estilo.
3. El compilador también introduce las oportunidades para la contracción. Como se ilustra mediante la variable T en el ejemplo multirrejillas, un compilador expandiría los arreglos para mejorar la paralelización. Tenemos que contraerlos cuando no sea necesaria la expansión de espacio.

Ejemplo 11.67: La expresión de arreglos $Z = W + X + Y$ se traduce a:

```
for (i=0; i<n; i++) T[i] = W[i] + X[i];
for (i=0; i<n; i++) Z[i] = T[i] + Y[i];
```

Si rescribimos el código de la siguiente manera:

```
for (i=0; i<n; i++) { T = W[i] + X[i]; Z[i] = T + Y[i] }
```

podemos aumentar su velocidad de manera considerable. Desde luego que al nivel del código en C, no tendríamos ni siquiera que utilizar la variable temporal T , pero podríamos escribir la asignación a $Z[i]$ como una sola instrucción. Sin embargo, aquí estamos tratando de modelar el nivel de código intermedio en el cual un procesador de vectores se encargaría de las operaciones. \square

Algoritmo 11.68: Contracción de arreglos.

ENTRADA: Un programa transformado por el Algoritmo 11.64.

SALIDA: Un programa equivalente con dimensiones reducidas del arreglo.

MÉTODO: Una dimensión de un arreglo puede contraerse a un solo elemento si:

1. Cada partición independiente utiliza sólo un elemento del arreglo.
2. La partición no utiliza el valor del elemento al momento de entrar a la partición.
3. El valor del elemento no está vivo al salir de la partición.

Identifique las dimensiones que pueden contraerse (aquellas que cumplen con las tres condiciones anteriores) y sustitúyalas con un solo elemento. \square

El Algoritmo 11.68 asume que el programa se ha transformado primero mediante el Algoritmo 11.64 para juntar todas las operaciones dependientes en una partición y ejecutar las particiones en forma secuencial. Busca aquellas variables de arreglo cuyos rangos vivos de sus elementos en distintas iteraciones están desunidas. Si esas variables no están vivas después del ciclo, contrae el arreglo y hace que el procesador opere en la misma ubicación escalar. Después de cualquier contracción, puede ser necesario expandir los arreglos en forma selectiva, para dar cabida al paralelismo y a otras optimizaciones de localidad.

El análisis del estado de vida que se requiere aquí es más complejo que el descrito en la sección 9.2.5. Si el arreglo se declara como variable global, o si es un parámetro, se requiere un análisis entre procedimientos para asegurar que no se utilice el valor en la salida. Además, debemos calcular el estado de vida de los elementos individuales del arreglo; sería muy impreciso tratar al arreglo en forma conservadora como un escalar.

11.10.3 Intercalación de particiones

A menudo, las distintas particiones en un ciclo leen los mismos datos, o leen y escriben en las mismas líneas de caché. En esta sección y en las dos que le siguen, veremos cómo optimizar la localidad cuando hay reutilización entre las particiones.

Reutilización en los bloques más internos

Adoptamos el modelo simple, en el cual los datos pueden encontrarse en la caché si se reutiliza dentro de un pequeño número de iteraciones. Si el ciclo más interno tiene un límite extenso o desconocido, sólo la reutilización a través de las iteraciones del ciclo más interno se traduce en un beneficio para la localidad. Los bloques crean ciclos internos con límites pequeños conocidos, permitiendo explotar la reutilización dentro y a través de bloques completos de cómputo. Por ende, los bloques tienen el efecto de sacar provecho en más dimensiones de reutilización.

Ejemplo 11.69: Considere el código de multiplicación de matrices que se muestra en la figura 11.5 y su versión con bloques en la figura 11.7. La multiplicación de matrices tiene reutilización a lo largo de cada dimensión de su espacio de iteraciones tridimensional. En el código original, el ciclo más interno tiene n iteraciones, en donde n es desconocida y puede ser grande. Nuestro modelo simple supone que sólo los datos reutilizados a través de las iteraciones en el ciclo más interno se encuentra en la caché.

En la versión con bloques, los tres ciclos más internos ejecutan un bloque tridimensional de cómputo, con B iteraciones en cada lado. El tamaño del bloque B lo elige el compilador de manera que sea lo bastante pequeño como para que todas las líneas de caché leídas y escritas dentro del bloque de cómputo puedan caber en la caché. Por ende, los datos reutilizados a través de las iteraciones en el tercer ciclo más externo pueden encontrarse en la caché. \square

Nos referimos al conjunto más interno de ciclos con pequeños límites conocidos como el *bloque más interno*. Es conveniente que el bloque más interno incluya todas las dimensiones del espacio de iteraciones que acarrean la reutilización, si es posible. No es tan importante maximizar las longitudes de cada lado del bloque. Para el ejemplo de multiplicación de matrices, el bloqueo tridimensional reduce la cantidad de datos a los que se accede para cada matriz, por un factor de B^2 . Si hay reutilización, es mejor acomodar los bloques de mayor dimensión con lados más cortos, que los bloques de menor dimensión con lados más largos.

Podemos optimizar la localidad del anidamiento de ciclos más interno, completamente permutable, bloqueando el subconjunto de ciclos que comparten la reutilización. También podemos generalizar la noción del bloqueo para explotar las reutilizaciones que se encuentran entre las iteraciones de ciclos paralelos externos. Observamos que los bloques principalmente intercalan la ejecución de un pequeño número de instancias en el ciclo más interno. En la multiplicación de matrices, cada instancia del ciclo más interno calcula un elemento de la respuesta tipo arreglo; hay n^2 de ellos. Los bloques intercalan la ejecución de un bloque de instancias, calculando B iteraciones de cada instancia a la vez. De manera similar, podemos intercalar las iteraciones en ciclos paralelos para aprovechar las reutilizaciones entre ellos.

A continuación definimos dos primitivas que pueden reducir la distancia entre reutilizaciones, a través de distintas iteraciones. Aplicamos estas primitivas en forma repetida, empezando desde el ciclo más externo hasta que todas las reutilizaciones se muevan, adyacentes unas a otras, en el bloque más interno.

Intercalación de ciclos internos en un ciclo paralelo

Considere el caso en el que un ciclo paralelizable externo contiene un ciclo interno. Para explotar la reutilización a través de las iteraciones del ciclo externo, intercalamos las ejecuciones

de un número fijo de instancias del ciclo interno, como se muestra en la figura 11.63. Al crear bloques internos bidimensionales, esta transformación reduce la distancia entre la reutilización de iteraciones consecutivas del ciclo externo.

```
for (i=0; i<n; i++)      for (ii=0; ii<n; ii+=4)
    for (j=0; j<n; j++)    for (j=0; j<n; j++)
        <S>                  for (i=ii; ii<min(n, ii+4); ii+=4)
                            <S>
```

(a) Programa fuente.

(b) Código transformado.

Figura 11.63: Intercalación de 4 instancias del ciclo interno

El paso que convierte un ciclo

```
for (i=0; i<n; i++)
    <S>
```

en

```
for (ii=0; ii<n; ii+=4)
    for (i=ii; ii<min(n, ii+4); ii+=4)
        <S>
```

se conoce como *seccionamiento (stripmining)*. En el caso en el que el ciclo externo en la figura 11.63 tiene un pequeño límite conocido, no es necesario seccionarlo, sino solo permutar los dos ciclos en el programa original.

Intercalación de instrucciones en un ciclo paralelo

Considere el caso en el que un ciclo paralelizable contiene una secuencia de instrucciones s_1, s_2, \dots, s_m . Si algunas de estas instrucciones son también ciclos, las instrucciones de iteraciones consecutivas aún pueden estar separadas por muchas operaciones. Podemos explotar la reutilización entre iteraciones al intercalar de nuevo sus ejecuciones, como se muestra en la figura 11.64. Esta transformación *distribuye* un ciclo seccionado a través de las instrucciones. De nuevo, si el ciclo externo tiene un pequeño número fijo de iteraciones, no es necesario seccionar el ciclo, sino sólo distribuir el ciclo original a través de todas las instrucciones.

Utilizamos $s_i(j)$ para denotar la ejecución de la instrucción s_1 en la iteración j . En vez del orden de ejecución secuencial original que se muestra en la figura 11.65(a), el código se ejecuta en el orden mostrado en la figura 11.65(b).

Ejemplo 11.70: Ahora vamos a regresar al ejemplo multirrejillas y mostraremos cómo explotamos la reutilización entre las iteraciones de los ciclos paralelos externos. Observamos que las referencias $DW[1, k, j, i]$, $DW[1, k-1, j, i]$ y $DW[1, k+1, j, i]$ en los ciclos más internos del código en la figura 11.62 tienen una localidad espacial bastante mala. Del análisis de reutilización,

```

for (i=0; i<n; i++) {
    <S1>
    <S2>
    ...
}

for (ii=0; ii<n; ii+=4) {
    for (i=ii; i<min(n,ii+4); i++)
        <S1>
        for (i=ii; i<min(n,ii+4); i++)
            <S2>
        ...
}

```

(a) Programa fuente.

(b) Código transformado.

Figura 11.64: La transformación de intercalación de instrucciones

como vimos en la sección 11.5, el ciclo con el índice i acarrea la localidad espacial y el ciclo con el índice k acarrea la reutilización de grupo. El ciclo con el índice k ya es el ciclo más interno, por lo que nos interesa intercalar las operaciones en DW a partir de un bloque de particiones con valores consecutivos de i .

Aplicamos la transformación para intercalar las instrucciones en el ciclo y obtener el código de la figura 11.66, y después aplicamos la transformación para intercalar los ciclos internos y obtener el código de la figura 11.67. Observe que, a medida que intercalamos B iteraciones del ciclo con el índice i , debemos expandir las variables AP , AM , T en arreglos que guarden B resultados a la vez. \square

11.10.4 Reunión de todos los conceptos

El Algoritmo 11.71 optimiza la localidad para un uniprocesador, y el Algoritmo 11.72 optimiza tanto el paralelismo como la localidad para un multiprocesador.

Algoritmo 11.71: Optimización de la localidad de datos en un uniprocesador.

ENTRADA: Un programa con accesos afines a los arreglos.

SALIDA: Un programa equivalente que maximiza la localidad de los datos.

MÉTODO: Realice los siguientes pasos:

1. Aplique el Algoritmo 11.64 para optimizar la localidad temporal de los resultados calculados.
2. Aplique el Algoritmo 11.68 para contraer los arreglos en donde sea posible.
3. Determine el subespacio de iteraciones que pueden compartir los mismos datos o líneas de caché, usando la técnica descrita en la sección 11.5. Para cada instrucción, identifique las dimensiones de los ciclos paralelos externos que tienen reutilización de datos.
4. Para cada ciclo paralelo externo que acarree la reutilización, mueva un bloque de las iteraciones hacia el bloque más interno, mediante la aplicación de las primitivas de intercalación en forma repetida.

$$\begin{aligned}
 & s_1(0), s_2(0), \dots, s_m(0), \\
 & s_1(1), s_2(1), \dots, s_m(1), \\
 & s_1(2), s_2(2), \dots, s_m(2), \\
 & s_1(3), s_2(3), \dots, s_m(3), \\
 & s_1(4), s_2(4), \dots, s_m(4), \\
 & s_1(5), s_2(5), \dots, s_m(5), \\
 & s_1(6), s_2(6), \dots, s_m(6), \\
 & s_1(7), s_2(7), \dots, s_m(7), \\
 & \dots;
 \end{aligned}$$

(a) Orden original.

$$\begin{aligned}
 & s_1(0), s_1(1), s_1(2), s_1(3), \\
 & s_2(0), s_2(1), s_2(2), s_2(3), \\
 & \dots, \\
 & s_m(0), s_m(1), s_m(2), s_m(3), \\
 & s_1(4), s_1(5), s_1(6), s_1(7), \\
 & s_2(4), s_2(5), s_2(6), s_2(7), \\
 & \dots, \\
 & s_m(4), s_m(5), s_m(6), s_m(7) \\
 & \dots;
 \end{aligned}$$

(b) Orden transformado.

Figura 11.65: Distribución de un ciclo seccionado

5. Aplique los bloques al subconjunto de dimensiones en el anidamiento de ciclos completamente permutable más interno que acarree la reutilización.
6. Use bloques en el anidamiento de ciclos completamente permutable externo para niveles más altos de jerarquías de memoria, como la caché de tercer nivel o la memoria física.
7. Expanda las escalares y los arreglos en donde sea necesario, mediante las longitudes de los bloques.

□

Algoritmo 11.72: Optimización del paralelismo y la localidad de los datos para multiprocesadores.

ENTRADA: Un programa con accesos afines a los arreglos.

SALIDA: Un programa equivalente que maximiza el paralelismo y la localidad de los datos.

MÉTODO: Haga lo siguiente:

1. Use el Algoritmo 11.64 para paralelizar el programa y crear un programa SPMD.

```

for (j = 2, j <= jl, j++)
    for (ii = 2, ii <= il, ii+=b) {
        for (i = ii; i <= min(ii+b-1,il); i++) {
            ib          = i-ii+1;
            AP[ib]      = ...;
            T           = 1.0/(1.0 +AP[ib]);
            D[2,ib]     = T*AP[ib];
            DW[1,2,j,i] = T*DW[1,2,j,i];
        }
        for (i = ii; i <= min(ii+b-1,il); i++) {
            for (k=3, k <= kl-1, k++)
                ib          = i-ii+1;
                AM          = AP[ib];
                AP[ib]      = ...;
                T           = ...AP[ib]-AM*D[ib,k-1]...
                D[ib,k]     = T*AP;
                DW[1,k,j,i] = T*(DW[1,k,j,i]+DW[1,k-1,j,i])...
            }
        ...
        for (i = ii; i <= min(ii+b-1,il); i++) {
            for (k=kl-1, k>=2, k--) {
                DW[1,k,j,i] = DW[1,k,j,i] +D[iw,k]*DW[1,k+1,j,i];
            /* Termina el código que va a ejecutar el procesador (j,i) */
        }
    }
}

```

Figura 11.66: Extracto de la figura 11.23 después del particionamiento, la contracción de arreglos y los bloques

2. Aplique el Algoritmo 11.71 al programa SPMD producido en el paso 1 para optimizar su localidad.

□

11.10.5 Ejercicios para la sección 11.10

Ejercicio 11.10.1: Realice la contracción de arreglos en las siguientes operaciones con vectores:

```

for (i=0; i<n; i++) T[i] = A[i] * B[i];
for (i=0; i<n; i++) D[i] = T[i] + C[i];

```

Ejercicio 11.10.2: Realice la contracción de arreglos en las siguientes operaciones con vectores:

```

for (j = 2, j <= jl, j++)
    for (ii = 2, ii <= il, ii+=b) {
        for (i = ii; i <= min(ii+b-1,il); i++) {
            ib          = i-ii+1;
            AP[ib]      = ...;
            T           = 1.0/(1.0 +AP[ib]);
            D[2,ib]     = T*AP[ib];
            DW[1,2,j,i] = T*DW[1,2,j,i];
        }
        for (k=3, k <= kl-1, k++)
            for (i = ii; i <= min(ii+b-1,il); i++) {
                ib          = i-ii+1;
                AM          = AP[ib];
                AP[ib]      = ...;
                T           = ...AP[ib]-AM*D[ib,k-1]...
                D[ib,k]     = T*AP;
                DW[1,k,j,i] = T*(DW[1,k,j,i]+DW[1,k-1,j,i])...
            }
        ...
        for (k=kl-1, k>=2, k--) {
            for (i = ii; i <= min(ii+b-1,il); i++)
                DW[1,k,j,i] = DW[1,k,j,i] +D[iw,k]*DW[1,k+1,j,i];
            /* Termina el código que va a ejecutar el procesador (j,i) */
        }
    }
}

```

Figura 11.67: Extracto de la figura 11.23 después del particionamiento, la contracción de arreglos y los bloques

```

for (i=0; i<n; i++) T[i] = A[i] + B[i];
for (i=0; i<n; i++) S[i] = C[i] + D[i];
for (i=0; i<n; i++) E[i] = T[i] * S[i];

```

Ejercicio 11.10.3: Seccione el siguiente ciclo externo:

```

for (i=n-1; i>=0; i--)
    for (j=0; j<n; j++)

```

en secciones con anchura de 10.

11.11 Otros usos de las transformaciones afines

Hasta ahora nos hemos enfocado en la arquitectura de las máquinas con memoria compartida, pero la teoría de las transformaciones afines de ciclos tiene muchas otras aplicaciones. Podemos aplicar transformaciones afines a otras formas de paralelismo, incluyendo las máquinas con memoria

distribuida, las instrucciones vectoriales, las instrucciones SIMD (Single Instruction Multiple Data; Una sola instrucción, múltiples datos), así como las máquinas emisoras de múltiples instrucciones. El análisis de reutilización presentado en este capítulo también es útil para la *preobtención* de datos, la cual es una técnica efectiva para mejorar el rendimiento de la memoria.

11.11.1 Máquinas con memoria distribuida

Para las máquinas con memoria distribuida, en donde los procesadores se comunican enviándose mensajes entre sí; es aún más importante que a los procesadores se les asignen unidades grandes e independientes de cómputo, como las generadas por el algoritmo de particionamiento afín. Además del particionamiento del cómputo, todavía quedan varias cuestiones adicionales de compilación:

1. *Repartición de datos.* Si los procesadores utilizan distintas porciones de un arreglo, sólo tienen que repartir suficiente espacio para contener la porción utilizada. Podemos usar la proyección para determinar la sección de arreglos utilizados por cada procesador. La entrada es el sistema de desigualdades lineales que representan los límites de ciclo, las funciones de acceso a los arreglos y las particiones afines que asignan las iteraciones a los IDs de los procesadores. Proyectamos los índices de ciclo para alejarlos y buscamos para cada ID de procesador el conjunto de ubicaciones de arreglos utilizadas.
2. *Código de comunicación.* Debemos generar código explícito para enviar y recibir datos hacia, y desde, otros procesadores. En cada punto de sincronización:
 - (a) Se determinan los datos que residen en un procesador, y que otros procesadores necesitan.
 - (b) Se genera el código que busca todos los datos a enviar y los empaqueta en un búfer.
 - (c) De manera similar, se determinan los datos que necesita el procesador, se desempaquetan los mensajes recibidos y se mueven los datos a las ubicaciones de memoria apropiadas.

De nuevo, si todos los accesos son afines, el compilador puede realizar estas tareas, usando el framework afín.

3. *Optimización.* No es necesario que todas las comunicaciones se lleven a cabo en los puntos de sincronización. Es preferible que cada procesador envíe los datos tan pronto como estén disponibles, y que cada procesador no empiece esperando los datos hasta que se necesiten. Dichas optimizaciones deben balancearse mediante el objetivo de no generar demasiados mensajes, ya que hay una sobrecarga no trivial asociada con el procesamiento de cada mensaje.

Las técnicas aquí descritas tienen otras aplicaciones también. Por ejemplo, un sistema embebido de propósito especial puede utilizar coprocesadores para descargar parte de sus cálculos. O, en vez de demandar la obtención de datos en la caché, un sistema embebido puede utilizar un controlador separado para cargar y descargar datos hacia, y fuera de, la caché o de otros búferes de datos, mientras que el procesador opera con otros datos. En estos casos pueden utilizarse técnicas similares para generar el código que desplace los datos.

11.11.2 Procesadores que emiten múltiples instrucciones

También podemos usar transformaciones afines de ciclos para optimizar el rendimiento de las máquinas que emiten varias instrucciones. Como vimos en la sección 10.5, el rendimiento de un ciclo canalizado por software se limita mediante dos factores: los ciclos en las restricciones de precedencia y el uso del recurso crítico. Al modificar la composición del ciclo más interno, podemos mejorar estos límites.

En primer lugar, tal vez podamos usar transformaciones de ciclos para crear ciclos más internos paralelizables, con lo cual se eliminan por completo los ciclos de precedencia. Suponga que un programa tiene dos ciclos, en donde el externo es paralelizable y el interno no lo es. Podemos permutar los dos ciclos para hacer el ciclo interno paralelizable, y así crear más oportunidades para el paralelismo a nivel de instrucción. Observe que no es necesario que las iteraciones en el ciclo más interno sean completamente paralelizables. Basta con que el ciclo de dependencias en el ciclo sea lo bastante corto como para que se utilicen por completo todos los recursos de hardware.

También podemos relajar el límite debido al uso de los recursos, mejorando el balance de uso dentro de un ciclo. Suponga que un ciclo sólo utiliza el sumador, y que otro ciclo sólo utiliza el multiplicador. O, suponga que un ciclo está limitado por memoria y que otro está limitado por cómputo. Es conveniente fusionar cada par de ciclos en estos ejemplos, para poder utilizar todas las unidades funcionales al mismo tiempo.

11.11.3 Instrucciones con vectores y SIMD

Además de la cuestión sobre múltiples instrucciones, hay otras dos formas importantes de paralelismo a nivel de instrucción: las operaciones con vectores y SIMD. En ambos casos, la cuestión de sólo una instrucción hace que la misma operación se aplique a un vector de datos.

Como dijimos antes, muchas de las primeras supercomputadoras utilizaban instrucciones con vectores. Las operaciones con vectores se llevan a cabo en forma canalizada; los elementos en el vector se obtienen en forma serial y los cómputos sobre los distintos elementos se traslanan. En las máquinas de vectores avanzadas, las operaciones con vectores pueden *encadenarse*: a medida que se producen los elementos de los resultados con vectores, se consumen de inmediato por las operaciones de otra instrucción con vectores, sin tener que esperar a que todos los resultados estén listos. Además, en máquinas avanzadas con hardware para *dispersar/recolectar*, los elementos de los vectores no necesitan ser contiguos; se utiliza un vector índice para especificar la ubicación de los elementos.

Las instrucciones SIMD especifican que se debe realizar la misma operación en ubicaciones contiguas de memoria. Estas instrucciones cargan datos de la memoria en paralelo, los almacenan en registros amplios y realizan cálculos con ellos usando hardware en paralelo. Muchas aplicaciones de medios, grafos y procesamiento de señales digitales pueden beneficiarse de estas operaciones. Los procesadores de medios de bajo rendimiento pueden alcanzar un paralelismo a nivel de instrucción con sólo emitir una instrucción SIMD a la vez. Los procesadores de mayor rendimiento pueden combinar la emisión de instrucciones SIMD con varias instrucciones para lograr un mayor rendimiento.

La generación de instrucciones con vectores y SIMD comparte muchas similitudes con la optimización de la localidad. A medida que encontramos particiones independientes que operan

sobre ubicaciones contiguas de memoria, seccionamos esas iteraciones e intercalamos esas operaciones en los ciclos más internos.

La generación de instrucciones SIMD presenta dos dificultades adicionales. En primer lugar, ciertas máquinas requieren que los datos SIMD que se obtienen de memoria estén alineados. Por ejemplo, podrían requerir que los operandos SIMD de 256 bytes se coloquen en direcciones que sean múltiplos de 256. Si el ciclo de origen opera sólo sobre un arreglo de datos, podemos generar un ciclo principal que opere sobre datos alineados y código adicional antes y después del ciclo para manejar esos elementos en el límite. Sin embargo, para los ciclos que operan sobre más de un arreglo, tal vez no sea posible alinear todos los datos al mismo tiempo. En segundo lugar, los datos utilizados por iteraciones consecutivas en un ciclo pueden no estar contiguos. Algunos ejemplos incluyen muchos algoritmos importantes de procesamiento de señales digitales, como los decodificadores de Viterbi y las transformadas rápidas de Fourier. Tal vez se requieran operaciones adicionales para revolver los datos alrededor, para sacar provecho de las instrucciones SIMD.

11.11.4 Preobtención

Ninguna optimización de la localidad de los datos puede eliminar todos los accesos a la memoria; por una parte, los datos usados por primera vez deben obtenerse de memoria. Para ocultar la latencia de las operaciones de memoria, se han adoptado *instrucciones de preobtención (prefetch)* en muchos procesadores de alto rendimiento. La preobtención es una instrucción de máquina que indica al procesador la probabilidad de usar ciertos datos pronto, y que es conveniente cargar esos datos en la caché, si no están ya presentes.

El análisis de reutilización descrito en la sección 11.5 puede usarse para estimar cuando hay probabilidad de fallos en la caché. Hay dos consideraciones importantes a la hora de generar instrucciones de preobtención. Si se va a acceder a ubicaciones contiguas de memoria, debemos emitir sólo una instrucción de preobtención para cada línea de caché. Las instrucciones de preobtención deben emitirse con la suficiente anticipación como para que los datos estén en la caché para cuando se vayan a utilizar. Sin embargo, no debemos emitir instrucciones de preobtención con demasiada anticipación. Éstas instrucciones pueden desplazar datos que podrían ser necesarios todavía; además, los datos preobtenidos podrían vaciarse antes de usarlos.

Ejemplo 11.73: Considere el siguiente código:

```
for (i=0; ii<3; i++)
    for (j=0; j<100; j++)
        A[i,j] = ...;
```

Suponga que la máquina destino tiene una instrucción de preobtención que puede obtener dos palabras de datos a la vez, y que la latencia de una instrucción de preobtención requiere un tiempo aproximado de ejecución equivalente a seis iteraciones del ciclo anterior. El código de preobtención para el ejemplo anterior se muestra en la figura 11.68.

Desenrollamos el ciclo más interno dos veces, por lo que puede emitirse una instrucción de preobtención para cada línea de caché. Utilizamos el concepto de canalización de software para preobtener los datos seis iteraciones antes de usarlos. El prólogo obtiene los datos que se

```

for (i=0; ii<3; i++) {
    for (j=0; j<6; j+=2)
        prefetch(&A[i,j]);
    for (j=0; j<94; j+=2) {
        prefetch(&A[i,j+6]);
        A[i,j] = ...;
        A[i,j+1] = ...;
    }
    for (j=94; j<100; j++)
        A[i,j] = ...;
}

```

Figura 11.68: Código modificado para la preobtención de datos

utilizan en las primeras seis iteraciones. El ciclo de estado estable preobtiene seis iteraciones por adelantado, a medida que realiza su cómputo. El epílogo no emite instrucciones de preobtención, sino que sólo ejecuta las iteraciones restantes. \square

11.12 Resumen del capítulo 11

- ◆ *Paralelismo y localidad de los arreglos:* Las oportunidades más importantes para las optimizaciones basadas en el paralelismo y la localidad provienen de los ciclos que acceden a arreglos. Estos ciclos tienen por lo regular dependencias limitadas entre los accesos a los elementos de un arreglo, y tienden a acceder a los arreglos en un patrón regular, permitiendo un uso eficiente de la caché para una buena localidad.
- ◆ *Accesos afines:* Casi toda la teoría y las técnicas para la optimización del paralelismo y la localidad suponen que los accesos a arreglos son afines: las expresiones para los índices de arreglos son funciones lineales de los índices de ciclo.
- ◆ *Espacios de iteraciones:* Un anidamiento de ciclos con d ciclos anidados define un espacio de iteraciones d -dimensional. Los puntos en el espacio son los d tuplas de valores que pueden asumir los índices de ciclo durante la ejecución del anidamiento de ciclos. En el caso afín, los límites sobre cada índice de ciclo son funciones lineales de los índices del ciclo externo, por lo que el espacio de iteraciones es un poliedro.
- ◆ *Eliminación de Fourier-Motzkin:* Una manipulación clave de los espacios de iteraciones es reordenar los ciclos que definen el espacio de iteraciones. Para ello se requiere la proyección de un espacio de iteraciones, representado por un poliedro, en un subconjunto de sus dimensiones. El algoritmo de Fourier-Motzkin sustituye los límites superior e inferior en una variable dada por las desigualdades entre los mismos límites.
- ◆ *Dependencias de datos y accesos a arreglos:* Un problema central que debemos resolver para poder manipular los ciclos de las optimizaciones de paralelismo y localidad es si dos accesos a un arreglo tienen una dependencia de datos (pueden tocar el mismo elemento del arreglo). Cuando los accesos y los límites de ciclo son afines, el problema puede expresarse

como si hubiera soluciones a una ecuación con matrices y vectores dentro del poliedro que define el espacio de iteraciones.

- ◆ *Rango de una matriz y reutilización de datos:* La matriz que describe un acceso a un arreglo puede indicarnos varias cosas importantes acerca de ese acceso. Si el rango de la matriz es lo más grande posible (el mínimo del número de filas y el número de columnas), entonces el acceso nunca toca el mismo elemento dos veces a medida que iteran los ciclos. Si el arreglo se almacena en formato de orden por fila (columna), entonces el rango de la matriz con la última (primera) fila eliminada nos indica si el acceso tiene buena localidad; es decir, se accede a los elementos en una línea de caché individual casi al mismo tiempo.
- ◆ *Dependencia de datos y ecuaciones diofantinas:* Sólo porque dos accesos al mismo arreglo están en contacto con la misma región del arreglo, no significa que en realidad accedan a un elemento en común. La razón es que cada arreglo puede omitir ciertos elementos; por ejemplo, uno accede a los elementos pares y el otro a los elementos impares. Para poder estar seguros de que hay una dependencia de datos, debemos resolver una ecuación diofantina (sólo soluciones enteras).
- ◆ *Solución de ecuaciones lineales diofantinas:* La técnica clave es calcular el máximo común divisor (GCD) de los coeficientes de las variables. Sólo si ese GCD divide el término constante, habrá soluciones enteras.
- ◆ *Restricciones de partición de espacio:* Para parallelizar la ejecución de un anidamiento de ciclos, debemos asignar las iteraciones del ciclo a un espacio de procesadores, que puede tener una o más dimensiones. Las restricciones de partición de espacio indican que si dos accesos en dos iteraciones distintas comparten una dependencia de datos (es decir, acceden al mismo elemento del arreglo), entonces deben asignarse al mismo procesador. Siempre y cuando la asignación de iteraciones a los procesadores sea afín, podemos formular el problema en términos de matriz-vector.
- ◆ *Transformaciones primitivas de código:* Las transformaciones que se utilizan para parallelizar los programas con accesos afines a los arreglos son combinaciones de siete primitivas: fusión de ciclos, fisión de ciclos, reindexado (sumar una constante a los índices de ciclo), escalado (multiplicar los índices de ciclo por una constante), inversión (de un índice de ciclo), permutación (del orden de los ciclos), y desplazamiento (rescribir los ciclos de forma que la línea de pasaje a través del espacio de iteraciones ya no sea a lo largo de uno de los ejes).
- ◆ *Sincronización de operaciones en paralelo:* Algunas veces podemos obtener más paralelismo si insertamos operaciones de sincronización entre los pasos de un programa. Por ejemplo, los anidamientos de ciclos consecutivos pueden tener dependencias de datos, pero las sincronizaciones entre los ciclos pueden permitir la parallelización de éstos por separado.
- ◆ *Canalización:* Esta técnica de parallelización permite a los procesadores compartir datos, al pasar ciertos datos en forma síncrona (por lo general, elementos de un arreglo) de un procesador a un procesador adyacente en el espacio de procesadores. El método puede mejorar la localidad de los datos a los que accede cada procesador.

- ◆ *Restricciones de partición de tiempo:* Para descubrir las oportunidades para la canalización, debemos descubrir soluciones a las restricciones de partición de tiempo. Éstas nos indican que cuando dos accesos a un arreglo pueden tocar el mismo elemento del arreglo, entonces el acceso en la iteración que ocurre primero debe asignarse a una etapa en la canalización que ocurra no después que la etapa a la cual se asignó el segundo acceso.
- ◆ *Resolución de restricciones de partición de tiempo:* El Lema de Farkas proporciona una técnica poderosa para buscar todas las asignaciones de particiones de tiempo afines que se permiten mediante un anidamiento de ciclos dado, con accesos a arreglos. En esencia, la técnica consiste en sustituir la formulación fundamental de las desigualdades lineales que expresan las restricciones de partición de tiempo por su doble.
- ◆ *Uso de bloques:* Esta técnica descompone cada uno de los diferentes ciclos en un anidamiento de ciclos, en dos ciclos cada uno. La ventaja es que si hacemos esto, podremos trabajar con secciones pequeñas (bloques) de un arreglo multidimensional, un bloque a la vez. Eso, a su vez, mejora la localidad del programa, dejando que todos los datos necesarios residan en la caché mientras se trabaja sobre un solo bloque.
- ◆ *Seccionamiento (Stripmining):* De manera similar al bloqueo, esta técnica descompone sólo un subconjunto de los ciclos de un anidamiento de ciclos en dos ciclos cada uno. Una posible ventaja es que se accede al arreglo multidimensional una “sección” a la vez, lo cual puede conducir a la mejor utilización posible de la caché.

11.13 Referencias para el capítulo 11

Para explicaciones detalladas de arquitecturas de microprocesadores, el lector puede consultar el texto de Hennessy y Patterson [9].

Lamport [13] y Kuck, Muraoka y Chen [6] introdujeron el concepto del análisis de dependencias de datos. Las primeras pruebas de dependencia de datos utilizaron la heurística para demostrar que un par de referencias son independientes al determinar si no hay soluciones a las ecuaciones diofantinas y los sistemas de desigualdades lineales reales: [5, 6, 26]. Maydan, Hennessy y Lam [18] formularon la prueba de dependencia de datos como programación lineal entera y mostraron que el problema puede resolverse con exactitud y eficiencia en la práctica. El análisis de dependencias de datos que se describe en este capítulo se basa en el trabajo que realizaron Maydan, Hennessy y Lam [18], y Pugh y Wonnacott [23], que a su vez utilizan técnicas de la eliminación de Fourier-Motzkin [7] y del algoritmo de Shostak [25].

Las décadas de 1970 y 1980 vieron el uso de las transformaciones de ciclo para mejorar la vectorización y la parallelización: fusión de ciclos [3], fisión de ciclos [1], seccionamiento (stripmining) [17], e intercambio de ciclos [28]. Hubo tres proyectos experimentales importantes de parallelización/vectorización en ese tiempo: Parafrase, que dirigió Kuck en la Universidad de Illinois Urbana-Champaign [21], el proyecto PFC, que dirigió Kennedy en la Universidad Rice [4], y el proyecto PTRAN de Allen en IBM Research [2].

McKellar y Coffman [19] hablaron por primera vez sobre el uso de bloques para mejorar la localidad de los datos. Lam, Rothbert y Wolf [12] proporcionaron el primer análisis empírico a profundidad del uso de bloques en las cachés para las arquitecturas modernas. Wolf y Lam [27]

utilizaron las técnicas del álgebra lineal para calcular la reutilización de los datos en los ciclos. Sarkar y Gao [24] presentaron la optimización de la contracción de arreglos.

Lamport [13] fue el primero en modelar los ciclos como espacios de iteraciones, y utilizó la hiperplaneación (un caso especial de una transformación afín) para encontrar paralelismo en los multiprocesadores. Las transformaciones afines tienen sus raíces en el diseño de algoritmos con arreglos sistólicos [11]. Diseñados como algoritmos paralelos implementados directamente en VLSI, los arreglos sistólicos requieren la minimización de la comunicación junto con la paralelización. Se desarrollaron técnicas algebraicas para asignar el cómputo en coordenadas de espacio y tiempo. Feautrier [8] presentó el concepto de un programa afín y el uso del Lema de Farkas en las transformaciones afines. El algoritmo de transformaciones afines que se describe en este capítulo está basado en el trabajo de Lim y colaboradores [15, 14, 16].

Porterfield [22] propuso uno de los primeros algoritmos de compiladores para preobtener datos. Mowry, Lam y Gupta [20] aplicaron el análisis de la reutilización para minimizar la sobre-carga de las instrucciones de preobtención y ganar una mejora general en el rendimiento.

1. Abu-Sufah, W., D. J. Kuck y D. H. Lawrie, “On the performance enhancement of paging systems through program analysis and transformations”, *IEEE Trans. on Computing* **C-30**:5 (1981), pp. 341-356.
2. Allen, F. E., M. Burke, P. Charles, R. Cytron y J. Ferrante, “An overview of the PTRAN analysis system for multiprocessing”, *J. Parallel and Distributed Computing* **5**:5 (1988), pp. 617-640.
3. Allen, F. E. y J. Cocke, “A Catalogue of optimizing transformations”, en *Design and Optimization of Compilers* (R. Rustin, ed.), pp. 1-30, Prentice-Hall, 1972.
4. Allen, R. y K. Kennedy, “Automatic translation of Fortran programs to vector form”, *ACM Transactions on Programming Languages and Systems* **9**:4 (1987), pp. 491-542.
5. Banerjee, U., *Data Dependence in Ordinary Programs*, tesis de Maestría, Departamento de Ciencias Computacionales, Universidad de Illinois Urbana-Champaign, 1976.
6. Banerjee, U., *Speedup of Ordinary Programs*, tesis Ph. D., Departamento de Ciencias Computacionales, Universidad de Illinois Urbana-Champaign, 1979.
7. Dantzig, G. y B. C. Eaves, “Eliminación de Fouier-Motzkin y su doble”, *J. Combinatorial Theory*, **A(14)** (1973), pp. 288-297.
8. Feautrier, P., “Some efficient solutions to the affine scheduling problem: I. One-dimensional time”, *International J. Parallel Programming* **21**:5 (1992), pp. 313-348.
9. Hennessy, J. L. y D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Tercera edición, Morgan Kaufman, San Francisco, 2003.
10. Kuck, D., Y. Muraoka y S. Chen, “On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup”, *IEEE Transactions on Computers* **C-21**:12 (1972), pp. 1293-1310.

11. Kung, H. T. y C. E. Leiserson, “Systolic arrays (for VLSI)”, en Duff, I. S. y G. W. Stewart (eds.), *Sparse Matrix Proceedings*, pp. 256-282. Sociedad para las Matemáticas industriales y aplicadas, 1978.
12. Lam, M. S., E. E. Rothberg y M. E. Wolf, “The cache performance and optimization of blocked algorithms”, *Proc. Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (1991), pp. 63-74.
13. Lamport, L., “The parallel execution of DO loops”, *Comm. ACM* **17**:2 (1974), pp. 83-93.
14. Lim, A. W., G. I. Cheong y M. S. Lam, “An affine partitioning algorithm to maximize parallelism and minimize communication”, *Proc. 13th International Conference on Supercomputing* (1999), pp. 228-237.
15. Lim, A. W. y M. S. Lam, “Maximizing parallelism and minimizing synchronization with affine transforms”, *Proc. 24th ACM SIGPLAN-SIG ACT Symposium on Principles of Programming Languages* (1997), pp. 201-214.
16. Lim, A. W., S.-W. Liao y M. S. Lam, “Blocking and array contraction across arbitrarily nested loops using affine partitioning”, *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2001), pp. 103-112.
17. Loveman, D. B., “Program improvement by source-to-source transformation”, *J. ACM* **24**:1 (1977), pp. 121-145.
18. Maydan, D. E., J. L. Hennessy y M. S. Lam, “An efficient method for exact dependence analysis”, *Proc. ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 1-14.
19. McKeller, A. C. y E. G. Coffman, “The organization of matrices and matrix operations in a paged multiprogramming environment”, *Comm. ACM*, **12**:3 (1969), pp. 153-165.
20. Mowry, T. C., M. S. Lam y A. Gupta, “Design and evaluation of a compiler algorithm for prefetching”, *Proc. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (1992), pp. 62-73.
21. Padua, D. A. y M. J. Wolfe, “Advanced compiler optimizations for supercomputers”, *Comm. ACM*, **29**:12 (1986), pp. 1184-1201.
22. Porterfield, A., *Software Methods for Improving Cache Performance on Supercomputer Applications*, tesis Ph. D., Departamento de Ciencias Computacionales, Universidad Rice, 1989.
23. Pugh, W. y D. Wonnacott, “Eliminating false positives using the omega test”, *Proc. ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pp. 140-151.
24. Sarkar, V. y G. Gao, “Optimization of array accesses by collective loop transformations”, *Proc. 5th International Conference on Supercomputing* (1991), pp. 194-205.

25. R. Shostak, “Deciding linear inequalities by computing loop residues”, *J. ACM*, **28**:4 (1981), pp. 769-779.
26. Towle, R. A., *Control and Data Dependence for Program Transformation*, tesis Ph. D., Departamento de Ciencias Computacionales, Universidad de Illinois Urbana-Champaign, 1976.
27. Wolf, M. E. y M. S. Lam, “A data locality optimizing algorithm”, *Proc. SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 30-44.
28. Wolfe, M. J., *Techniques for Improving the Inherent Parallelism in Programs*, tesis de Maestría, Departamento de Ciencias Computacionales, Universidad de Illinois Urbana-Champaign, 1978.

Capítulo 12

Análisis interprocedural

En este capítulo destacaremos la importancia del análisis interprocedural, al hablar sobre varios problemas importantes de optimización que no pueden resolverse con el análisis intraprocedural. Empezaremos por describir las formas comunes del análisis interprocedural y explicaremos las dificultades en su implementación. Después describiremos las aplicaciones para este análisis. Para los lenguajes de programación muy populares como C y Java, el análisis de alias de apuntadores es la clave para cualquier análisis interprocedural. Por lo tanto, durante la mayor parte de este capítulo hablaremos sobre las técnicas necesarias para calcular los alias de apuntadores. Para empezar presentaremos Datalog, una notación que oculta en gran parte la complejidad de un análisis eficiente de apuntadores. Después describiremos un algoritmo para este análisis, y mostraremos cómo utilizar la abstracción de los diagramas de decisiones binarias (BDDs) para implementar el algoritmo con eficiencia.

La mayoría de las optimizaciones de los compiladores, incluyendo las descritas en los capítulos 9, 10 y 11, se realizan sobre los procedimientos, uno a la vez. A dicho análisis se le conoce como *intraprocedural*. Estos análisis suponen de manera conservadora que los procedimientos invocados pueden alterar el estado de todas las variables visibles para los procedimientos y que pueden crear todos los efectos adicionales posibles, como la modificación de cualquiera de las variables visibles para el procedimiento, o la generación de excepciones que provoquen la limpieza de la pila (stack unwinding) de llamadas. Por ende, el análisis intraprocedural es bastante simple, aunque impreciso. Algunas optimizaciones no necesitan del análisis interprocedural, mientras que otras casi no pueden producir información útil sin él.

Un análisis interprocedural opera a través de un programa completo, haciendo que fluya información del emisor a los procesos llamados, y viceversa. Una técnica bastante simple pero útil es la de poner los procedimientos *en línea*; es decir, sustituir la invocación de un procedimiento por el cuerpo del mismo procedimiento, con modificaciones adecuadas para tomar en cuenta el paso de los parámetros y el valor de retorno. Este método se aplica sólo si conocemos el destino de la llamada al procedimiento.

Si los procedimientos se invocan de manera indirecta a través de un apuntador, o mediante el mecanismo despachador de métodos prevaleciente en la programación orientada a objetos,

el análisis de los apuntadores o referencias del programa puede, en algunos casos, determinar los destinos de las invocaciones indirectas. Si hay un destino único, pueden aplicarse los procedimientos en línea.

Aun si se determina un destino único para cada invocación a un procedimiento, el uso de los procedimientos en línea debe aplicarse con prudencia. En general, no es posible poner en línea los procedimientos recursivos de manera directa, e inclusive sin la recursividad, el uso de procedimientos en línea puede expandir el código en forma exponencial.

12.1 Conceptos básicos

En esta sección presentaremos los grafos de llamadas: grafos que nos indican qué procedimientos pueden llamar a qué otros procedimientos. También expondremos la idea de “sensibilidad al contexto”, en donde se requieren análisis de flujo de datos para conocer cuál ha sido la secuencia de llamadas a procedimientos. Es decir, el análisis sensible al contexto incluye (una sinopsis de) la secuencia actual de los registros de activación en la pila, junto con el punto actual en el programa, cuando hay que diferenciar entre distintos “lugares” en el programa.

12.1.1 Grafos de llamadas

Un *grafo de llamadas* para un programa es un conjunto de nodos y aristas de manera que:

1. Hay un nodo para cada procedimiento en el programa.
2. Hay un nodo para cada *sitio de llamada*; es decir, un lugar en el programa en donde se invoca a un procedimiento.
3. Si el sitio de llamada *c* puede llamar al procedimiento *p*, entonces hay una arista que va del nodo para *c* hasta el nodo para *p*.

Muchos programas escritos en lenguajes como C y Fortran hacen las llamadas a los procedimientos de manera directa, por lo que el destino de llamada de cada invocación puede determinarse en forma estática. En ese caso, cada sitio de llamada tiene una arista que va a un solo procedimiento en el grafo de llamadas. No obstante, si el programa incluye el uso de un parámetro de procedimiento o apuntador de función, por lo general, no se conoce el destino sino hasta que se ejecuta el programa y, de hecho, puede variar de una invocación a otra. Entonces, un sitio de llamada puede indicar enlaces que van a muchos o a todos los procedimientos en el grafo de llamadas.

Las llamadas indirectas son la norma para los lenguajes de programación orientados a objetos. En especial, cuando hay redefinición de métodos en las subclases, un uso del método *m* puede referirse a cualquiera de varios métodos distintos, dependiendo de la subclase del objeto receptor al cuál se aplicó. El uso de dichas invocaciones a métodos *virtuales* significa que debemos conocer el tipo del receptor antes de poder determinar qué método se invocó.

Ejemplo 12.1: La figura 12.1 muestra un programa en C que declara a *pf* como un apuntador global a una función cuyo tipo es “de entero a entero”. Hay dos funciones de este tipo, *fun1* y *fun2*, y una función principal que no es del tipo al que *pf* apunta. La figura muestra tres sitios de llamada, denotados como *c1*, *c2* y *c3*; las etiquetas no forman parte del programa.

```

int (*pf)(int);

int fun1(int x) {
    if (x < 10)
        return (*pf)(x+1);
    else
        return x;
}

int fun2(int y) {
    pf = &fun1;
}
c2:    return (*pf)(y);
}

void main() {
    pf = &fun2;
c3:    (*pf)(5);
}

```

Figura 12.1: Un programa con un apuntador de función

El análisis más simple del objeto al que `pf` podría apuntar sólo observaría los tipos de las funciones. Las funciones `fun1` y `fun2` son del mismo tipo al que `pf` apunta, mientras que `main` no. Por ende, en la figura 12.2(a) se muestra un grafo de llamadas conservador. Un análisis más cuidadoso del programa observaría que se hace que `pf` apunte a `fun2` en `main`, y que apunte a `fun1` en `fun2`. Pero no hay otras asignaciones para ningún apuntador, por lo que, en especial, no hay forma de que `pf` apunte a `main`. Este razonamiento produce el mismo grafo de llamadas que el de la figura 12.2(a).

Un análisis aún más preciso indicaría que en `c3` sólo es posible que `pf` apunte a `fun2`, ya que esa llamada va precedida de inmediato por esa asignación a `pf`. De manera similar, en `c2` sólo es posible que `pf` apunte a `fun1`. Como resultado, la llamada inicial a `fun1` sólo puede provenir de `fun2`, y `fun1` no modifica a `pf`, por lo que cada vez que estamos dentro de `fun1`, `pf` apunta a `fun1`. En especial, en `c1` podemos estar seguros de que `pf` apunta a `fun1`. Así, la figura 12.2(b) es un grafo de llamadas correcto y más preciso. \square

En general, la presencia de referencias o apuntadores a funciones o métodos requiere que obtengamos una aproximación estática de los valores potenciales de todos los parámetros de procedimientos, apuntadores de funciones y tipos de objetos receptores. Para realizar una aproximación precisa, es necesario el análisis interprocedural. El análisis es iterativo, y empieza con los destinos observables de manera estática. A medida que se descubren más destinos, el análisis incorpora las nuevas aristas en el grafo de llamadas y repite el descubrimiento de más destinos hasta que se llega a una convergencia.

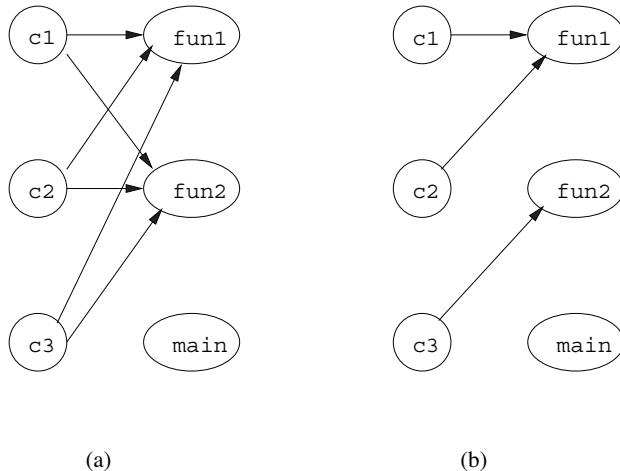


Figura 12.2: Grafos de llamadas derivados de la figura 12.1

12.1.2 Sensibilidad al contexto

El análisis interprocedural es retador, ya que el comportamiento de cada procedimiento depende del contexto en el que se llame. El ejemplo 12.2 utiliza el problema de la propagación constante interprocedural en un pequeño programa, para ilustrar la importancia de los contextos.

Ejemplo 12.2: Considere el fragmento de programa en la figura 12.3. La función f se invoca en tres sitios de llamada: $c1$, $c2$ y $c3$. La constante 0 se pasa como el parámetro actual en $c1$, y la constante 243 se pasa en $c2$ y $c3$ en cada iteración; se devuelven las constantes 1 y 244, respectivamente. Así, la función f se invoca con una constante en cada uno de los contextos, pero el valor de la constante es dependiente del contexto.

Como veremos más adelante, no es posible saber que a t_1 , t_2 y t_3 se les asignan valores constantes (por lo tanto a $X[i]$), a menos que reconozcamos que cuando se llama en el contexto c_1 , f devuelve 1, y cuando se llama en los otros dos contextos, f devuelve 244. Un análisis simple concluiría que f puede devolver 1 o 244 de cualquier llamada. \square

Un método simple pero bastante impreciso para el análisis interprocedural, conocido como *análisis insensible al contexto*, es tratar a cada instrucción de llamada y retorno como operaciones “*goto*”. Creamos un *súper* grafo de control de flujo en donde, además de las aristas de control de flujo intraprocedural normales, se crean aristas adicionales que conectan:

1. Cada sitio de llamada con el inicio del procedimiento al que llama.
 2. Las instrucciones de retorno que regresan a los sitios de llamada.¹

¹En realidad, el retorno es a la instrucción que sigue después del sitio de llamada.

```

        for (i = 0; i < n; i++) {
c1:          t1 = f(0);
c2:          t2 = f(243);
c3:          t3 = f(243);
          X[i] = t1+t2+t3;
}

int f (int v) {
    return (v+1);
}

```

Figura 12.3: Un fragmento de programa que ilustra la necesidad del análisis sensible al contexto

Se agregan instrucciones de asignación para asignar cada parámetro actual a su parámetro formal correspondiente, y para asignar el valor devuelto a la variable que recibe el resultado. Entonces podemos aplicar un análisis estándar que se pretende usar dentro de un procedimiento al súper grafo de control de flujo, para encontrar los resultados interprocedural insensibles al contexto. Aunque es simple, este modelo abstrae la importante relación entre los valores de entrada y salida en las invocaciones a procedimientos, lo cual hace que el análisis sea impreciso.

Ejemplo 12.3: El súper grafo de control de flujo para el programa en la figura 12.3 se muestra en la figura 12.4. El bloque B_6 es la función f . El bloque B_3 contiene el sitio de llamada $c1$; establece el parámetro formal v a 0 y después salta al inicio de f , en B_6 . De manera similar, B_4 y B_5 representan los sitios de llamada $c2$ y $c3$, respectivamente. En B_4 , al cual se llega desde el final de f (bloque B_6), tomamos el valor de retorno de f y lo asignamos a $t1$. Después establecemos el parámetro formal v a 243 y llamamos a f de nuevo, saltando a B_6 . Observe que no hay arista de B_3 a B_4 . El control debe fluir a través de f en el camino de B_3 a B_4 .

B_5 es similar a B_4 . Recibe el retorno de f , asigna el valor de retorno a $t2$ e inicia la tercera llamada a f . El bloque B_7 representa el retorno de la tercera llamada y la asignación a $X[i]$.

Si tratamos a la figura 12.4 como si fuera el grafo de flujo de un solo procedimiento, entonces concluiríamos que al llegar a B_6 , v puede tener el valor 0 o 243. Por ende, lo más que podemos concluir acerca de $valret$ es que se le asigna 1 o 244, pero ningún otro valor. De manera similar, sólo podemos concluir acerca de $t1$, $t2$ y $t3$ que pueden ser 1 o 244. Por ende, $X[i]$ parece ser 3, 246, 489 o 732. En contraste, un análisis sensible al contexto separaría los resultados para cada uno de los contextos de llamada y produciría la respuesta intuitiva descrita en el ejemplo 12.2: $t1$ siempre es 1, $t2$ y $t3$ siempre son 244, y $X[i]$ es 489. \square

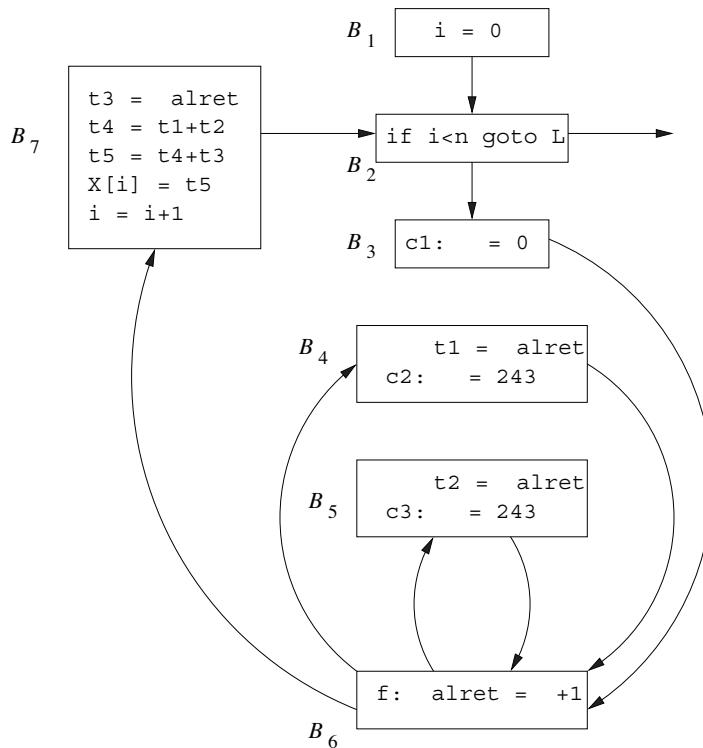


Figura 12.4: El grafo de control de flujo para la figura 12.3, en donde las llamadas a funciones se tratan como control de flujo

12.1.3 Cadenas de llamadas

En el ejemplo 12.2, podemos diferenciar entre los contextos con sólo conocer el sitio de llamada que llama al procedimiento f . En general, un contexto de llamada se define mediante el contenido de toda la pila de llamadas. A la cadena de sitios de llamada en la pila se le conoce como *cadena de llamadas*.

Ejemplo 12.4: La figura 12.5 es una ligera modificación de la figura 12.3. Aquí hemos sustituido las llamadas a f por llamadas a g , que a su vez llama a f con el mismo argumento. Hay un sitio de llamada adicional, $c4$, en donde g llama a f .

Hay tres cadenas de llamadas a f : $(c1, c4)$, $(c2, c4)$ y $(c3, c4)$. Como podemos ver en este ejemplo, el valor de v en la función f no depende del sitio inmediato o del último sitio $c4$ en la cadena de llamadas. En vez de ello, las constantes se determinan mediante el primer elemento en cada una de las cadenas de llamadas. \square

El ejemplo 12.4 ilustra que la información relevante para el análisis puede introducirse de manera anticipada en la cadena de llamadas. De hecho, algunas veces es necesario considerar

toda la cadena de llamadas completa para calcular la respuesta más precisa, como se ilustra en el ejemplo 12.5.

```

        for (i = 0; i < n; i++) {
c1:            t1 = g(0);
c2:            t2 = g(243);
c3:            t3 = g(243);
            X[i] = t1+t2+t3;
        }

        int g (int v) {
c4:            return f(v);
        }

        int f (int v) {
            return (v+1);
        }
    
```

Figura 12.5: Fragmento de programa que ilustra las cadenas de llamadas

```

        for (i = 0; i < n; i++) {
c1:            t1 = g(0);
c2:            t2 = g(243);
c3:            t3 = g(243);
            X[i] = t1+t2+t3;
        }

        int g (int v) {
            if (v > 1) {
c4:                return g(v-1);
            } else {
c5:                return f(v);
            }
        }

        int f (int v) {
            return (v+1);
        }
    
```

Figura 12.6: Programa recursivo que requiere el análisis de las cadenas de llamadas completas

Ejemplo 12.5: Este ejemplo ilustra cómo la habilidad para razonar acerca de las cadenas de llamadas sin límites puede producir resultados más precisos. En la figura 12.6 podemos ver que si g se llama con un valor positivo c , entonces g se invocará de manera recursiva c veces. Cada vez que se llama a g , el valor de su parámetro v disminuye por 1. Entonces, el valor del parámetro v de g en el contexto cuya cadena de llamadas es $c2(c4)^n$ es $234 - n$. Por ende, el efecto de g es incrementar 0 o cualquier argumento negativo por 1, y devolver 2 en cualquier argumento de 1 o mayor.

Hay tres posibles cadenas de llamadas para f . Si empezamos con la llamada en $c1$, entonces g llama a f de inmediato, por lo que $(c1, c5)$ es una de esas cadenas. Si empezamos en $c2$ o $c3$, entonces llamamos a g un total de 243 veces, y después llamamos a f . Estas cadenas de llamadas son $(c2, c4, c4, \dots, c5)$ y $(c3, c4, c4, \dots, c5)$, en donde en cada caso hay 242 $c4$ s en la secuencia. En el primero de estos contextos, el valor del parámetro v de f es 0, mientras que en los otros dos contextos es 1. \square

Al diseñar un análisis sensible al contexto, tenemos una elección en la precisión. Por ejemplo, en vez de calificar los resultados mediante la cadena de llamadas completa, podemos optar por sólo diferenciar un contexto de otro en base a sus k sitios de llamada más inmediatos. Esta técnica se conoce como análisis de contexto limitado por k . El análisis insensible al contexto es tan sólo un caso especial del análisis de contexto limitado por k , en donde k es 0. Podemos encontrar todas las constantes en el ejemplo 12.2, usando un análisis limitado por 1, y todas las constantes en el ejemplo 12.4 usando un análisis limitado por 2. Sin embargo, ningún análisis limitado por k puede encontrar todas las constantes en el ejemplo 12.5, siempre y cuando la constante 243 se sustituya por dos constantes distintas y arbitrariamente grandes.

En vez de elegir un valor fijo k , otra posibilidad es ser completamente sensible al contexto para todas las cadenas de llamadas *acíclicas*, que son cadenas que no contienen ciclos recursivos. Para las cadenas de llamadas con recursividad, podemos colapsar todos los ciclos recursivos para poder limitar la cantidad de los distintos contextos analizados. En el ejemplo 12.5, las llamadas iniciadas en el sitio de llamada $c2$ pueden aproximarse mediante la siguiente cadena de llamadas: $(c2, c4*, c5)$. Observe que, con este esquema, aun para los programas sin recursividad, el número de distintos contextos de llamada puede ser exponencial en el número de procedimientos en el programa.

12.1.4 Análisis sensible al contexto basado en la clonación

Otro método para el análisis sensible al contexto es clonar el procedimiento en forma conceptual, uno para cada contexto único de interés. Después podemos aplicar un análisis insensible al contexto al grafo de llamadas clonado. Los ejemplos 12.6 y 12.7 muestran el equivalente de una versión clonada de los ejemplos 12.4 y 12.5, respectivamente. En realidad no necesitamos clonar el código; tan sólo podemos usar una representación interna eficiente para llevar la cuenta de los resultados del análisis de cada clon.

Ejemplo 12.6: La versión clonada de la figura 12.5 se muestra en la figura 12.7. Como cada contexto de llamada se refiere a un clon distinto, no hay confusión. Por ejemplo, $g1$ recibe 0 como entrada y produce 1 como salida, y $g2$ y $g3$ reciben 243 como entrada y producen 244 como salida. \square

```

        for (i = 0; i < n; i++) {
c1:          t1 = g1(0);
c2:          t2 = g2(243);
c3:          t3 = g3(243);
          X[i] = t1+t2+t3;
        }
        int g1 (int v) {
c4.1:          return f1(v);
        }
        int g2 (int v) {
c4.2:          return f2(v);
        }
        int g3 (int v) {
c4.3:          return f3(v);
        }

        int f1 (int v) {
          return (v+1);
        }
        int f2 (int v) {
          return (v+1);
        }
        int f3 (int v) {
          return (v+1);
        }
    }

```

Figura 12.7: Versión clonada de la figura 12.5

Ejemplo 12.7: La versión clonada del ejemplo 12.5 se muestra en la figura 12.8. Para el procedimiento g , creamos un clon para representar todas las instancias de g que se llaman por primera vez de los sitios $c1$, $c2$ y $c3$. En este caso, el análisis determinaría que la invocación en el sitio de llamada $c1$ devuelve 1, suponiendo que el análisis puede deducir que con $v = 0$, la prueba $v > 1$ falla. Sin embargo, este análisis no maneja la recursividad lo bastante bien como para producir las constantes para los sitios de llamada $c2$ y $c3$. \square

12.1.5 Análisis sensible al contexto basado en el resumen

El análisis interprocedural basado en el resumen es una extensión del análisis basado en regiones. Básicamente, en un análisis basado en el resumen cada procedimiento se representa mediante una descripción concisa (“resumen”) que encapsula cierto comportamiento observable del procedimiento. El propósito principal del resumen es evitar reanalizar el cuerpo de un procedimiento en cada sitio de llamada que pueda invocar al procedimiento.

Primero vamos a considerar el caso en donde no hay recursividad. Cada procedimiento se modela como una región con un solo punto de entrada, en donde cada par emisor-receptor

```
        for (i = 0; i < n; i++) {
c1:            t1 = g1(0);
c2:            t2 = g2(243);
c3:            t3 = g3(243);
            X[i] = t1+t2+t3;
        }

        int g1 (int v) {
            if (v > 1) {
c4.1:                return g1(v-1);
            } else {
c5.1:                return f1(v);
            }

            int g2 (int v) {
                if (v > 1) {
c4.2:                    return g2(v-1);
                } else {
c5.2:                    return f2(v);
                }

                int g3 (int v) {
                    if (v > 1) {
c4.3:                        return g3(v-1);
                    } else {
c5.3:                        return f3(v);
                    }

                    int f1 (int v) {
                        return (v+1);
                    }
                    int f2 (int v) {
                        return (v+1);
                    }
                    int f3 (int v) {
                        return (v+1);
                    }
                }
            }
        }
    }
```

Figura 12.8: Versión clonada de la figura 12.6

comparte una relación de región exterior-interior. La única diferencia en comparación con la versión intraprocedural es que, en el caso interprocedural, la región de un procedimiento puede anidarse dentro de varias regiones externas distintas.

El análisis consiste en dos partes:

1. Una fase ascendente, que calcula una función de transferencia para resumir el efecto de un procedimiento.
2. Una fase descendente, que propaga la información del emisor para calcular los resultados de los receptores.

Para obtener resultados completamente sensibles al contexto, la información de los distintos contextos de llamada debe propagarse hacia abajo, a los receptores, en forma individual. Para un cálculo más eficiente pero menos preciso, puede combinarse la información de todos los emisores, usando un operador de reunión, y después propagarse hacia abajo, a los receptores.

Ejemplo 12.8: Para una propagación de constantes, cada procedimiento se resume mediante una función de transferencia que especifica cómo propagaría las constantes a través de su cuerpo. En el ejemplo 12.2, podemos resumir f como una función que, dada una constante c como un parámetro actual para v , devuelve la constante $c+1$. Con base en esta información, el análisis determinaría que t_1 , t_2 y t_3 tienen los valores constantes 1, 244 y 244, respectivamente. Observe que este análisis no sufre la imprecisión debido a las cadenas de llamadas inalcanzables.

Recuerde que el ejemplo 12.4 extiende al ejemplo 12.2 al hacer que g llame a f . Por ende, podríamos concluir que la función de transferencia para g es igual que la función de transferencia para f . De nuevo concluimos que t_1 , t_2 y t_3 tienen los valores constantes 1, 244 y 244, respectivamente.

Ahora, vamos a considerar cuál es el valor del parámetro v en la función f para el ejemplo 12.2. Como primera parte, podemos combinar todos los resultados para todos los contextos de llamada. Como v puede tener los valores 0 o 243, podemos simplemente concluir que v no es una constante. Esta conclusión es justa, ya que no hay constante que pueda sustituir a v en el código.

Si deseamos resultados más precisos, podemos calcular resultados específicos para los contextos de interés. La información debe pasarse hacia abajo, desde el contexto de interés para determinar la respuesta sensible al contexto. Este paso es similar a la pasada descendente en el análisis basado en regiones. Por ejemplo, el valor de v es 0 en el sitio de llamada c_1 , y 243 en los sitios c_2 y c_3 . Para obtener la ventaja de la propagación de constantes dentro de f , debemos capturar esta distinción mediante la creación de dos clones, en donde el primero está especializado para el valor de entrada 0 y el segundo con el valor 243, como se muestra en la figura 12.9. \square

Con el ejemplo 12.8 podemos ver que, al final, si deseamos compilar el código de manera distinta en los distintos contextos, de todas formas tenemos que clonar el código. La diferencia es que en el método basado en la clonación, ésta se realiza antes del análisis, con base en las cadenas de llamadas. En el método basado en el resumen, la clonación se realiza después del análisis, usando los resultados del mismo como base.

```

        for (i = 0; i < n; i++) {
c1:      t1 = f0(0);
c2:      t2 = f243(243);
c3:      t3 = f243(243);
        X[i] = t1+t2+t3;
    }

    int f0 (int v) {
        return (1);
    }

    int f243 (int v) {
        return (244);
    }

```

Figura 12.9: Resultado de propagar todos los posibles argumentos constantes a la función *f*

Aun si no se aplica la clonación, en el método basado en el resumen las inferencias acerca del efecto de un procedimiento llamado se hacen con precisión, sin el problema de los caminos no realizables.

En vez de clonar una función, también podríamos poner el código en línea. Esto tiene el efecto adicional de eliminar la sobrecarga de las llamadas a los procedimientos también.

Podemos manejar la recursividad calculando la solución de punto fijo. En presencia de la recursividad, primero buscamos los componentes fuertemente conectados en el grafo de llamadas. En la fase de abajo hacia arriba, no visitamos un componente fuertemente conectado a menos que se hayan visitado todos sus sucesores. Para un componente fuertemente conectado que no sea trivial, calculamos en forma iterativa las funciones de transferencia para cada procedimiento en el componente hasta llegar a una convergencia; es decir, actualizamos en forma iterativa las funciones de transferencia hasta que no ocurran más cambios.

12.1.6 Ejercicios para la sección 12.1

Ejercicio 12.1.1: En la figura 12.10 hay un programa en C con dos apuntadores a funciones, *p* y *q*. *N* es una constante que podría ser menor o mayor que 10. Observe que el programa resulta en una secuencia infinita de llamadas, pero eso no es motivo de preocupación para los fines de este problema.

- Identifique todos los sitios de llamada en este programa.
- Para cada sitio de llamada, ¿a dónde puede apuntar *p*? ¿A dónde puede apuntar *q*?
- Dibuje el grafo de llamadas para este programa.
- ! d) Describa todas las cadenas de llamadas para *f* y *g*.

```

int (*p)(int);
int (*q)(int);

int f(int i) {
    if (i < 10)
        {p = &g; return (*q)(i);}
    else
        {p = &f; return (*p)(i);}
}

int g(int j) {
    if (j < 10)
        {q = &f; return (*p)(j);}
    else
        {q = &g; return (*q)(j);}
}

void main() {
    p = &f;
    q = &g;
    (*p)((*q)(N));
}

```

Figura 12.10: Programa para el ejercicio 12.1.1

Ejercicio 12.1.2: En la figura 12.11 hay una función `id` que viene siendo la “función identidad”; devuelve exactamente lo que recibe como argumento. También vemos un fragmento de código que consiste en una bifurcación, a la cual le sigue una asignación que suma $x + y$.

- Al examinar el código, ¿qué podemos averiguar acerca del valor de z al final?
- Construya el grafo de flujo para el fragmento de código, tratando las llamadas a `id` como control de flujo.
- Si ejecutamos un análisis de propagación de constantes, como en la sección 9.4, en su grafo de flujo del inciso (b), ¿qué valores constantes se determinan?
- ¿Cuáles son todos los sitios de llamada en la figura 12.11?
- ¿Cuáles son todos los contextos en los que se llama `id`?
- Rescriba el código de la figura 12.11, clonando una nueva versión de `id` para cada contexto en el que se llame.
- Construya el grafo de flujo de su código de (f), tratando las llamadas como control de flujo.

```

int id(int x) { return x;}

...
if (a == 1) { x = id(2); y = id(3); }
else         { x = id(3); y = id(2); }
z = x+y;
...

```

Figura 12.11: Fragmento de código para el ejercicio 12.12

- h) Realice un análisis de propagación de constantes en su grafo de flujo de (g). ¿Qué valores constantes se determinan ahora?

12.2 ¿Por qué análisis interprocedural?

Debido a la dificultad del análisis interprocedural, vamos ahora a señalar el problema importante acerca de cuándo y cómo es conveniente usar este análisis. Aunque utilizamos la propagación de constantes para ilustrar el análisis interprocedural, esta optimización interprocedural no está disponible con facilidad ni es muy beneficiosa cuando ocurre. La mayoría de los beneficios de la propagación de constantes se pueden obtener con sólo realizar un análisis intraprocedural y poner en línea las llamadas a procedimientos de las secciones de código que se ejecutan con más frecuencia.

Sin embargo, existen muchas razones por las que el análisis interprocedural es esencial. A continuación describiremos varias aplicaciones importantes de este análisis.

12.2.1 Invocación de métodos virtuales

Como dijimos antes, los programas orientados a objeto tienen muchos métodos pequeños. Si sólo optimizamos un método a la vez, entonces hay pocas oportunidades de optimización. Al resolver la invocación a un método se habilita la optimización. Un lenguaje como Java carga sus clases en forma dinámica. Como resultado, no sabemos en tiempo de compilación a cuáles de los (tal vez) muchos métodos llamados *m* hace referencia un uso de “*m*” en una invocación como *x.m()*.

Muchas implementaciones de Java utilizan un compilador “just-in-time” para compilar sus bytecodes en tiempo de ejecución. Una optimización común es perfilar la ejecución y determinar cuáles son los tipos comunes de receptores. Así, podemos poner en línea los métodos que se invocan con más frecuencia. El código incluye una comprobación dinámica sobre el tipo y ejecuta los métodos en línea si el objeto en tiempo de ejecución tiene el tipo esperado.

Es posible otro método para resolver los usos del nombre *m* de un método, siempre y cuando todo el código fuente esté disponible en tiempo de compilación. Entonces, es posible realizar un análisis interprocedural para determinar los tipos de los objetos. Si el tipo para una variable *x* resulta ser único, entonces puede resolverse un uso de *x.m()*.

Sabemos con exactitud a qué método hace referencia en este contexto. En ese caso, podemos poner el código en línea para este método m , y el compilador ni siquiera tiene que incluir una prueba para el tipo de x .

12.2.2 Análisis de alias de apuntadores

Aun si no deseamos ejecutar las versiones interprocedurales de los análisis comunes de flujos de datos, como las definiciones de alcance, estos análisis pueden de hecho beneficiarse del análisis de apuntadores interprocedural. Todos los análisis presentados en el capítulo 9 se aplican sólo a las variables escalares locales que no pueden tener alias. Sin embargo, es común el uso de apuntadores, en especial en lenguajes como C. Al saber si los apuntadores pueden ser *alias* (si pueden apuntar a la misma ubicación) podemos mejorar la precisión de las técnicas del capítulo 9.

Ejemplo 12.9: Considere la siguiente secuencia de tres instrucciones, que podrían formar un bloque básico:

```
*p = 1;
*q = 2;
x = *p;
```

Sin saber si p y q pueden apuntar a la misma ubicación (es decir, si pueden o no ser alias) no podemos concluir que x es igual a 1 al final del bloque. \square

12.2.3 Paralelización

Como vimos en el capítulo 11, la manera más efectiva de paralelizar una aplicación es buscar la granularidad más gruesa del paralelismo, como la que se encuentra en los ciclos más externos de un programa. Para esta tarea, el análisis interprocedural es de gran importancia. Existe una diferencia considerable entre las optimizaciones *escalares* (las que se basan en valores de variables simples, como vimos en el capítulo 9) y la paralelización. En la paralelización, sólo una dependencia de datos espuria puede hacer que todo un ciclo completo no sea paralelizable, y reducir en gran parte la efectividad de la optimización. Dicha amplificación de imprecisiones no se ve en las optimizaciones escalares. En la optimización escalar, sólo debemos buscar la mayoría de las oportunidades de optimización. Pasar por alto una o dos oportunidades pocas veces hace la gran diferencia.

12.2.4 Detección de errores de software y vulnerabilidades

El análisis interprocedural no sólo es importante para optimizar código. Pueden usarse las mismas técnicas en el análisis del software existente para muchos tipos de errores de codificación. Estos errores pueden hacer que el software sea desconfiable; los errores de codificación que los hackers pueden explotar para tomar el control de (o de alguna otra forma dañar a) un sistema computacional pueden representar riesgos importantes de vulnerabilidad en la seguridad.

El análisis estático es útil para detectar ocurrencias de muchos patrones de error comunes. Por ejemplo, un elemento de datos debe protegerse mediante un candado. Como otro ejemplo, la deshabilitación de una interrupción en el sistema operativo debe ir seguida de una rehabilitación de la interrupción. Como las inconsistencias que abarcan los límites de los procedimientos son una fuente considerable de errores, el análisis interprocedural es de gran importancia. Prefix y Metal son dos herramientas prácticas que utilizan el análisis interprocedural con efectividad para buscar muchos errores de programación en programas extensos. Dichas herramientas buscan errores en forma estática y pueden mejorar de manera considerable la confiabilidad del software. Sin embargo, estas herramientas son tanto incompletas como poco sólidas, en el sentido de que tal vez no encuentren todos los errores, y no todas las advertencias reportadas son errores reales. Por desgracia, para reportar todos los errores potenciales el gran número de advertencias falsas convertiría a las herramientas en algo inutilizable. Sin embargo, aun cuando estas herramientas no son perfectas, su uso sistemático ha demostrado mejorar en gran medida la confiabilidad del software.

Al tratarse de vulnerabilidades en la seguridad, es muy conveniente que encontremos todos los errores potenciales en un programa. En el año 2006, dos de las formas “más populares” de intrusiones que utilizaron los hackers para poner en peligro un sistema fueron:

1. La falta de validación de entrada en las aplicaciones Web: la inyección de código SQL es una de las formas más populares de dicha vulnerabilidad, en donde los hackers obtienen el control de una base de datos al manipular las entradas que aceptan las aplicaciones Web.
2. Los desbordamientos de búfer en los programas en C y C++. Como C y C++ no comproban si los accesos a los arreglos están dentro de los límites, los hackers pueden escribir cadenas bien elaboradas en áreas no planeadas y así obtener el control de la ejecución del programa.

En la siguiente sección veremos cómo podemos usar el análisis interprocedural para proteger los programas contra dichas vulnerabilidades.

12.2.5 Inyección de código SQL

La inyección de código SQL se refiere a la vulnerabilidad en la que los hackers pueden manipular la entrada del usuario para una aplicación Web y obtener acceso no planeado a la base de datos. Por ejemplo, los bancos desean que sus usuarios puedan hacer transacciones en línea, siempre y cuando suministren su contraseña correcta. Una arquitectura común para dicho sistema es hacer que el usuario introduzca cadenas en un formulario Web, y después hacer que esas cadenas formen parte de una consulta de base de datos escrita en el lenguaje SQL. Si los desarrolladores de sistemas no son cuidadosos, las cadenas que proporciona el usuario pueden alterar el significado de la instrucción SQL en formas inesperadas.

Ejemplo 12.10: Suponga que un banco ofrece a sus clientes acceso a una relación como:

```
DatosCta(nombre, contrasenia, saldo)
```

Es decir, esta relación es una tabla de tripletas, cada una de las cuales consiste en el nombre de un cliente, la contraseña y el saldo de la cuenta. La intención es que los clientes puedan ver

el saldo de su cuenta, sólo si pueden proporcionar su nombre y contraseña correctos. Que un hacker vea el saldo de una cuenta no es lo peor que podría ocurrir, pero este ejemplo simple es típico de las situaciones más complicadas, en donde el hacker podría realizar pagos desde la cuenta.

El sistema podría implementar una consulta de saldo de la siguiente manera:

1. Los usuarios invocan a un formulario Web, en el cual pueden introducir su nombre y contraseña.
2. El nombre se copia a una variable *n* y la contraseña a una variable *p*.
3. Más adelante, tal vez en algún otro procedimiento, se ejecuta la siguiente secuencia SQL:

```
SELECT saldo FROM DatosCta
WHERE nombre = ':n' and contrasenia = ':p'
```

Para los lectores que no estén familiarizados con SQL, esta consulta dice: “Buscar en la tabla *DatosCta* una fila con el primer componente (nombre) igual a la cadena actual en la variable *n* y el segundo componente (contraseña) igual a la cadena actual en la variable *p*; imprimir el tercer componente (saldo) de esa fila”. Observe que SQL utiliza comillas sencillas, no dobles, para delimitar cadenas, y que los signos de punto y coma en frente de *n* y *p* indican que son variables del lenguaje circundante.

Suponga que el hacker, que desea buscar el saldo de la cuenta de Charles Dickens, suministra los siguientes valores para las cadenas *n* y *p*:

```
n = Charles Dickens' --      p = que importa
```

El efecto de estas extrañas cadenas es convertir la consulta en lo siguiente:

```
SELECT saldo FROM DatosCta
WHERE nombre = 'Charles Dickens' --' and contrasenia = 'que importa'
```

En muchos sistemas de bases de datos, los símbolos *--* son un token para introducir comentarios, y tienen el efecto de convertir en comentario lo que vaya después en esa línea. Como resultado, la consulta ahora pide al sistema de base de datos que imprima el saldo para cada persona cuyo nombre sea *'Charles Dickens'*, sin importar la contraseña que aparezca con ese nombre en un registro formado por las variables nombre-contraseña-saldo. Es decir, eliminando los comentarios la consulta sería:

```
SELECT saldo FROM DatosCta
WHERE nombre = 'Charles Dickens'
```

□

En el ejemplo 12.10, las cadenas “malas” se mantienen en dos variables, que pueden pasarse entre un procedimiento y otro. Sin embargo, en casos más realistas, estas cadenas podrían copiarse varias veces, o combinarse con otras para formar la consulta completa. No podemos esperar detectar los errores de codificación que crean vulnerabilidades de inyección de código SQL sin realizar un análisis interprocedural completo de todo el programa.

12.2.6 Desbordamiento de búfer

Un *ataque por desbordamiento de búfer* ocurre cuando los datos cuidadosamente planeados que suministra el usuario escriben más allá del búfer destinado, y manipulan la ejecución del programa. Por ejemplo, un programa en C podría leer una cadena *s* del usuario, y después copiarla en un búfer *b*, mediante la siguiente llamada a una función:

```
strcpy(b,s);
```

Si la cadena *s* es más larga que el búfer *b*, entonces se modificarán los valores de las ubicaciones que no sean parte de *b*. Es probable que eso en sí haga que el programa funcione de manera incorrecta, o que por lo menos produzca la respuesta incorrecta, ya que se habrán modificado ciertos datos que utiliza el programa.

O peor aún, el hacker que seleccionó la cadena *s* puede elegir un valor que haga algo más que provocar un error. Por ejemplo, si el búfer está en la pila en tiempo de ejecución, entonces está cerca de la dirección de retorno para su función. Un valor de *s* elegido con dolo puede sobrescribir la dirección de retorno, y cuando la función regrese, irá a un lugar elegido por el hacker. Si los hackers tienen un conocimiento detallado del sistema operativo y el hardware del entorno, pueden llegar a ejecutar un comando que les otorgue el control de la máquina. En ciertas situaciones, incluso pueden tener la habilidad de hacer que la dirección de retorno falsa transfiera el control a cierto código que forma parte de la cadena *s*, con lo cual se permite la inserción de cualquier tipo de programa en el código en ejecución.

Para evitar desbordamientos de búfer, hay que demostrar en forma estática que cada operación de escritura de arreglos se encuentra dentro de los límites, o debe realizarse en forma dinámica una comprobación apropiada de los límites del arreglo. Como estas comprobaciones de los límites tienen que insertarse a mano en los programas en C y C++, es fácil olvidar insertar la prueba o hacerla mal. Se han desarrollado herramientas de heurística para comprobar si por lo menos cierta prueba, aunque no sea necesariamente correcta, se ha realizado antes de llamar a una instrucción `strcpy`.

La comprobación dinámica de los límites es inevitable, ya que es imposible determinar en forma estática el tamaño de la entrada del usuario. Todo lo que puede hacer un análisis estático es asegurar que las comprobaciones dinámicas se hayan insertado en forma apropiada. Por ende, una estrategia razonable es hacer que el compilador inserte la comprobación dinámica de los límites en cada operación de escritura, y usar el análisis estático como medio para optimizar todas las comprobaciones de límites que sean posibles. Ya no es necesario atrapar todas las potenciales violaciones; además, sólo debemos optimizar aquellas regiones de código que se ejecutan con frecuencia.

Insertar la comprobación de límites en programas en C no es algo trivial, incluso si no nos importa el costo. Un apuntador podría apuntar a la parte media de cierto arreglo, sin que nosotros conociéramos la extensión de ese arreglo. Se han desarrollado técnicas para llevar el registro de la extensión del búfer al que apunta cada apuntador en forma dinámica. Esta información permite al compilador insertar comprobaciones de límites de arreglos para todos los accesos. Algo muy importante es que no es conveniente detener un programa cada vez que se detecta un desbordamiento de búfer. De hecho, estos desbordamientos ocurren en la práctica, y es probable que un programa falle si deshabilitamos todos los desbordamientos de búfer. La solución es extender el tamaño del arreglo en forma dinámica para darles cabida.

Podemos usar el análisis interprocedural para agilizar el costo de las comprobaciones de los límites de los arreglos dinámicos. Por ejemplo, suponga que sólo nos interesa atrapar los desbordamientos de búfer que involucran a las cadenas de entrada del usuario; podemos usar el análisis estático para determinar qué variables pueden guardar el contenido que proporciona el usuario. Al igual que la inyección de código SQL, es útil poder rastrear una entrada a medida que se copia a través de los procedimientos para eliminar las comprobaciones de límites innecesarias.

12.3 Una representación lógica del flujo de datos

Hasta este momento, nuestra representación de los problemas y soluciones del flujo de datos puede determinarse como “teoría de conjuntos”. Es decir, representamos la información como conjuntos y calculamos los resultados usando operadores como la unión y la intersección. Por ejemplo, cuando presentamos el problema de las definiciones de alcance en la sección 9.2.4, calculamos $\text{ENT}[B]$ y $\text{SAL}[B]$ para un bloque B , y los describimos como conjuntos de definiciones. Representamos el contenido del bloque B mediante sus conjuntos gen y eliminar.

Para lidiar con la complejidad del análisis interprocedural, ahora vamos a presentar una notación más general y concisa basada en la lógica. En vez de decir algo como “la definición D está en $\text{ENT}[B]$ ”, vamos a usar una notación como $\text{ent}(B, D)$ para indicar lo mismo. Al hacer esto, podemos expresar “reglas” concisas acerca de inferir los hechos del programa. También nos permite implementar estas reglas con eficiencia, de una forma que generalice el método de vector de bits para las operaciones teóricas de conjuntos. Por último, el método lógico nos permite combinar lo que parecen ser varios análisis independientes en un algoritmo integrado. Por ejemplo, en la sección 9.5 describimos la eliminación de la redundancia parcial mediante una secuencia de cuatro análisis de flujo de datos y otros dos pasos intermedios. En la notación lógica, todos estos pasos podrían combinarse en una colección de reglas lógicas que se resuelven al mismo tiempo.

12.3.1 Introducción a Datalog

Datalog es un lenguaje que utiliza una notación parecida a Prolog, pero cuya semántica es mucho más simple. Para empezar, los elementos de Datalog son *átomos* de la forma $p(X_1, X_2, \dots, X_n)$. Aquí,

1. p es un *predicado*: un símbolo que representa a un tipo de instrucción como “una definición llega al inicio de un bloque”.
2. X_1, X_2, \dots, X_n son términos como variables o constantes. También debemos permitir expresiones simples como argumentos de un predicado.²

Un *átomo base* (*ground atom*) es un predicado que sólo tiene constantes como argumentos. Todo átomo base afirma un hecho específico, y su valor es verdadero o falso. A menudo

²De manera formal, dichos términos se crean a partir de símbolos de funciones y complican la implementación de Datalog de manera considerable. Sin embargo, sólo vamos a usar algunos operadores, como la suma o resta de constantes, en contextos que no compliquen las cosas.

es conveniente representar un predicado mediante una *relación*, o tabla de sus átomos base verdaderos. Cada átomo base se representa mediante una sola fila, o *tupla*, de la relación. Las columnas de la relación se nombran mediante *atributos*, y cada tupla tiene un componente para cada atributo. Los atributos corresponden a los componentes de los átomos base representados por la relación. Cualquier átomo base en la relación es verdadero, y los átomos base que no están en la relación son falsos.

Ejemplo 12.11: Vamos a suponer que el predicado $ent(B, D)$ significa “la definición D llega al inicio del bloque B ”. Entonces podríamos suponer que, para un grafo de flujo específico, $ent(b_1, d_1)$ es verdadero, al igual que $ent(b_2, d_1)$ y $ent(b_2, d_2)$. También podríamos suponer que para este grafo de flujo, todos los demás hechos ent son falsos. Entonces, la relación en la figura 12.12 representa el valor de este predicado para este grafo de flujo.

B	D
b_1	d_1
b_2	d_1
b_2	d_2

Figura 12.12: Representación del valor de un predicado por una relación

Los atributos de la relación son B y D . Las tres tuplas de la relación son (b_1, d_1) , (b_2, d_1) y (b_2, d_2) . \square

También veremos a veces un átomo, que en realidad es una comparación entre variables y constantes. Un ejemplo sería $X \neq Y$ o $X = 10$. En estos ejemplos, el predicado es en realidad el operador de comparación. Es decir, podemos considerar que $X = 10$ está escrita en forma de predicado: *igual*(X , 10). Sin embargo, hay una diferencia importante entre los predicados de comparación y los demás. Un predicado de comparación tiene su interpretación estándar, mientras que un predicado ordinario como ent sólo significa lo que está definido para significar mediante un programa Datalog (que describiremos a continuación).

Una *literal* es un átomo o un átomo negado. Indicamos la negación con la palabra NOT en frente del átomo. Por ende, NOT $ent(B, D)$ es una afirmación de que la definición D no llega al inicio del bloque B .

12.3.2 Reglas de Datalog

Las reglas son una forma de expresar las inferencias locales. En Datalog, las reglas también sirven para sugerir cómo debe llevarse a cabo un cálculo de los hechos verdaderos. La forma de una regla es:

$$H :- B_1 \ \& \ B_2 \ \& \ \dots \ \& \ B_n$$

Los componentes son los siguientes:

- H y B_1, B_2, \dots, B_n son literales; ya sea átomos o átomos negados.

Convenciones de Datalog

Vamos a utilizar las siguientes convenciones para los programas en Datalog:

1. Las variables empiezan con letra mayúscula.
2. Todos los demás elementos empiezan con letras minúsculas u otros símbolos como los dígitos. Estos elementos incluyen predicados y constantes que son argumentos de predicados.

- H es el *encabezado* y B_1, B_2, \dots, B_n forman el *cuerpo* de la regla.
- A cada una de las B_i 's se le conoce algunas veces como una *submeta* de la regla.

Debemos leer el símbolo `:-` como “si”. El significado de una regla es “el encabezado es verdadero si el cuerpo es verdadero”. Dicho en forma más precisa, *aplicamos* una regla a un conjunto dado de átomos base como se muestra a continuación. Considere todas las posibles sustituciones de constantes por las variables de la regla. Si esta sustitución hace verdadera cada una de las submetas del cuerpo (suponiendo que todos y sólo los átomos base dados sean verdaderos), entonces podemos inferir que el encabezado con esta sustitución de constantes por variables es un hecho verdadero. Las sustituciones que no hacen verdaderas a todas las submetas no nos proporcionan información; el encabezado puede o no ser verdadero.

Un *programa en Datalog* es una colección de reglas. Este programa se aplica a los “datos”; es decir, a un conjunto de átomos base para algunos de los predicados. El resultado del programa es el conjunto de átomos base inferidos mediante la aplicación de las reglas, hasta que no puedan hacerse más inferencias.

Ejemplo 12.12: Un ejemplo simple de un programa en Datalog es el cálculo de los caminos en un grafo, dadas sus aristas (dirigidas). Es decir, hay un predicado *arista*(X, Y) que significa “hay una arista que va del nodo X al nodo Y ”. Otro predicado *camino*(X, Y) significa que hay un camino que va de X a Y . Las reglas que definen los caminos son:

- 1) $\text{camino}(X, Y) \ :- \text{arista}(X, Y)$
- 2) $\text{camino}(X, Y) \ :- \text{camino}(X, Z) \ \& \ \text{camino}(Z, Y)$

La primera regla dice que una arista individual es un camino. Es decir, cada vez que sustituimos la variable X por una constante a y la variable Y por una constante b , y *arista*(a, b) es verdadero (es decir, hay una arista que va del nodo a al nodo b), entonces *camino*(a, b) también es verdadero (es decir, hay un camino que va de a a b). La segunda regla dice que si hay un camino que va de cierto nodo X a cierto nodo Z , y también hay un camino que va de Z al nodo Y , entonces hay un camino que va de X a Y . Esta regla expresa el “cierre transitivo”. Observe que puede formarse cualquier camino si tomamos las aristas a lo largo del camino y aplicamos la regla del cierre transitivo en forma repetida.

Por ejemplo, suponga que los siguientes hechos (átomos base) son verdaderos: *arsita*(1, 2), *arista*(2, 3) y *arista*(3, 4). Entonces, podemos usar la primera regla con tres distintas sustituciones para inferir *camino*(1, 2), *camino*(2, 3) y *camino*(3, 4). Como ejemplo, al sustituir $X = 1$ y $Y = 2$ se instancia la primera regla para que sea *camino*(1, 2) : – *arista*(1, 2). Como *arista*(1, 2) es verdadero, podemos inferir *camino*(1, 2).

Con estos tres hechos *camino*, podemos usar la segunda regla varias veces. Si sustituimos $X = 1$, $Z = 2$ y $Y = 3$, instanciamos la regla para que sea *camino*(1, 3) : – *camino*(1, 2) & *camino*(2, 3). Como ambas submetas del cuerpo se han inferido, se sabe que son verdaderas, por lo que podemos inferir el encabezado: *camino*(1, 3). Entonces, la sustitución $X = 1$, $Z = 3$ y $Y = 4$ nos permite inferir el encabezado *camino*(1, 4); es decir, hay un camino que va del nodo 1 al nodo 4. \square

12.3.3 Predicados intensionales y extensionales

Es convencional en los programas en Datalog distinguir unos predicados de otros de la siguiente manera:

1. Los predicados EDB (extensional database), o *base de datos extensional*, son aquellos que se definen a priori. Es decir, sus hechos verdaderos se proporcionan en una relación o tabla, o se proporcionan mediante el significado del predicado (como sería el caso para un predicado de comparación, por ejemplo).
2. Los predicados IDB (intensional database), o *base de datos intensional*, se definen sólo mediante las reglas.

Un predicado debe ser IDB o EDB, y sólo puede tener una de éstas. Como resultado, cualquier predicado que aparece en el encabezado de una o más reglas debe ser un predicado IDB. Los predicados que aparecen en el cuerpo pueden ser IDB o EDB. Por ejemplo, en el ejemplo 12.12, *arista* es un predicado EDB y *camino* es un predicado IDB. Recuerde que se nos proporcionaron ciertos hechos *arista*, como *arista*(1, 2), pero los hechos *camino* fueron inferidos por las reglas.

Cuando se utilizan programas en Datalog para expresar los algoritmos de flujo de datos, los predicados EDB se calculan a partir del mismo grafo de flujo. Entonces, los predicados IDB se expresan mediante reglas, y el problema de flujo de datos se resuelve al inferir todos los posibles hechos IDB a partir de las reglas y de los hechos EDB dados.

Ejemplo 12.13: Vamos a considerar cómo deben expresarse las definiciones de alcance en Datalog. En primer lugar, tiene sentido pensar a nivel de instrucción, en vez de hacerlo a nivel de bloque; es decir, la construcción de conjuntos gen y eliminar de un bloque básico se integrarán con el cálculo de las mismas definiciones de alcance. Por ende, el bloque b_1 sugerido en la figura 12.13 es común. Observe que identificamos los puntos dentro del bloque numerado como 0, 1, ..., n , si n es el número de instrucciones en el bloque. La i -ésima definición está “en” el punto i , y no hay una definición establecida en el punto 0.

Un punto en el programa debe representarse mediante un par (b, n) , en donde b es el nombre de un bloque y n es un entero entre 0 y el número de instrucciones en el bloque b . Nuestra formulación requiere dos predicados EDB:

	0	$x = y + z$
b_1	1	$*p = u$
	2	$x = v$
	3	

Figura 12.13: Un bloque básico con puntos entre las instrucciones

1. $def(B, N, X)$ es verdadera si, y sólo si la N -ésima instrucción en el bloque B puede definir la variable X . Por ejemplo, en la figura 12.13 $def(b_1, 1, X)$ es verdadera, $def(b_1, 3, X)$ es verdadera, y $def(b_1, 2, Y)$ es verdadera para toda posible variable Y a la que p pueda apuntar en ese punto. Por el momento vamos a suponer que Y puede ser cualquier variable del tipo al que apunta p .
2. $suc(B, N, C)$ es verdadero si, y sólo si el bloque C es sucesor del bloque B en el grafo de flujo, y B tiene N instrucciones. Es decir, el control puede fluir desde el punto N de B , hasta el punto 0 de C . Por ejemplo, suponga que b_2 es un predecesor del bloque b_1 en la figura 12.13, y que b_2 tiene 5 instrucciones. Entonces, $suc(b_2, 5, b_1)$ es verdadero.

Hay un predicado IDB, $da(B, N, C, M, X)$. Se pretende que sea verdadera si, y sólo si la definición de la variable X en la M -ésima instrucción del bloque C llega al punto N en el bloque B . Las reglas que definen el predicado da están en la figura 12.14.

- 1) $da(B, N, B, N, X) :- def(B, N, X)$
- 2) $da(B, N, C, M, X) :- da(B, N - 1, C, M, X) \&$
 $def(B, N, Y) \&$
 $X \neq Y$
- 3) $da(B, 0, C, M, X) :- da(D, N, C, M, X) \&$
 $suc(D, N, B)$

Figura 12.14: Reglas para el predicado da

La regla (1) establece que si la N -ésima instrucción del bloque B define a X , entonces esa definición de X llega al N -ésimo punto de B (es decir, el punto que está justo después de la instrucción). Esta regla corresponde al concepto de “gen” en nuestra formulación teórica de conjunto anterior sobre las definiciones de alcance.

La regla (2) representa la idea de que una definición pasa a través de una instrucción a menos que se “elimine”, y la única manera de eliminar una definición es redefinir su variable con un 100% de certeza. En detalle, la regla (2) dice que la definición de la variable X de la M -ésima instrucción del bloque C llega al punto N del bloque B si:

- a) llega al punto anterior $N - 1$ de B , y

- b) hay por lo menos una variable Y , aparte de X , que puede estar definida en la N -ésima instrucción de B .

Por último, la regla (3) expresa el flujo de control en el grafo. Dice que la definición de X en la M -ésima instrucción del bloque C llega al punto 0 de B si hay algún bloque D con N instrucciones, de tal forma que la definición de X llegue al final de D , y B es un sucesor de D . \square

El predicado EDB *suc* del ejemplo 12.13 puede leerse claramente del grafo de flujo. Podemos obtener *def* del grafo de flujo también, si somos conservadores y suponemos que un apuntador puede apuntar a cualquier parte. Si deseamos limitar el rango de un apuntador a las variables del tipo apropiado, entonces podemos obtener la información del tipo de la tabla de símbolos, y utilizar una relación *def* más pequeña. Una opción es hacer a *def* un predicado IDB y definirlo mediante las reglas. Estas reglas utilizarán predicados EDB más primitivos, los cuales se pueden determinar a partir del grafo de flujo y la tabla de símbolos.

Ejemplo 12.14: Suponga que introducimos dos nuevos predicados EDB:

1. *asignar*(B, N, X) es verdadero siempre que la N -ésima instrucción del bloque B tiene a X a la izquierda. Observe que X puede ser una variable o una expresión simple con un valor l , como $*p$.
2. *tipo*(X, T) es verdadero si el tipo de X es T . De nuevo, X puede ser cualquier expresión con un valor l , y T puede ser cualquier expresión para un tipo válido.

Entonces, podemos escribir reglas para *def*, convirtiéndolo en un predicado IDB. La figura 12.15 es una expansión de la figura 12.14, con dos de las posibles reglas para *def*. La regla (4) establece que la N -ésima instrucción del bloque B define a X , si X se asigna mediante la N -ésima instrucción. La regla (5) dice que X también puede definirse mediante la N -ésima instrucción del bloque B , si esa instrucción asigna a $*P$, y X es cualquiera de las variables del tipo al que apunta P . Otros tipos de asignaciones necesitarían otras reglas para *def*.

Como ejemplo de la forma en que haríamos inferencias mediante el uso de las reglas de la figura 12.15, volvamos a examinar el bloque b_1 de la figura 12.13. La primera instrucción asigna un valor a la variable x , por lo que el hecho *asignar*($b_1, 1, x$) estaría en el EDB. La tercera instrucción también asigna a x , por lo que *asignar*($b_1, 3, x$) es otro hecho EDB. La segunda instrucción asigna de manera indirecta a través de p , por lo que un tercer hecho EDB sería *asignar*($b_1, 2, *p$). Entonces, la regla (4) nos permite inferir *def*($b_1, 1, x$) y *def*($b_1, 3, x$).

Suponga que p es del tipo apuntador a entero (*int), y que x y y son enteros. Entonces podemos usar la regla (5), con $B = b_1$, $N = 2$, $P = p$, $T = \text{int}$ y X igual a x o y , para inferir *def*($b_1, 2, x$) y *def*($b_1, 2, y$). De manera similar, podemos inferir lo mismo acerca de cualquier otra variable cuyo tipo sea entero o que obligue a un entero. \square

- 1) $da(B, N, B, N, X) :- def(B, N, X)$
- 2) $da(B, N, C, M, X) :- da(B, N - 1, C, M, X) \& def(B, N, Y) \& X \neq Y$
- 3) $da(B, 0, C, M, X) :- da(D, N, C, M, X) \& suc(D, N, B)$
- 4) $def(B, N, X) :- asignar(B, N, X)$
- 5) $def(B, N, X) :- asignar(B, N, *P) \& tipo(X, T) \& tipo(P, *T)$

Figura 12.15: Reglas para los predicados *da* y *def*

12.3.4 Ejecución de programas en Datalog

Cada conjunto de reglas de Datalog define las relaciones para sus predicados IDB, como una función de las relaciones que se proporcionan por sus predicados EDB. Empiece con la suposición de que las relaciones IDB están vacías (es decir, los predicados IDB son falsos para todos los posibles argumentos). Entonces, aplique las reglas en forma repetida, infiriendo nuevos hechos cada vez que las reglas requieran que lo hagamos. Cuando el proceso converja, hemos terminado y las relaciones IDB resultantes forman la salida del programa. Este proceso se formaliza en el siguiente algoritmo, que es similar a los algoritmos iterativos que vimos en el capítulo 9.

Algoritmo 12.15: Evaluación simple de programas en Datalog.

ENTRADA: Un programa en Datalog y conjuntos de hechos para cada predicado EDB.

SALIDA: Conjuntos de hechos para cada predicado IDB.

MÉTODO: Para cada predicado p en el programa, haga que R_p sea la relación de hechos que sean verdaderos para ese predicado. Si p es un predicado EDB, entonces R_p es el conjunto de hechos dados para ese predicado. Si p es un predicado IDB, debemos calcular R_p . Ejecute el algoritmo de la figura 12.16. \square

Ejemplo 12.16: El programa en el ejemplo 12.12 calcula los caminos en un grafo. Para aplicar el Algoritmo 12.15, empezamos con el predicado EDB *arista* que contiene todas las aristas del grafo, y con la relación para *camino* vacío. En la primera ronda, la regla (2) no produce nada, ya que no hay hechos *camino*. Pero la regla (1) hace que todos los hechos *arista* se conviertan en hechos *camino* también. Es decir, después de la primera ronda conocemos *camino(a, b)* si, y sólo si hay una arista que va de *a* a *b*.

```

for (cada predicado IDB  $p$ )
     $R_p = \emptyset$ ;
while (ocurran cambios a cualquier  $R_p$ ) {
    considerar todas las posibles sustituciones de constantes para
    las variables en todas las reglas;
    determinar, para cada sustitución, si todas las
    submetas del cuerpo son verdaderas, usando las  $R_p$ s
    actuales para determinar la verdad de los predicados EDB e IDB;
    if (una sustitución hace que el cuerpo de una regla sea verdadero)
        agregar el encabezado a  $R_q$ , si  $q$  es el predicado de encabezado;
}

```

Figura 12.16: Evaluación de programas en Datalog

En la segunda ronda, la regla (1) no produce nuevos hechos de caminos, ya que la relación EDB *arista* nunca cambia. Sin embargo, ahora la regla (2) nos permite reunir dos caminos de longitud 1 para crear caminos de longitud 2. Es decir, después de la segunda ronda, *camino*(a, b) es verdadera si, y sólo si hay un camino de longitud 1 o 2 de a a b . De manera similar, en la tercera ronda podemos combinar caminos de longitud 2 o menos para descubrir todos los caminos de longitud 4 o menos. En la cuarta ronda, descubrimos caminos de longitud hasta 8, y en general, después de la i -ésima ronda, *camino*(a, b) es verdadera si, y sólo si hay un camino de a a b de longitud 2^{i-1} o menor. \square

12.3.5 Evaluación incremental de programas en Datalog

Hay una posible mejora de eficiencia en el Algoritmo 12.15. Observe que un nuevo hecho IDB sólo puede descubrirse en la ronda i si es el resultado de la sustitución de constantes en una regla, de tal forma que por lo menos una de las submetas se convierta en un hecho que acaba de descubrirse en la ronda $i - 1$. La prueba de esa afirmación es que si todos los hechos entre las submetas se conocieran en la ronda $i - 2$, entonces el “nuevo” hecho se habría descubierto cuando hiciéramos la misma sustitución de constantes en la ronda $i - 1$.

Para sacar provecho de esta observación, introduzca para cada predicado IDB p un predicado *nuevoP* que sólo contenga los hechos p recién descubiertos de la ronda anterior. Cada regla que tenga uno o más predicados IDB entre las submetas se sustituye por una colección de reglas. Cada regla en la colección se forma sustituyendo exactamente una ocurrencia de cierto predicado IDB q en el cuerpo por *nuevoQ*. Por último, para todas las reglas sustituimos el predicado de encabezado h por *nuevoH*. Se dice que las reglas resultantes están en *forma incremental*.

Las relaciones para cada predicado IDB p acumula a todos los hechos p , como en el Algoritmo 12.15. En una ronda:

1. Aplicamos las reglas para evaluar los predicados *nuevoP*.

Evaluación incremental de conjuntos

También es posible resolver los problemas de flujo de datos teóricos de conjuntos en forma incremental. Por ejemplo, en las definiciones de alcance, una definición sólo puede ser recién descubierta en $\text{ENT}[B]$ en la i -ésima ronda si acaba de descubrirse que está en $\text{SAL}[P]$ para cierto predecesor P de B . La razón por la que en general no tratamos de resolver dichos problemas de flujo de datos de manera incremental es que la implementación en vectores de bits de los conjuntos es muy eficiente. Por lo general, es más fácil pasar a través de los vectores completos que decidir si un hecho es nuevo o no.

2. Entonces, se resta p de $nuevoP$, para asegurar que los hechos en $nuevoP$ sean realmente nuevos.
3. Se agregan los hechos en $nuevoP$ a p .
4. Se establecen todas las relaciones $nuevasX$ a \emptyset para la siguiente ronda.

Formalizaremos estas ideas en el Algoritmo 12.18. Sin embargo, primero veremos un ejemplo.

Ejemplo 12.17: Considere el programa en Datalog del ejemplo 12.12 otra vez. La forma incremental de las reglas se proporciona en la figura 12.17. La regla (1) no cambia excepto en el encabezado, ya que no tiene submetas IDB en el cuerpo. Sin embargo, la regla (2), con dos submetas IDB, se convierte en dos reglas distintas. En cada regla, una de las ocurrencias de *camino* en el cuerpo se sustituye por *nuevoCamino*. En conjunto, estas reglas hacen valer la idea de que por lo menos uno de los dos caminos concatenados por la regla debe haberse descubierto en la ronda anterior. \square

- 1) $nuevoCamino(X, Y) :- arista(X, Y)$
- 2a) $nuevoCamino(X, Y) :- camino(X, Z) \& nuevoCamino(Z, Y)$
- 2b) $nuevoCamino(X, Y) :- nuevoCamino(X, Z) \& camino(Z, Y)$

Figura 12.17: Reglas incrementales para el programa de caminos en Datalog

Algoritmo 12.18: Evaluación incremental de programas en Datalog.

ENTRADA: Un programa en Datalog y conjuntos de hechos para cada predicado EDB.

SALIDA: Conjuntos de hechos para cada predicado IDB.

MÉTODO: Para cada predicado p en el programa, haga que R_p sea la relación de hechos que sean verdaderos para ese predicado. Si p es un predicado EDB, entonces R_p es el conjunto de hechos dados para ese predicado. Si p es un predicado IDB, debemos calcular R_p . Además, para cada predicado IDB p , haga que R_{nuevoP} sea una relación de “nuevos” hechos para el predicado p .

1. Modifique las reglas en la forma incremental antes descrita.
2. Ejecute el algoritmo de la figura 12.18.

□

```

for (cada predicado IDB  $p$ ) {
     $R_p = \emptyset;$ 
     $R_{nuevoP} = \emptyset;$ 
}
repeat {
    considerar todas las posibles sustituciones de constantes para
    las variables en todas las reglas;
    determinar, para cada sustitución, si todas las
    submetas del cuerpo son verdaderas, usando las  $R_p$ s y  $R_{nuevoP}$ s
    actuales para determinar la verdad de los predicados EDB e IDB;
    if (una sustitución hace que el cuerpo de una regla sea verdadero)
        agregar el encabezado a  $R_{nuevoH}$ , en donde  $h$  es el predicado
        de encabezado;
    for (cada predicado  $p$ ) {
         $R_{nuevoP} = R_{nuevoP} - R_p;$ 
         $R_p = R_p \cup R_{nuevoP};$ 
    }
} until (todos los  $R_{nuevoP}$ s estén vacíos);

```

Figura 12.18: Evaluación de programas en Datalog

12.3.6 Reglas problemáticas en Datalog

Existen ciertas reglas o programas en Datalog que técnicamente no tienen significado, por lo que no deben usarse. Los dos riesgos más importantes son:

1. *Reglas inseguras*: aquellas que tienen una variable en el encabezado que no aparezca en el cuerpo, de una manera que restrinja a esa variable para que sólo reciba valores que aparezcan en el EDB.
2. *Programas no estratificados*: conjuntos de reglas que tienen una recursividad que involucra a una negación.

Ahora vamos a explicar con más detalle cada uno de estos riesgos.

Seguridad en las reglas

Cualquier variable que aparezca en el encabezado de una regla debe también aparecer en el cuerpo. Además, la apariencia debe estar en una submeta que sea un átomo IDB o EDB ordinario. No es aceptable si la variable aparece sólo en un átomo negado, o sólo en un operador de comparación. La razón de esta política es evitar las reglas que nos permiten inferir un número infinito de hechos.

Ejemplo 12.19: La siguiente regla:

$$p(X, Y) :- q(Z) \& \text{NOT } r(X) \& X \neq Y$$

es insegura por dos razones. La variable X aparece sólo en la submeta negada $r(X)$ y la comparación $X \neq Y$. Y sólo aparece en la comparación. La consecuencia es que p es verdadera para un número infinito de pares (X, Y) , siempre y cuando $r(X)$ sea falsa y Y sea cualquier cosa distinta de X . \square

Datalog estratificado

Para que un programa pueda tener sentido, la recursividad y la negación deben separarse. El requerimiento formal es el siguiente. Debemos ser capaces de dividir los predicados *IDB* en *estratos*, por lo que si hay una regla con el predicado de encabezado p y una submeta de la forma $\text{NOT } q(\dots)$, entonces q es un predicado EDB o IDB en un estrato más inferior que p . Mientras se cumpla esta regla, podemos evaluar los estratos, de menor a mayor, mediante el Algoritmo 12.15 o 12.18, y después tratar las relaciones de los predicados IDB de los estratos como si fueran EDB para los cálculos de los estratos superiores. No obstante, si violamos esta regla, entonces el algoritmo iterativo no puede llegar a la convergencia, como se muestra en el siguiente ejemplo.

Ejemplo 12.20: Considere el programa en Datalog que consiste en una única regla:

$$p(X) :- e(X) \& \text{NOT } p(X)$$

Suponga que e es un predicado EDB, y que sólo $e(1)$ es verdadero. ¿ $p(1)$ es verdadero?

Este programa no está estratificado. Sea cual fuere el estrato en el que coloquemos a p , su regla tiene una submeta que se niega y tiene un predicado IDB (es decir, el mismo p) que sin duda no se encuentra en un estrato menor a p .

Si aplicamos el algoritmo iterativo, empezamos con $R_p = \emptyset$, por lo que al principio la respuesta es “no; $p(1)$ no es verdadero”. Sin embargo, la primera iteración nos permite inferir a $p(1)$, ya que tanto $e(1)$ como $\text{NOT } p(1)$ son verdaderos. Pero entonces la segunda iteración nos indica que $p(1)$ es falso. Es decir, sustituir 1 por X en la regla no nos permite inferir $p(1)$, ya que la segunda submeta $\text{NOT } p(1)$ es falsa. De manera similar, la tercera iteración indica que $p(1)$ es verdadera, la cuarta dice que es falsa, y así sucesivamente. Concluimos que este programa no estratificado carece de significado, y no lo consideraremos un programa válido. \square

12.3.7 Ejercicios para la sección 12.3

Ejercicio 12.3.1: En este problema, vamos a considerar un análisis de flujo de datos de definiciones de alcance que es más simple que el del ejemplo 12.13. Suponga que cada instrucción es por sí sola un bloque, y asuma al principio que cada instrucción define sólo a una variable. El predicado EDB $pred(I, J)$ indica que la instrucción I es un predecesor de la instrucción J . El predicado EDB $define(I, X)$ indica que la variable definida por la instrucción I es X . Vamos a usar los predicados IDB $ent(I, D)$ y $sal(I, D)$ para indicar que la definición D llega al inicio o al final de la instrucción I , respectivamente. Tenga en cuenta que una definición es en realidad un número de instrucción. La figura 12.19 es un programa en Datalog que expresa el algoritmo usual para calcular definiciones de alcance.

```

1)  eliminar( $I, D$ )  :-  define( $I, X$ ) & define( $D, X$ )
2)  sal( $I, I$ )  :-  define( $I, X$ )
3)  sal( $I, D$ )  :-  ent( $I, D$ ) & NOT eliminar( $I, D$ )
4)  ent( $I, D$ )  :-  sal( $J, D$ ) & pred( $J, I$ )

```

Figura 12.19: Programa en Datalog para un análisis simple de definiciones de alcance

Observe que la regla (1) establece que una instrucción se elimina a sí misma, pero la regla (2) asegura que una instrucción está en su propio “conjunto salida” de todas formas. La regla (3) es la función de transferencia normal, y la regla (4) permite la confluencia, ya que I puede tener varios predecesores.

Si problema es modificar las reglas para manejar el caso común en el que una definición es ambigua; por ejemplo, una asignación a través de un apuntador. En esta situación, $define(I, X)$ puede ser verdadera para varias X s distintas y una I . Una definición se representa mejor mediante un par (D, X) , en donde D es una instrucción y X es una de las variables que pueden definirse en D . Como resultado, ent y sal se convierten en predicados de tres argumentos; por ejemplo, $ent(I, D, X)$ indica que la (posible) definición de X en la instrucción D llega al inicio de la instrucción I .

Ejercicio 12.3.2: Escriba un programa en Datalog que sea similar a la figura 12.19, para calcular las expresiones disponibles. Además del predicado $define$, use un predicado $eval(I, X, O, Y)$ que diga que la instrucción I hace que se evalúe la expresión XOY . Aquí, O es el operador en la expresión; por ejemplo, $+$.

Ejercicio 12.3.3: Escriba un programa en Datalog que sea similar a la figura 12.19, para calcular las variables vivas. Además del predicado $define$, suponga que hay un predicado $usa(I, X)$ que indica que la instrucción I usa a la variable X .

Ejercicio 12.3.4: En la sección 9.5 definimos un cálculo de flujo de datos que involucraba a seis conceptos: anticipado, disponible, primero, postergable, último y usado. Suponga que escribimos un programa en Datalog para definir cada uno de éstos en términos de conceptos

EDB que pueden derivarse del programa (por ejemplo, información de gen y eliminar) y otros de estos seis conceptos. ¿Cuáles de los seis dependen de cuáles otros? ¿Cuáles de estas dependencias están negadas? ¿Estaría estratificado el programa en Datalog resultante?

Ejercicio 12.3.5: Suponga que el predicado EDB $arista(X, Y)$ consiste en los siguientes hechos:

$$\begin{array}{lll} arista(1, 2) & arista(2, 3) & arista(3, 4) \\ arista(4, 1) & arista(4, 5) & arista(5, 6) \end{array}$$

- Simule el programa en Datalog del ejemplo 12.12 con estos datos, usando la estrategia simple de evaluación del Algoritmo 12.15. Muestre los hechos $camino$ que se descubren en cada ronda.
- Simule el programa en Datalog de la figura 12.17 con estos datos, como parte de la estrategia de evaluación incremental del Algoritmo 12.18. Muestre los hechos $camino$ que se descubren en cada ronda.

Ejercicio 12.3.6: La siguiente regla:

$$p(X, Y) :- q(X, Z) \& r(Z, W) \& \text{NOT } p(W, Y)$$

es parte de un programa P en Datalog más grande.

- Identifique el encabezado, el cuerpo y las submetas de esta regla.
- ¿Qué predicados son sin duda predicados IDB del programa P ?
- ¿Qué predicados son sin duda predicados EDB de P ?
- ¿La regla es segura?
- ¿ P está estratificado?

Ejercicio 12.3.7: Convierta las reglas de la figura 12.14 a la forma incremental.

12.4 Un algoritmo simple de análisis de apuntadores

En esta sección, comenzaremos la discusión de un análisis muy simple de alias de apuntadores insensible al flujo, suponiendo que no hay llamadas a procedimientos. En las secciones siguientes mostraremos cómo manejar los procedimientos, primero de manera insensible al contexto, y después de manera sensible al contexto. La sensibilidad al flujo agrega mucha complejidad, y es menos importante en cuanto a la sensibilidad al contexto para los lenguajes como Java, en donde los métodos tienden a ser pequeños.

La pregunta fundamental que debemos hacer en el análisis de alias de apuntadores es si un par dado de apuntadores puede aliarse. Una manera de responder es calcular para cada apuntador la respuesta a la pregunta: “¿a qué objetos puede apuntar este apuntador?” Si dos apuntadores pueden apuntar al mismo objeto, entonces pueden aliarse.

12.4.1 Por qué es difícil el análisis de apuntadores

El análisis de alias de apuntadores para los programas en C es bastante difícil, ya que los programas en C pueden realizar cálculos arbitrarios con apuntadores. De hecho, podemos leer un entero y asignarlo a un apuntador, lo cual convertiría a este apuntador en un alias potencial de todas las demás variables apuntadores en el programa. Los apuntadores en Java, conocidos como referencias, son mucho más simples. No se permiten operaciones aritméticas, y los apuntadores sólo pueden apuntar al inicio de un objeto.

El análisis de alias de apuntadores debe ser interprocedural. Sin este análisis, debemos asumir que cualquier método al que se llame puede modificar el contenido de todas las variables apuntador accesibles, con lo que cualquier análisis de alias de apuntadores intraprocedural sería inefectivo.

Los lenguajes que permiten llamadas indirectas a funciones presentan un reto adicional para el análisis de alias de apuntadores. En C, podemos llamar a una función de manera indirecta, para lo cual llamamos a un apuntador a función desreferenciado. Debemos saber a qué datos puede apuntar el apuntador a función para poder analizar la función a la que se llamó. Y sin duda, después de analizar esta función, es posible descubrir más funciones a las que puede apuntar el apuntador a función y, por lo tanto, el proceso debe iterarse.

Aunque la mayoría de las funciones se llaman de manera indirecta en C, los métodos virtuales en Java hacen que muchas invocaciones sean indirectas. Dada una invocación `x.m()` en un programa en Java, puede haber muchas clases a las que podría pertenecer el objeto `x` y que tienen un método llamado `m`. Entre más preciso sea nuestro conocimiento del tipo actual de `x`, más preciso será nuestro grafo de llamadas. De manera ideal, podemos determinar en tiempo de compilación la clase exacta de `x` y así saber con precisión a qué método se refiere `m`.

Ejemplo 12.21: Considere la siguiente secuencia de instrucciones en Java:

```
Object = o;
o = new String();
n = o.length();
```

Aquí, `o` se declara como un `Object`. Sin analizar a dónde hace referencia `o`, hay que considerar como posibles destinos a todos los posibles métodos llamados “`length`”, declarados para todas las clases. Saber que `o` apunta a un objeto `String` reduce el análisis interprocedural a sólo el método declarado para `String`. □

Es posible aplicar aproximaciones para reducir el número de destinos. Por ejemplo, podemos determinar de manera estática cuáles son todos los tipos de objetos creados, y podemos limitar el análisis a ellos. Pero podemos ser más precisos si descubrimos el grafo de llamadas al instante, con base en el análisis tipo “apunta a” que se obtiene al mismo tiempo. Los grafos de llamadas más precisos no sólo conllevan a resultados más precisos, sino que también pueden reducir en gran parte el tiempo de análisis, que de otra forma se requeriría.

El análisis tipo “apunta a” es complicado. No es uno de esos problemas “fáciles” de flujo de datos, en donde sólo debemos simular el efecto de recorrer un ciclo de instrucciones una vez. En vez de ello, a medida que descubrimos nuevos destinos para un apuntador, hay que volver a analizar todas las instrucciones que asignan el contenido de ese apuntador a otro apuntador.

Por cuestión de simplicidad, nos enfocaremos principalmente en Java. Empezaremos con el análisis insensible al flujo y el análisis insensible al contexto, suponiendo por ahora que no se hacen llamadas a métodos en el programa. Después, describiremos cómo podemos descubrir el grafo de llamadas al instante, a medida que se calculan los resultados del análisis tipo “apunta a”. Por último, describimos la manera de manejar la sensibilidad al contexto.

12.4.2 Un modelo para apuntadores y referencias

Vamos a suponer que nuestro lenguaje tiene las siguientes maneras de representar y manipular referencias:

1. Ciertas variables del programa son de tipo “apuntador a T ” o “referencia a T ”, en donde T es un tipo. Estas variables son estáticas o están vivas en la pila en tiempo de ejecución. Las llamamos simplemente *variables*.
2. Hay un montículo de objetos. Todas las variables apuntan a objetos del montículo, no a otras variables. Estos objetos se conocen como *objetos montículo*.
3. Un objeto del montículo puede tener *campos*, y el valor de un campo puede ser una referencia a un objeto montículo (pero no a una variable).

Java está bien modelado mediante esta estructura, por lo que utilizaremos la sintaxis de Java en los ejemplos. Tenga en cuenta que C se modela con menos precisión, ya que las variables apuntador pueden apuntar a otras variables apuntador en C, y en principio, cualquier valor en C puede verse obligado a convertirse en un apuntador.

Como vamos a realizar un análisis insensible, sólo debemos afirmar que una variable v dada puede apuntar a un objeto h del montículo dado; no tenemos que lidiar con la cuestión de en qué parte del programa puede v apuntar a h , o en qué contexto puede v apuntar a h . Sin embargo, tenga en cuenta que las variables pueden nombrarse mediante su nombre completo. En Java, este nombre puede incorporar el módulo, clase, método y bloque dentro de un método, así como el mismo nombre de la variable. Así, podemos distinguir muchas variables que tienen el mismo identificador.

Los objetos del montículo no tienen nombres. A menudo se utiliza la aproximación para nombrar a los objetos, ya que puede crearse un número ilimitado de objetos en forma dinámica. Una convención es hacer referencia a los objetos mediante la instrucción en la cual se crearon. Como una instrucción puede ejecutarse muchas veces y crear un nuevo objeto cada vez, una afirmación como “ v puede apuntar a h ” en realidad significa “ v puede apuntar a uno o más de los objetos creados en la instrucción etiquetada como h ”.

El objetivo del análisis es determinar a dónde pueden apuntar cada variable y cada campo de cada objeto del montículo. A esto se le conoce como *análisis tipo “apunta a”*; dos apuntadores se alían si sus conjuntos “apunta a” se intersectan. Aquí describimos un análisis *basado en la inclusión*; es decir, una instrucción como $v = w$ hace que la variable v apunte a todos los objetos a los que apunta w , pero no viceversa. Aunque este método puede parecer obvio, existen otras alternativas para la forma en que definimos el análisis tipo “apunta a”. Por ejemplo, podemos definir un análisis *basado en equivalencia* de forma que una instrucción como $v = w$ convertiría a las variables v y w en una clase de equivalencia, apuntando a todas las variables a las que cada una de ellas puede apuntar. Aunque esta formulación no aproxima bien los alias,

proporciona una respuesta rápida, y a menudo apropiada, para la pregunta sobre qué variables apuntan al mismo tipo de objetos.

12.4.3 Insensibilidad al flujo

Empezaremos por mostrar un ejemplo muy simple para ilustrar el efecto de ignorar el flujo de control en el análisis tipo “apunta a”.

Ejemplo 12.22: En la figura 12.20 se crean tres objetos, *h*, *i* y *j*, y se asignan a las variables *a*, *b* y *c*, respectivamente. Por ende, sin duda *a* apunta a *h*, *b* apunta a *i*, y *c* apunta a *j* al final de la línea (3).

```

1) h: a = new Object();
2) i: b = new Object();
3) j: c = new Object();
4)     a = b;
5)     b = c;
6)     c = a;

```

Figura 12.20: Código en Java para el ejemplo 12.22

Si seguimos las instrucciones (4) a (6), descubriremos que después de la línea (4), *a* apunta sólo a *i*. Después de la línea (5), *b* apunta sólo a *j* y después de la línea (6), *c* apunta sólo a *i*. \square

El análisis anterior es sensible al flujo, ya que seguimos el flujo de control y calculamos a lo que puede apuntar cada variable después de cada instrucción. En otras palabras, además de considerar qué información tipo “apunta a” es la que “genera” cada instrucción, también consideramos qué información tipo “apunta a” es la que “elimina” cada instrucción. Por ejemplo, la instrucción *b* = *c*; elimina el hecho anterior “*b* apunta a *j*” y genera la nueva relación “*b* apunta a lo que *c* apunta”.

Un análisis insensible al flujo ignora el flujo de control, que en esencia asume que cada instrucción en el programa puede ejecutarse en cualquier orden. Calcula sólo un mapa global tipo “apunta a”, indicando qué es a lo que posiblemente puede apuntar cada variable en cualquier punto de la ejecución del programa. Si una variable puede apuntar a dos objetos distintos después de dos instrucciones distintas en un programa, sólo registramos que puede apuntar a ambos objetos. En otras palabras, en el análisis insensible al flujo, una asignación no “elimina” relaciones tipo “apunta a”, sino que sólo puede “generar” más relaciones tipo “apunta a”. Para calcular los resultados insensibles al flujo, agregamos en forma repetida los efectos tipo “apunta a” de cada instrucción en las relaciones tipo “apunta a”, hasta que no se encuentran nuevas relaciones. Es evidente que la falta de sensibilidad al flujo debilita los resultados del análisis en forma considerable, pero tiende a reducir el tamaño de la representación de los resultados y hace que el algoritmo converja con más rapidez.

Ejemplo 12.23: Regresando al ejemplo 12.22, las líneas (1) a (3) nos indican de nuevo que a puede apuntar a h , b puede apuntar a i , y c puede apuntar a j . Con las líneas (4) y (5), a puede apuntar tanto a h como a i , y b puede apuntar tanto a i como a j . Con la línea (6), c puede apuntar a h , i y j . Esta información afecta a la línea (5), que a su vez afecta a la línea (4). Al final, nos quedamos con la inútil conclusión de que cualquier cosa puede apuntar a cualquier otra cosa. \square

12.4.4 La formulación en Datalog

Ahora vamos a formalizar un análisis de alias de apuntadores insensible al flujo, con base en la explicación anterior. Por ahora ignoraremos las llamadas a procedimientos y nos concentraremos en los cuatro tipos de instrucciones que pueden afectar a los apuntadores:

1. *Creación de objetos.* $h : T \ v = \text{new } T()$; Esta instrucción crea un nuevo objeto del montículo, y la variable v puede apuntar a éste.
2. *Instrucción de copia.* $v = w$; Aquí, v y w son variables. La instrucción hace que v apunte a cualquier objeto del montículo al que w apunte en ese momento; es decir, w se copia en v .
3. *Almacenamiento de campo.* $v.f = w$; El tipo del objeto al que apunta v debe tener un campo f , y este campo debe ser de cierto tipo de referencia. Haga que v apunte al objeto del montículo h , y que w apunte a g . Esta instrucción hace que el campo f en h apunte ahora a g . Observe que la variable v no se modifica.
4. *Carga de campo.* $v = w.f$; Aquí, w es una variable que apunta a cierto objeto del montículo que tiene un campo f , y f apunta a cierto objeto del montículo h . La instrucción hace que la variable v apunte a h .

Observe que los accesos a los campos compuestos en el código fuente, como $v = w.f.g$, se descompondrán en dos instrucciones primitivas de carga de campos:

```
v1 = w.f;
v = v1.g;
```

Ahora vamos a expresar el análisis de manera formal en las reglas de Datalog. En primer lugar, sólo hay dos predicados IDB que debemos calcular:

1. $apnt(V, H)$ indica que la variable V puede apuntar al objeto del montículo H .
2. $hapnt(H, F, G)$ indica que el campo F del objeto del montículo H puede apuntar al objeto del montículo G .

Las relaciones EDB se construyen a partir del mismo programa. Como la ubicación de instrucciones en un programa es irrelevante cuando el análisis es insensible al flujo, sólo tenemos que afirmar en la EDB la existencia de instrucciones que tienen ciertas formas. A continuación, haremos una simplificación conveniente. En vez de definir las relaciones EDB para que guarden la información obtenida del programa, vamos a usar una instrucción entre comillas para sugerir

la relación o relaciones EDB que representan la existencia de dicha instrucción. Por ejemplo, “ $H : T V = \text{new } T()$ ” es un hecho EDB que afirma que en la instrucción H hay una asignación que hace que la variable V apunte a un nuevo objeto de tipo T . Suponemos que en la práctica, habría una relación EDB correspondiente que se llenaría con átomos base, uno para cada instrucción de esta forma en el programa.

Con esta convención, todo lo que tenemos que escribir en el programa en Datalog es una regla para cada uno de los cuatro tipos de instrucciones. El programa se muestra en la figura 12.21. La regla (1) dice que la variable V puede apuntar al objeto del montículo H si la instrucción H es una asignación de un nuevo objeto a V . La regla (2) dice que si hay una instrucción de copia $V = W$, y W puede apuntar a H , entonces V puede apuntar a H .

- 1) $apnt(V, H) :- "H : T V = \text{new } T()"$
- 2) $apnt(V, H) :- "V = W" \& apnt(W, H)$
- 3) $hapnt(H, F, G) :- "V.F = W" \& apnt(W, G) \& apnt(V, H)$
- 4) $apnt(V, H) :- "V = W.F" \& apnt(W, G) \& hapnt(G, F, H)$

Figura 12.21: Programa en Datalog para el análisis de apuntadores insensible al flujo

La regla (3) dice que si hay una instrucción de la forma $V.F = W$, W puede apuntar a G , y V puede apuntar a H , entonces el campo F de H puede apuntar a G . Por último, la regla (4) dice que si hay una instrucción de la forma $V = W.F$, W puede apuntar a G , y el campo F de G puede apuntar a H , entonces V puede apuntar a H . Observe que $apnt$ y $hapnt$ son mutuamente recursivos, pero este programa en Datalog puede evaluarse mediante cualquiera de los algoritmos iterativos que vimos en la sección 12.3.4.

12.4.5 Uso de la información de los tipos

Como Java tiene seguridad en los tipos, las variables sólo pueden apuntar a los tipos que son compatibles con los tipos declarados. Por ejemplo, al asignar un objeto que pertenezca a una superclase del tipo declarado de una variable se produciría una excepción en tiempo de ejecución. Considere el ejemplo simple de la figura 12.22, en donde S es una subclase de T . Este programa generará una excepción en tiempo de ejecución si p es verdadera, ya que a a no se le puede asignar un objeto de la clase T . Por ende, podemos concluir en forma estática que debido a la restricción de los tipos, a sólo puede apuntar a h y no a g .

```

S a;
T b;
if (p) {
g:    b = new T();
} else
h:    b = new S();
}
a = b;

```

Figura 12.22: Programa en Java con un error de tipo

Así, introducimos en nuestro análisis tres predicados EDB que reflejan una información importante sobre los tipos en el código que se está analizando. Vamos a utilizar lo siguiente:

1. $vTipo(V, T)$ indica que la variable V se declara para tener el tipo T .
2. $hTipo(H, T)$ indica que el objeto del montículo H se asigna con el tipo T . El tipo de un objeto creado no puede conocerse con precisión si, por ejemplo, el objeto se devuelve mediante un método simple. Dichos tipos se modelan de manera conservadora como todos los tipos posibles.
3. $asignable(T, S)$ indica que un objeto de tipo S puede asignarse a una variable con el tipo T . Por lo general, esta información se recopila de la declaración de los subtipos en el programa, pero también incorpora información acerca de las clases predefinidas del lenguaje. $asignable(T, T)$ siempre es verdadera.

Podemos modificar las reglas de la figura 12.21 para permitir inferencias, sólo si la variable asignada obtiene un objeto del montículo de un tipo que pueda asignarse. Las reglas se muestran en la figura 12.23.

La primera modificación es a la regla (2). Las últimas tres submetas dicen que podemos sólo concluir que V puede apuntar a H si hay tipos T y S que la variable V y el objeto del montículo H puedan tener respectivamente, de tal forma que puedan asignarse objetos de tipo S a las variables que sean referencias al tipo T . A la regla (4) se le agregó una restricción adicional similar. Observe que no hay restricción adicional en la regla (3), ya que todas las operaciones de almacenamiento deben pasar a través de las variables. Cualquier restricción de tipos sólo atraparía un caso extra, cuando el objeto base es una constante nula.

12.4.6 Ejercicios para la sección 12.4

Ejercicio 12.4.1: En la figura 12.24, h y g son etiquetas que se utilizan para representar objetos recién creados, y no forman parte del código. Podemos asumir que los objetos de tipo T tienen un campo f . Use las reglas de Datalog de esta sección para inferir todos los posibles hechos $apnt$ y $hapnt$.

- 1) $apnt(V, H) :- "H: T\ V = \text{new } T()"$
- 2) $apnt(V, H) :- "V = W" \ \&$
 $apnt(W, H) \ \&$
 $vTipo(V, T) \ \&$
 $hTipo(H, S) \ \&$
 $asignable(T, S)$
- 3) $hapnt(H, F, G) :- "V.F = W" \ \&$
 $apnt(W, G) \ \&$
 $apnt(V, H)$
- 4) $apnt(V, H) :- "V = W.F" \ \&$
 $apnt(W, G) \ \&$
 $hapnt(G, F, H) \ \&$
 $vTipo(V, T) \ \&$
 $hTipo(H, S) \ \&$
 $asignable(T, S)$

Figura 12.23: Agregar restricciones de tipo al análisis de apuntadores insensible al flujo

```

h: T a = new T();
g: T b = new T();
T c = a;
a.f = b;
b.f = c;
T d = c.f;

```

Figura 12.24: Código para el ejercicio 12.4.1

! Ejercicio 12.4.2: Si aplicamos el algoritmo de esta sección al siguiente código:

```

h: T a = new T();
g: b = new T();
T c = a;

```

inferiríamos que tanto *a* como *b* pueden apuntar a *h* y a *g*. Si el código se hubiera escrito así:

```

h: T a = new T();
g: b = new T();
T c = b;

```

inferiríamos correctamente que *a* puede apuntar a *h*, y que *b* y *c* pueden apuntar a *g*. Sugiera un análisis de flujo de datos intraprocedural que pueda evitar este tipo de imprecisión.

```

t p(t x) {
    h: T a = new T();
    a.f = x;
    return a;
}

void main() {
    g: T b = new T();
    b = p(b);
    b = b.f;
}

```

Figura 12.25: Código de ejemplo para el análisis de apuntadores

! Ejercicio 12.4.3: Podemos extender el análisis de esta sección para que sea interprocedural si simulamos la llamada y el retorno como si fueran operaciones de copia, como en la regla (2) de la figura 12.21. Es decir, una llamada copia los valores actuales a sus correspondientes valores formales, y el retorno copia la variable que contiene el valor de retorno a la variable a la que se le asigna el resultado de la llamada. Considere el programa de la figura 12.25.

- Realice un análisis insensible sobre este código.
- Algunas de las inferencias hechas en (a) son en realidad “falsas”, ya que no representan ningún evento que pueda ocurrir en tiempo de ejecución. El problema puede rastrearse hasta las múltiples asignaciones a la variable *b*. Rescriba el código de la figura 12.25 de manera que ninguna variable se asigne más de una vez. Vuelva a ejecutar el análisis y muestre que cada hecho *apnt* y *hapnt* inferido puede ocurrir en tiempo de ejecución.

12.5 Análisis interprocedural insensible al contexto

Ahora consideraremos las invocaciones a los métodos. Primero explicaremos cómo puede usarse el análisis tipo “apunta a” para calcular un grafo de llamadas preciso, el cual es útil para calcular los resultados precisos del tipo “apunta a”. Después formalizaremos el descubrimiento de grafos de llamadas al instante y mostraremos cómo se puede usar Datalog para describir el análisis en forma concisa.

12.5.1 Efectos de la invocación a un método

Los efectos de la llamada a un método como *x = y.n(z)* en Java, en las relaciones del tipo “apunta a”, puede calcularse de la siguiente manera:

- Determine el tipo del objeto receptor, que es el objeto al que apunta *y*. Suponga que su tipo es *t*. Haga que *m* sea el método de nombre *n* en la superclase más estrecha de *t* que

tenga un método llamado n . Observe que, en general, el método que se invoque sólo se puede determinar en forma dinámica.

2. A los parámetros formales de m se les asignan los objetos a los que apuntan los parámetros actuales. Estos parámetros actuales incluyen no sólo los parámetros que se pasan directamente, sino también el mismo objeto receptor. Cada invocación a un método asigna el objeto receptor a la variable `this`.³ Nos referimos a las variables `this` como los 0-ésimos parámetros formales de los métodos. En $x = y.n(z)$, hay dos parámetros formales: el objeto al que apunta y se asigna a la variable `this`, y el objeto al que apunta z se asigna al primer parámetro formal declarado de m .
3. El objeto devuelto de m se asigna a la variable del lado izquierdo de la instrucción de asignación.

En el análisis sensible al contexto, los parámetros y los valores devueltos se modelan mediante instrucciones de copia. La pregunta interesante que queda por contestar es cómo determinar el tipo del objeto receptor. Podemos determinar de manera conservadora el tipo, de acuerdo con la declaración de la variable; por ejemplo, si la variable declarada tiene el tipo t , entonces sólo se pueden invocar los métodos llamados n en los subtipos de t . Por desgracia, si la variable declarada tiene el tipo `Object`, entonces todos los métodos con el nombre n son destinos potenciales. En los programas reales que utilizan mucho las jerarquías de objetos e incluyen muchas bibliotecas grandes, dicho método puede producir muchos destinos de llamada espurios, con lo cual el análisis se hace lento e impreciso.

Debemos saber a dónde apuntan las variables para poder calcular los destinos de llamada; pero a menos que conozcamos los destinos de llamada, no podemos averiguar a dónde pueden apuntar todas las variables. Esta relación recursiva requiere que descubramos los destinos de llamada al instante, a medida que calculamos el conjunto “apunta a”. El análisis continúa hasta que no haya nuevos destinos de llamada y que no se encuentren nuevas relaciones “apunta a”.

Ejemplo 12.24: En el código de la figura 12.26, r es un subtipo de s , que a su vez es un subtipo de t . Con sólo utilizar la información del tipo declarado, `a.n()` podemos invocar cualquiera de los tres métodos declarados con el nombre n , ya que s y r son ambos subtipos del tipo declarado de a , t . Además, parece que a puede apuntar a los objetos g , h e i después de la línea (5).

Al analizar las relaciones tipo “apunta a”, primero determinamos que a puede apuntar a j , un objeto de tipo t . Por ende, el método declarado en la línea (1) es un destino de llamada. Al analizar la línea (1), determinamos que a también puede apuntar a g , un objeto de tipo r . Por ende, el método declarado en la línea (3) también puede ser un destino de llamada, y a puede ahora apuntar a i , otro objeto de tipo r . Como no hay más nuevos destinos de llamada, el análisis termina sin analizar el método declarado en la línea (2) y sin concluir que a puede apuntar a h . \square

³Recuerde que las variables se diferencian mediante el método al que pertenecen, por lo que no hay sólo una variable llamada `this`, sino una de estas variables para cada método en el programa.

```

class t {
1) g:    t n() { return new r(); }
}
class s extends t {
2) h:    t n() { return new s(); }
}
class r extends s {
3) i:    t n() { return new r(); }
}

main () {
4) j:    t a = new t();
5)        a = a.n();
}

```

Figura 12.26: Una invocación a un método virtual

12.5.2 Descubrimiento del grafo de llamadas en Datalog

Para formular las reglas de Datalog para el análisis interprocedural insensible al contexto, introducimos tres predicados EDB, cada uno de los cuales puede obtenerse con facilidad del código fuente:

1. *actual(S, I, V)* dice que *V* es el *I*-ésimo parámetro actual que se utiliza en el sitio de llamada *S*.
2. *formal(M, I, V)* dice que *V* es el *I*-ésimo parámetro formal declarado en el método *M*.
3. *ajk(T, N, M)* dice que *M* es el método que se llama cuando *N* se invoca en un objeto receptor de tipo *T*. (*ajk* significa “análisis de jerarquía de clases”.)

Cada arista del grafo de llamadas se representa mediante un predicado IDB llamado *invoca*. Al descubrir más aristas en el grafo de llamadas, se crean más relaciones tipo “apunta a” a medida que se pasan como entrada los parámetros y se pasan como salida los valores de retorno. Este efecto se sintetiza mediante las reglas que se muestran en la figura 12.27.

La primera regla calcula el destino de llamada del sitio de llamada. Es decir, “*S : V.N(...)*” dice que hay un sitio de llamada etiquetado como *S*, que invoca al método llamado *N* en el objeto receptor al que apunta *V*. Las submetas dicen que si *V* puede apuntar al objeto del montículo *H*, que se asigna como el tipo *T*, y *M* es el método que se utiliza cuando *N* se invoca en objetos de tipo *T*, entonces el sitio de llamada *S* puede invocar al método *M*.

La segunda regla dice que si el sitio *S* puede llamar al método *M*, entonces cada parámetro formal de *M* puede apuntar a cualquier parte a la que pueda apuntar el parámetro actual de la llamada. La regla para manejar los valores devueltos se deja como ejercicio.

Al combinar estas dos reglas con las que se explican en la sección 12.4 se crea un análisis tipo “apunta a” insensible al contexto, el cual utiliza un grafo de llamadas que se calcula al instante. Este análisis tiene el efecto adicional de crear un grafo de llamadas utilizando un análisis

```

1) invokes(S, M) :- “S : V.N(...)” &
   pts(V, H) &
   hType(H, T) &
   cha(T, N, M)
2) pts(V, H) :- invokes(S, M) &
   formal(M, I, V) &
   actual(S, I, W) &
   pts(W, H)

```

Figura 12.27: Datal program for call-graph discovery

tipo “apunta a” insensible al contexto y al flujo. Este grafo de llamadas es mucho más preciso que uno que se calcula sólo en base a las declaraciones de tipos y el análisis sintáctico.

12.5.3 Carga dinámica y reflexión

Los lenguajes como Java permiten la carga dinámica de clases. Es imposible analizar todo el código posible que ejecuta un programa, y por ende imposible proporcionar cualquier aproximación conservadora de los grafos de llamadas o de los alias de apuntadores en forma estática. El análisis estático sólo puede proporcionar una aproximación con base en el código analizado. Recuerde que todos los análisis aquí descritos se pueden aplicar en el nivel de bytecodes de Java y, por lo tanto, no es necesario examinar el código fuente. Esta opción es muy importante, ya que los programas en Java tienden a usar muchas bibliotecas.

Aun si suponemos que se analiza todo el código que se va a ejecutar, hay una complicación más que hace imposible el análisis conservador: la reflexión. La reflexión permite a un programa determinar en forma dinámica los tipos de los objetos a crear, los nombres de los métodos invocados, así como los nombres de los campos a los que se accede. El tipo, el método y los nombres de los campos pueden calcularse o derivarse de la entrada del usuario, por lo que en general, la única aproximación posible es suponer el universo.

Ejemplo 12.25: El siguiente código muestra un uso común de la reflexión:

```

1) String nombreClase = ...;
2) Class c = Class.forName(nombreClase);
3) Object o = c.newInstance();
4) T t = (T) o;

```

El método `forName` en la biblioteca `Class` recibe una cadena que contiene el nombre de la clase y devuelve la clase. El método `newInstance` devuelve una instancia de esa clase. En vez de dejar el objeto `o` con el tipo `Object`, este objeto se convierte en una superclase `T` de todas las clases esperadas. □

Aunque muchas aplicaciones extensas de Java utilizan la reflexión, tienden a usar especificaciones comunes, como el que se muestra en el ejemplo 12.25. Siempre y cuando la aplicación no redefina el cargador de clases, podemos conocer la clase del objeto si conocemos el valor de `nombreClase`. Si el valor de `nombreClase` se define en el programa, como las cadenas son inmutables en Java, saber a qué apunta `nombreClase` nos proporciona el nombre de la clase. Esta técnica es otro uso del análisis tipo “apunta a”. Si el valor de `nombreClase` se basa en la entrada del usuario, entonces el análisis tipo “apunta a” puede ayudar a localizar en dónde se introduce el valor, y el desarrollador puede limitar el alcance de su valor.

De manera similar, podemos explotar la instrucción de conversión de tipos, la línea (4) en el ejemplo 12.25, para aproximar el tipo de los objetos creados en forma dinámica. Suponiendo que no se ha redefinido el manejador de excepciones de conversión de tipo, el objeto debe pertenecer a una subclase de la clase T .

12.5.4 Ejercicios para la sección 12.5

Ejercicio 12.5.1: Para el código de la figura 12.26:

- a) Construya las relaciones EDB *actual*, *formal* y *ajk*.
- b) Haga todas las posibles inferencias de los hechos *apnt* y *hapnt*.

! Ejercicio 12.5.2: ¿Cómo agregaría a los predicados EDB y a las reglas de la sección 12.5.2 predicados y reglas adicionales, para tomar en cuenta el hecho de que si la llamada a un método devuelve un objeto, entonces la variable a la cual se asigna el resultado de la llamada puede apuntar a cualquier lugar a donde pueda apuntar la variable que contiene el valor de retorno?

12.6 Análisis de apuntadores sensible al contexto

Como vimos en la sección 12.1.2, la sensibilidad al contexto puede mejorar en gran parte la precisión del análisis interprocedural. Hablamos acerca de dos métodos para este análisis, uno basado en la clonación (sección 12.1.4) y otro basado en los resúmenes (sección 12.1.5). ¿Cuál debemos usar?

Hay varias dificultades para calcular los resúmenes de la información tipo “apunta a”. En primer lugar, los resúmenes son extensos. El resumen de cada método debe incluir el efecto de todas las actualizaciones que pueden hacer la función y todos los métodos llamados, en términos de los parámetros de entrada. Es decir, un método puede cambiar los conjuntos “apunta a” de todos los datos que se pueden alcanzar a través de las variables estáticas, los parámetros entrantes y todos los objetos creados por el método y todos los métodos a los que llama. Aunque se han propuesto esquemas complicados, no se conoce una solución que pueda escalar a programas extensos. Incluso si los resúmenes pueden calcularse en una pasada de abajo hacia arriba, el cálculo de los conjuntos “apunta a” para todos los exponencialmente diversos contextos en una típica pasada descendente presenta un problema aún mayor. Dicha información es necesaria para las consultas globales, como buscar todos los puntos en el código que entran en contacto con cierto objeto.

En esta sección veremos un análisis sensible al contexto, basado en la clonación. Lo único que hace el análisis basado en la clonación es clonar los métodos, uno para cada contexto de interés. Después aplicamos el análisis insensible al contexto al grafo de llamadas clonado. Aunque este método parece simple, el problema está en los detalles de manejar el extenso número de clones. ¿Cuántos contextos hay? Aun si utilizamos la idea de colapsar todos los ciclos recursivos, como vimos en la sección 12.1.3, es común encontrar 10^{14} contextos en una aplicación en Java. El reto es representar los resultados de estos diversos contextos.

Vamos a separar la explicación de la sensibilidad al contexto en dos partes:

1. ¿Cómo podemos manejar la sensibilidad al contexto en forma lógica? Esta parte es fácil, ya que tan sólo aplicamos el algoritmo insensible al contexto al grafo de llamadas clonado.
2. ¿Cómo representar los exponencialmente diversos contextos? Una manera es representar la información como diagramas de decisiones binarios (BDDs), una estructura de datos muy optimizada que se ha utilizado para muchas otras aplicaciones.

Este método para la sensibilidad al contexto es un excelente ejemplo de la importancia de la abstracción. Como veremos en breve, para eliminar la complejidad algorítmica aprovechamos los años de trabajo invertidos en la abstracción del BDD. Podemos especificar un análisis tipo “apunta a” sensible al contexto en sólo unas cuantas líneas de Datalog, que a su vez aprovecha los varios miles de líneas de código existente para la manipulación de BDDs. Este método tiene varias ventajas importantes. En primer lugar, hace posible la expresión fácil de análisis posteriores que utilizan los resultados del análisis tipo “apunta a”. Después de todo, los resultados del análisis “apunta a” por sí solos no son interesantes. En segundo lugar, facilita en forma considerable la escritura correcta del análisis, ya que aprovecha muchas líneas de código bien depurado.

12.6.1 Contextos y cadenas de llamadas

El análisis tipo “apunta a” sensible al contexto antes descrito asume que ya se ha calculado un grafo de llamadas. Este paso ayuda a hacer posible una representación compacta de los diversos contextos de llamada. Para obtener el grafo de llamadas, primero ejecutamos un análisis tipo “apunta a” insensible al contexto que calcula el grafo de llamadas al instante, como vimos en la sección 12.5. Ahora describiremos cómo crear un grafo de llamadas clonado.

Un contexto es una representación de la cadena de llamadas que forma el historial de las llamadas a las funciones activas. Otra forma de ver el contexto es como un resumen de la secuencia de llamadas, cuyos registros de activación se encuentran en ese momento en la pila en tiempo de ejecución. Si no hay funciones recursivas en la pila, entonces la cadena de llamadas (la secuencia de ubicaciones desde las cuales se hicieron las llamadas en la pila) es una representación completa. También es una representación aceptable, en el sentido de que sólo hay un número finito de contextos distintos, aunque ese número puede ser exponencial en el número de funciones en el programa.

No obstante, si hay funciones recursivas en el programa, entonces el número de posibles cadenas de llamadas es infinito, y no podemos permitir que todas las posibles cadenas de llamadas representen distintos contextos. Hay varias formas en las que podemos limitar el número

de distintos contextos. Por ejemplo, podemos escribir una expresión regular que describa todas las posibles cadenas de llamadas y que convierta esa expresión regular en un autómata finito determinista, usando los métodos de la sección 3.7. Así, los contextos pueden identificarse con los estados de este autómata.

Aquí vamos a adoptar un esquema más simple, que captura el historial de las llamadas no recursivas pero considera que las llamadas recursivas son “demasiado difíciles de desenmarañar”. Empezaremos por buscar todos los conjuntos mutuamente recursivos de funciones en el programa. El proceso es simple y no lo elaboraremos con detalle aquí. Piense en un grafo cuyos nodos son las funciones, con una arista que va de p a q si la función p llama a la función q . Los componentes fuertemente conectados (SCCs) de este grafo son los conjuntos de funciones mutuamente recursivas. Como un caso especial común, una función p que se llama a sí misma, pero que no está en un SCC con cualquier otra función es un SCC por sí sola. Las funciones no recursivas son también SCCs por sí solas. A un SCC le llamamos *no trivial* si tiene más de un miembro (el caso mutuamente recursivo), o si tiene un solo miembro recursivo. Los SCCs que son funciones no recursivas individuales son SCCs *triviales*.

La siguiente es nuestra modificación de la regla que establece que cualquier cadena de llamadas es un contexto. Dada una cadena de llamadas, se elimina la ocurrencia de un sitio de llamada s si:

1. s está en una función p .
2. La función q se llama en el sitio s (es posible que $q = p$).
3. p y q se encuentran en el mismo componente fuerte (es decir, p y q son mutuamente recursivas, o $p = q$ y p es recursiva).

El resultado es que cuando se hace una llamada a un miembro de un SCC S no trivial, el sitio para esa llamada se vuelve parte del contexto, pero las llamadas dentro de S a otras funciones en el mismo SCC no forman parte del contexto. Por último, cuando se hace una llamada fuera de S , registramos ese sitio de llamada como parte del contexto.

Ejemplo 12.26: En la figura 12.28 hay un bosquejo de cinco funciones con algunos sitios de llamada y ciertas llamadas entre ellos. Un análisis de las llamadas muestra que q y r son mutuamente recursivas. Sin embargo, p , s y t no son recursivas en definitiva. Por ende, nuestros contextos serán listas de todos los sitios de llamada excepto $s3$ y $s5$, en donde se realizan las llamadas entre q y r .

Vamos a considerar todas las formas en las que podríamos llegar de p a t ; es decir, todos los contextos en los que ocurren llamadas a t :

1. p podría llamar a s en $s2$, y después s podría llamar a t en $s7$ o $s8$. Por ende, dos posibles cadenas de llamadas son $(s2, s7)$ y $(s2, s8)$.
2. p podría llamar a q en $s1$. Entonces, q y r podrían llamarse entre sí en forma recursiva, un cierto número de veces. Podríamos descomponer el ciclo:
 - (a) En $s4$, en donde q llama directamente a t . Esta opción nos conduce a sólo un contexto, $(s1, s4)$.

```

void p() {
    h: a    = new T();
    s1: T b = q(a);
    s2:      s(b);
}

T  q(T w) {
    s3: c    = r(w);
    i: T d = new T();
    s4:      t(d);
    return d;
}

T  r(T x) {
    s5: T e = q(x);
    s6:      s(e);
    return e;
}

void s(T y) {
    s7: T f = t(y);
    s8:      f = t(f);
}

T  t(T z) {
    j: T g = new T();
    return g;
}

```

Figura 12.28: Funciones y sitios de llamada para un ejemplo abierto

- (b) En s_6 , en donde r llama a s . Aquí podríamos llegar a t mediante la llamada a s_7 , o mediante la llamada a s_8 . Eso nos proporciona dos contextos más, (s_1, s_6, s_7) y (s_1, s_6, s_8) .

Hay, por lo tanto, cinco contextos distintos en los que se puede llamar a t . Observe que todos estos contextos omiten los sitios de llamada recursivos, s_3 y s_5 . Por ejemplo, el contexto (s_1, s_4) en realidad representa el conjunto infinito de cadenas de llamadas $(s_1, s_3, (s_5, s_3)^n, s_4)$ para todas las $n \geq 0$. \square

Ahora describiremos cómo derivamos el grafo de llamadas clonado. Cada método clonado se identifica mediante el método en el programa M y un contexto C . Las aristas pueden derivarse agregando los correspondientes contextos a cada uno de las aristas en el grafo de llamadas original. Recuerde que hay una arista en el grafo de llamadas original que enlaza al sitio de llamada S con el método M , si el predicado $invoca(S, M)$ es verdadero. Para agregar contextos

con el fin de identificar los métodos en el grafo de llamadas clonado, podemos definir un predicado $CSinvoca$ correspondiente, de tal forma que $CSinvoca(S, C, M, D)$ sea verdadero si el sitio de llamada S en el contexto C llama al contexto D del método M .

12.6.2 Agregar contexto a las reglas de Datalog

Para buscar relaciones “apunta a” sensibles al contexto, sólo debemos aplicar el mismo análisis tipo “apunta a” insensible al contexto al grafo de llamadas clonado. Como un método en el grafo de llamadas clonado se representa mediante el método original y su contexto, revisamos todas las reglas de Datalog de manera acorde. Por cuestión de simplicidad, las reglas que se muestran a continuación no incluyen la restricción de tipos, y los símbolos $-$ son todas las variables nuevas.

- 1) $apnt(V, C, H) :- "H : T V = \text{new } T()" \& CSinvoca(H, C, -, -)$
- 2) $apnt(V, C, H) :- "V = W" \& apnt(W, C, H)$
- 3) $hapnt(H, F, G) :- "V.F = W" \& apnt(W, C, G) \& apnt(V, C, H)$
- 4) $apnt(V, C, H) :- "V = W.F" \& apnt(W, C, G) \& hapnt(G, F, H)$
- 5) $apnt(V, D, H) :- CSinvoca(S, C, M, D) \& formal(M, I, V) \& actual(S, I, W) \& apnt(W, C, H)$

Figura 12.29: Programa en Datalog para el análisis tipo “apunta a” sensible al contexto

Un argumento adicional, que representa al contexto, se debe proporcionar al predicado IDB $apnt(V, C, H)$ dice que la variable V en el contexto C puede apuntar al objeto del montículo H . Todas las reglas se explican por sí solas, tal vez con la excepción de la regla 5. Esta regla dice que si el sitio de llamada S en el contexto C llama al método M del contexto D , entonces los parámetros formales en el método M del contexto D pueden apuntar a los objetos a los que apuntan los parámetros actuales correspondientes en el contexto C .

12.6.3 Observaciones adicionales acerca de la sensibilidad

Lo que hemos descrito es una formulación de sensibilidad al contexto, la cual ha demostrado ser lo suficientemente práctica como para manejar muchos programas extensos reales en Java,

usando los trucos que se describen brevemente en la siguiente sección. Sin embargo, el algoritmo no puede aún manejar las aplicaciones en Java más extensas.

Los objetos montículo en esta formulación se nombran en base a su sitio de llamada, pero sin sensibilidad al contexto. Esta simplificación puede provocar problemas. Considere el modismo de fábrica de objetos (object-factory) en el que todos los objetos del mismo tipo se asignan por la misma rutina. El esquema actual haría que todos los objetos de esa clase compartan el mismo nombre. En esencia, es muy simple manejar tales casos si ponemos en línea el código de asignación. En general, lo ideal sería aumentar la sensibilidad al contexto al nombrar los objetos. Aunque es fácil agregar la sensibilidad al contexto de los objetos a la formulación en Datalog, obtener el análisis para escalar a programas más extensos es otro asunto.

Otra forma importante de sensibilidad es la sensibilidad a los objetos. Una técnica sensible a los objetos puede diferenciar entre los métodos invocados sobre distintos objetos receptores. Considere el caso de un sitio de llamada en un contexto de llamada, en el que se descubre que una variable apunta a dos objetos receptores distintos de la misma clase. Sus campos pueden apuntar a distintos objetos. Sin hacer diferencias entre los objetos, una copia entre los campos de la referencia al objeto `this` creará relaciones espurias, a menos que sepáremos el análisis de acuerdo a los objetos receptores. La sensibilidad a los objetos es más útil que la sensibilidad al contexto para ciertos análisis.

12.6.4 Ejercicios para la sección 12.6

```

void p() {
    h: T a = new T();
    i: T b = new T();
    c1: T c = q(a,b);
}

T q(T x, T y) {
    j: T d = new T();
    c2: d = q(x,d);
    c3: d = q(d,y);
    c4: d = r(d);
    return d;
}

T r(T z) {
    return z;
}

```

Figura 12.30: Código para los ejercicios 12.6.1 y 12.6.2

Ejercicio 12.6.1: ¿Cuáles son todos los contextos que se diferenciarían si aplicamos los métodos de esta sección al código de la figura 12.30?

! Ejercicio 12.6.2: Realice un análisis sensible al contexto del código de la figura 12.30.

! Ejercicio 12.6.3: Extienda las reglas de Datalog de esta sección para incorporar la información de tipos y subtipos, siguiendo el método de la sección 12.5.

12.7 Implementación en Datalog mediante BDDs

Los *diagramas de decisiones binarios* (Binary Decision Diagrams, BDDs) son un método para representar las funciones booleanas mediante grafos. Como hay 2^{2^n} funciones booleanas de n variables, ningún método de representación va a ser muy conciso en todas las funciones booleanas. Sin embargo, las funciones booleanas que aparecen en la práctica tienen en general mucha regularidad. Por ende, es común que podamos encontrar un BDD conciso para las funciones que en realidad deseamos representar.

Resulta que las funciones booleanas descritas por los programas en Datalog que hemos desarrollado para analizar programas no son la excepción. Aunque los BDDs concisos que representan información sobre un programa se deben encontrar a menudo mediante el uso de heurística y las técnicas usadas en los paquetes de manipulación de BDDs comerciales, el método del BDD ha sido bastante exitoso en la práctica. En especial, supera a los métodos basados en sistemas de administración de bases de datos convencionales, ya que éstos están diseñados para los patrones de datos más irregulares que aparecen en los datos comerciales típicos.

La discusión acerca de la tecnología BDD que se ha desarrollado a través de los años es algo que va más allá de este libro. Aquí le presentaremos la notación BDD. Después vamos a sugerir cómo podemos representar los datos relacionales como BDDs y cómo podríamos manipular los BDDs para reflejar las operaciones que se llevan a cabo para ejecutar programas en Datalog, mediante algoritmos como el Algoritmo 12.18. Por último, describiremos cómo representar los exponencialmente diversos contextos en los BDDs, la clave para el éxito del uso de BDDs en el análisis sensible al contexto.

12.7.1 Diagramas de decisiones binarios

Un BDD representa a una función booleana mediante un GDA con raíz. Cada uno de los nodos interiores del GDA está etiquetado por una de las variables de la función representada. Al final hay dos hojas, una etiquetada como 0 y la otra como 1. Cada nodo interior tiene dos aristas hacia los hijos; a estas aristas se les conoce como “inferior” y “superior”. La arista inferior se asocia con el caso en donde la variable en el nodo tiene el valor 0, y el arista superior se asocia con el caso en donde la variable tiene el valor 1.

Dada una asignación de verdad para las variables, podemos empezar en la raíz y en cada nodo, por decir un nodo etiquetado como x , seguir la arista inferior o superior, dependiendo de si el valor de verdad para x es 0 o 1, respectivamente. Si llegamos a la hoja etiquetada como 1, entonces la función representada es verdadera para esta asignación de verdad; en caso contrario es falsa.

Ejemplo 12.27: En la figura 12.31 podemos ver un BDD. En breve veremos la función que representa. Observe que hemos etiquetado todas las aristas “inferiores con 0 y todas las aristas “superiores” con 1. Considere la asignación de verdad para las variables $wxyz$ que establece

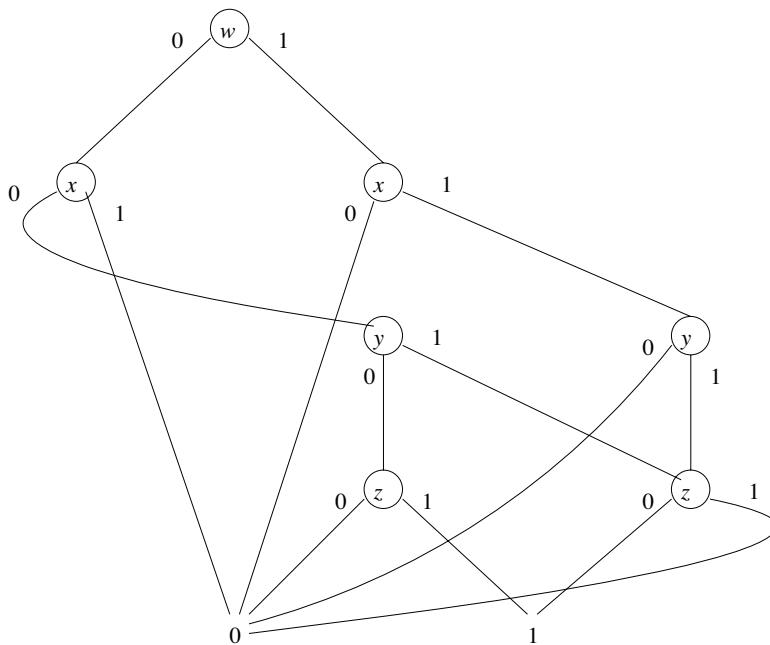


Figura 12.31: Un diagrama de decisiones binario

$w = x = y = 0$ y $z = 1$. Empezando en la raíz, como $w = 0$ tomamos la arista inferior, el cual nos lleva al extremo izquierdo de los nodos etiquetados como x . Ya que $x = 0$, seguimos de nuevo la arista inferior desde este nodo, el cual nos lleva al extremo izquierdo de los nodos etiquetados como y . Ya que $y = 0$, a continuación nos movemos al extremo izquierdo de los nodos etiquetados como z . Ahora, como $z = 1$, tomamos la arista superior y terminamos en la hoja etiquetada como 1. Nuestra conclusión es que la función es verdadera para esta asignación de verdad.

Ahora, considere la asignación de verdad $wxyz = 0101$; es decir, $w = y = 0$ y $x = z = 1$. De nuevo empezamos en la raíz. Como $w = 0$ otra vez nos movemos al extremo izquierdo de los nodos etiquetados como x . Pero ahora, como $x = 1$, seguimos la arista superior que salta hasta la hoja 0. Es decir, no sólo sabemos que la asignación de verdad 0101 hace a la función falsa, sino que como ni siquiera vimos y o z , cualquier asignación de verdad de la forma 01 yz también hará que la función tenga el valor de 0. Esta habilidad de “corto circuito” es una de las razones por las que los BDDs tienden a ser representaciones concisas de las funciones booleanas. \square

En la figura 12.31 los nodos interiores se encuentran en rangos; cada rango tiene nodos con una variable específica como etiqueta. Aunque no es un requerimiento absoluto, es conveniente limitarnos a los *BDDs ordenados*. En un BDD ordenado, hay un orden x_1, x_2, \dots, x_n para las variables, y cada vez que hay una arista de un nodo padre etiquetado como x_i a un nodo hijo etiquetado como x_j , entonces $i < j$. Más adelante veremos que es más fácil operar con los BDDs ordenados, y de aquí en adelante supondremos que todos los BDDs están ordenados.

Observe también que los BDDs son, no árboles. Por lo general, las hojas 0 y 1 no sólo tendrán muchos padres, sino que los nodos interiores también pueden tener varios padres. Por ejemplo, el extremo derecho de los nodos etiquetados como z en la figura 12.31 tiene dos padres. Esta combinación de nodos que resultaría en la misma decisión es otra razón por las que los BDDs tienden a ser concisos.

12.7.2 Transformaciones en BDDs

En la explicación anterior pasamos por alto dos simplificaciones en los BDDs que ayudan a que sean más concisos:

1. *Corto circuito*: Si las aristas superior e inferior de un nodo N van al mismo nodo M , entonces podemos eliminar a N . Las aristas que entran a N van ahora a M .
2. *Mezcla de nodos*: Si dos nodos N y M tienen aristas bajos que van al mismo nodo, y también tienen aristas superiores que van al mismo nodo, entonces podemos mezclar a N con M . Las aristas que entran a N o a M van al nodo mezclado.

También es posible ejecutar estas transformaciones en dirección opuesta. En especial, podemos introducir un nodo a lo largo de una arista de N a M . Ambas aristas que provienen del nodo introducido van a M , y la arista que proviene de N ahora va al nodo introducido. Sin embargo, observe que la variable asignada al nuevo nodo debe ser una de las que se encuentran entre las variables de N y M en el orden. La figura 12.32 muestra las dos transformaciones en forma esquemática.

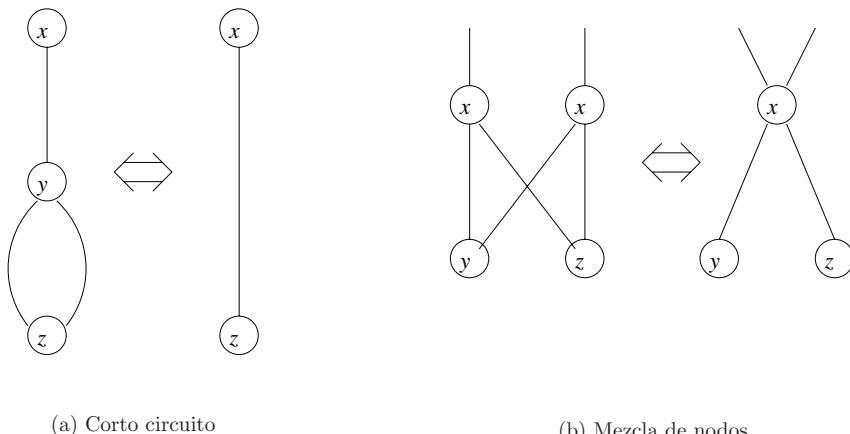


Figura 12.32: Transformaciones en BDDs

12.7.3 Representación de las relaciones mediante BDDs

Las relaciones con las que hemos estado tratando tienen componentes que se toman de los “dominios”. Un dominio para un componente de una relación es el conjunto de posibles valores que pueden tener las tuplas en ese componente. Por ejemplo, la relación $apnt(V, H)$ tiene el dominio de todas las variables del programa para su primer componente, y el dominio de todas las instrucciones de creación de objetos para el segundo componente. Si un dominio tiene más de 2^{n-1} posibles valores pero no más de 2^n valores, entonces requiere n bits o variables booleanas para representar los valores en ese dominio.

Una tupla en una relación puede entonces considerarse como una asignación de verdad a las variables que representan valores en los dominios para cada uno de los componentes de la tupla. Podemos ver una relación como una función booleana que devuelve el valor verdadero para todas y sólo esas asignaciones de verdad que representan a las tuplas en la relación. Un ejemplo aclarará estas ideas.

Ejemplo 12.28: Considere una relación $r(A, B)$ tal que el dominio de A y B está compuesto por $\{a, b, c, d\}$. Debemos codificar a mediante los bits 00, b mediante 01, c mediante 10, y d mediante 11. Hagamos que las tuplas de la relación r sean:

A	B
a	00
a	01
d	10

Usemos las variables booleanas wx para codificar el primer componente (A) y las variables yz para codificar el segundo componente (B). Entonces, la relación r se convierte en:

w	x	y	z
0	0	0	1
0	0	1	0
1	1	1	0

Es decir, la relación r se ha convertido en la función booleana que es verdadera para las tres asignaciones de verdad $wxyz = 0001, 0010$ y 1110 . Observe que estas tres secuencias de bits son exactamente las que etiquetan los caminos de la raíz a la hoja 1 en la figura 12.31. Es decir, el BDD en esa figura representa a la relación r , si se utiliza la codificación antes descrita. \square

12.7.4 Operaciones relacionales como operaciones BDD

Ahora podemos ver cómo representar las relaciones como BDDs. Pero para implementar un algoritmo como el 12.18 (evaluación incremental de programas en Datalog), tenemos que manipular los BDDs de una forma que refleje cómo se manipulan las mismas relaciones. He aquí las operaciones principales sobre las relaciones que debemos realizar:

1. *Inicialización:* Debemos crear un BDD que represente una sola tupla de una relación. Ensamblaremos éstos en BDDs que representen relaciones extensas, mediante el uso de la unión.
2. *Unión:* Para usar la unión de las relaciones, tomamos el OR lógico de las funciones booleanas que representan a las relaciones. Esta operación se necesita no sólo para construir las relaciones iniciales, sino también para combinar los resultados de varias reglas para el mismo predicado de encabezado, y para acumular nuevos hechos en el conjunto de hechos anteriores, como en el Algoritmo 12.18 incremental.
3. *Proyección:* Al evaluar el cuerpo de una regla, debemos construir la relación de encabezado que está implícita mediante las tuplas verdaderas del cuerpo. En términos del BDD que representa a la relación, tenemos que eliminar los nodos que se etiquetan mediante esas variables booleanas que no representan componentes del encabezado. Tal vez también tengamos que renombrar las variables en el BDD para que correspondan con las variables booleanas para los componentes de la relación de encabezado.
4. *Combinación:* Para encontrar las asignaciones de los valores a las variables que hacen que el cuerpo de una regla sea verdadero, debemos “combinar” las relaciones correspondientes a cada una de las submetas. Por ejemplo, suponga que tenemos dos submetas $r(A, B) \ \& \ s(B, C)$. La combinación de las relaciones para estas submetas es el conjunto de triplets (a, b, c) , de tal forma que (a, b) es una tupla en la relación para r , y (b, c) es una tupla en la relación para s . Más adelante veremos que, después de renombrar las variables booleanas en los BDDs de manera que los componentes para las dos Bs coincidan en los nombres de las variables, la operación sobre los BDDs es similar al AND lógico, que a su vez es similar a la operación OR sobre los BDDs que implementa a la combinación.

BDDs para tuplas individuales

Si deseamos inicializar una relación, debemos tener una forma de construir un BDD para la función que sea verdadera para una sola asignación de verdad. Suponga que las variables booleanas son x_1, x_2, \dots, x_n , y la asignación de verdad es a_1, a_2, \dots, a_n , en donde cada a_i puede ser 0 o 1. El BDD tendrá un nodo N_i para cada x_i . Si $a_i = 0$, entonces la arista superior que proviene de N_i conduce hasta la hoja 0, y la arista inferior conduce hasta N_{i+1} , o hasta la hoja 1 si $i = n$. Si $a_i = 1$, entonces hacemos lo mismo, pero se invierten las aristas superior e inferior.

Esta estrategia proporciona un BDD que comprueba si cada x_i tiene el valor correcto, para $i = 1, 2, \dots, n$. Tan pronto como encontramos un valor incorrecto, saltamos directamente a la hoja 0. Sólo terminamos en la hoja 1 si todas las variables tienen su valor correcto.

Como ejemplo, vea la figura 12.33(b). Este BDD representa la función que es verdadera si, y sólo si $x = y = 0$; es decir, la asignación verdadera 00.

Unión

Vamos a proporcionar con detalle un algoritmo para obtener el OR lógico de los BDDs; es decir, la unión de las relaciones representadas por los BDDs.

Algoritmo 12.29: Unión de BDDs.

ENTRADA: Dos BDDs ordenados con el mismo conjunto de variables, en el mismo orden.

SALIDA: Un BDD que representa la función que es el OR lógico de las dos funciones booleanas representadas por los BDDs de entrada.

MÉTODO: Vamos a describir un procedimiento recursivo para combinar dos BDDs. La inducción está en el tamaño del conjunto de variables que aparecen en los BDDs.

BASE: Cero variables. Ambos BDDs deben ser hojas, etiquetados como 0 o 1. La salida es la hoja etiquetada como 1 si cualquiera de las entradas es 1, o la hoja etiquetada como 0 si ambas son 0.

INDUCCIÓN: Suponga que hay k variables y_1, y_2, \dots, y_k que se encuentran entre los dos BDDs. Haga lo siguiente:

1. Si es necesario, use el corto circuito inverso para agregar una nueva raíz, de manera que ambos BDDs tengan una raíz etiquetada como y_1 .
2. Haga que las dos raíces sean N y M ; haga que sus hijos inferiores sean N_0 y M_0 , y que sus hijos superiores sean N_1 y M_1 . Aplique este algoritmo en forma recursiva a los BDDs con raíz en N_0 y M_0 . Además, aplique este algoritmo en forma recursiva a los BDDs con raíz en N_1 y M_1 . El primero de estos BDDs representa a la función que es verdadera para todas las asignaciones de verdad que tienen $y_1 = 0$ y que hacen verdadero a uno o ambos de los BDDs dados. El segundo representa lo mismo para las asignaciones de verdad con $y_1 = 1$.
3. Cree un nodo etiquetado como y_1 . Su hijo inferior es la raíz del primer BDD construido en forma recursiva, y su hijo superior es la raíz del segundo BDD.
4. Mezcle las dos hojas etiquetadas como 0 y las dos hojas etiquetadas como 1 en el BDD combinado que acaba de construir.
5. Aplique la mezcla y el corto circuito en donde sea posible, para simplificar el BDD.

□

Ejemplo 12.30: En la figura 12.33(a) y (b) hay dos BDDs simples. El primero representa a la función x OR y , y el segundo representa a la función:

$$\text{NOT } x \text{ AND NOT } y$$

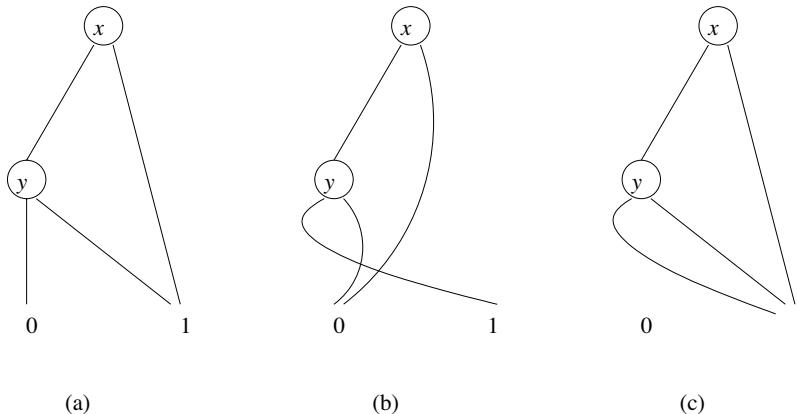


Figura 12.33: Construcción del BDD para un OR lógico

Observe que su OR lógico es la función 1 que siempre es verdadera. Para aplicar el Algoritmo 12.29 a estos dos BDDs, consideramos los hijos inferiores de las dos raíces y los hijos superiores de las dos raíces; vamos a tomar estos últimos primero.

El hijo superior de la raíz en la figura 12.33(a) es 1, y en la figura 12.33(b) es 0. Como estos hijos se encuentran al nivel de hoja, no tenemos que insertar nodos etiquetados como y a lo largo de cada arista, aunque el resultado sería el mismo si hubiéramos optado por hacerlo. El caso base para la unión de 0 y 1 es producir una hoja etiquetada como 1 que se convertirá en el hijo superior de la nueva raíz.

Los hijos inferiores de las raíces en las figuras 12.33(a) y (b) están etiquetados como y , por lo que podemos calcular su BDD de unión en forma recursiva. Estos dos nodos tienen hijos inferiores etiquetados como 0 y 1, por lo que la combinación de sus hijos inferiores es la hoja etiquetada como 1. De igual forma, sus hijos superiores son 1 y 0, por lo que la combinación es de nuevo la hoja 1. Cuando agregamos una nueva raíz etiquetada como x , tenemos el BDD que se ve en la figura 12.33(c).

No hemos terminado, ya que la figura 12.33(c) se puede simplificar. El nodo etiquetado como y tiene en ambos hijos el nodo 1, por lo que podemos eliminar el nodo y y hacer que la hoja 1 sea el hijo inferior de la raíz. Ahora, ambos hijos de la raíz son la hoja 1, por lo que podemos eliminar la raíz. Es decir, el BDD más simple para la unión es la hoja 1, por sí sola. \square

12.7.5 Uso de BDDs para el análisis tipo “apunta a”

El proceso de lograr que funcione el análisis tipo “apunta a” insensible al contexto es ya de por sí poco trivial. El orden de las variables del BDD puede modificar en forma considerable el tamaño de la representación. Se requieren muchas consideraciones, así como la prueba y error, para obtener un orden que permita al análisis completarse con rapidez.

Es aún más difícil lograr que se ejecute el análisis tipo “apunta a” sensible al contexto, debido a los exponencialmente diversos contextos en el programa. En especial, si asignamos

números en forma arbitraria para representar contextos en un grafo de llamadas, no podemos manejar ni siquiera los programas pequeños en Java. Es importante que los contextos se enumeren de tal forma que la codificación binaria del análisis tipo “apunta a” pueda hacerse muy compacta. Dos contextos del mismo método con caminos de llamada similares comparten muchas funcionalidades, por lo que es conveniente enumerar los n contextos de un método en forma consecutiva. De manera similar, como los pares de emisor-receptor para el mismo sitio de llamada comparten muchas similitudes, es conveniente enumerar los contextos de tal forma que la diferencia numérica entre cada par de emisor-receptor de un sitio de llamada sea siempre una constante.

Aun con un esquema de numeración inteligente para los contextos de llamada, sigue siendo difícil analizar con eficiencia los programas extensos en Java. El aprendizaje activo de las máquinas ha demostrado ser útil para derivar un orden de las variables lo bastante eficiente como para manejar las aplicaciones extensas.

12.7.6 Ejercicios para la sección 12.7

Ejercicio 12.7.1: Usando la codificación de símbolos en el ejemplo 12.28, desarrolle un BDD que represente la relación consistente en los tuples (b, b) , (c, a) y (b, a) . Puede ordenar las variables booleanas en cualquier forma que le proporcione el BDD más conciso.

! Ejercicio 12.7.2: Como función de n , ¿cuántos nodos hay en el BDD más conciso que representa a la función or exclusivo en n variables? Es decir, la función es verdadera si un número impar de las n variables son verdaderas y falsa si un número par de ellas son verdaderas.

Ejercicio 12.7.3: Modifique el Algoritmo 12.29 de manera que produzca la intersección (AND lógico) de dos BDDs.

!! Ejercicio 12.7.4: Busque algoritmos para realizar las siguientes operaciones relacionales en los BDDs ordenados que las representan:

- Proyecte algunas de las variables booleanas. Es decir, la función representada debe ser verdadera para una asignación de verdad α si hay alguna asignación de verdad para las variables faltantes que, en conjunto con α , hacen que la función original sea verdadera.
- Combine dos relaciones r y s , combinando una tupla de r con uno de s cada vez que estas tuplas concuerden con los atributos que r y s tienen en común. En realidad es suficiente considerar el caso en el que las relaciones sólo tienen dos componentes, y uno de cada relación coincide; es decir, las relaciones son $r(A, B)$ y $s(B, C)$.

12.8 Resumen del capítulo 12

- ◆ *Análisis interprocedural:* Un análisis de flujo de datos que rastrea la información a través de los límites de los procedimientos se considera como interprocedural. Muchos análisis, como los análisis tipo “apunta a”, sólo pueden realizarse en un forma significativa si son interprocedural.

- ◆ *Sitios de llamada:* Los programas llaman a los procedimientos en ciertos puntos conocidos como sitios de llamada. El procedimiento llamado en un sitio puede ser evidente, o puede ser ambiguo, en caso de que la llamada sea indirecta a través de un apuntador o una llamada de un método virtual que tiene varias implementaciones.
- ◆ *Grafos de llamadas:* Un grafo de llamadas para un programa es un grafo bipartido con nodos para los sitios de llamada y nodos para los procedimientos. Una arista va de un nodo de sitio de llamada hacia un nodo de procedimiento, si es que se puede hacer una llamada a ese procedimiento en el sitio.
- ◆ *Poner en línea:* Siempre y cuando no haya recursividad en un programa, podemos en principio sustituir todas las llamadas a procedimientos por copias de su código, y usar el análisis intraprocedural en el programa resultante. Este análisis es, en efecto, interprocedural.
- ◆ *Sensibilidad al flujo y sensibilidad al contexto:* Un análisis de flujo de datos que produce hechos que dependen en la ubicación en el programa se considera como sensible al flujo. Si el análisis produce hechos que dependen del historial de las llamadas a procedimientos, se considera como sensible al contexto. Un análisis de flujo de datos puede ser sensible al flujo o al contexto, a ambos o a ninguno.
- ◆ *Análisis sensible al contexto basado en la clonación:* En principio, una vez que establecemos los distintos contextos en los que se puede llamar un procedimiento, podemos imaginar que hay un clon de cada procedimiento para cada contexto. De esa forma, un análisis insensible al contexto sirve como un análisis sensible al contexto.
- ◆ *Análisis sensible al contexto basado en el resumen:* Otro método para el análisis interprocedural extiende la técnica de análisis basado en regiones que describimos para el análisis intraprocedural. Cada procedimiento tiene una función de transferencia y se trata como una región, en cada lugar en donde se llama a ese procedimiento.
- ◆ *Aplicaciones del análisis interprocedural:* Una aplicación importante que requiere del análisis interprocedural es la detección de las vulnerabilidades del software. A menudo, éstas se caracterizan por hacer que un procedimiento lea datos de una fuente de entrada no confiable, y que otro procedimiento utilice esos datos de forma que puedan explotarse.
- ◆ *Datalog:* El lenguaje Datalog es una notación simple para las reglas if-then que pueden usarse para describir los análisis de flujo de datos de alto nivel. Las colecciones de reglas de Datalog, o de programas en Datalog, pueden evaluarse mediante el uso de uno de varios algoritmos estándar.
- ◆ *Reglas de Datalog:* Una regla de Datalog consiste en un cuerpo (antecedente) y un encabezado (consecuente). El cuerpo es uno o más átomos, y el encabezado es un átomo. Los átomos son predicados que se aplican a los argumentos que son variables o constantes. Los átomos del cuerpo se conectan mediante un AND lógico, y un átomo en el cuerpo puede negarse.

- ◆ *Predicados IDB y EDB:* A los predicados EDB en un programa en Datalog se les proporcionan sus hechos verdaderos a-priori. En un análisis de flujo de datos, estos predicados corresponden a los hechos que pueden obtenerse del código que se está analizando. Los predicados IDB se definen mediante las mismas reglas y corresponden, en un análisis de flujo de datos, a la información que tratamos de extraer del código que se está analizando.
- ◆ *Evaluación de los programas en Datalog:* Para aplicar las reglas, sustituimos las constantes por variables que hacen que el cuerpo sea verdadero. Cada vez que hacemos esto, inferimos que el encabezado, con la misma sustitución para las variables, también es verdadero. Esta operación se repite hasta que no puedan inferirse más hechos.
- ◆ *Evaluación incremental de los programas en Datalog:* Puede obtenerse una mejora en la eficiencia al realizar una evaluación incremental. Realizamos una serie de rondas. En una ronda, consideramos sólo sustituciones de constantes por variables que hacen que, por lo menos, un átomo del cuerpo sea un hecho que acaba de descubrirse en la ronda anterior.
- ◆ *Análisis de apuntadores en Java:* Podemos modelar el análisis de apuntadores en Java mediante un framework en el cual hay variables de referencia que apuntan a objetos del montículo, los cuales pueden tener campos que apuntan a otros objetos del montículo. Un análisis de apuntadores insensible puede escribirse como un programa en Datalog que infiere dos tipos de hechos: una variable puede apuntar a un objeto montículo, o un campo de un objeto montículo puede apuntar a otro objeto del montículo.
- ◆ *Información de tipos para mejorar el análisis de apuntadores:* Podemos obtener un análisis de apuntadores más preciso si aprovechamos el hecho de que las variables de referencia sólo pueden apuntar a objetos del montículo que sean del mismo tipo que la variable o un subtipo.
- ◆ *Análisis de apuntadores interprocedural:* Para realizar el análisis interprocedural, debemos agregar reglas que reflejen la forma en que los parámetros pasan y devuelven los valores asignados a las variables. En esencia, estas reglas son las mismas que para copiar una variable de referencia a otra.
- ◆ *Descubrimiento del grafo de llamadas:* Como Java tiene métodos virtuales, el análisis interprocedural requiere que primero limitemos qué procedimientos pueden llamarse en un sitio de llamada dado. La principal forma de descubrir los límites sobre qué puede llamarse y en qué parte, es analizar los tipos de los objetos y aprovechar el hecho de que el método actual al que hace referencia una llamada a un método virtual debe pertenecer a una clase apropiada.
- ◆ *Análisis sensible al contexto:* Cuando los procedimientos son recursivos, debemos condensar la información contenida en las cadenas de llamadas en un número finito de contextos. Una manera efectiva de hacer esto es retirar de la cadena de llamadas cualquier sitio de llamada en donde un procedimiento llame a otro procedimiento (tal vez a sí mismo) con el que sea mutuamente recursivo. Usando esta representación, podemos modificar las reglas

para el análisis de apuntadores interprocedural, de manera que el contexto se acarree en predicados; este método simula el análisis basado en la clonación.

- ◆ *Diagramas de decisiones binarios:* Los BDDs son una representación concisa de las funciones booleanas mediante GDAs con raíz. Los nodos interiores corresponden a las variables booleanas y tienen dos hijos, inferior (que representa el valor de verdad 0) y superior (que representa 1). Hay dos hojas etiquetadas como 0 y 1. Una asignación de verdad hace que la función representada sea verdadera si, y sólo si el camino de la raíz en la cual vamos al hijo inferior si la variable en un nodo es 0 y al hijo superior en caso contrario, conduce a la hoja 1.
- ◆ *BDDs y relaciones:* Un BDD puede servir como una representación concisa de uno de los predicados en un programa en Datalog. Las constantes se codifican como asignaciones de verdad para una colección de variables booleanas, y la función representada por el BDD es verdadera si, y sólo si las variables booleanas representan un hecho verdadero para ese predicado.
- ◆ *Implementación del análisis de flujo de datos mediante BDDs:* Cualquier análisis de flujo de datos que pueda expresarse como reglas de Datalog puede implementarse mediante manipulaciones en los BDDs que representan a los predicados involucrados en esas reglas. A menudo, esta representación conduce a una implementación más eficiente del análisis de flujo de datos que cualquier otro método conocido.

12.9 Referencias para el capítulo 12

Podemos encontrar algunos de los conceptos básicos en un análisis interprocedural en [1, 6, 7 y 21]. Callahan y sus colegas [11] describen un algoritmo de propagación de constantes interprocedural.

Steengaard [22] publicó el primer análisis de alias de apuntadores escalable. Es insensible al contexto, insensible al flujo y está basado en equivalencias. Andersen [2] derivó una versión insensible al contexto del análisis tipo “apunta a” basado en la inclusión. Después, Heintze y Tardieu [15] describieron un algoritmo eficiente para este análisis. Fähndrich, Rehof y Das [14] presentaron un análisis insensible al contexto, insensible al flujo y basado en equivalencias que escala a programas extensos como `gcc`. Ghiya y Hendren [13] diseñaron un algoritmo tipo “apunta a” basado en la clonación, insensible al contexto, insensible al flujo y basado en la inclusión, que es notable entre los intentos anteriores de crear un análisis tipo “apunta a” insensible al contexto, basado en la inclusión.

Los diagramas de decisiones binarios (BDDs) aparecieron por primera vez en Bryant [9]. Berndl y sus colegas [4] lo usaron por primera vez para el análisis de flujo de datos. La aplicación de BDDs al análisis de apuntadores insensible se reporta por Zhu [25] y Berndl, junto con sus colegas [8]. Whaley y Lam [24] describen el primer algoritmo sensible al contexto, insensible al flujo y basado en inclusiones, que ha demostrado su utilidad en aplicaciones reales. El documento describe una herramienta llamada `bddbddb` que traduce en código BDD de forma automática el análisis descrito en Datalog. Milanova, Rountev y Ryder [18] introdujeron la sensibilidad a los objetos.

Para una explicación sobre Datalog, consulte a Ullman y Widom [23]. Consulte también a Lam y sus colaboradores [16] para una explicación sobre la conexión del análisis de flujo de datos a Datalog.

El comprobador de código Metal se describe por Engler y sus colaboradores [12]; Bush, Pincus y Sielaff [10] crearon el comprobador PREFix. Ball y Rajamani [4] desarrollaron un motor de análisis de programas llamado SLAM, usando la comprobación de modelos y la ejecución simbólica para simular todos los posibles comportamientos de un sistema. Ball y sus colaboradores [5] han creado una herramienta de análisis estático llamada SDV, basada en SLAM, para buscar los errores de uso de la API en programas controladores de dispositivos en C, mediante la aplicación de BDDs a la comprobación de modelos.

Livshits y Lam [17] describen cómo puede usarse el análisis tipo “apunta a” sensible al contexto para buscar vulnerabilidades de SQL en las aplicaciones Web en Java. Ruwase y Lam [20] describen cómo llevar el registro de las extensiones de arreglos e insertar comprobaciones dinámicas de límites en forma automática. Rinard y sus colaboradores [19] describen cómo extender arreglos en forma dinámica, para dar cabida al contenido desbordado. Avots y sus colaboradores [3] extienden a C el análisis en Java tipo “apunta a” sensible al contexto, y muestran cómo puede usarse para reducir el costo de la detección dinámica de desbordamientos de búfer.

1. Allen, F. E., “Interprocedural data flow analysis”, *Proc. IFIP Congress 1974*, pp. 398-402, Holanda del Norte, Ámsterdam, 1974.
2. Andersen, L., *Program Analysis and Specialization for the C Programming Language*, tesis Ph. D., DIKU, Universidad de Copenhague, Dinamarca, 1994.
3. Avots, D., M. Dalton, V. B. Livshits y M. S. Lam, “Improving software security with a C pointer analysis”, *ICSE 2005: Proc. 27th International Conference on Software Engineering*, pp. 332-341.
4. Ball, T. y S. K. Rajamani, “A symbolic model checker for boolean programs”, *Proc. SPIN 2000 Workshop on Model Checking of Software*, pp. 113-130.
5. Ball, T., E. Bounimova, B. Cook, V. Levin, J. Lichtenber, C. McGarvey, B. Ondrussek, S. Rajamani y A. Ustuner, “Thorough static analysis of device drivers”, *EuroSys* (2006), pp. 73-85.
6. Banning, J. P., “An efficient way to find the side effects of procedural calls and the aliases of variables”, *Proc. Sixth Annual Symposium on Principles of Programming Languages* (1979), pp. 29-41.
7. Barth, J. M., “A practical interprocedural data flow analysis algorithm”, *Comm. ACM* **21**:9 (1978), pp. 724-736.
8. Berndl, M., O. Lohtak, F. Qian, L. Hendren y N. Umanee, “Points-to analysis using BDD’s”, *Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pp. 103-114.
9. Bryant, R. E., “Graph-based algorithms for Boolean function manipulation”, *IEEE Trans. on Computers* **C-35**:8 (1986), pp. 677-691.

10. Bush, W. R., J. D. Pincus y D. J. Siefaff, "A static analyzer for finding dynamic programming errors", *Software-Practice and Experience*, **30**:7 (2000), pp. 775-802.
11. Callahan, D., K. D. Cooper, K. Kennedy y L. Torczon, "Interprocedural constant propagation", *Proc. SIGPLAN 1986 Symposium on Compiler Construction, SIGPLAN Notices*, **21**:7 (1986), pp. 152-161.
12. Engler, D., B. Chelf, A. Chou y S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions", *Proc. Sixth USENIX Conference on Operating Systems Design and Implementation* (2000), pp. 1-16.
13. Emami, M., R. Ghiya y L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers", *Proc. SIGPLAN Conference on Programming Language Design and Implementation* (1994), pp. 224-256.
14. Fähndrich, M., J. Rehof y M. Das, "Scalable context-sensitive flow analysis using instantiation constraints", *Proc. SIGPLAN Conference on Programming Language Design and Implementation* (2000), pp. 253-263.
15. Heintze, N. y O. Tardieu, "Ultra-fast aliasing analysis using CLA: a million lines of C code in a second", *Proc of the SIGPLAN Conference on Programming Language Design and Implementation* (2001), pp. 254-263.
16. Lam, M. S., J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin y C. Unkel, "Context-sensitive program analysis as database queries", *Proc 2005 ACM Symposium on Principles of Database Systems*, pp. 1-12.
17. Livshits, V. B. y M. S. Lam, "Finding security vulnerabilities in Java applications using static analysis", *Proc. 14th USENIX Security Symposium* (2005), pp. 271-286.
18. Milanova, A., A. Rountev y B. G. Ryder, "Parametrized object sensitivity for points-to and side-effect analyses for Java", *Proc. 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1-11.
19. Rinard, M., C. Cadar, D. Dumitran, D. Roy y T. Leu, "A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors)", *Proc. 2004 Annual Computer Security Applications Conference*, pp. 82-90.
20. Ruwase, O. y M. S. Lam, "A practical dynamic buffer overflow detector", *Proc. 11th Annual Network and Distributed System Security Symposium* (2004), pp. 159-169.
21. Sharir, M. y A. Pnueli, "Two approaches to interprocedural data flow analysis", en *Program Flow Analysis: Theory and Applications*, de S. Muchnick y N. Jones (eds.), capítulo 7, pp. 189-234. Prentice-Hall, Upper Saddle River NJ, 1981.
22. Steensgaard, B., "Points-to analysis in linear time", *Twenty-Third ACM Symposium on Principles of Programming Languages* (1996).

23. Ullman, J. D. y J. Widom, *A First Course in Database Systems*, Prentice-Hall, Upper Saddle River NJ, 2002.
24. Whaley, J. y M. S. Lam, “Cloning-based context sensitive pointer alias analysis using binary decision diagrams”, *Proc. ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pp. 131-144.
25. Zhu, J., “Symbolic Pointer Analysis”, *Proc. International Conference in Computer-Aided Design* (2002), pp. 150-157.

Apéndice A

Un front-end completo

El front-end completo para el compilador de este apéndice se basa en el compilador simple de las secciones 2.5 a 2.8, descrito de manera informal. La diferencia principal del capítulo 2 es que el front-end genera código de salto para las expresiones booleanas, como en la sección 6.6. Empezamos con la sintaxis del lenguaje de código fuente, descrito por una gramática que debe adaptarse para el análisis sintáctico descendente

El código en Java para el traductor consiste en cinco paquetes: `main`, `analizadorLexico`, `simbolos`, `analizador` e `inter`. El paquete `inter` contiene las clases para las construcciones del lenguaje en la sintaxis abstracta. Como el código para el analizador sintáctico interactúa con el resto de los paquetes, se describirá al final. Cada paquete se almacena como un directorio separado, con un archivo por clase.

Al pasar por el analizador sintáctico, el programa fuente consiste en un flujo de tokens, por lo que la orientación a objetos tiene poco que ver con el código para el analizador sintáctico. Al salir de éste, el programa fuente consiste en un árbol sintáctico, en el cual las construcciones o nodos se implementan como objetos. Estos objetos se encargan de lo siguiente: construir un nodo del árbol sintáctico, comprobar tipos y generar código intermedio de tres direcciones (vea el paquete `inter`).

A.1 El lenguaje de código fuente

Un programa en el lenguaje consiste en un bloque con declaraciones e instrucciones opcionales. El token **basico** representa a los tipos básicos.

```
programa  →  bloque
bloque   →  { decls instrs }
decls    →  decls decl | ε
decl     →  tipo id ;
tipo     →  tipo [ num ] | basico
instrs   →  instrs stmt | ε
```

Al tratar a las asignaciones como instrucciones, en vez de operadores dentro de expresiones, se simplifica la traducción.

Comparación entre orientación a objetos y orientación a fases

Con un método orientado a objetos, todo el código para una construcción se recolecta en la clase del constructor. De manera alternativa, con un método orientado a fases, el código se agrupa por fase, de manera que un procedimiento de comprobación de tipos tendría una instrucción `case` para cada construcción, y un procedimiento de generación de código tendría una instrucción `case` para cada construcción, y así sucesivamente.

La concesión es que un método orientado a objetos facilita el proceso de modificar o agregar una construcción, como las instrucciones “`for`”, y el método orientado a fases facilita el proceso de modificar o agregar una fase, como una comprobación de tipos. Con los objetos, para agregar una nueva construcción se escribe una clase autocontenido, pero una modificación a una fase, como el insertar código para las coerciones, requiere modificaciones a través de todas las clases afectadas. Con las fases, una nueva construcción puede provocar como resultado modificaciones a través de los procedimientos para las fases.

```

instr  →  loc = bool ;
         |  if ( bool ) instr
         |  if ( bool ) instr else instr
         |  while ( bool ) instr
         |  do instr while ( bool ) ;
         |  break ;
         |  bloque
loc   →  loc [ bool ] | id

```

Las producciones para las expresiones manejan la asociatividad y la precedencia de los operadores. Utilizan un símbolo no terminal para cada nivel de precedencia, y un símbolo no terminal (*factor*) para las expresiones entre paréntesis, identificadores, referencias a arreglos y constantes.

```

bool  →  bool || comb | comb
comb  →  comb && igualdad | igualdad
igualdad  →  igualdad == rel | igualdad != rel | rel
rel   →  expr < expr | expr <= expr | expr >= expr |
         expr > expr | expr
expr  →  expr + term | expr - term | term
term   →  term * unario | term / unario | unario
unario  →  ! unario | - unario | factor
factor  →  ( bool ) | loc | num | real | true | false

```

A.2 Main

La ejecución empieza en el método `main` en la clase `Main`. El método `main` crea un analizador léxico (`analizadorLexico`) y un analizador sintáctico (`analizador`), y después llama al método `programa` en el analizador sintáctico:

```

1) package main;           // Archivo Main.java
2) import java.io.*; import analizadorLexico.*; import analizador.*;
3) public class Main {
4)     public static void main(String[] args) throws IOException {
5)         AnalizadorLexico lex = new AnalizadorLexico();
6)         Analizador analizar = new Analizador(lex);
7)         analizar.programa();
8)         System.out.write('\n');
9)     }
10) }

```

A.3 Analizador léxico

El paquete `analizadorLexico` es una extensión del código para el analizador léxico de la sección 2.6.5. La clase `Etiqueta` define las constantes para los tokens:

```

1) package analizadorLexico;           // Archivo Etiqueta.java
2) public class Etiqueta {
3)     public final static int
4)         AND = 256, BASIC = 257, BREAK = 258, DO = 259, ELSE = 260,
5)         EQ = 261, FALSE = 262, GE = 263, ID = 264, IF = 265,
6)         INDEX = 266, LE = 267, MINUS = 268, NE = 269, NUM = 270,
7)         OR = 271, REAL = 272, TEMP = 273, TRUE = 274, WHILE = 275;
8) }

```

Tres de las constantes, `INDEX`, `MINUS` y `TEMP`, no son tokens léxicos; se utilizarán en los árboles sintácticos.

Las clases `Token` y `Num` son como en la sección 2.6.5, sólo que se agregó el método `toString`:

```

1) package analizadorLexico;           // Archivo Token.java
2) public class Token {
3)     public final int etiqueta;
4)     public Token(int t) {etiqueta = t;}
5)     public String toString() {return "" + (char)etiqueta;}
6) }

1) package analizadorLexico;           // Archivo Num.java
2) public class Num extends Token {
3)     public final int valor;
4)     public Num(int v) {super(Etiqueta.NUM); valor = v; }
5)     public String toString() { return "" + valor; }
6) }

```

La clase `Palabra` administra los lexemas para las palabras reservadas, identificadores y tokens compuestos como `&&`. También es útil para administrar la forma escrita de los operadores en el código intermedio, como el menos unario; por ejemplo, el texto fuente `-2` tiene la forma intermedia `minus 2`.

```

1) package analizadorLexico;           // Archivo Palabra.java
2) public class Palabra extends Token {
3)     public String lexema = "";
4)     public Palabra(String s, int etiqueta) { super(etiqueta); lexema = s; }
5)     public String toString() { return lexema; }
6)     public static final Palabra
7)         and = new Palabra( "&&", Etiqueta.AND ), or = new Palabra( "||", Etiqueta.OR ),
8)         eq = new Palabra( "==" , Etiqueta.EQ ), ne = new Palabra( "!=" , Etiqueta.NE ),

```

```

9)     le = new Palabra( "<=", Etiqueta.LE ), ge = new Palabra( ">=", Etiqueta.GE ),
10)    minus = new Palabra( "minus", Etiqueta_MINUS ),
11)    True = new Palabra( "true", Etiqueta.TRUE ),
12)    False = new Palabra( "false", Etiqueta.FALSE ),
13)    temp = new Palabra( "t", Etiqueta.TEMP );
14) }
```

La clase **Real** es para los números de punto flotante:

```

1) package analizadorLexico;           // Archivo Real.java
2) public class Real extends Token {
3)     public final float valor;
4)     public Real(float v) {super(Etiqueta.REAL); valor = v; }
5)     public String toString() {return "" + valor; }
6) }
```

El método principal en la clase **AnalizadorLexico**, la función **explorar**, reconoce números, identificadores y palabras reservadas, como vimos en la sección 2.6.5.

Las líneas 9-13 en la clase **AnalizadorLexico** reservan las palabras clave seleccionadas. Las líneas 14-16 reservan los lexemas para los objetos definidos en cualquier otra parte. Los objetos **Palabra.True** y **Palabra.False** se definen en la clase **Palabra**. Los objetos para los tipos básicos **int**, **char**, **bool** y **float** se definen en la clase **Tipo**, una subclase de **Palabra**. La clase **Tipo** es del paquete **simbolos**.

```

1) package analizadorLexico;           // Archivo AnalizadorLexico.java
2) import java.io.*; import java.util.*; import simbolos.*;
3) public class AnalizadorLexico {
4)     public static int linea = 1;
5)     char preanalisis = ' ';
6)     Hashtable palabras = new Hashtable();
7)     void reservar(Palabra w) { palabras.put(w.lexema, w); }
8)     public AnalizadorLexico() {
9)         reservar( new Palabra("if", Etiqueta.IF) );
10)        reservar( new Palabra("else", Etiqueta.ELSE) );
11)        reservar( new Palabra("while", Etiqueta.WHILE) );
12)        reservar( new Palabra("do", Etiqueta.DO) );
13)        reservar( new Palabra("break", Etiqueta.BREAK) );
14)        reservar( Palabra.True ); reservar( Palabra.False );
15)        reservar( Tipo.Int ); reservar( Tipo.Char );
16)        reservar( Tipo.Bool ); reservar( Tipo.Float );
17)    }
```

La función **readch()** (línea 18) se utiliza para leer el siguiente carácter de entrada y colarlo en la variable **preanalisis**. El nombre **readch** se reutiliza o se sobrecarga (líneas 19-24) para ayudar a reconocer los tokens compuestos. Por ejemplo, una vez que se ve la entrada **<**, la llamada **readch('=')** lee el siguiente carácter y lo coloca en **preanalisis**, y comprueba si es **=**.

```

18)     void readch() throws IOException { preanalisis = (char) System.in.read(); }
19)     boolean readch(char c) throws IOException {
20)         readch();
21)         if( preanalisis != c ) return false;
22)         preanalisis = ' ';
23)         return true;
24)     }
```

La función **explorar** empieza ignorando el espacio en blanco (líneas 26-30). Reconoce los tokens compuestos como <= (líneas 31-44) y los números como 365 y 3.14 (líneas 45-58), antes de recolectar palabras (líneas 59-70).

```

25)     public Token explorar() throws IOException {
26)         for( ; ; readch() ) {
27)             if( preanalisis == ' ' || preanalisis == '\t' ) continue;
28)             else if( preanalisis == '\n' ) linea = linea + 1;
29)             else break;
30)         }
31)         switch( preanalisis ) {
32)             case '&':
33)                 if( readch('&') ) return Palabra.and; else return new Token('&');
34)             case '|':
35)                 if( readch('|') ) return Palabra.or; else return new Token('|');
36)             case '=':
37)                 if( readch('=>') ) return Palabra.eq; else return new Token('=>');
38)             case '!':
39)                 if( readch('=>') ) return Palabra.ne; else return new Token('!=');
40)             case '<':
41)                 if( readch('=>') ) return Palabra.le; else return new Token('<');
42)             case '>':
43)                 if( readch('=>') ) return Palabra.ge; else return new Token('>');
44)         }
45)         if( Character.isDigit( preanalisis ) ) {
46)             int v = 0;
47)             do {
48)                 v = 10*v + Character.digit( preanalisis, 10 ); readch();
49)             } while( Character.isDigit( preanalisis ) );
50)             if( preanalisis != '.' ) return new Num(v);
51)             float x = v; float d = 10;
52)             for(;;) {
53)                 readch();
54)                 if( ! Character.isDigit( preanalisis ) ) break;
55)                 x = x + Character.digit( preanalisis, 10 ) / d; d = d*10;
56)             }
57)             return new Real(x);
58)         }
59)         if( Character.isLetter( preanalisis ) ) {
60)             StringBuffer b = new StringBuffer();
61)             do {
62)                 b.append( preanalisis ); readch();
63)             } while( Character.isLetterOrDigit( preanalisis ) );
64)             String s = b.toString();
65)             Palabra w = (Palabra)palabras.get(s);
66)             if( w != null ) return w;
67)             w = new Palabra(s, Etiqueta.ID);
68)             palabras.put(s, w);
69)             return w;
70)         }

```

Por último, los caracteres restantes se devuelven como tokens (líneas 71-72).

```

71)     Token tok = new Token( preanalisis ); preanalisis = ' ';
72)     return tok;
73) }
74) }

```

A.4 Tablas de símbolos y tipos

El paquete `simbolos` implementa a las tablas de símbolos y los tipos.

En esencia, la clase `Ent` permanece sin cambios, como en la figura 2.37. Mientras que la clase `AnalizadorLexico` asigna cadenas a palabras, la clase `Ent` asigna tokens de palabras a objetos de la clase `Id`, la cual se define en el paquete `inter`, junto con las clases para expresiones e instrucciones.

```

1) package simbolos;                                // Archivo Ent.java
2) import java.util.*; import analizadorLexico.*; import inter.*;
3) public class Ent {
4)     private Hashtable tabla;
5)     protected Ent ant;
6)     public Ent(Ent n) { tabla = new Hashtable(); ant = n; }
7)     public void put(Token w, Id i) { tabla.put(w, i); }
8)     public Id get(Token w) {
9)         for( Env e = this; e != null; e = e.ant ) {
10)             Id encontro = (Id)(e.tabla.get(w));
11)             if( encontro != null ) return encontro;
12)         }
13)         return null;
14)     }
15) }
```

Definimos la clase `Tipo` como una subclase de `Palabra`, ya que los nombres de los tipos básicos como `int` son sólo palabras reservadas, que el analizador léxico asigna de los lexemas a los objetos apropiados. Los objetos para los tipos básicos son `Tipo.Int`, `Tipo.Float`, `Tipo.Char` y `Tipo.Bool` (líneas 7-10). En todos ellos el campo heredado `etiqueta` se establece en `Etiqueta.BASIC`, por lo que el analizador sintáctico las trata a todas por igual.

```

1) package simbolos;                                // Archivo Tipo.java
2) import analizadorLexico.*;
3) public class Tipo extends Palabra {
4)     public int anchura = 0;           // anchura se usa para asignación de almacenamiento
5)     public Tipo(String s, int etiqueta, int w) { super(s, etiqueta); anchura = w; }
6)     public static final Tipo
7)         Int = new Tipo( "int", Etiqueta.BASIC, 4 ),
8)         Float = new Tipo( "float", Etiqueta.BASIC, 8 ),
9)         Char = new Tipo( "char", Etiqueta.BASIC, 1 ),
10)        Bool = new Tipo( "bool", Etiqueta.BASIC, 1 );
```

Las funciones `numerico` (líneas 11-14) y `max` (líneas 15-20) son útiles para las conversiones de tipos.

```

11)    public static boolean numerico(Tipo p) {
12)        if (p == Tipo.Char || p == Tipo.Int || p == Tipo.Float) return true;
13)        else return false;
14)    }
15)    public static Tipo max(Tipo p1, Tipo p2) {
16)        if ( ! numerico(p1) || ! numerico(p2) ) return null;
17)        else if (p1 == Tipo.Float || p2 == Tipo.Float) return Tipo.Float;
18)        else if (p1 == Tipo.Int || p2 == Tipo.Int) return Tipo.Int;
19)        else return Tipo.Char;
20)    }
21) }
```

Se permiten las conversiones entre los tipos “numéricos” `Tipo.Char`, `Tipo.Int` y `Tipo.Float`. Cuando se aplica un operador aritmético a dos tipos numéricos, el resultado es el “máximo” (máx) de los dos tipos.

Los arreglos son el único tipo construido en el lenguaje fuente. La llamada a `super` en la línea 7 establece el campo `anchura`, que es esencial para los cálculos de direcciones. También establece `lexema` y `tok` a valores predeterminados que no se utilizan.

```

1) package symbols;                                // Archivo Arreglo.java
2) import analizadorLexico.*;
3) public class Arreglo extends Tipo {
4)     public Tipo de;                            // arreglo *de* tipo
5)     public int tamanio = 1;                     // número de elementos
6)     public Arreglo(int tm, Tipo p) {
7)         super("[]", Etiqueta.INDEX, tm*p.anchura); tamanio = tm; de = p;
8)     }
9)     public String toString() { return "[" + tamanio + "] " + de.toString(); }
10) }
```

A.5 Código intermedio para las expresiones

El paquete `inter` contiene la jerarquía de clases `Nodo`, la cual tiene dos subclases: `Expr` para los nodos de expresiones e `Instr` para los nodos de instrucciones. Esta sección presenta a `Expr` y sus subclases. Algunos de los métodos en `Expr` tratan con valores booleanos y código de salto; los veremos en la sección A.6, junto con el resto de las subclases de `Expr`.

Los nodos en el árbol sintáctico se implementan como objetos de la clase `Nodo`. Para reportar errores, el campo `linealex` (línea 4, archivo `Nodo.java`) guarda el número de línea de código fuente de la construcción en este nodo. Las líneas 7-10 se utilizan para emitir el código de tres direcciones.

```

1) package inter;                                // Archivo Nodo.java
2) import analizadorLexico.*;
3) public class Nodo {
4)     int linealex = 0;
5)     Nodo() { linealex = AnalizadorLexico.linea; }
6)     void error(String s) {throw new Error("cerca de la linea "+lineales+": "+s); }
7)     static int etiquetas = 0;
8)     public int nuevaEtiqueta() { return ++etiquetas; }
9)     public void emitirEtiqueta(int i) { System.out.print("L" + i + ":" ); }
10)    public void emitir(String s) { System.out.println("\t" + s); }
11) }
```

Las construcciones de expresiones se implementan mediante subclases de `Expr`. La clase `Expr` tiene los campos `op` y `tipo` (líneas 4-5, archivo `Expr.java`), que representan al operador y el tipo, respectivamente, en un nodo.

```

1) package inter;                                // Archivo Expr.java
2) import analizadorLexico.*; import simbolos.*;
3) public class Expr extends Nodo {
4)     public Token op;
5)     public Tipo tipo;
6)     Expr(Token tok, Tipo p) { op = tok; tipo = p; }
```

El método `gen` (línea 7) devuelve un “término” que puede caber del lado derecho de una instrucción de tres direcciones. Dada la expresión $E = E_1 + E_2$, el método `gen` devuelve un término $x_1 + x_2$, en donde x_1 y x_2 son direcciones para los valores de E_1 y E_2 , respectivamente. El valor de retorno `this` es apropiado, si este objeto es una dirección; por lo general, las subclases de `Expr` vuelven a implementar a `gen`.

El método `reducir` (línea 8) calcula o “reduce” una expresión hasta una sola dirección; es decir, devuelve una constante, un identificador o un nombre temporal. Dada la expresión E , el método `reduce` devuelve un nombre temporal t que contiene el valor de E . De nuevo, `this` es un valor de retorno apropiado, si este objeto es una dirección.

Aplazaremos la explicación sobre los métodos `salto` y `emitirsaltos` (líneas 9-18) hasta la sección A.6; éstos generan el código de salto para las expresiones booleanas.

```

7)   public Expr gen() { return this; }
8)   public Expr reducir() { return this; }
9)   public void salto(int t, int f) { emitirsaltos(toString(), t, f); }
10)  public void emitirsaltos(String prueba, int t, int f) {
11)    if( t != 0 && f != 0 ) {
12)      emitir("if " + prueba + " goto L" + t);
13)      emitir("goto L" + f);
14)    }
15)    else if( t != 0 ) emitir("if " + prueba + " goto L" + t);
16)    else if( f != 0 ) emitir("iffalse " + prueba + " goto L" + f);
17)    else ; // nada, ya que tanto t como f pasan directo
18)  }
19)  public String toString() { return op.toString(); }
20) }
```

La clase `Id` hereda las implementaciones predeterminadas de `gen` y `reducir` en la clase `Expr`, ya que un identificador es una dirección.

```

1) package inter;                                // Archivo Id.java
2) import analizadorLexico.*; import simbolos.*;
3) public class Id extends Expr {
4)   public int desplazamiento;      // dirección relativa
5)   public Id(Palabra id, Tipo p, int b) { super(id, p); desplazamiento = b; }
6) }
```

El nodo para un identificador de la clase `Id` es una hoja. La llamada `super(id, p)` (línea 5, archivo `Id.java`) almacena a `id` y `p` en los campos heredados `op` y `tipo`, respectivamente. El campo `desplazamiento` (línea 4) contiene la dirección relativa de este identificador.

La clase `Op` proporciona una implementación de `reducir` (líneas 5-10, archivo `Op.java`) que heredan las subclases `Arit` para los operadores aritméticos, `Unario` para los operadores unarios, y `Acceso` para los accesos a arreglos. En cada caso, `reducir` llama a `gen` para generar un término, emite una instrucción para asignar el término a un nuevo nombre temporal y devuelve ese temporal.

```

1) package inter;                                // Archivo Op.java
2) import analizadorLexico; import simbolos.*;
3) public class Op extends Expr {
4)   public Op(Token tok, Tipo p) { super(tok, p); }
5)   public Expr reducir() {
6)     Expr x = gen();
```

```

7)      Temp t = new Temp(tipo);
8)      emitir( t.toString() + " = " + x.toString() );
9)      return t;
10) }
11) }

```

La clase `Arit` implementa los operadores binarios como `+` y `*`. El constructor `Arit` empieza por llamar a `super(tok, null)` (línea 6), en donde `tok` es un token que representa al operador y `null` es un receptáculo para el tipo. El tipo se determina en la línea 7 mediante el uso de `Tipo.max`, que comprueba si los dos operandos pueden forzarse a un tipo numérico común; el código para `Tipo.max` está en la sección A.4. Si pueden forzar, `tipo` se establece al tipo del resultado; en caso contrario se reporta un error (línea 8). Este compilador simple comprueba los tipos, pero no inserta las conversiones de tipos.

```

1) package inter;                                     // Archivo Arit.java
2) import analizadorLexico.*; import simbolos.*;
3) public class Arit extends Op {
4)     public Expr expr1, expr2;
5)     public Arit(Token tok, Expr x1, Expr x2) {
6)         super(tok, null); expr1 = x1; expr2 = x2;
7)         tipo = Tipo.max(expr1.tipo, expr2.tipo);
8)         if (tipo == null) error("error de tipo");
9)     }
10)    public Expr gen() {
11)        return new Arit(op, expr1.reducir(), expr2.reducir());
12)    }
13)    public String toString() {
14)        return expr1.toString() + " " + op.toString() + " " + expr2.toString();
15)    }
16) }

```

El método `gen` construye el lado derecho de una instrucción de tres direcciones, al reducir la subexpresión a direcciones y aplicar el operador a las direcciones (línea 11, archivo `Arit.java`). Por ejemplo, suponga que `gen` se llama en la raíz para `a+b*c`. Las llamadas a `reducir` devuelven `a` como la dirección para la subexpresión `a` y un nombre temporal `t` como la dirección para `b*c`. Mientras tanto, `reducir` emite la instrucción `t=b*c`. El método `gen` devuelve un nuevo nodo `Arit`, con el operador `*` y las direcciones `a` y `t` como operandos.¹

Vale la pena recalcar que los nombres temporales tienen tipo, junto con todas las demás expresiones. Por lo tanto, el constructor `Temp` se llama con el tipo como un parámetro (línea 6, archivo `Temp.java`).²

```

1) package inter;                                     // Archivo Temp.java
2) import analizadorLexico.*; import simbolos.*;
3) public class Temp extends Expr {

```

¹ Para reportar errores, el campo `linealex` en la clase `Nodo` registra el número de línea en el que se encontró durante el análisis léxico actual cuando se construye un nodo. Dejamos al lector el trabajo de rastrear los números de línea cuando se construyen nuevos nodos durante la generación de código intermedio.

² Un método alternativo podría ser que el constructor recibiera un nodo de expresión como un parámetro, para que pudiera copiar el tipo y la posición léxica del nodo de expresión.

```

4)     static int conteo = 0;
5)     int numero = 0;
6)     public Temp(Tipo p) { super(Palabra.temp, p); numero = ++conteo; }
7)     public String toString() { return "t" + numero; }
8) }

```

La clase **Unario** es la contraparte de la clase **Arit**, con un operando:

```

1) package inter;                                // Archivo Unario.java
2) import analizadorLexico.*; import simbolos.*;
3) public class Unario extends Op {
4)     public Expr expr;
5)     public Unario(Token tok, Expr x) {      // maneja el menos, para ! vea Not
6)         super(tok, null); expr = x;
7)         tipo = Tipo.max(Tipo.Int, expr.tipo);
8)         if (tipo == null) error("error de tipo");
9)     }
10)    public Expr gen() { return new Unario(op, expr.reducir()); }
11)    public String toString() { return op.toString() + " " + expr.toString(); }
12) }

```

A.6 Código de salto para las expresiones booleanas

El código de salto para una expresión booleana *B* se genera mediante el método **salto**, el cual recibe dos etiquetas **t** y **f** como parámetros, a los cuales se les conoce como las salidas verdadera y falsa de *B*, respectivamente. El código contiene un salto a **t** si *B* se evalúa como verdadero, y un salto a **f** si *B* se evalúa como falso. Por convención, la etiqueta especial **0** significa que el control pasa a través de *B* hacia la siguiente instrucción después del código para *B*.

Empezamos con la clase **Constante**. El constructor **Constante** en la línea 4 recibe un token **tok** y un tipo **p** como parámetros. Construye una hoja en el árbol sintáctico con la etiqueta **tok** y el tipo **p**. Por conveniencia, el constructor **Constante** está sobrecargado (línea 5) para crear un objeto constante a partir de un entero.

```

1) package inter;                                // Archivo Constante.java
2) import analizadorLexico.*; import simbolos.*;
3) public class Constante extends Expr {
4)     public Constante(Token tok, Tipo p) { super(tok, p); }
5)     public Constante(int i) { super(new Num(i), Tipo.Int); }
6)     public static final Constante
7)         True = new Constante(Palabra.True, Tipo.Bool),
8)         False = new Constante(Palabra.False, Tipo.Bool);
9)     public void salto(int t, int f) {
10)         if (this == True && t != 0) emitir("goto L" + t);
11)         else if (this == False && f != 0) emitir("goto L" + f);
12)     }
13) }

```

El método **salto** (líneas 9-12, archivo *Constante.java*) recibe dos parámetros, las etiquetas **t** y **f**. Si esta constante es el objeto estático **True** (definido en la línea 7) y **t** no es la etiqueta especial **0**, entonces se genera un salto a **t**. En caso contrario, si éste es el objeto **False** (definido en la línea 8) y **f** es distinta de cero, entonces se genera un salto a **f**.

La clase `Logica` proporciona cierta funcionalidad para las clases `Or`, `And` y `Not`. Los campos `x` y `y` (línea 4) corresponden a los operandos de un operador lógico. (Aunque la clase `Not` implementa a un operador unario, por conveniencia, es una subclase de `Logica`.) El constructor `Logica(tok, a, b)` (líneas 5-10) construye un nodo sintáctico con el operador `tok` y los operandos `a` y `b`. Al hacer esto, utiliza la función `comprobar` para asegurar que tanto `a` como `b` sean booleanas. Hablaremos sobre el método `gen` al final de esta sección.

```

1) package inter;                                // Archivo Logica.java
2) import analizadorLexico.*; import simbolos.*;
3) public class Logica extends Expr {
4)     public Expr expr1, expr2;
5)     Logica(Token tok, Expr x1, Expr x2) {
6)         super(tok, null);
7)         expr1 = x1; expr2 = x2;
8)         tipo = comprobar(expr1.tipo, expr2.tipo);
9)         if (tipo == null) error("error de tipo");
10)    }
11)    public Tipo comprobar(Tipo p1, Tipo p2) {
12)        if (p1 == Tipo.Bool && p2 == Tipo.Bool) return Tipo.Bool;
13)        else return null;
14)    }
15)    public Expr gen() {
16)        int f = nuevaEtiqueta(); int a = nuevaEtiqueta();
17)        Temp temp = new Temp(tipo);
18)        this.salto(0,f);
19)        emitir(temp.toString() + " = true");
20)        emitir("goto L" + a);
21)        emitirEtiqueta(f); emitir(temp.toString() + " = false");
22)        emitirEtiqueta(a);
23)        return temp;
24)    }
25)    public String toString() {
26)        return expr1.toString() + " " + op.toString() + " " + expr2.toString();
27)    }
28) }
```

En la clase `Or`, el método `salto` (líneas 5-10) genera el código de salto para una expresión booleana $B = B_1 \vee B_2$. Por el momento, suponga que ni la salida verdadera `t` ni la salida falsa `f` de B son la etiqueta especial 0. Como B es verdadera si B_1 es verdadera, la salida verdadera de B_1 debe ser `t` y la salida falsa corresponde a la primera instrucción de B_2 . Las salidas verdadera y falsa de B_2 son iguales que las de B .

```

1) package inter;                                // Archivo Or.java
2) import analizadorLexico.*; import simbolos.*;
3) public class Or extends Logica {
4)     public Or(Token tok, Expr x1, Expr x2) { super(tok, x1, x2); }
5)     public void salto(int t, int f) {
6)         int etiqueta = t != 0 ? t : nuevaEtiqueta();
7)         expr1.salto(etiqueta, 0);
8)         expr2.salto(t, f);
9)         if (t == 0) emitirEtiqueta(etiqueta);
10)    }
11) }
```

En el caso general, *t*, la salida verdadera de *B*, puede ser la etiqueta especial 0. La variable *etiqueta* (línea 6, archivo *Or.java*) asegura que la salida verdadera de *B*₁ se establezca de manera apropiada al final del código para *B*. Si *t* es 0, entonces *etiqueta* se establece a una nueva etiqueta que se emite después de la generación de código para *B*₁ y *B*₂.

El código para la clase *And* es similar al código para *Or*.

```

1) package inter;                                // Archivo And.java
2) import analizadorLexico.*; import simbolos.*;
3) public class And extends Logica {
4)     public And(Token tok, Expr x1, Expr x2) { super(tok, x1, x2); }
5)     public void salto(int t, int f) {
6)         int etiqueta = f != 0 ? f : nuevaEtiqueta();
7)         expr1.salto(etiqueta, 0);
8)         expr2.salto(t,f);
9)         if( f == 0 ) emitirEtiqueta(etiqueta);
10)    }
11) }
```

La clase *Not* tiene tanto en común con los demás operadores booleanos, que la haremos una subclase de *Logica*, aun cuando *Not* implementa a un operador unario. La superclase espera dos operandos, por lo que *b* aparece dos veces en la llamada a *super* en la línea 4. Sólo se utiliza *y* (declarada en la línea 4, archivo *Logica.java*) en los métodos en las líneas 5-6. En la línea 5, el método *salto* simplemente llama a *y.salto* con las salidas verdadera y falsa invertidas.

```

1) package inter;                                // Archivo Not.java
2) import analizadorLexico.*; import simbolos.*;
3) public class Not extends Logica {
4)     public Not(Token tok, Expr x2) { super(tok, x2, x2); }
5)     public void salto(int t, int f) { expr2.salto(f, t); }
6)     public String toString() { return op.toString()+" "+expr2.toString(); }
7) }
```

La clase *Rel* implementa los operadores <, <=, ==, !=, >= y >. La función *comprobar* (líneas 5-9) comprueba que los dos operandos tienen el mismo tipo y no son arreglos. Por cuestión de simplicidad, no se permiten las coerciones.

```

1) package inter;                                // Archivo Rel.java
2) import analizadorLexico.*; import simbolos.*;
3) public class Rel extends Logica {
4)     public Rel(Token tok, Expr x1, Expr x2) { super(tok, x1, x2); }
5)     public Tipo comprobar(Tipo p1, Tipo p2) {
6)         if ( p1 instanceof Arreglo || p2 instanceof Arreglo ) return null;
7)         else if( p1 == p2 ) return Tipo.Bool;
8)         else return null;
9)     }
10)    public void salto(int t, int f) {
11)        Expr a = expr1.reducir();
12)        Expr b = expr2.reducir();
13)        String prueba = a.toString() + " " + op.toString() + " " + b.toString();
14)        emitirsaltos(prueba, t, f);
15)    }
16) }
```

El método **salto** (líneas 10-15, archivo *Rel.java*) empieza generando código para las subexpresiones **x** y **y** (líneas 11-12). Después llama al método **emitirSaltos** definido en las líneas 10-18, archivo *Expr.java*, en la sección A.5. Si ni **t** ni **f** son la etiqueta especial 0, entonces **emitirSaltos** ejecuta lo siguiente:

```
12)         emitir("if " + prueba + " goto L" + t);           // Archivo Expr.java
13)         emitir("goto L" + f);
```

A lo más se genera una instrucción si **t** o **f** son la etiqueta especial 0 (de nuevo, del archivo *Expr.java*):

```
15)         else if( t != 0 ) emitir("if " + prueba + " goto L" + t);
16)         else if( f != 0 ) emitir("iffalse " + prueba + " goto L" + f);
17)         else ; // nada, ya que tanto t como f pasan directo
```

Para otro uso de **emitirSaltos**, considere el código para la clase **Acceso**. El lenguaje fuente permite asignar valores booleanos a identificadores y elementos de arreglos, por lo que una expresión booleana puede ser un acceso a un arreglo. La clase **Acceso** tiene el método **gen** para generar código “normal” y el método **salto** para el código de salto. El método **salto** (línea 11) llama a **emitirSaltos** después de reducir este acceso a arreglo a un nombre temporal. El constructor (líneas 6-9) se llama con un arreglo a aplanado, un índice **i**, y el tipo **p** de un elemento en el arreglo aplanado. La comprobación de tipos se realiza durante el cálculo de las direcciones del arreglo.

```
1) package inter;                                     // Archivo Acceso.java
2) import analizadorLexico.*; import simbolos.*;
3) public class Acceso extends Op {
4)     public Id arreglo;
5)     public Expr indice;
6)     public Acceso(Id a, Expr i, Tipo p) {           // p es el tipo de elemento despues de
7)         super(new Palabra("[]", Etiqueta.INDEX), p); // aplanar el arreglo
8)         arreglo = a; indice = i;
9)     }
10)    public Expr gen() { return new Acceso(arreglo, indice.reducir(), tipo); }
11)    public void salto(int t,int f) { emitirSaltos(reducir().toString(),t,f); }
12)    public String toString() {
13)        return arreglo.toString() + " [ " + indice.toString() + " ]";
14)    }
15) }
```

El código de salto también puede usarse para devolver un valor booleano. La clase **Logica**, que se muestra antes en esta sección, tiene un método **gen** (líneas 15-24) que devuelve un nombre temporal **temp**, cuyo valor se determina mediante el flujo de control a través del código de salto para esta expresión. En la salida verdadera de esta expresión booleana, a **temp** se le asigna **true**; en la salida falsa, a **temp** se le asigna **false**. El nombre temporal se declara en la línea 17. El código de salto para esta expresión se genera en la línea 18, en donde la salida verdadera es la siguiente instrucción y la falsa es una nueva etiqueta **f**. La siguiente instrucción asigna **true** a **temp** (línea 19), seguida de un salto a una nueva etiqueta **a** (línea 20). El código en la línea 21 emite la etiqueta **f** y una instrucción que asigna **false** a **temp**. El fragmento de código termina con la etiqueta **a**, que se genera en la línea 22. Por último, **gen** devuelve **temp** (línea 23).

A.7 Código intermedio para las instrucciones

Cada construcción de instrucción se implementa mediante una subclase de `Instr`. Los campos para los componentes de una construcción están en la subclase relevante; por ejemplo, la clase `While` tiene campos para una expresión de prueba y una subinstrucción, como veremos más adelante.

Las líneas 3-4 en el siguiente código para la clase `Instr` tratan con la construcción de árboles sintácticos. El constructor `Instr()` no hace nada, ya que el trabajo se realiza en las subclases. El objeto estático `Instr.Null` (línea 4) representa una secuencia vacía de instrucciones.

```

1) package inter;                                // Archivo Instr.java
2) public class Instr extends Nodo {
3)     public Instr() { }
4)     public static Instr Null = new Instr();
5)     public void gen(int b, int a) {}    // se llama con etiquetas inicio y despues
6)     int despues = 0;                  // almacena la etiqueta despues
7)     public static Instr Circundante = Instr.Null; // se utiliza para instrs break
8) }
```

Las líneas 5-7 tratan con la generación de código de tres direcciones. El método `gen` se llama con dos etiquetas `b` y `a`, en donde `b` marca el inicio del código para esta instrucción y `a` marca la primera instrucción después del código para esta instrucción. El método `gen` (línea 5) es un receptáculo para los métodos `gen` en las subclases. Las subclases `While` y `Do` guardan su etiqueta `a` en el campo `despues` (línea 6), por lo que cualquier instrucción `break` encerrada puede usarla para salir de su construcción circundante. El objeto `Instr.Circundante` se utiliza durante el análisis sintáctico para llevar el registro de la construcción circundante. Para un lenguaje fuente con instrucciones “`continue`”, podemos usar el mismo método para llevar el registro de la construcción circundante para una instrucción “`continue`”.

El constructor para la clase `If` construye un nodo para una instrucción `if (E) S`. Los campos `expr` e `instr` contienen los nodos para `E` y `S`, respectivamente. Observe que `expr` en letras minúsculas nombra a un campo de la clase `Expr`; de manera similar, `instr` nombra a un campo de la clase `Instr`.

```

1) package inter;                                // Archivo If.java
2) import simbolos.*;
3) public class If extends Instr {
4)     Expr expr; Instr instr;
5)     public If(Expr x, Instr s) {
6)         expr = x; instr = s;
7)         if( expr.tipo != Tipo.Bool ) expr.error("se requiere booleano en if");
8)     }
9)     public void gen(int b, int a) {
10)         int etiqueta = nuevaEtiqueta(); // etiqueta para el codigo de instr
11)         expr.salto(0, a);           // pasa por alto en true, va hacia a en false
12)         emitirEtiqueta(etiqueta); instr.gen(etiqueta, a);
13)     }
14) }
```

El código para un objeto `If` consiste en el código de salto para `expr`, seguido del código para `instr`. Como vimos en la sección A.6, la llamada `expr.salto(0,f)` en la línea 11 especi-

fica que el control debe pasar por alto el código para `expr` si ésta se evalúa como verdadera, y debe fluir hacia la etiqueta `a` en caso contrario.

La implementación de la clase `Else`, que maneja instrucciones condicionales con partes del `else`, es similar a la de la clase `If`:

```

1) package inter;                                // Archivo Else.java
2) import simbolos.*;
3) public class Else extends Instr {
4)   Expr, expr; Instr instr1, instr2;
5)   public Else(Expr x, Instr s1, Instr s2) {
6)     expr = x; instr1 = s1; instr2 = s2;
7)     if( expr.tipo != Tipo.Bool ) expr.error("se requiere booleano en if");
8)   }
9)   public void gen(int b, int a) {
10)     int etiqueta1 = nuevaEtiqueta();           // etiqueta1 para instr1
11)     int etiqueta2 = nuevaEtiqueta();           // etiqueta2 para instr2
12)     expr.salto(0,etiqueta2);                  // pasa hacia instr1 en true
13)     emitirEtiqueta(etiqueta1); instr1.gen(etiqueta1, a); emitir("goto L" + a);
14)     emitirEtiqueta(etiqueta2); instr2.gen(etiqueta2, a);
15)   }
16) }
```

La construcción de un objeto `While` se divide entre el constructor `While()`, el cual crea un nodo con hijos nulos (línea 5) y una función de inicialización `inic(x, s)`, la cual establece el hijo `expr` a `x` y el hijo `instr` a `s` (líneas 6-9). La función `gen(b, a)` para generar código de tres direcciones (líneas 10-16) está a la par con la correspondiente función `gen()` en la clase `If`. La diferencia es que la etiqueta `a` se guarda en el campo `despues` (línea 11) y que el código para `instr` va seguido de un salto a `b` (línea 15) para la siguiente iteración del ciclo `while`.

```

1) package inter;                                // Archivo While.java
2) import simbolos.*;
3) public class While extends Instr {
4)   Expr expr; Instr instr;
5)   public While() { expr = null; instr = null; }
6)   public void inic(Expr x, Instr s) {
7)     expr = x; instr = s;
8)     if( expr.tipo != Tipo.Bool ) expr.error("se requiere booleano en while");
9)   }
10)  public void gen(int b, int a) {
11)    despues = a;                           // guarda la etiqueta a
12)    expr.salto(0, a);
13)    int etiqueta = nuevaEtiqueta();        // etiqueta para instr
14)    emitirEtiqueta(etiqueta); instr.gen(etiqueta, b);
15)    emitir("goto L" + b);
16)  }
17) }
```

La clase `Do` es muy similar a la clase `While`.

```

1) package inter;                                // Archivo Do.java
2) import simbolos.*;
3) public class Do extends Instr {
4)   Expr expr; Instr instr;
```

```

5)  public Do() { expr = null; instr = null; }
6)  public void inic(Instr s, Expr x) {
7)    expr = x; instr = s;
8)    if( expr.tipo != Tipo.Bool ) expr.error("se requiere booleano en do");
9)  }
10) public void gen(int b, int a) {
11)   despues = a;
12)   int etiqueta = nuevaEtiqueta(); // etiqueta para expr
13)   instr.gen(b,etiqueta);
14)   emitirEtiqueta(etiqueta);
15)   expr.salto(b,0);
16) }
17) }

```

La clase `Est` implementa asignaciones con un identificador del lado izquierdo y una expresión a la derecha. La mayor parte del código en la clase `Est` es para construir un nodo y comprobar tipos (líneas 5-13). La función `gen` emite una instrucción de tres direcciones (líneas 14-16).

```

1) package inter; // Archivo Est.java
2) import analizadorLexico.*; import simbolos.*;
3) public class Est extends Instr {
4)   public Id id; public Expr expr;
5)   public Est(Id i, Expr x) {
6)     id = i; expr = x;
7)     if ( comprobar(id.tipo, expr.tipo) == null ) error("error de tipo");
8)   }
9)   public Tipo comprobar(Tipo p1, Tipo p2) {
10)     if ( Tipo.numerico(p1) && Tipo.numerico(p2) ) return p2;
11)     else if ( p1 == Tipo.Bool && p2 == Tipo.Bool ) return p2;
12)     else return null;
13)   }
14)   public void gen(int b, int a) {
15)     emitir( id.toString() + " = " + expr.gen().toString());
16)   }
17) }

```

La clase `EstElem` implementa las asignaciones para un elemento de un arreglo:

```

1) package inter; // Archivo EstElem.java
2) import analizadorLexico.*; import simbolos.*;
3) public class EstElem extends Instr {
4)   public Id arreglo; public Expr indice; public Expr expr;
5)   public EstElem(Acceso x, Expr y) {
6)     arreglo = x.arreglo; indice = x.indice; expr = y;
7)     if ( comprobar(x.tipo, expr.tipo) == null ) error("error de tipo");
8)   }
9)   public Tipo comprobar(Tipo p1, Tipo p2) {
10)     if ( p1 instanceof Arreglo || p2 instanceof Arreglo ) return null;
11)     else if ( p1 == p2 ) return p2;
12)     else if ( Tipo.numerico(p1) && Tipo.numerico(p2) ) return p2;
13)     else return null;
14)   }
15)   public void gen(int b, int a) {
16)     String s1 = indice.reducir().toString();
17)     String s2 = expr.reducir().toString();

```

```

18)         emitir(arreglo.toString() + " [ " + s1 + " ] = " + s2);
19)     }
20) }

```

La clase `Sec` implementa una secuencia de instrucciones. Las pruebas para las instrucciones nulas en las líneas 6-7 son para evitar las etiquetas. Observe que no se genera código para la instrucción nula, `Instr.Null`, ya que el método `gen` en la clase `Instr` no hace nada.

```

1) package inter;                                // Archivo Sec.java
2) public class Sec extends Instr {
3)     Instr instr1; Instr instr2;
4)     public Sec(Instr s1, Instr s2) { instr1 = s1; instr2 = s2; }
5)     public void gen(int b, int a) {
6)         if ( instr1 == Instr.Null ) instr2.gen(b, a);
7)         else if ( instr2 == Instr.Null ) instr1.gen(b, a);
8)         else {
9)             int etiqueta = nuevaEtiqueta();
10)            instr1.gen(b,etiqueta);
11)            emitirEtiqueta(etiqueta);
12)            instr2.gen(etiqueta, a);
13)        }
14)    }
15) }

```

Una instrucción `break` envía el control hacia fuera de un ciclo o instrucción `switch` circundante. La clase `Break` utiliza el campo `instr` para guardar la construcción de la instrucción circundante (el analizador asegura que `Instr.Circundante` denote el nodo del árbol sintáctico para la construcción circundante). El código para un objeto `Break` es un salto a la etiqueta `instr.despues`, que marca la instrucción justo después del código para `instr`.

```

1) package inter;                                // Archivo Break.java
2) public class Break extends Instr {
3)     Instr instr;
4)     public Break() {
5)         if( Instr.Circundante == null ) error("break no encerrada");
6)         instr = Instr.Circundante;
7)     }
8)     public void gen(int b, int a) {
9)         emitir( "goto L" + instr.despues);
10)    }
11) }

```

A.8 Analizador sintáctico

El analizador sintáctico lee un flujo de tokens y construye un árbol sintáctico llamando a las funciones constructor apropiadas de las secciones A.5-A.7. La tabla de símbolos actual se mantiene como en el esquema de traducción en la figura 2.38 de la sección 2.7.

El paquete `analizador` contiene una clase, `Analizador`:

```

1) package analizador;                         // Archivo Analizador.java
2) import java.io.*; import analizadorLexico.*; import simbolos.*; import inter.*;

```

```

3) public class Analizador {
4)     private AnalizadorLexico lex; //analizador léxico para este analizador sintáctico
5)     private Token busca; // marca de búsqueda por adelantado
6)     Ent sup = null; // tabla de símbolos actual o superior
7)     int usado = 0; // almacenamiento usado para las declaraciones
8)     public Analizador(AnalizadorLexico lex) throws IOException { lex = 1; mover(); }
9)     void mover() throws IOException { busca = lex.escanear(); }
10)    void error(String s) { throw new Error("cerca de linea "+lex.linea+": "+s); }
11)    void coincidir(int t) throws IOException {
12)        if( busca.etiqueta == t ) mover();
13)        else error("error de sintaxis");
14)    }

```

Al igual que el traductor de expresiones simple de la sección 2.5, la clase `Analizador` tiene un procedimiento para cada no terminal. Los procedimientos se basan en una gramática que se forma al eliminar la recursividad izquierda de la gramática del lenguaje fuente en la sección A.1.

El análisis sintáctico empieza con una llamada al procedimiento `programa`, el cual llama a `bloque()` (línea 16) para analizar el flujo de entrada y construir el árbol sintáctico. Las líneas 17-18 generan el código intermedio.

```

15)    public void programa() throws IOException { // programa -> bloque
16)        Instr s = bloque();
17)        int inicio = s.nuevaEtiqueta(); int despues = s.nuevaEtiqueta();
18)        s.emitirEtiqueta(inicio); s.gen(inicio, despues); s.emitirEtiqueta(despues);
19)    }

```

El manejo de la tabla de símbolos se muestra de manera explícita en el procedimiento `bloque`.³ La variable `sup` (declarada en la línea 5) contiene la tabla de símbolos superior; la variable `entGuardado` (línea 21) es un enlace a la tabla de símbolos anterior.

```

20)    Instr bloque() throws IOException { // bloque -> { decls instrs }
21)        coincidir('{'); Ent entGuardado = sup; sup = new Ent(sup);
22)        decls(); Instr s = instrs();
23)        coincidir('}'); sup = entGuardado;
24)        return s;
25)    }

```

Las declaraciones producen entradas en la tabla de símbolos para los identificadores (vea la línea 36). Aunque no se muestra aquí, las declaraciones también pueden producir instrucciones para reservar almacenamiento para los identificadores en tiempo de ejecución.

```

26)    void decls() throws IOException {
27)        while( busca.etiqueta == Etiqueta.BASIC ) { // D -> tipo ID;
28)            Tipo p = tipo(); Token tok = busca; coincidir(Etiqueta.ID); coincidir('\'');
29)            Id id = new Id((Palabra)tok, p, usado);
30)            sup.put( tok, id );
31)            usado = usado + p.anchura;
32)        }
33)    }
34)    Tipo tipo() throws IOException {
35)        Tipo p = (Tipo)busca; // espera busca.etiqueta == Etiqueta.BASIC

```

³ Una alternativa atractiva es agregar los métodos `push` y `pop` a la clase `Ent`, para que se pueda acceder a la tabla actual a través de una variable estática `Ent.sup`.

```

36)      coincidir(Etiqueta.BASIC);
37)      if( busca.etiqueta != '[' ) devuelve p; // T -> basico
38)      else return dims(p); // devuelve el tipo del arreglo
39)  }
40)  Tipo dims(Tipo p) throws IOException {
41)      coincidir('['); Token tok = busca; coincidir(Etiqueta.NUM); coincidir(']');
42)      if( busca.etiqueta == '[' )
43)          p = dims(p);
44)      return new Arreglo(((Num)tok).valor, p);
45)  }

```

El procedimiento `instr` tiene una instrucción `switch` con instrucciones `case` que corresponden a las producciones para la no terminal `Instr`. Cada `case` construye un nodo para una construcción, usando las funciones constructor que vimos en la sección A.7. Los nodos para las instrucciones `while` y `do` se construyen cuando el analizador ve la palabra clave de apertura. Los nodos se construyen antes de que se analice la instrucción para permitir que cualquier instrucción `break` encerrada apunte de vuelta a su ciclo circundante. Los ciclos anidados se manejan mediante el uso de la variable `Instr.Circundante` en la clase `Instr` y `instrGuardada` (declarada en la línea 52) para mantener el ciclo circundante actual.

```

46)  Instr instrs() throws IOException {
47)      if ( busca.etiqueta == '}' ) return Instr.Null;
48)      else return new Sec(instr(), instrs());
49)  }
50)  Instr instr() throws IOException {
51)      Expr x; Instr s, s1, s2;
52)      Instr instrGuardada; // guarda ciclo circundante p/instrucciones break
53)      switch( busca.etiqueta ) {
54)          case ';':
55)              mover();
56)              return Instr.Null;
57)          case Etiqueta.IF:
58)              coincidir(Etiqueta.IF); coincidir('('); x = bool(); coincidir(')');
59)              s1 = instr();
60)              if( busca.etiqueta != Etiqueta.ELSE ) return new If(x, s1);
61)              coincidir(Etiqueta.ELSE);
62)              s2 = instr();
63)              return new Else(x, s1, s2);
64)          case Etiqueta.WHILE:
65)              While nodowhile = new While();
66)              instrGuardada = Instr.Circundante; Instr.Circundante = nodowhile;
67)              coincidir(Etiqueta.WHILE); coincidir('('); x = bool(); coincidir(')');
68)              s1 = instr();
69)              nodowhile.inic(x, s1);
70)              Instr.Circundante = instrGuardada; // restablece Instr.Circundante
71)              return nodowhile;
72)          case Etiqueta.DO:
73)              Do nododo = new Do();
74)              instrGuardada = Instr.Circundante; Instr.Circundante = nododo;
75)              coincidir(Etiqueta.DO);
76)              s1 = instr();
77)              coincidir(Etiqueta.WHILE); coincidir('('); x = bool(); coincidir(')');
78)              coincidir(')');
79)              nododo.inic(s1, x);
80)              Instr.Circundante = instrGuardada; // restablece Instr.Circundante
81)              return nododo;
81)          case Etiqueta.BREAK;

```

```

82)         coincidir(Etiqueta.BREAK); coincidir(';');
83)         return new Break();
84)     case '{':
85)         return bloque();
86)     default:
87)         return asignar();
88)     }
89) }

```

Por conveniencia, el código para las asignaciones aparece en un procedimiento auxiliar, `asignar`.

```

90)     Instr asignar() throws IOException {
91)         Instr instr; Token t = busca;
92)         coincidir(Etiqueta.ID);
93)         Id id = sup.get(t);
94)         if( id == null ) error(t.toString() + " no declarado");
95)         if( busca.etiqueta == '=' ) {           // S -> id = E ;
96)             mover(); instr = new Est(id, bool());
97)         }
98)         else {                                // S -> L = E ;
99)             Acceso x = desplazamiento(id);
100)            coincidir('>'); instr = new EstElem(x, bool());
101)        }
102)        coincidir(';');
103)        return instr;
104)    }

```

El análisis sintáctico de las expresiones aritméticas y booleanas es similar. En cada caso, se crea un nodo de árbol sintáctico apropiado. La generación de código para cualquiera de las dos expresiones es distinta, como vimos en las secciones A.5 y A.6.

```

105)     Expr bool() throws IOException {
106)         Expr x = unir();
107)         while( busca.etiqueta == Etiqueta.OR ) {
108)             Token tok = busca; mover(); x = new Or(tok, x, unir());
109)         }
110)         return x;
111)     }
112)     Expr unir() throws IOException {
113)         Expr x = igualdad();
114)         while( busca.etiqueta == Etiqueta.AND ) {
115)             Token tok = busca; mover(); x = new And(tok, x, igualdad());
116)         }
117)         return x;
118)     }
119)     Expr igualdad() throws IOException {
120)         Expr x = rel();
121)         while( busca.etiqueta == Etiqueta.EQ || busca.etiqueta == Etiqueta.NE ) {
122)             Token tok = busca; mover(); x = new Rel(tok, x, rel());
123)         }
124)         return x;
125)     }
126)     Expr rel() throws IOException {
127)         Expr x = expr();
128)         switch( busca.etiqueta ) {

```

```

129)     case '<': case Etiqueta.LE: case Etiqueta.GE: case '>':
130)         Token tok = busca; mover(); return new Rel(tok, x, expr());
131)     default:
132)         return x;
133)     }
134) }
135) Expr expr() throws IOException {
136)     Expr x = term();
137)     while( busca.etiqueta == '+' || busca.etiqueta == '-' ) {
138)         Token tok = busca; mover(); x = new Arit(tok, x, term());
139)     }
140)     return x;
141) }
142) Expr term() throws IOException {
143)     Expr x = unario();
144)     while(busca.etiqueta == '*' || busca.etiqueta == '/') {
145)         Token tok = busca; mover(); x = new Arit(tok, x, unario());
146)     }
147)     return x;
148) }
149) Expr unario() throws IOException {
150)     if( busca.etiqueta == '-' ) {
151)         mover(); return new Unario(Palabra.minus, unario());
152)     }
153)     else if( busca.etiqueta == '!' ) {
154)         Token tok = busca; mover(); return new Not(tok, unario());
155)     }
156)     else return factor();
157) }

```

El resto del código en el analizador sintáctico trata con los “factores” en las expresiones. El procedimiento auxiliar **desplazamiento** genera código para los cálculos de las direcciones de los arreglos, como vimos en la sección 6.4.3.

```

158) Expr factor() throws IOException {
159)     Expr x = null;
160)     switch( busca.etiqueta ) {
161)         case '(':
162)             mover(); x = bool(); coincidir(')');
163)             return x;
164)         case Etiqueta.NUM:
165)             x = new Constante(busca, Tipo.Int);     mover(); return x;
166)         case Etiqueta.REAL:
167)             x = new Constante(busca, Tipo.Float);  mover(); return x;
168)         case Etiqueta.TRUE:
169)             x = Constante.True;                   mover(); return x;
170)         case Etiqueta.FALSE:
171)             x = Constante.False;                 mover(); return x;
172)         default:
173)             error("error de sintaxis");
174)             return x;
175)         case Etiqueta.ID:
176)             String s = busca.toString();
177)             Id id = sup.get(busca);
178)             if( id == null ) error(busca.toString() + " no declarado");
179)             mover();

```

```

180)         if (busca.etiqueta != '[' ) return id;
181)         else return desplazamiento(id);
182)     }
183) }
184) Acceso desplazamiento(Id a) throws IOException { // I -> [E] | [E] I
185)     Expr i; Expr w Expr t1, t2; Expr ubic; // hereda id
186)     Tipo tipo = a.tipo;
187)     coincidir('['); i = bool(); coincidir(']'); // primer indice, I -> [ E ]
188)     tipo = ((Arreglo)tipo).de;
189)     w = new Constante(tipo.anchura);
190)     t1 = new Arit(new Token('*'), i, w);
191)     ubic = t1;
192)     while( busca.etiqueta == '[' ) { // multi-dimensional I -> [ E ] I
193)         coincidir('['); i = bool(); coincidir(']');
194)         tipo = ((Arreglo)tipo).de;
195)         w = new Constante(tipo.anchura);
196)         t1 = new Arit(new Token('*'), i, w);
197)         t2 = new Arit(new Token('+'), ubic, t1);
198)         ubic = t2;
199)     }
200)     return new Acceso(a, ubic, tipo);
201) }
202) }

```

A.9 Creación del front-end

El código para los paquetes aparece en cinco directorios: `main`, `analizadorLexico`, `simbolos`, `analizador` e `inter`. Los comandos para crear el compilador varían de sistema en sistema. Los siguientes comandos son para una implementación en UNIX:

```

javac analizadorLexico/*.java
javac simbolos/*.java
javac inter/*.java
javac analizador/*.java
javac main/*.java

```

El comando `javac` crea archivos `.class` para cada clase. Después podemos probar el traductor escribiendo `java main.Main`, seguido del programa fuente que se desea traducir; por ejemplo, el contenido del archivo `prueba`:

```

1) { // Archivo prueba
2)     int i; int j; float v; float x; float[100] a;
3)     while( true ) {
4)         do i = i+1; while( a[i] < v );
5)         do j = j-1; while( a[j] > v );
6)         if( i >= j ) break;
7)         x = a[i]; a[i] = a[j]; a[j] = x;
8)     }
9) }

```

En esta entrada, el front-end produce lo siguiente:

```
1) L1:L3: i = i + 1
2) L5:      t1 = i * 8
3)      t2 = a [ t1 ]
4)      if t2 < v goto L3
5) L4:      j = j - 1
6) L7:      t3 = j * 8
7)      t4 = a [ t3 ]
8)      if t4 > v goto L4
9) L6:      iffalse i >= j goto L8
10) L9:     goto L2
11) L8:     t5 = i * 8
12)     x = a [ t5 ]
13) L10:    t6 = i * 8
14)     t7 = j * 8
15)     t8 = a [ t7 ]
16)     a [ t6 ] = t8
17) L11:    t9 = j * 8
18)     a [ t9 ] = x
19)
20) L2:
```

Pruébelo.

Apéndice B

Búsqueda de soluciones linealmente independientes

Algoritmo B.1: Busca un conjunto máximo de soluciones linealmente independientes para $A\vec{x} \geq \vec{0}$, y las expresa como filas de la matriz B .

ENTRADA: Una matriz A de $m \times n$.

SALIDA: Una matriz B de soluciones linealmente independientes para $A\vec{x} \geq \vec{0}$.

MÉTODO: El algoritmo se muestra en seudocódigo a continuación. Observe que $X[y]$ denota la y -ésima fila de la matriz X , $X[y : z]$ denota las filas y a z de la matriz X , y $X[y : z][u : v]$ denota el rectángulo de la matriz X en las filas de la y a la z y las columnas de la u a la v . \square

```

 $M = A^T;$ 
 $r_0 = 1;$ 
 $c_0 = 1;$ 
 $B = I_{n \times n}; /*$  una matriz identidad de  $n$  por  $n$  */

while ( true ) {

    /* 1. Convertir a  $M[r_0 : r' - 1][c_0 : c' - 1]$  en una matriz diagonal con
       entradas diagonales positivas y  $M[r' : n][c_0 : m] = 0$ .
        $M[r' : n]$  son soluciones. */

     $r' = r_0;$ 
     $c' = c'_0;$ 
    while ( existe  $M[r][c] \neq 0$  de tal forma que
             $r - r'$  y  $c - c'$  sean ambas  $\geq 0$  ) {

        Mover pivote  $M[r][c]$  a  $M[r'][c']$  mediante intercambio
            de fila y columna;
        Intercambiar fila  $r$  con fila  $r'$  en  $B$ ;
        if ( $M[r'][c'] < 0$  ) {
             $M[r'] = -1 * M[r'];$ 
             $B[r'] = -1 * B[r'];$ 
        }
        for (  $fila = r_0$  a  $n$  ) {
            if (  $fila \neq r'$  y  $M[fila][c'] \neq 0$  {
                 $u = -(M[fila][c'] / M[r'][c']);$ 
                 $M[fila] = M[fila] + u * M[r'];$ 
                 $B[fila] = B[fila] + u * B[r'];$ 
            }
        }
         $r' = r' + 1;$ 
         $c' = c' + 1;$ 
    }
}

```

```

/* 2. Buscar una solución aparte de  $M[r' : n]$ . Debe ser una
   combinación no negativa de  $M[r_0 : r' - 1][c_0 : m]$  */
Buscar  $k_{r_0}, \dots, k_{r'-1} \geq 0$  de tal forma que
 $k_{r_0}M[r_0][c' : m] + \dots + k_{r'-1}M[r' - 1][c' : m] \geq 0$ ;
if ( existe una solución no trivial, por decir  $k_r > 0$  ) {
   $M[r] = K_{r_0}M[r_0] + \dots + k_{r'-1}M[r' - 1];$ 
  NoMásSols = false;
} else /*  $M[r' : n]$  son las únicas soluciones */
  NoMasSols = true;

/* 3. Hacer que  $M[r_0 : r_n - 1][c_0 : m] \geq 0$  */
if ( NoMásSols ) { /* Mover soluciones  $M[r' : n]$  a  $M[r_0 : r_n - 1]$  */
  for (  $r = r'$  hasta  $n$  )
    Intercambiar filas  $r$  y  $r_0 + r - r'$  en  $M$  y  $B$ ;
     $r_n = r_0 + n - r' + 1$ ;
  else { /* Usar suma de filas para encontrar más soluciones */
     $r_n = n + 1$ ;
    for (  $col = c'$  hasta  $m$  )
      if ( existe  $M[fila][col] < 0$  tal que  $fila \geq r_0$  )
        if ( existe  $M[r][col] > 0$  tal que  $r \geq r_0$  )
          for (  $fila = r_0$  hasta  $r_n - 1$  )
            if (  $M[fila][col] < 0$  )
               $u = \lceil (-M[fila][col]/M[r][col]) \rceil$ ;
               $M[fila] = M[fila] + u * M[r];$ 
               $B[fila] = B[fila] + u * B[r];$ 
            }
        else
          for (  $fila = r_n - 1$  hasta  $r_0$  en pasos de  $-1$  )
            if (  $M[fila][col] < 0$  )
               $r_n = r_n - 1$ ;
              Intercambiar  $M[fila]$  con  $M[r_n]$ ;
              Intercambiar  $B[fila]$  con  $B[r_n]$ ;
        }
  }
}

/* 4. Hacer que  $M[r_0 : r_n - 1][1 : c_0 - 1] \geq 0$  */
for (  $fila = r_0$  hasta  $r_n - 1$  )
  for (  $col = 1$  hasta  $c_0 - 1$  )
    if (  $M[fila][col] < 0$  )
      Elegir una  $r$  tal que  $M[r][col] > 0$  y  $r < r_0$ ;
       $u = \lceil (-M[fila][col]/M[r][col]) \rceil$ ;
       $M[fila] = M[fila] + u * M[r];$ 
       $B[fila] = B[fila] + u * B[r];$ 
    }
}

```

```
/* 5. Si es necesario, repetir con las filas  $M[r_n : n]$  */  
if ( (NoMásSols o  $r_n > n$  o  $r_n == r_0$ ) {  
    Eliminar las filas desde  $r_n$  hasta  $n$  de  $B$ ;  
    return  $B$ ;  
}  
else {  
     $c_n = m + 1$ ;  
    for (  $col = m$  hasta  $1$  en pasos de  $-1$  )  
        if ( no hay  $M[r][col] > 0$  tal que  $r < r_n$  ) {  
             $c_n = c_n - 1$ ;  
            Intercambiar columna  $col$  con  $c_n$  en  $M$ ;  
        }  
     $r_0 = r_n$ ;  
     $c_0 = c_n$ ;  
}  
}
```

Índice

A

Abstracto, árbol sintáctico
 Vea Sintáctico, árbol
Abu-Sufah, W., 900
Acción, 58-59, 249, 327
Aceptación, 149
Acíclica
 cadena de llamadas, 910
 prueba, 821-822
Acíclico, camino, 667
Ada, 391
AFD
 Vea Determinista, autómata finito
Afín
 acceso a arreglo, 781, 801-804, 815-826
 expresión, 687-770
 partición de espacio, 830-838
 particionamiento, 781
 transformación, 778-782, 846-851
AFN
 Vea No determinista, autómata finito
Aho, A. V., 189-190, 301, 579-580
Aho-Corasick, algoritmo de, 138-140
Al instante, generación, 340-343, 380-381
Alcance, 86
 definiciones de, 601-608, 615
Alfabeto, 117
Algebraicas, identidades, 536-552
Alias, 35, 713, 903, 917, 933
Alineación, 374, 428
Allen, F. E., 704, 899-900, 962
Allen, R., 900

Almacenamiento
 distribución del, 373
 instrucción de, 512
 Vea también Dinámico,
 almacenamiento; Estático, almacenamiento; Antidependencia;
 Salida, dependencia de
Alpha, 703
Altura de un semienrejado, 623, 626, 628
Ambigua, gramática, 47, 203-204, 210-212,
 255, 278-283, 291-294
Amdahl, ley de, 774
Ampliación, 388-389
Análisis, 4
 del flujo de datos, framework, 618
 sintáctico, árbol de, 45-48, 201-204
 Vea también Anotado, árbol de análisis sintáctico
Analizador sintáctico, 8, 41, 45, 60-61,
 110-111, 191-302, 981-986
 sintáctico, estado del, 241-242
 Vea también Conjunto de elementos;
 Ascendente, analizador sintáctico;
 Descendente, analizador sintáctico
 de
Analizadores sintácticos, generador de
 Vea Antlr, CUP, Yacc
Ancestro, 46
Anchura de un tipo, 374
Andersen, L., 962
Anidadas, declaraciones de procedimientos,
 442-445
Anotado, árbol de análisis sintáctico, 54

Anticipada, expresión, 645-648, 653
 Antidependencia, 711, 816
 Antisimetría, 619
 Antlr, 300, 302
 Apuntador, 365, 373, 514, 539, 935
 Vea también Colgante, apuntador;
 Pila, apuntador de
 Apuntadores, análisis de, 713, 903, 917, 933-951
 Árbol, 46, 56
 de activación, 430-433
 Árboles, reescritura de, 558-567
 Arista
 Vea Avance, arista de; Posterior, arista, Crítica, arista; Cruzada, arista, Retirada, arista de
 Aritmética, expresión, 49-50, 68-69, 378-381, 971-974
 Arquitectura, 19-22
 Arreglo, 373-375, 381-384, 537-539, 541, 584, 712-713, 770, 920
 Vea también Afín, acceso a arreglo
 Arreglos, contracción de, 884-887
 Ascendente, analizador sintáctico, 233-240
 Vea también LR, analizador sintáctico;
 Desplazamiento-reducción, analizador sintáctico
 ASCII, 117
 Asignación de memoria, 453
 Asociatividad, 48, 122, 279-281, 293, 619
 Átomo, 921
 Atribuida, gramática, 306
 Atributo, 54, 112
 L, definición de, 313-314, 331-352
 S, definición de, 306, 312-313, 324
 Vea también Heredado, atributo;
 Principal, atributo; Sintetizado, atributo
 Aumentada, gramática, 243
 Auslander, M. A., 580
 Autoincremento, 739
 Autómata, 147
 Vea también Determinista, autómata finito; LR(0), autómata;
 No determinista, autómata finito

Avance, arista de, 661
 Avots, D., 962-963

B

Backpatching, 410-417
 Backus, J. W., 300-301
 Backus-Naur, forma
 Vea BNF
 Baker, algoritmo de, 475-476, 482
 Baker, H. G. Jr., 502
 Ball, T., 962
 Banerjee, U., 900
 Banning, J. P., 962
 Barth, J. M., 962
 Base
 átomo, 921
 dirección, 381
 registro, 762
 Básico
 bloque, 525-541, 597, 600-601, 721-726
 tipo, 371
 Bauer, F. L., 354-355
 BDD, 951-958
 Bddbddb, 961
 Bergin, T. J., 38
 Berndl, M., 961-962
 Bernstein, D., 766-767
 Bifurcación
 Vea Salto
 Bifurcar y delimitar, 824-825
 Binaria, traducción, 22
 Binario, alfabeto, 117
 Birman, A., 301
 Bison, 300
 Bloque, 29, 86-87, 95
 Vea también Básico, bloque
 Bloques
 estructura
 Vea Estático, alcance
 uso de, 770-771, 785-787, 877-880, 888
 BNF
 Vea Libre de contexto, gramática
 Booleana, expresión, 399-400, 403-409, 411-413, 974-977
 Bounimova, E., 962

Break, instrucción, 416-417
 Brooker, R. A., 354
 Bryant, R. E., 961-962
 Búfer, 115-117
 desbordamiento de, 918, 920-921
 Burke, M., 900
 Bus, 772-773
 Bush, W. R., 962-963
 Búsqueda en profundidad, árbol de expansión con, 660
 Bytecode, 2

C

C, 13, 18, 25, 28-29, 381, 498, 903, 934
 C++, 13, 18, 34, 498
 Caché, 20, 454-455, 457, 772, 783-785
 interferencia, 788
 Cedar, C., 963
 Cadena, 118-119, 373
 Callahan, D., 961, 963
 Camino
 Vea Acíclico, camino; Crítico, camino;
 Ejecución, camino de; Reunión
 sobre los caminos, solución; Peso
 de un camino
 Campo, 377, 584, 935
 almacenamiento de, 937
 carga de, 937
 Canalización
 Vea Instrucciones, canalización de;
 Canalizaciones; Software, canalización por
 Canalizaciones, 861-864
 Canónica, derivación
 Vea Por la derecha, derivación de
 Canónico
 analizador sintáctico LR, 259, 265-266,
 283
 LR(0), conjunto de elementos, 243, 247
 LR(1), conjunto de elementos, 260-264
 Cantor, D. C., 300-301
 Carácter, clase, 123, 126
 Carbin, M., 963
 Carga, instrucción de, 512

Cargador, 3
 Centinela, 116
 derecha, forma, 201, 256
 forma, 200
 izquierda, forma, 201
 CFG
 Vea Gramática
 Chaitin, G. J., 580
 Chandra, A. K., 580
 Charles, P., 900
 Chelf, B., 963
 Chen, S., 766-767, 899, 901
 Cheney, algoritmo de, 479-482
 Cheney, C. J., 502-503
 Cheong, G. I., 901
 Chomsky, Forma Normal de, 232, 300
 Chomsky, N., 300-301
 Chou, A., 963
 Chow, F., 579-580
 Church, A., 502-503
 Cíclica, basura, 488
 Ciclo, 531, 554, 556, 655-656, 775
 desenrollamiento de, 735, 740-741, 743
 expresión invariante de, 641-642
 prueba de residuo de, 822-823
 Vea también Ejecución total, ciclo
 de; Completamente permutables,
 ciclos; Natural, ciclo
 Ciclos
 anidamiento de, 780, 791, 792, 862
 fisión de; *Vea* Fisión
 fusión de; *Vea* Fusión
 región de, 674
 Cierre, 119, 121-122, 243-245, 261-262
 de funciones de transferencia, 679
 Vea también Positivo, cierre
 Circular, dependencia, 307
 CISC, 21, 507-508
 Clase, 33, 376
 variable de, 25-26
 Clonación, 910-911
 Coalescencia de trozos, 459-460
 Cocke, J., 301, 579-580, 704, 900
 Cocke-Younger-Kasami, algoritmo de, 232,
 301

- Código
 - generación de, 10-11, 505-581, 707-767
 - Vea también* Programación
 - movimiento de, 592
 - Vea también* Hacia abajo, movimiento de código; optimización de, 5, 10, 15-19, 368, 583-705, 769-963
 - P, 386
 - programación de
 - Vea* Programación
 - Coerción, 99, 388-389
 - Coffman, E. G., 900-901
 - Coherente, protocolo de caché, 772-773
 - Cola, recursividad, 73
 - Colgante
 - apuntador, 461
 - else, 210-212, 281-283
 - Collins, G. E., 503
 - Coloración, 556-557
 - Columnas, orden por, 382, 785
 - Combinación/unión, 621, 955
 - Comentario, 77
 - Compilación, tiempo de, 25
 - Completamente permutables, ciclos, 861, 864-867, 875-876
 - Composición, 624, 678, 693
 - Computadora
 - arquitectura de *Vea* Arquitectura
 - con conjunto complejo de instrucciones
 - Vea* CISC
 - Común, subexpresión, 533-535, 588-590, 611, 639-641
 - Comunicación, 828, 881-882, 894
 - Concatenación, 119, 121-122
 - Concurrente, recolección de basura, 495-497
 - Condición
 - comutativa, 122, 619
 - distributiva, 122
 - Condicional, salto, 513, 527
 - Configuración, 249
 - Conflicto, 144, 565
 - Vea también* Desplazamiento-reducción, conflicto;
 - Conjunto
 - de corte, 645
 - de elementos, 243-246, 257
 - Vea también* Canónico, LR(0), conjunto de elementos; Canónico, LR(1), conjunto de elementos de las partes de un conjunto (Power Set), 620
 - Conjuntos, asociatividad de, 457
 - Conservador, análisis de flujo de datos, 603
 - Constante, 78-79
 - Constantes
 - plegado de, 536, 632-637
 - propagación de
 - Vea* Constantes, plegado de
 - Contenedores, uso de, trozos, 458
 - Contexto, sensibilidad al, 906-908, 945, 951
 - Contigua, evaluación, 574
 - Continue, instrucción, 416-417
 - Control
 - enlace de, 434
 - equivalencia de, 728
 - Convexo, poliedro, 789-790, 795-796
 - Cook, B., 962
 - Cooper, K. D., 580, 963
 - Copia
 - instrucción de, 544, 937
 - propagación de, 590-591
 - Copiado, recolector de basura de, 478-482, 488, 497-498
 - Corasick, M. J., 189-190
 - Corto circuito, 953
 - Cosecha, 46-47, 201
 - Cousot, P., 704
 - Cousot, R., 704
 - Crítica, arista, 643
 - Crítico
 - camino, 725
 - ciclo, 758
 - Cruzada, arista, 662
 - Cuádruple, 366-368
 - Cuarta generación, lenguaje de, 13
 - Cuerpo, 43, 197, 923
 - Cuerpo, región del, 673
 - CUP, 300, 302

- CYK, algoritmo
 Vea Cocke-Younger-Kasami, algoritmo de
Cytron, R., 704, 900
- D**
- Dain, J., 300-301
Dalton, M., 962
Dantzig, G., 900
Das, M., 961, 963
Datalog, 921-933
Datalog, programa en, 923
Datos
 abstracción de, 18
 dependencia de, 711-715, 732, 747-749, 771, 781-782, 804-805, 815-826
 Vea también Antidependencia; Salida, dependencia de; Verdadera, dependencia
 espacio de, 779-780
 grafo de dependencia de, 722-723
 localidad de, 891-892
 Vea también Localidad
 reutilización de, 804-815, 887-888
Davidson, E. S., 767
Declaración, 32, 373, 376
Declarativo, lenguaje, 13
Decodificar, 708
Def, 609
Definición, 32
 de una variable, 601
Delimitadora
 condición, 615
 etiqueta, 459
Dependencia
 de control, restricción de, 710, 716-717
 grafo de, 310-312
Derecha, asociatividad por la, 48
Derecho, lado
 Vea Cuerpo
DeRemer, F., 300-301
Derivación, 44-46, 199-202
 Vea también Por la izquierda, derivación de; Por la derecha, derivación de
Derramamiento de registros, 716
Desasignación de memoria, 453, 460-463
Descendente, analizador sintáctico de, 61-68, 217-233
 Vea también Predictivo, analizador sintáctico; Recursivo, analizador sintáctico descendente
Descendiente, 46
Desplazamiento (skewing), 377-378, 849-850
 -reducción, analizador sintáctico, 236-240
 -reducción, conflicto, 238-240, 293
Desreferencia, 461
Destino
 código
 Vea Objeto, código
 conjunto, 488
 lenguaje, 1
Determinista, autómata finito, 149-156, 164-166, 170-186, 206
Diagrama de decisiones binario
 Vea BDD
Diferido, movimiento de código
 Vea Parcial, eliminación de redundancia
Dijkstra, E. W., 502-503
Dinámica
 carga, 944
 política, 25
 programación, 573-577
 Vea también Cocke-Younger-Kasami, algoritmo de
 RAM, 456
Dinámico
 acceso, 816
 alcance, 31-33
 almacenamiento, 429
 Vea también Montículo; Tiempo de ejecución, pila en programador, 719, 737
Diófantina, ecuación, 818-820
Dirección, 364, 374
Direcciones
 descriptor de, 543, 545-547
 espacio de, 427

Dirigido, grafo acíclico
Vea GDA
 Distributivo, framework, 625, 635-636
 Dominador, 656-659, 672, 728
 Dominadores, árbol de, 657
 Dominio
 de un análisis de flujo de datos, 599, 615
 de una relación, 954
 Donnelly, C., 301
 Dumitran, D., 963

E

ϵ
 producción, 63, 65-66
 Vea Vacía, cadena
 Earley, J., 301
 Eaves, B. C., 900
 EDB
 Vea también Extensional, predicado de
 base de datos
 Ejecución
 camino de, 597, 628
 cruzada, ciclos de, 743-745
 total, ciclo de, 738
 Elemento, 242-243
 Vea también Núcleo, elemento;
 Conjunto de elementos; Válido,
 elemento
 Eliminar, 601, 603, 611
 Emami, M., 961, 963
 Encabezado, 42, 197, 665, 672, 923
 Engler, D., 963
 Enlace de acceso, 434, 445-449
 Enrejados, diagrama de, 621-622
 Ensamblador, 3
 lenguaje, 13, 508
 Entera, programación lineal, 817-825
 Entorno, 26-28
 Entrada, nodo de, 531, 605
 Epílogo, 742
 Eqn, 331
 Equivalencias, análisis basado en, 935

Errores
 corrección de, 113-114, 192-196, 228-231
 producción de, 196
 recuperación de, 283-284, 295-297
 Ershov, A. P., 426, 579-580, 705
 Ershov, número de, 567-572
 Escalado, 848, 850
 Escaneado, estado, 474
 Escaneo, 110
 Vea también Léxico, analizador
 Escritura, barrera de, 486-487
 Espacial
 localidad, 455-457, 465, 477, 884
 reutilización, 806, 809-811
 Espacio
 en blanco, 41, 77-78
 restricción de partición de, 831-838
 Vea Datos, espacio de; Iteraciones,
 espacio de; Nulo, espacio;
 Procesadores, espacio de
 Especulativa, ejecución, 708, 717-719
 Estable
 conjunto, 488
 estado, 742
 Estado, 147, 205
 de aceptación
 Vea Final, estado
 del almacenamiento del programa, 26-28
 inicial
 Vea Inicial, estado
 Vea también Muerto, estado;
 Minimización de estados; Analiza-
 dor sintáctico, estado del
 Estática
 comprobación, 97-98, 357
 Vea también Tipos, comprobación de
 forma de asignación individual,
 369-370
 política, 25
 RAM, 456
 repartición, 518, 524
 Estático
 acceso, 816
 alcance, 25, 28-31, 442
 Vea también Alcance
 almacenamiento, 429, 442

Estratificado, programa en Datalog, 930-931
Estructura
 Vea Clase
Etiqueta, 46, 364, 366
Euclidiano, algoritmo, 820
Expresión, 94, 96-97, 101-105, 359, 568-572
 disponible, 610-615, 648-649, 653
 Vea también Aritmética, expresión; Booleana, expresión; Infijo, expresión; Postfijo, expresión; Prefijo, expresión; Tipo, expresión de
Extensional, predicado de base de datos, 924
Extremo posterior, 4, 357

F

Fahndrich, M., 961, 963
Farkas, lema de, 872-875
Fase, 11
Feautrier, P., 900
Feldman, S. I., 426
Fenichel, R. R., 502-503
Ferrante, J., 704, 900
Fila
 Vea Tupla
Filas, orden por, 382, 785
Final, estado, 130-131, 147, 205
Finito, autómata
 Vea Autómata
Firma, 361
Fischer, C. N., 580
Fisher, J. A., 766-767
Física
 dirección, 427
 memoria, 454-455
Fisión, 848, 850, 854
Flex, 189-190
Flotante, basura, 484
Floyd R. W., 300-301
Flujo
 de control, 399-409, 413-417, 525
 de control, ecuación de, 600, 605
 de datos, análisis de, 18, 23, 597-705, 921

grafo de, 529-531
 Vea también Reducible, grafo de flujo; Síper, grafo de flujo de control
 sensibilidad al 933, 936-937
Formal, parámetro, 33, 942
Fortran, 113, 382, 779, 886
Fortran, H., 703
Fourier-Motzkin, algoritmo, 796-797
Fragmentación, 457-460
Framework
 Vea Flujo de datos, framework de análisis de; Distributivo, framework; Monótono, framework
Fraser, C. W., 580
Frege, G., 502-503
Frente de onda, 876-877
Front end, 4, 40-41 357, 986
Frontera
 Vea Cosecha
Fuente, lenguaje, 1
Fuertemente,
 conectado, componente, 751, 859
 tipificado, lenguaje, 387
Fuerza, reducción de
 Vea Reducción en fuerza
Función, 29
 tipo de, 371, 423
 Vea también Procedimiento
Funcional, lenguaje, 443
Fusión, 848, 850

G

Ganapathi, M., 579-580
Gao, G., 902
GCD, 818-820
GDA, 359-362, 533-541, 951
Gear, C. W., 705
Gen, 603, 611
Gen-eliminar, forma, 603
Generacional, recolección de basura, 483, 488-489
Geschke, C. M., 705
Ghiya, R., 961, 963
Gibson, R. G., 38

Giratorio, archivo de registro, 762
 Glaeser, C. D., 767
 Glanville, R. S., 579-580
 Global
 optimización de código
 Vea Código, optimización de
 variable, 442
 GNU, 38, 426
 Gosling, J., 426
 GOTO, 246, 249, 261
 Grafo
 Vea Llamadas, grafo de; GDA; Datos,
 grafo de dependencia de; Dependencia,
 grafo de; Flujo, grafo de;
 Dependencias del programa, grafo
 de; Coloración
 Graham, S. L., 579-580
 Gramática, 42-50, 197-199, 204-205
 Vea también Ambigua, gramática;
 Aumentada, gramática
 Gramatical, símbolo, 199
 Granularidad, 917
 del paralelismo, 773-775
 Gross, T. R., 766-767
 Grune, D., 302
 Grupo, reutilización de, 806, 811-813
 Gupta, A., 900-901

H

Hacia abajo, movimiento de código, 731-732
 Hacia arriba, movimiento de código, 730-732
 Hacia atrás, flujo, 610, 615, 618, 627, 669
 Hacia delante, flujo, 615, 618, 627, 668
 Hallem, S., 963
 Halstead, M. H., 426
 Hanson, D. R., 580
 Hardware
 renombramiento de registros de, 714
 síntesis de, 22
 Hecht, M. S., 705
 Heintze, N., 961, 963
 Hendren, L. J., 961-963
 Hennessy, J. L., 38, 579-580, 766-767, 899, 901

Heredado, atributo, 55, 304-305, 307
 Herencia, 18
 Hewitt, C., 502-503
 Hijo, 46
 Hoare, C. A. R., 302
 Hobbs, S. O., 705
 Hoja, 45-46
 región, 673
 Hopcroft, J. E., 189-190, 302
 Hopkins, M. E., 580
 Hoyo, 457
 Hudson, R. L., 502-503
 Hudson, S. E., 302
 Huffman, D. A., 189-190
 Huskey, H. D., 426

I

IDB
 Vea Intensional, predicado de base de
 datos
 Idempotente, 122, 619
 Identidad, función, 624
 Identificador, 28, 79-80
 If, instrucción, 401
 Imperativo, lenguaje, 13
 Inalcanzable
 código; *Vea Muerto*, código
 estado, 473
 Inclusión, análisis basado en la, 935
 Incremental
 evaluación, 928-930
 colección de basura, 483-487
 traducción; *Vea Al instante*, generación
 Incremento, instrucción de, 509
 Indexada, dirección, 513
 Índice, 365
 Indirecta, dirección, 513
 Indirectos, triples, 368-369
 Individual, producción, 232
 Inducción, variable de, 592-596, 687-688
 Inexplorado, estado, 474
 Inferior, elemento, 619, 622
 Infijo, expresión, 40, 52-53
 Ingberman, P. Z., 302
 Iniciación, intervalo de, 745

- Inicial
 estado, 131, 147, 205
 símbolo, 43, 45, 197
- Inicialización, 615
- Inmediato, dominador, 657-658
- Insegura, regla de Datalog, 930
- Inseguro, lenguaje, 498
- Instrucción, 93-94, 100-101, 978-981
 Vea también Break, instrucción;
 Continue, instrucción; If, instrucción;
 Switch, instrucción; While,
 instrucción
- Instrucciones, canalización de, 708-709
 Vea también Software, canalización por
- Intencional, predicado de base de datos, 924
- Intercalación, 887-890
- Intermedio, código, 9, 91-105, 357-426, 507,
 971-981
- Intérprete, 2
- Interprocedural, análisis, 713, 903, 964
- Interrupción, 526
- Intersección, 612-613, 615, 620, 650
- Intraprocedural, análisis, 903
- Inversión, 849-850
- Irons, E. T., 354
- Iteraciones, espacio de, 779-780, 788-799
- Iterativo, algoritmo de flujo de datos,
 605-607, 610, 614, 626-628
- Izquierda
 asociatividad por la, 48
 factorización por la, 214-215
 recursividad, 67-68, 71, 212-214, 328-
 331
- Izquierdo, lado
 Vea Encabezado
- J**
- Jacobs, C. J. H., 302
- Java, 2, 13, 18-19, 25, 34, 76, 381, 903,
 934, 944
 máquina virtual de, 507-508
- Jazayeri, M., 354
- Jerárquica, reducción, 761-762
- Jerárquico, tiempo, 857-859
- JFlex, 189-190
- Johnson, R. K., 705
- Johnson, S. C., 300-302, 355, 426, 502-503,
 579-580
- Justo a tiempo, compilación, 508
- JVM
 Vea Java, máquina virtual de
- K**
- Kam, J. B., 705
- Kasami, T., 301-302, 705
- Kennedy, K., 899-900, 963
- Kernighan, B. W., 189-190
- Killdall, G., 704-705
- Kleene, S. C., 189-190
 cierre de; *Vea* Cierre
- Knoop, J., 705
- Knuth, D. E., 189-190, 300, 302, 354-355,
 502-503
- Knuth-Morris-Pratt, algoritmo de, 136-138
- Korenjak, A. J., 300, 302
- Kosaraju, S. R., 705
- Kuck, D. J., 766-767, 899-901
- Kung, H. T., 901
- L**
- LALR, analizador sintáctico, 259, 266-275,
 283, 287
- Lam, M. S., 767, 899-902, 961-964
- Lamport, L., 503, 766-767, 899-901
- Lattice, 621
 Vea también Semi-lattice
- Lawrie, D. H., 900
- Lea, 458
- LeBlanc, R. J., 580
- Lectura, barrera de, 486
- Leiserson, C. E., 901
- Lenguaje, 44, 118
 Vea también Java; Fuente, lenguaje;
 Destino, lenguaje
- Lesk, M. E., 189-190
- Leu, T., 963
- Levin, V., 962
- Lewis, P. M. II, 300, 302, 355

- Lex, 126-127, 140-145, 166-167, 189-190, 294-295
- Lexema, 111
- Léxico
- alcance; *Vea* Estático, alcance
 - analizador, 5-7, 41, 76-84, 86, 109-190, 209-210, 294-295, 967-969
 - error, 194
- Lexicográfico, orden, 791
- Ley
- Vea* Asociatividad, Condición commutativa, Condición distributiva, Idempotente
- Liao, S.-W., 901
- Libre
- de contexto, gramática; *Vea* Gramática
 - estado, 473
 - lista, 459-460, 471
 - trozo, 457
- Lichtenber, J., 962
- Líder, 526-527
- Lieberman, H., 502-503
- Lim, A. W., 901
- Límites, comprobación de, 19, 24, 920-921
- Lineal, programación
- Vea* Entera, programación lineal
- Listas, programación de, 723-726
- Literal, 922
- Livshits, V. B., 962-963
- LL, analizador sintáctico
- Vea* Predictivo, analizador sintáctico
- LL, gramática, 223
- Llamada, 365, 423-424, 467, 518-522, 539, 541
- por nombre, 35
 - por referencia, 34
 - por valor, 34
 - sitio de, 904, 950
- Llamadas
- cadena de, 908-910, 946-949
 - grafo de, 904-906, 943-944
 - secuencia de, 436-438
- LLgen, 300
- Local, optimización de código
- Vea* Básico, bloque
- Localidad, 455, 769
- Vea* también Espacial, localidad;
 - Temporal, localidad
- Lógica, dirección, 427
- Lógico, error, 194
- Lohtak, O., 962
- Longitud variable, datos de, 438-440
- Loveman, D. B., 901
- Lowry, E. S., 579-580, 705
- LR de lectura anticipada, analizador sintáctico
- Vea* LALR, analizador sintáctico
- LR(0), autómata, 243, 247-248, 252
- LR, analizador sintáctico, 53-252, 275-277, 325, 348-352
- Vea* también Canónico, analizador
 - sintáctico LR; LALR, analizador sintáctico;
 - SLR, analizador sintáctico
- M**
- Macro, 13
- Manejador, 235-236
- Mapeo directo, caché de, 457, 788
- Máquina, lenguaje, 508
- Marcar
- y compactar, 476-482
 - y limpiar, 471-476, 482
- Markstein, P. W., 580
- Martin, A. J., 503
- Martin, M. C., 963
- Más general, unificador, 393
- Vea* también Unificación
- Matrices, multiplicación de, 782-788
- Máximo, punto fijo, 626-628, 630-631
- Maydan, D. E., 899, 901
- Mayor
- divisor común; *Vea* GCD
 - límite inferior, 620, 622
 - orden, función de, 444
- Mayúsculas y minúsculas, sensibilidad a, 125
- McArthur, R., 426
- McCarthy, J., 189-190, 502-503
- McClure, R. M., 302
- McCullough, W. S., 189-190
- McGarvey, C., 962
- McKellar, A. C., 900-901

- McNaughton, R., 189-190
McNaughton-Yamada-Thompson, algoritmo, 159-161
Medlock, C. W., 579-580, 705
Mejor ajuste, 458
Memorización, 823
Memoria, 20, 772-773
 Vea también Montículo; Física, memoria; Almacenamiento; Virtual, memoria
Memoria
 fuga de, 25, 461
 jerarquía de, 20, 454-455
Menor límite superior, 621
META, 300
Metal, 918, 962
Método, 29
 invocación a, 33
 llamada a; *Vea Llamada*
 Vea también Procedimiento; Virtual, método
MGU
 Vea Más general, unificador
Milanova, A., 962-963
Milner, R., 426
Minimización de estados, 180-185
Minsky, M., 503
Mirilla, optimización tipo, 549-552
ML, 387, 443-445
Mock, O., 426
Modo de pánico
 recolección de basura en, 492-493.
 recuperación en, 195-196, 228-230,
 283-284
Modular
 expansión de variables, 758-761
 tabla de reservación de recursos,
 746-747, 758
Monótono, framework, 624-628, 635
Montículo, 428-430, 452-463, 518, 935
Moore, E. F., 189-190
MOP
 Vea Reunión sobre los caminos,
 solución
Morel, E., 705
Morris, D., 354
Morris, J. H., 189-190
Moss, J. E. B., 502-503
Motwani, R., 189-190, 302
Mowry, T. C., 900-901
Muerta, variable, 608
Muerto
 código, 533, 535, 550, 591-592
 estado, 172, 183
Multiprocesador, 772-773, 895
 Vea también SIMD; Un solo programa, varios datos
Muraoka, T., 766-767, 899, 901
Mutador, 464
Muy larga, palabra de instrucción
 Vea VLIW
Muy ocupada, expresión
 Vea Anticipada, expresión
- N**
- NAA, 690
NAC, 633
Natural, ciclo, 667, 673
Naur, P., 300, 302
Neliac, 425
Nivel de frase, recuperación a, 196, 231
No determinista, autómata finito, 147-148,
 152-175, 205, 257
No reducible, grafo de flujo
 Vea Reducible, grafo de flujo
No terminal, 42-43, 45, 197-198
 símbolo, 349
No uniforme, acceso a memoria, 773
Nodo, 46
 mezcla de, 953
Nombre, 26-28
Núcleo, 777
 elemento de, 245, 272-273
Nulidad, 808
Nullable, 175-177
Nulo, espacio
NUMA
 Vea No uniforme, acceso a memoria

O

O grande (Big-oh), 159

Objeto

 código, 358

Vea también Código, generación de programa, 427-428

Objetos

 creación de, 937

 propiedad de, 462

 sensibilidad a, 950

Obtener, 708

Ogden, W. F., 354

Olsztyn, J., 426

Ondrussek, B., 962

Optimización

Vea Código, optimización de

Oración, 200

Ordenado, BDD, 952

Orientado a objetos, lenguaje

Vea C++; Java

P

Paakki, J., 354-355

Padre, 46

Palabra clave, 50-51, 79-80, 132-133

Panini, 300

Papua, D. A., 902

Parafrasear, 899

Paralelismo, 19-20, 707-902, 917

Paralelo, recolección de basura en, 495-497

Paramétrico, polimorfismo, 391

Vea también Polimorfismo

Parámetro(s), 422

 actual, 33, 434, 942

 paso de, 33-35, 365

Parcial

 eliminación de redundancia, 639-655

 orden, 619-621, 623

 recolección de basura, 483, 487-494

Parcialmente muerta, variable, 655

Parcialmente ordenado, conjunto

Vea Poset

Parr, T., 302

Pasada, 11

Paso

 de mensajes, máquina de, 773, 894

 directo, código de, 406

Patel, J. H., 767

Patrón, 111

 coincidencia de, en árboles, 563-567

Patterson, D. A., 38, 579-580, 766-767, 899, 901

Pausa

 corta, recolección de basura, 483-494

 tiempo de, 465

Vea también Pausa corta, recolección de basura

PDG

Vea Programa, grafo de dependencias de

Pelegri-Llopert, E., 580

Permutación, 849-850

Peso de un camino, 822

Peterson, W. W., 705

PFC, 899

Phoenix, 38

Pierce, B. C., 426

Pila, 325, 518, 520

Vea también Tiempo de ejecución, pila en

 apuntador de, 437

 máquina de, 507

Pincus, J. D., 962-963

Pitts, W., 189-190

Pnueli, A., 964

Poliedro

Vea Convexo, poliedro

Polimorfismo, 391-395

Poner en línea, 903-904, 914

Por adelantado, 78, 144-145, 171-172, 272-275

Por la derecha, derivación de, 201

Por la izquierda, derivación de, 201

Por profundidad

 búsqueda, 57

 orden, 660

Porterfield, A., 900, 902

Posdominador, 728

Poset, 619

Positivo, cierre, 123

Postergable, expresión, 646, 649, 651-654

Posterior, arista, 662, 664-665
 Postfijo
 esquema de traducción, 324-327
 expresión, 40, 53-54
 Postorden, recorrido, 58, 432
 Vea también Por profundidad, orden
 Pratt, V. R., 189-190
 PRE
 Vea Parcial, eliminación de
 redundancia
 Precedencia, 48, 121-122, 279-281, 293-294
 Predecesor, 529
 Predicada, ejecución, 718, 761
 Predicado, 921-922
 Predictivo, analizador sintáctico, 64-68,
 222-231, 343-348
 Prefijo, 119, 918, 962
 expresión, 327
 Preobtención, 718, 896
 Preobtener, 457
 Preorden, recorrido, 58, 432
 Preprocesador, 3
 Primer ajuste, 458
 Primera
 expresión, 649-650, 654
 generación, lenguaje de, 13
 PRIMERO, 220-222
 Principal, atributo, 341
 Priorizado, orden topológico, 725-726
 Private, 31
 Privatizable, variable, 758
 Procedimiento, 29, 422-424
 llamada a; *Vea* Llamada
 parámetros de, 448-449
 Procesadores, espacio de, 779-781, 838-841
 Producción, 42-43, 45, 197, 199
 Vea también Errores, producción de
 Productos, semienrejado de, 622-623
 Proebsting, T. A., 580
 Profundidad de un grafo de flujo, 665
 Programa, grafo de dependencias
 de, 854-857
 Programación, 710-711, 716
 lenguaje de, 12-14, 25-35
 Vea también Ada, C, C++, Fortran,
 Java, ML

Prólogo, 742
 Propia, reutilización, 806-811
 Prosser, R. T., 705
 Protected, 31
 Proximidades, compactación de las, 736
 Proyección, 955
 PTRAN, 900
 Public, 31
 Pugh, W., 899, 902
 Pulso
 Vea Reloj
 Punto fijo
 Vea Máximo, punto fijo
 Purificar, 25, 452

Q

Qian, F., 962
 Quicksort, 431-432, 585
 Quinta generación, lenguaje de, 13

R

Rabin, M. O., 189-190
 Raíz, 46
 conjunto, 466-467, 488
 Rajamani, S. K., 962
 Randell, B., 502-503
 Rango
 completo, matriz de, 808
 de una matriz, 807-809
 Rastreo, recolección de basura basada en
 470-471
 Vea también Marcar y compactar;
 Marcar y limpiar
 Rau, B. R., 767
 Recolección de basura, 25, 430, 463-499
 Vea también Marcar y compactar;
 Marcar y limpiar; Pausa corta,
 recolección de basura con
 Recordado, conjunto, 491
 Recorrido, 56-57
 Vea también Por profundidad,
 búsqueda; Postorden, recorrido;
 Preorden, recorrido

- Recursivo
 analizador sintáctico de descenso, 64, 219-222
 descenso, 338-343
 tipo, 372
- Recursos
 restricción de, 711
 tabla de reservación de, 719-720
Vea también Modular, tabla de reservación de recursos
- Reducción, 234, 324, 388-389
 en fuerza, 536, 552, 592-596
- Reducible, grafo de flujo, 662, 664, 673-677, 684-685
- Reducido, computadora con conjunto de instrucciones
Vea RISC
- Redundancia completa, 645
- Referencia
 conteo de, 462-463, 466, 468-470
 variable de, 34, 686-689
Vea Apuntador
- Reflexión, 944-945
- Reflexividad, 619
- Región, 672-686, 694-699, 733-734, 911
- Regiones, asignación basada en, 463
- Registro, 18, 20, 371, 376-378, 454-455, 542-543, 584, 714-715
 de activación, 433-452
 descriptor de, 543, 545-547
Vea también Seudoregistro, Giratorio, archivo de registro
- Registros
 asignación de, 510, 556
 par de, 510
 renombramiento de; *Vea* Hardware, renombramiento de registros
 repartición de, 510-512, 533-557, 570-572, 716, 743
- Regla, 922-923
- Regular
 definición, 123
 expresión, 116-122, 159-163, 179-180, 189, 210
- Rehof, J., 961, 963
- Reindexado, 848, 850
- Relación, 922, 954
- Relativa, dirección, 371, 373, 381
- Reloj, 708
- Renvoise, C., 705
- Reservada, palabra
Vea Palabra clave
- Restricción
Vea Dependencia de control, restricción de; Datos, dependencia de; Recursos, restricción de
- Resumen, análisis basado en el, 911-914
- Retirada, arista de, 661, 664-665
- Retorno, 365, 467, 518-522, 906, 942
 valor de, 434
- Reunión, 605, 615, 618-619, 622-623, 633, 678, 695
 sobre los caminos, solución, 629-631
- Reutilización
Vea Datos, reutilización de
- Revestimiento, 560-563
- Rinard, M., 962-963
- RISC, 21, 507-508
- Ritchie, D. M., 426, 502-503
- Rodeh, M., 766-767
- Rosen, B. K., 704
- Rosenkrantz, D. J., 355
- Rothberg, E. E., 900-901
- Rounds, W. C., 354
- Rountev, A., 962-963
- Roy, D., 963
- Russell, L. J., 502-503
- Ruwase, O., 962-963
- Ryder, B. G., 962-963
- S**
- Sadgupta, S., 767
- Salida
 bloque de, 677
 dependencia de, 711, 816
 nodo de, 605
- Salto, 513, 527, 551-552
 código de, 408, 974-977
- Samelson, K., 354-355
- Sarkar, V., 902

- SCC
Vea Fuertemente conectado, componente
- Scholten, C. S., 503
- Schorre, D. V., 302
- Schwartz, J. T., 579, 581, 704
- Scott, D., 189-190
- Scott, M. L., 38
- SDD
Vea Sintaxis, definición orientada por la SDT
Vea Sintaxis, traducción orientada por la
- SDV, 962
- Secuencias de comandos, lenguaje de, 13-14
- Secundario
 almacenamiento, 20
 efecto, 306, 314-316, 727
- Sedgewick, R., 585
- Seguimiento, 220-222
- Siguiente pos, 177-179
- Segunda generación, lenguaje de, 13
- Seguridad
Vea Conservador, análisis de flujo de datos
- Semántica, 40
 Regla; *Vea* Sintaxis, definición orientada por la
- Semántico
 análisis, 8-9
 error, 194
- Semienrejado, 618-623
- Sensibilidad
Vea Contexto, sensibilidad al; Flujo, sensibilidad al
- Sensible al contexto, análisis, 906-907, 945-950
- Sethi, R., 38, 579, 581
- Seudoregistro, 713
- Shannon, C., 189-190
- Sharif, M., 964
- Shostak, R., 902
- Sielaff, D. J., 962-963
- Siguiente ajuste, 458-459
- Simbólica, constante, 793
- Simbólico
 análisis, 686-699
 mapa, 690
- Símbolos, tabla de, 4-5, 11, 85-91, 423, 970-971
- SIMD, 21, 895-896
- Simétrico, multiprocesador, 772
- Simple, definición orientada por la sintaxis, 56
- Simulación, 23
- Sin* ϵ , gramática, 232
- Sin contexto, lenguaje, 200, 215-216
- Sincronización, 828, 832, 853-854, 880-882
- Sintáctico
 análisis; *Vea* Analizador sintáctico
 árbol, 41, 69-70, 92-93, 318-321, 358, 367, 981-986
- Sintaxis, 40
 definición orientada por la, 54-56, 304-316
 error de, 194
 traducción orientada por la, 40, 57-60, 324-352
Vea también Gramática
- Síntesis, 4
- Sintetizado, atributo, 54-56, 304-305
- SLAM, 962
- SLR, analizador sintáctico, 252-257, 283
- SMP
Vea Simétrico, multiprocesador
- Sobrecarga, 99, 390-391
- Software
 canalización por, 738-763, 895
 productividad de, 23-25
 vulnerabilidad del; *Vea* Vulnerabilidad del software
- Sólido, sistema de tipos, 387
- Solución ideal para un problema de flujo de datos, 628-630
- SOR
Vea Sucesiva, sobre relajación
- SPMD
Vea Un solo programa, varios datos
- SQL, 22-23
 inyección de, 918-919

SSA

Vea Estática, forma de asignación individual

Stallman, R., 301

Stearns, R. E., 300, 302, 355

Steel, T., 426

Steensgaard, B., 961, 964

Steffens, E. F. M., 503

Strong, J., 426

Subcadena, 119

Subconjuntos, construcción de, 153-154

Submeta, 923

Subsecuencia, 119

Sucesiva, sobre relajación, 863

Sucesor, 529

Sufijo, 119

Súper grafo de control de flujo, 906

Superescalar, máquina, 710

Superior, elemento, 619, 622

Switch, instrucción, 418-421

T

T1-T2, reducción, 677

Tabla

Vea Relación; Recursos, tabla de reservación de; Símbolos, tabla de; Transiciones, tabla de

Takizuka, T., 767

Tamura, E., 767

Tardieu, O., 961, 963

Tareas, paralelismo de, 776

Temporal

 localidad, 455-457, 777, 884-885

 reutilización, 806

Tercera generación, lenguaje de, 13

Terminal, 42-43, 45, 197-198, 305

TeX, 331

Thompson, K., 189-190

Tiempo, restricción de partición de, 868-875, 989-992

Tiempo de ejecución, 25

 entorno, 427

 pila en, 428-451, 468

Tipo, 938

 expresión de, 371-372, 393, 395

variable de, 391

Vea también Básico, tipo; Función, tipo de; Recursivo, tipo

Tipos

 comprobación de, 24, 98-99, 370, 386-398

 conversión de, 388-390

Vea también Coerción

 equivalencia de, 372-373

 inferencia de, 387, 391-395

 seguridad de, 19, 464-465

 síntesis de, 387

Tjiang, S. W. K., 579-580

TMG, 300

Token, 41, 43, 76, 111

Tokura, N., 705

Tokuro, M., 767

Topológico, orden, 312

Vea también Priorizado, orden topológico

Torczon, L., 580, 963

Towle, R. A., 902

Transferencia

 barrera de, 486

 función de, 599-600, 603-604, 615, 623-624, 629, 634, 676-679, 691-693

Transición, 205

 diagrama de, 130-131, 147-148

 función de, 147, 150

Transiciones, tabla de, 148-149, 185-186

Transitividad, 619-620

Tren, algoritmo de, 483, 490-493

Tres direcciones, código de, 42, 99, 363-369

Triple, 367-368

Tritter, A., 426

Trozo, 457-459

Tupla, 954

U

Ubicación, 26-28

Ullman, J. D., 189-190, 301-302, 579, 581, 705, 962, 964

Última expresión, 649, 654

Ultimapos, 175-177

Umanee, N., 962

Un solo programa, varios datos, 776
Una sola instrucción, varios datos
 Vea SIMD
UNCOL, 425
UNDEF, 633
Unificación, 393, 395-398
Unión, 119, 121-122, 605, 613, 615, 620, 650, 955-957
Unkel, C., 963
Uso, 609
 antes de la definición, 602
 conteo de, 554-555
Ustuner, A., 962
Utilizada
 expresión, 649, 653-654
 variable, 528-529

V

Vacía, cadena, 44, 118, 121
Válido, elemento, 256
Valor, 26-27
Valor-1, 26, 98
 Vea también Ubicación
Valor-número, 360-362, 390
Valor-r, 26, 98
Variable, 26-28
 Vea también No terminal; Referencia, variable de
Variables
 expansión de; *Vea Modular*, expansión de variables
 independientes, prueba de, 820-821
Vectores, máquina de, 886, 895-896
Veneno, bit, 718
Verdadera, dependencia, 711, 815
Viable, prefijo, 256-257
Virtual
 máquina, 2
 Vea también Java, máquina virtual de
 memoria, 454-455
 método, 904, 916-917, 934, 941-944
Visualización, 449-451
Viva, variable, 528-529, 608-610, 615
VLIW, 19-21, 710

Von Neumann, lenguaje de, 13
Vulnerabilidad del software, 917-921
Vyssotsky, V., 704-705

W

Weber, H., 300, 302
Wegman, M. N., 704
Wegner, P., 704-705
Wegstein, J., 426
Weinberger, P. J., 189-190
Weinstock, C. B., 705
Wexelblat, R. L., 38
Whaley, J., 961, 963-964
While, instrucción, 401
Widom, J., 962, 964
Wilderness, trozo, 458
Wilson, P. R., 502-503
Wirth, N., 300, 302, 426
Wolf, M. E., 900-902
Wolfe, M. J., 902
Wonnacott, D., 899, 902
Wood, G., 767
Wulf, W. A., 705

Y

Yacc, 287-297, 354
Yamada, H., 189-190
Yochelson, J. C., 502-503
Younger, D. H., 301-302

Z

Zadeck, F. K., 704
Zhu, J., 961, 964

Alfred V. Aho es el profesor Lawrence Gussman de ciencias de la computación en la Columbia University. El profesor Aho ha obtenido varios premios, incluyendo el Great Teacher Award para el año 2003, de la Sociedad de graduados de Columbia y la medalla John von Neumann del Instituto de Ingenieros en Electricidad y Electrónica (IEEE). Es miembro de la Academia Nacional de Ingeniería (NAE) y miembro titular de la Asociación de Maquinaria de la Computación (ACM), y del IEEE.

Monica S. Lam es profesora de ciencias de la computación en la Stanford University, fue jefa de científicos en Tensilica, y es presidenta fundadora de moka5. Dirigió el proyecto SUIF, el cual produjo uno de los compiladores de investigación más populares, y fue pionera en numerosas técnicas de compiladores que se utilizan en la industria.

Ravi Sethi fundó la organización de investigación en Avaya y es presidente de Avaya Labs. Fue vicepresidente ejecutivo en Bell Laboratories y director técnico de software de comunicaciones en Lucent Technologies. Ha desempeñado varios puestos de enseñanza en la Pennsylvania State University y en la Arizona University, y ha impartido clases en la Princeton University y la Rutgers University. Es miembro titular de la ACM.

Jeffrey D. Ullman es presidente de Gradiance Corp., y profesor emérito Stanford W. Ascherman en ciencias de la computación en la Stanford University. Entre sus intereses de investigación se encuentran la teoría de bases de datos, la integración de bases de datos, la minería de datos y la educación mediante la infraestructura de información. Es miembro de la NAE, miembro titular de la ACM, y ganador de los premios Karlstrom y Knuth.

