Contenido

1 INTRODUCCIÓN 1

1.1 ¿Por qué compiladores? Una breve historia 2
1.2 Programas relacionados con los compiladores 4
1.3 Proceso de traducción 7
1.4 Estructuras de datos principales en un compilador 13
1.5 Otras cuestiones referentes a la estructura del compilador 14
1.6 Arranque automático y portabilidad 18
1.7 Lenguaje y compilador de muestra TINY 21
1.8 C-Minus: Un lenguaje para un proyecto de compilador 26
Ejercicios 27
Notas y referencias 29

2 RASTREO O ANÁLISIS LÉXICO 31

2.1 El proceso del análisis léxico 32
2.2 Expresiones regulares 34
2.3 Autómatas finitos 47
2.4 Desde las expresiones regulares hasta los DFA 64
2.5 Implementación de un analizador léxico TINY ("Diminuto") 75
2.6 Uso de Lex para generar automáticamente un analizador léxico 81
Ejercicios 91
Ejercicios de programación 93
Notas y referencias 94

3 GRAMÁTICAS LIBRES DE CONTEXTO Y ANÁLISIS SINTÁCTICO 95

3.1 El proceso del análisis sintáctico 96
3.2 Gramáticas libres de contexto 97
3.3 Árboles de análisis gramatical y árboles sintácticos abstractos 106
3.4 Ambigüedad 114
3.5 Notaciones extendidas: EBNF y diagramas de sintaxis 123
3.6 Propiedades formales de los lenguajes libres de contexto 128
3.7 Sintaxis del lenguaje TINY 133
Ejercicios 138
Notas y referencias 142

4 ANÁLISIS SINTÁCTICO DESCENDENTE 143

4.1 Análisis sintáctico descendente mediante método descendente recursivo 144
4.2 Análisis sintáctico LL(1) 152
4.3 Conjuntos primero y siguiente 168
4.4 Un analizador sintáctico descendente recursivo para el lenguaje TINY 180
4.5 Recuperación de errores en analizadores sintácticos descendentes 183
Ejercicios 189
Ejercicios de programación 193
Notas y referencias 196

5 ANÁLISIS SINTÁCTICO ASCENDENTE 197

5.1 Perspectiva general del análisis sintáctico ascendente 198
5.2 Autómatas finitos de elementos LR(0) y análisis sintáctico LR(0) 201
5.3 Análisis sintáctico SLR(1) 210
5.4 Análisis sintáctico LALR(1) y LR(1) general 217
5.5 Yacc: un generador de analizadores sintácticos LALR(1) 226
5.6 Generación de un analizador sintáctico TINY utilizando Yacc 243
5.7 Recuperación de errores en analizadores sintácticos ascendentes 245
Ejercicios 250
Ejercicios de programación 254
Notas y referencias 256

6 ANÁLISIS SEMÁNTICO 257

6.1 Atributos y gramáticas con atributos 259
6.2 Algoritmos para cálculo de atributos 270
6.3 La tabla de símbolos 295
6.4 Tipos de datos y verificación de tipos 313
6.5 Un analizador semántico para el lenguaje TINY 334
Ejercicios 339
Ejercicios de programación 342
Notas y referencias 343

7 AMBIENTES DE EJECUCIÓN 345

7.1 Organización de memoria durante la ejecución del programa 346
7.2 Ambientes de ejecución completamente estáticos 349
7.3 Ambientes de ejecución basados en pila 352
7.4 Memoria dinámica 373
7.5 Mecanismos de paso de parámetros 381

7.6 Un ambiente de ejecución para el lenguaje TINY 386
Ejercicios 388
Ejercicios de programación 395
Notas y referencias 396

8 GENERACIÓN DE CÓDIGO 397

8.1 Código intermedio y estructuras de datos para generación de código 398
8.2 Técnicas básicas de generación de código 407
8.3 Generación de código de referencias de estructuras de datos 416
8.4 Generación de código de sentencias de control y expresiones lógicas 428
8.5 Generación de código de llamadas de procedimientos y funciones 436
8.6 Generación de código en compiladores comerciales: dos casos de estudio 443
8.7 TM: Una máquina objetivo simple 453
8.8 Un generador de código para el lenguaje TINY 459
8.9 Una visión general de las técnicas de optimización de código 468
8.10 Optimizaciones simples para el generador de código de TINY 481
Ejercicios 484
Ejercicios de programación 488
Notas y referencias 489

Apéndice A: PROYECTO DE COMPILADOR 491

A.1 Convenciones léxicas de C− 491
A.2 Sintaxis y semántica de C− 492
A.3 Programas de muestra en C− 496
A.4 Un ambiente de ejecución de la Máquina Tiny para el lenguaje C− 497
A.5 Proyectos de programación utilizando C− y TM 500

Apéndice B: LISTADO DEL COMPILADOR TINY 502

Apéndice C: LISTADO DEL SIMULADOR DE LA MÁQUINA TINY 545

Bibliografía 558

Índice 562

Prefacio

Este libro es una introducción al campo de la construcción de compiladores. Combina un estudio detallado de la teoría subyacente al enfoque moderno para el diseño de compiladores, junto con muchos ejemplos prácticos y una descripción completa, con el código fuente, de un compilador para un lenguaje pequeño. Está específicamente diseñado para utilizarse en un curso introductorio sobre el diseño de compiladores o construcción de compiladores a un nivel universitario avanzado. Sin embargo, también será de utilidad para profesionales que se incorporen o inicien un proyecto de escritura de compilador, en la medida en que les dará todas las herramientas necesarias y la experiencia práctica para diseñar y programar un compilador real.

Existen ya muchos textos excelentes para este campo. ¿Por qué uno más? La respuesta es que la mayoría de los textos actuales se limita a estudiar uno de los dos aspectos importantes de la construcción de compiladores. Una parte de ellos se restringe a estudiar la teoría y los principios del diseño de compiladores, y sólo incluye ejemplos breves de la aplicación de la teoría. La otra parte se concentra en la meta práctica de producir un compilador real, ya sea para un lenguaje de programación real, o para una versión reducida de alguno, e incursiona sólo superficialmente en la teoría en la cual se basa el código para explicar su origen y comportamiento. Considero que ambos enfoques tienen carencias. Para comprender en realidad los aspectos prácticos del diseño de compiladores, se necesita haber comprendido la teoría, y para apreciar realmente la teoría, se requiere verla en acción en una configuración práctica real o cercana a la realidad.

Este texto se encarga de proporcionar el balance adecuado entre la teoría y la práctica, y de suministrar suficientes detalles de implementación real para ofrecer una visión real de las técnicas sin abrumar al lector. En este texto proporciono un compilador completo para un lenguaje pequeño escrito en C y desarrollado utilizando las diferentes técnicas estudiadas en cada capítulo. Además, ofrezco descripciones detalladas de técnicas de codificación para ejemplos adicionales de lenguajes a medida que se estudian los temas relacionados. Finalmente, cada capítulo concluye con un extenso conjunto de ejercicios, que se dividen en dos secciones. La primera contiene los diversos ejercicios que se resuelven con papel y lápiz y que implican poca programación. La segunda contiene aquellos que involucran una cantidad importante de programación.

Al escribir un texto así se debe tener en cuenta los diferentes sitios que ocupa un curso de compiladores en diferentes currículos de ciencias de la computación. En algunos programas se requiere haber tomado un curso de teoría de autómatas; en otros, uno sobre lenguajes de programación; mientras que en otros es suficiente con haber tomado el curso de estructuras de datos. Este texto sólo requiere que el lector haya tomado el curso habitual sobre estructuras de datos y que conozca un poco del lenguaje C, incluso está planeado

Capítulo 1

Introducción

1.1 ¿Por qué compiladores? Una breve historia
1.2 Programas relacionados con los compiladores
1.3 Proceso de traducción
1.4 Estructuras de datos principales en un compilador
1.5 Otras cuestiones referentes a la estructura del compilador
1.6 Arranque automático y portabilidad
1.7 Lenguaje y compilador de muestra TINY
1.8 C-Minus: un lenguaje para un proyecto de compilador

Los compiladores son programas de computadora que traducen un lenguaje a otro. Un compilador toma como su entrada un programa escrito en su **lenguaje fuente** y produce un programa equivalente escrito en su **lenguaje objetivo**. Por lo regular, el lenguaje fuente es un **lenguaje de alto nivel**, tal como C o C++, mientras que el lenguaje objetivo es **código objeto** (también llamado en ocasiones **código de máquina**) para la máquina objetivo, es decir, código escrito en las instrucciones de máquina correspondientes a la computadora en la cual se ejecutará. Podemos visualizar este proceso de manera esquemática como sigue:

Un compilador es un programa muy complejo con un número de líneas de código que puede variar de 10,000 a 1,000,000. Escribir un programa de esta naturaleza, o incluso comprenderlo, no es una tarea fácil, y la mayoría de los científicos y profesionales de la computación nunca escribirán un compilador completo. No obstante, los compiladores se utilizan en casi todas las formas de la computación, y cualquiera que esté involucrado profesionalmente con las computadoras debería conocer la organización y el funcionamiento básicos de un compilador. Además, una tarea frecuente en las aplicaciones de las computadoras es el desarrollo de programas de interfaces e intérpretes de comandos, que son más pequeños que los compiladores pero utilizan las mismas técnicas. Por lo tanto, el conocimiento de estas técnicas tiene una aplicación práctica importante.

El propósito de este texto no sólo es el de proporcionar este conocimiento básico, sino también el de ofrecer al lector todas las herramientas necesarias, así como la experiencia práctica, para diseñar y programar un compilador real. Para conseguir esto es

necesario estudiar las técnicas teóricas, principalmente las provenientes de la teoría de los autómatas, que hacen de la construcción de compiladores una tarea manejable. Al abordar esta teoría evitaremos suponer que el lector tiene un conocimiento previo de la teoría de autómatas. En vez de eso aquí aplicaremos un punto de vista diferente al que se aplica en un texto estándar de teoría de autómatas, en el sentido de que éste está dirigido específicamente al proceso de compilación. Sin embargo, un lector que haya estudiado teoría de autómatas encontrará el material teórico bastante familiar y podrá avanzar más rápidamente a través de estas secciones. En particular, un lector que tenga buenos fundamentos en la teoría de autómatas puede saltarse las secciones 2.2, 2.3, 2.4 y 3.2 o revisarlas superficialmente. En cualquier caso, el lector debería estar familiarizado con matemáticas discretas y estructuras básicas de datos. También es esencial que conozca un poco de arquitectura de máquinas y lenguaje ensamblador, en particular para el capítulo sobre la generación de código.

El estudio de las técnicas de codificación práctica en sí mismo requiere de una cuidadosa planeación, ya que incluso con buenos fundamentos teóricos los detalles de la codificación pueden ser complejos y abrumadores. Este texto contiene una serie de ejemplos simples de construcciones de lenguaje de programación que se utilizan para elaborar el análisis de las técnicas. El lenguaje que empleamos para éste se denomina TINY. También proporcionamos (en el apéndice A) un ejemplo más extenso, que se compone de un subconjunto pequeño, pero suficientemente complejo, de C, que denominamos C-Minus, el cual es adecuado para un proyecto de clase. De manera adicional tenemos numerosos ejercicios; éstos incluyen ejercicios simples para realizar con papel y lápiz, extensiones del código en el texto y ejercicios de codificación más involucrados.

En general, existe una importante interacción entre la estructura de un compilador y el diseño del lenguaje de programación que se está compilando. En este texto sólo estudiaremos de manera incidental cuestiones de diseño de lenguajes. Existen otros textos disponibles para profundizar más en las cuestiones de diseño y conceptos de lenguajes de programación. (Véase la sección de notas y referencias al final de este capítulo.)

Comenzaremos con una breve revisión de la historia y razón de ser de los compiladores, junto con una descripción de programas relacionados con ellos. Después examinaremos la estructura, mediante un ejemplo concreto simple, de un compilador y los diversos procesos de traducción y estructuras de datos asociadas con él. Al final daremos una perspectiva general de otras cuestiones relacionadas con la estructura de compiladores, incluyendo el arranque automático de transferencia ("bootstrapping") y portabilidad, para concluir con una descripción de los principales ejemplos de lenguajes que se emplean en el resto del libro.

1.1 ¿POR QUÉ COMPILADORES? UNA BREVE HISTORIA

Con el advenimiento de la computadora con programa almacenado, iniciado por John von Neumann a finales de la década de 1940, se hizo necesario escribir secuencias de códigos, o programas, que darían como resultado que estas computadoras realizaran los cálculos deseados. Al principio estos programas se escribían en **lenguaje de máquina**: códigos numéricos que representaban las operaciones reales de la máquina que iban a efectuarse. Por ejemplo,

```
C7 06 0000 0002
```

representa la instrucción para mover el número 2 a la ubicación 0000 (en sistema hexadecimal) en los procesadores Intel 8x86 que se utilizan en las PC de IBM. Por supuesto, la escritura de tales códigos es muy tediosa y consume mucho tiempo, por lo que esta forma de codificación pronto fue reemplazada por el **lenguaje ensamblador**, en el cual las instrucciones y

las localidades de memoria son formas simbólicas dadas. Por ejemplo, la instrucción en lenguaje ensamblador

```
MOV X , 2
```

es equivalente a la instrucción de máquina anterior (suponiendo que la localidad de memoria simbólica X es 0000). Un **ensamblador** traduce los códigos simbólicos y las localidades de memoria del lenguaje ensamblador a los códigos numéricos correspondientes del lenguaje de máquina.

El lenguaje ensamblador mejoró enormemente la velocidad y exactitud con la que podían escribirse los programas, y en la actualidad todavía se encuentra en uso, en especial cuando se necesita una gran velocidad o brevedad en el código. Sin embargo, el lenguaje ensamblador tiene varios defectos: aún no es fácil de escribir y es difícil de leer y comprender. Además, el lenguaje ensamblador depende en extremo de la máquina en particular para la cual se haya escrito, de manera que el código escrito para una computadora debe volver a escribirse por completo para otra máquina. Como es evidente, el siguiente paso fundamental en la tecnología de programación fue escribir las operaciones de un programa de una manera concisa que se pareciera mucho a la notación matemática o lenguaje natural de manera que fueran independientes de cualquier máquina en particular y todavía se pudieran traducir mediante un programa para convertirlas en código ejecutable. Por ejemplo, el anterior código del lenguaje ensamblador se puede escribir de manera concisa e independiente de una máquina en particular como

```
X = 2
```

Al principio se temía que esto no fuera posible, o que si lo fuera, el código objeto sería tan poco eficiente que resultaría inútil.

El desarrollo del lenguaje FORTRAN y su compilador, llevado a cabo por un equipo en IBM dirigido por John Backus entre 1954 y 1957 demostró que estos temores eran infundados. No obstante, el éxito de este proyecto se debió sólo a un gran esfuerzo, ya que la mayoría de los procesos involucrados en la traducción de lenguajes de programación no fueron bien comprendidos en el momento.

Más o menos al mismo tiempo en que el primer compilador se estaba desarrollando, Noam Chomsky comenzó a estudiar la estructura del lenguaje natural. Sus hallazgos finalmente hicieron que la construcción de compiladores se volviera mucho más fácil e incluso pudiera ser automatizado hasta cierto punto. Los estudios de Chomsky condujeron a la clasificación de los lenguajes de acuerdo con la complejidad de sus **gramáticas** (las reglas que especifican su estructura) y la potencia de los algoritmos necesarios para reconocerlas. La **jerarquía de Chomsky**, como ahora se le conoce, se compone de cuatro niveles de gramáticas, denominadas gramáticas tipo 0, tipo 1, tipo 2 y tipo 3, cada una de las cuales es una especialización de su predecesora. Las gramáticas de tipo 2, o **gramáticas libres de contexto**, demostraron ser las más útiles para lenguajes de programación, en la actualidad son la manera estándar para representar la estructura de los lenguajes de programación. El estudio del **problema del análisis sintáctico** (la determinación de algoritmos eficientes para el reconocimiento de lenguajes libres de contexto) se llevó a cabo en las décadas de los 60 y 70 y condujo a una solución muy completa de este problema, que en la actualidad se ha vuelto una parte estándar de la teoría de compiladores. Los lenguajes libres de contexto y los algoritmos de análisis sintáctico se estudian en los capítulos 3, 4 y 5.

Los **autómatas finitos** y las **expresiones regulares**, que corresponden a las gramáticas de tipo 3 de Chomsky, se encuentran estrechamente relacionados con las gramáticas libres de contexto. Al comenzar a generarse casi al mismo tiempo que el trabajo de Chomsky, su estudio condujo a métodos simbólicos para expresar la estructura de las palabras, o tokens, de un lenguaje de programación. En el capítulo 2 se analizan los autómatas finitos y las expresiones regulares.

Mucho más complejo ha sido el desarrollo de métodos para la generación de código objeto eficaz, que comenzó con los primeros compiladores y continúa hasta nuestros días. Estas técnicas suelen denominarse, incorrectamente, **técnicas de optimización**, pero en realidad deberían llamarse **técnicas de mejoramiento de código**, puesto que casi nunca producen un código objeto verdaderamente óptimo y sólo mejoran su eficacia. En el capítulo 8 se describen los fundamentos de estas técnicas.

A medida que el problema del análisis sintáctico se comprendía bien, se dedicó mucho trabajo a desarrollar programas que automatizarían esta parte del desarrollo de compiladores. Estos programas originalmente se llamaron compiladores de compilador, pero se hace referencia a ellos de manera más acertada como **generadores de analizadores sintácticos**, ya que automatizan sólo una parte del proceso de compilación. El más conocido de estos programas es Yacc (por las siglas del término en inglés "yet another compiler-compiler", "otro compilador de compilador más"), el cual fue escrito por Steve Johnson en 1975 para el sistema Unix y se estudiará en el capítulo 5. De manera similar, el estudio de los autómatas finitos condujo al desarrollo de otra herramienta denominada **generador de rastreadores** (o **generador de analizadores léxicos**), cuyo representante más conocido es Lex (desarrollado para el sistema Unix por Mike Lesk más o menos al mismo tiempo que Yacc). Lex se estudiará en el capítulo 2.

A fines de los 70 y principios de los 80 diversos proyectos se enfocaron en automatizar la generación de otras partes de un compilador, incluyendo la generación del código. Estos intentos han tenido menos éxito, posiblemente debido a la naturaleza compleja de las operaciones y a nuestra poca comprensión de las mismas. No las estudiaremos con detalle en este texto.

Los avances más recientes en diseño de compiladores han incluido lo siguiente. En primer lugar, los compiladores han incluido la aplicación de algoritmos más sofisticados para inferir y/o simplificar la información contenida en un programa, y éstos han ido de la mano con el desarrollo de lenguajes de programación más sofisticados que permiten esta clase de análisis. Un ejemplo típico de éstos es el algoritmo de unificación de verificación de tipo de Hindley-Milner, que se utiliza en la compilación de lenguajes funcionales. En segundo lugar, los compiladores se han vuelto cada vez más una parte de un **ambiente de desarrollo interactivo**, o IDE (por sus siglas en inglés de "interactive development environment"), basado en ventanas, que incluye editores, ligadores, depuradores y administradores de proyectos. Hasta ahora ha habido escasa estandarización de estos IDE, pero el desarrollo de ambientes de ventanas estándar nos está conduciendo en esa dirección. El estudio de estos temas rebasa el alcance de este texto (pero véase la siguiente sección para una breve descripción de algunos de los componentes de un IDE). Para sugerencias de literatura relacionada véase la sección de notas y referencias al final del capítulo. Sin embargo, a pesar de la cantidad de actividades de investigación en los últimos años, los fundamentos del diseño de compiladores no han cambiado demasiado en los últimos veinte años, y se han convertido cada vez más en una parte del núcleo estándar en los currículos para las ciencias de la computación.

1.2 PROGRAMAS RELACIONADOS CON LOS COMPILADORES

En esta sección describiremos brevemente otros programas que están relacionados, o que se utilizan, con los compiladores y que con frecuencia vienen junto con ellos en un ambiente de desarrollo de lenguaje completo. (Ya mencionamos algunos.)

INTÉRPRETES

Un intérprete es un traductor de lenguaje, igual que un compilador, pero difiere de éste en que ejecuta el programa fuente inmediatamente, en vez de generar un código objeto que se ejecuta después de que se completa la traducción. En principio, cualquier lenguaje de programación se puede interpretar o compilar, pero se puede preferir un intérprete a un compilador dependiendo del lenguaje que se esté usando y de la situación en la cual

se presenta la traducción. Por ejemplo, BASIC es un lenguaje que por lo regular es interpretado en vez de compilado. De manera similar, los lenguajes funcionales, como LISP, tienden a ser interpretados. Los intérpretes también se utilizan con frecuencia en situaciones relacionadas con la enseñanza o con el desarrollo de software, donde los programas son probablemente traducidos y vueltos a traducir muchas veces. Por otra parte, es preferible usar un compilador si lo que importa es la velocidad de ejecución, ya que el código objeto compilado es siempre más rápido que el código fuente interpretado, en ocasiones hasta por un factor de 10 o más. No obstante, los intérpretes comparten muchas de sus operaciones con los compiladores, y ahí pueden incluso ser traductores híbridos, de manera que quedan en alguna parte entre los intérpretes y los compiladores. Analizaremos a los intérpretes de manera ocasional, pero en este texto nos enfocaremos principalmente en la compilación.

ENSAMBLADORES

Un ensamblador es un traductor para el lenguaje ensamblador de una computadora en particular. Como ya lo advertimos, el lenguaje ensamblador es una forma simbólica del lenguaje de máquina de la computadora y es particularmente fácil de traducir. En ocasiones un compilador generará lenguaje ensamblador como su lenguaje objetivo y dependerá entonces de un ensamblador para terminar la traducción a código objeto.

LIGADORES

Tanto los compiladores como los ensambladores a menudo dependen de un programa conocido como ligador, el cual recopila el código que se compila o ensambla por separado en diferentes archivos objeto, a un archivo que es directamente ejecutable. En este sentido, puede hacerse una distinción entre código objeto (código de máquina que todavía no se ha ligado) y código de máquina ejecutable. Un ligador también conecta un programa objeto con el código de funciones de librerías estándar, así como con recursos suministrados por el sistema operativo de la computadora, tales como asignadores de memoria y dispositivos de entrada y salida. Es interesante advertir que los ligadores ahora realizan la tarea que originalmente era una de las principales actividades de un compilador (de aquí el uso de la palabra *compilador*: construir mediante la recopilación o *compilación* de fuentes diferentes). En este texto no estudiaremos el proceso de ligado porque depende demasiado de los detalles del sistema operativo y del procesador. Tampoco haremos siempre una distinción clara entre el código objeto no ligado y el código ejecutable, porque esta distinción no tiene importancia para nuestro estudio de las técnicas de compilación.

CARGADORES

Con frecuencia un compilador, ensamblador o ligador producirá un código que todavía no está completamente organizado y listo para ejecutarse, pero cuyas principales referencias de memoria se hacen relativas a una localidad de arranque indeterminada que puede estar en cualquier sitio de la memoria. Se dice que tal código es **relocalizable** y un cargador resolverá todas las direcciones relocalizables relativas a una dirección base, o de inicio, dada. El uso de un cargador hace más flexible el código ejecutable, pero el proceso de carga con frecuencia ocurre en segundo plano (como parte del entorno operacional) o conjuntamente con el ligado. Rara vez un cargador es en realidad un programa por separado.

PREPROCESADORES

Un preprocesador es un programa separado que es invocado por el compilador antes de que comience la traducción real. Un preprocesador de este tipo puede eliminar los comentarios, incluir otros archivos y ejecutar sustituciones de **macro** (una macro es una descripción abreviada de una secuencia repetida de texto). Los preprocesadores pueden ser requeridos por el lenguaje (como en C) o pueden ser agregados posteriores que proporcionen facilidades adicionales (como el preprocesador Ratfor para FORTRAN).

EDITORES

Los compiladores por lo regular aceptan programas fuente escritos utilizando cualquier editor que pueda producir un archivo estándar, tal como un archivo ASCII. Más recientemente, los compiladores han sido integrados junto con editores y otros programas en un ambiente de desarrollo interactivo o IDE. En un caso así, un editor, mientras que aún produce archivos estándar, puede ser orientado hacia el formato o estructura del lenguaje de programación en cuestión. Tales editores se denominan **basados en estructura** y ya incluyen algunas de las operaciones de un compilador, de manera que, por ejemplo, pueda informarse al programador de los errores a medida que el programa se vaya escribiendo en lugar de hacerlo cuando está compilado. El compilador y sus programas acompañantes también pueden llamarse desde el editor, de modo que el programador pueda ejecutar el programa sin tener que abandonar el editor.

DEPURADORES

Un depurador es un programa que puede utilizarse para determinar los errores de ejecución en un programa compilado. A menudo está integrado con un compilador en un IDE. La ejecución de un programa con un depurador se diferencia de la ejecución directa en que el depurador se mantiene al tanto de la mayoría o la totalidad de la información sobre el código fuente, tal como los números de línea y los nombres de las variables y procedimientos. También puede detener la ejecución en ubicaciones previamente especificadas denominadas **puntos de ruptura**, además de proporcionar información de cuáles funciones se han invocado y cuáles son los valores actuales de las variables. Para efectuar estas funciones el compilador debe suministrar al depurador la información simbólica apropiada, lo cual en ocasiones puede ser difícil, en especial en un compilador que intente optimizar el código objeto. De este modo, la depuración se convierte en una cuestión de compilación, la que, sin embargo, rebasa el alcance de este libro.

PERFILADORES

Un perfilador es un programa que recolecta estadísticas sobre el comportamiento de un programa objeto durante la ejecución. Las estadísticas típicas que pueden ser de interés para el programador son el número de veces que se llama cada procedimiento y el porcentaje de tiempo de ejecución que se ocupa en cada uno de ellos. Tales estadísticas pueden ser muy útiles para ayudar al programador a mejorar la velocidad de ejecución del programa. A veces el compilador utilizará incluso la salida del perfilador para mejorar de manera automática el código objeto sin la intervención del programador.

ADMINISTRADORES DE PROYECTO

Los modernos proyectos de software por lo general son tan grandes que tienen que ser emprendidos por grupos de programadores en lugar de por un solo programador. En tales casos es importante que los archivos que se están trabajando por personas distintas se encuentren coordinados, y éste es el trabajo de un programa de administración de proyectos. Por ejemplo, un administrador de proyecto debería coordinar la mezcla de diferentes versiones del mismo archivo producido por programadores diferentes. También debería mantener una historia de las modificaciones para cada uno de los grupos de archivos, de modo que puedan mantenerse versiones coherentes de un programa en desarrollo (esto es algo que también puede ser útil en un proyecto que lleva a cabo un solo programador). Un administrador de proyecto puede escribirse en una forma independiente del lenguaje, pero cuando se integra junto con un compilador, puede mantener información acerca del compilador específico y las operaciones de ligado necesarias para construir un programa ejecutable completo. Dos programas populares de administración de proyectos en sistemas Unix son **sccs** y **rcs** (**source code control system**, "sistema de control para código fuente") y (**revision control system**, "sistema de control para revisión").

1.3 PROCESO DE TRADUCCIÓN

Un compilador se compone internamente de varias etapas, o **fases**, que realizan distintas operaciones lógicas. Es útil pensar en estas fases como en piezas separadas dentro del compilador, y pueden en realidad escribirse como operaciones codificadas separadamente aunque en la práctica a menudo se integren juntas. Las fases de un compilador se ilustran en la figura 1.1, junto con los tres componentes auxiliares que interactúan con alguna de ellas o

Figura 1.1
Fases de un compilador

con todas: la tabla de literales, la tabla de símbolos y el manejador de errores. Aquí describiremos brevemente cada una de las fases, las cuales se estudiarán con más detalle en los capítulos siguientes. (Las tablas de literales y de símbolos se analizarán más ampliamente en la siguiente sección y el manejador de errores en la sección 1.5.)

ANALIZADOR LÉXICO O RASTREADOR (*SCANNER*)

Esta fase del compilador efectúa la lectura real del programa fuente, el cual generalmente está en la forma de un flujo de caracteres. El rastreador realiza lo que se conoce como **análisis léxico**: recolecta secuencias de caracteres en unidades significativas denominadas **tokens**, las cuales son como las palabras de un lenguaje natural, como el inglés. De este modo, se puede imaginar que un rastreador realiza una función similar al deletreo.

Como ejemplo, considere la siguiente línea de código, que podría ser parte de un programa en C:

```
a[index] = 4 + 2
```

Este código contiene 12 caracteres diferentes de un espacio en blanco pero sólo 8 tokens:

`a`	identificador
`[`	corchete izquierdo
`index`	identificador
`]`	corchete derecho
`=`	asignación
`4`	número
`+`	signo más
`2`	número

Cada token se compone de uno o más caracteres que se reúnen en una unidad antes de que ocurra un procesamiento adicional.

Un analizador léxico puede realizar otras funciones junto con la de reconocimiento de tokens. Por ejemplo, puede introducir identificadores en la tabla de símbolos, y puede introducir **literales** en la tabla de literales (las literales incluyen constantes numéricas tales como 3.1415926535 y cadenas de texto entrecomilladas como "¡Hola, mundo!").

ANALIZADOR SINTÁCTICO (*PARSER*)

El analizador sintáctico recibe el código fuente en la forma de tokens proveniente del analizador léxico y realiza el **análisis sintáctico**, que determina la estructura del programa. Esto es semejante a realizar el análisis gramatical sobre una frase en un lenguaje natural. El análisis sintáctico determina los elementos estructurales del programa y sus relaciones. Los resultados del análisis sintáctico por lo regular se representan como un **árbol de análisis gramatical** o un **árbol sintáctico**.

Como ejemplo, consideremos otra vez la línea de código en C que ya habíamos dado. Representa un elemento estructural denominado expresión, la cual es una expresión de asignación compuesta de una expresión con subíndice a la izquierda y una expresión aritmética entera a la derecha. Esta estructura se puede representar como un árbol de análisis gramatical de la forma siguiente:

Advierta que los nodos internos del árbol de análisis gramatical están etiquetados con los nombres de las estructuras que representan y que las hojas del árbol representan la secuencia de tokens de la entrada. (Los nombres de las estructuras están escritos en un tipo de letra diferente para distinguirlos de los tokens.)

Un árbol de análisis gramatical es un auxiliar útil para visualizar la sintaxis de un programa o de un elemento de programa, pero no es eficaz en su representación de esa estructura. Los analizadores sintácticos tienden a generar un árbol sintáctico en su lugar, el cual es una condensación de la información contenida en el árbol de análisis gramatical. (En ocasiones los árboles sintácticos se denominan **árboles sintácticos abstractos** porque representan una abstracción adicional de los árboles de análisis gramatical.) Un árbol sintáctico abstracto para nuestro ejemplo de una expresión de asignación en C es el siguiente:

Advierta que en el árbol sintáctico muchos de los nodos han desaparecido (incluyendo los nodos de tokens). Por ejemplo, si sabemos que una expresión es una operación de subíndice, entonces ya no será necesario mantener los paréntesis cuadrados `[` y `]` que representan esta operación en la entrada original.

ANALIZADOR SEMÁNTICO

La semántica de un programa es su "significado", en oposición a su sintaxis, o estructura. La semántica de un programa determina su comportamiento durante el tiempo de ejecución, pero la mayoría de los lenguajes de programación tienen características que se pueden determinar antes de la ejecución e incluso no se pueden expresar de manera adecuada como sintaxis y analizarse mediante el analizador sintáctico. Se hace referencia

a tales características como **semántica estática**, y el análisis de tal semántica es la tarea del analizador semántico. (La semántica "dinámica" de un programa, es decir, aquellas propiedades del programa que solamente se pueden determinar al ejecutarlo, no se pueden determinar mediante un compilador porque éste no ejecuta el programa.) Las características típicas de la semántica estática en los lenguajes de programación comunes incluyen las declaraciones y la verificación de tipos. Las partes extra de la información (como los tipos de datos) que se calculan mediante el analizador semántico se llaman **atributos** y con frecuencia se agregan al árbol como anotaciones o "decoraciones". (Los atributos también se pueden introducir en la tabla de símbolos.)

En nuestro ejemplo de ejecución de la expresión en C

```
a[index] = 4 + 2
```

la información de tipo específica que se tendría que obtener antes del análisis de esta línea sería que **a** sea un arreglo de valores enteros con subíndices proveniente de un subintervalo de los enteros y que **index** sea una variable entera. Entonces el analizador semántico registraría el árbol sintáctico con los tipos de todas las subexpresiones y posteriormente verificaría que la asignación tuviera sentido para estos tipos, y declararía un error de correspondencia de tipo si no fuera así. En nuestro ejemplo todos los tipos tienen sentido, y el resultado del análisis semántico en el árbol sintáctico podría representarse por el siguiente árbol con anotaciones:

OPTIMIZADOR DE CÓDIGO FUENTE

Los compiladores a menudo incluyen varias etapas para el mejoramiento, u optimización, del código. El punto más anticipado en el que la mayoría de las etapas de optimización se pueden realizar es precisamente después del análisis semántico, y puede haber posibilidades para el mejoramiento del código que dependerán sólo del código fuente. Indicamos esta posibilidad al proporcionar esta operación como una fase por separado en el proceso de compilación. Los compiladores individuales muestran una amplia variación no sólo en los tipos de optimizaciones realizadas sino también en la colocación de las fases de optimización.

En nuestro ejemplo incluimos una oportunidad para la optimización a nivel de fuente; a saber, la expresión **4 + 2** se puede calcular previamente por el compilador para dar como resultado **6**. (Esta optimización particular es conocida como **incorporación de constantes**.) Claro que existen posibilidades mucho más complejas (algunas de las cuales se mencionarán en el capítulo 8). En nuestro ejemplo esta optimización se puede realizar de manera directa sobre el árbol sintáctico (con anotaciones) al colapsar el subárbol secundario de la derecha del nodo raíz a su valor constante:

Muchas optimizaciones se pueden efectuar directamente sobre el árbol, pero en varios casos es más fácil optimizar una forma linealizada del árbol que esté más cercana al código ensamblador. Existen muchas variedades diferentes de tal código, pero una elección estándar es el **código en tres direcciones**, denominado así porque contiene las direcciones de (y hasta) tres localidades en memoria. Otra selección popular es el **código P**, el cual se ha utilizado en muchos compiladores de Pascal.

En nuestro ejemplo el código en tres direcciones para la expresión original en C podría parecerse a esto:

```
t = 4 + 2
a[index] = t
```

(Advierta el uso de una variable temporal adicional `t` para almacenar el resultado intermedio de la suma.) Ahora el optimizador mejoraría este código en dos etapas, en primer lugar calculando el resultado de la suma

```
t = 6
a[index] = t
```

y después reemplazando a `t` por su valor para obtener la sentencia en tres direcciones

```
a[index] = 6
```

En la figura 1.1 indicamos la posibilidad de que el optimizador del código fuente pueda emplear código en tres direcciones al referirnos a su salida como a un **código intermedio**. Históricamente, el código intermedio ha hecho referencia a una forma de representación de código intermedia entre el código fuente y el código objeto, tal como el código de tres direcciones o una representación lineal semejante. Sin embargo, también puede referirse de manera más general a *cualquier* representación interna para el código fuente utilizado por el compilador. En este sentido, también se puede hacer referencia al árbol sintáctico como un código intermedio, y efectivamente el optimizador del código fuente podría continuar el uso de esta representación en su salida. En ocasiones este sentido más general se indica al hacer referencia al código intermedio como **representación intermedia**, o **RI**.

GENERADOR DE CÓDIGO

El generador de código toma el código intermedio o RI y genera el código para la máquina objetivo. En este texto escribiremos el código objetivo en la forma de lenguaje ensamblador para facilitar su comprensión, aunque la mayoría de los compiladores

generan el código objeto de manera directa. Es en esta fase de la compilación en la que las propiedades de la máquina objetivo se convierten en el factor principal. No sólo es necesario emplear instrucciones que existan en la máquina objetivo, sino que las decisiones respecto a la representación de los datos desempeñarán ahora también un papel principal, tal como cuántos bytes o palabras de memoria ocuparán las variables de tipos de datos enteros y de punto flotante.

En nuestro ejemplo debemos decidir ahora cuántos enteros se almacenarán para generar el código para la indización del arreglo. Por ejemplo, una posible secuencia de código muestra para la expresión dada podría ser (en un hipotético lenguaje ensamblador)

```
MOV   R0, index    ;; valor de index -> R0
MUL   R0, 2        ;; doble valor en R0
MOV   R1, &a       ;; dirección de a -> R1
ADD   R1, R0       ;; sumar R0 a R1
MOV   *R1, 6       ;; constante 6 -> dirección en R1
```

En este código utilizamos una convención propia de C para direccionar modos, de manera que `&a` es la dirección de `a` (es decir, la dirección base del arreglo) y que `*R1` significa direccionamiento indirecto de registro (de modo que la última instrucción almacena el valor 6 en la dirección contenida en R1). En este código también partimos del supuesto de que la máquina realiza direccionamiento de byte y que los enteros ocupan dos bytes de memoria (de aquí el uso del 2 como el factor de multiplicación en la segunda instrucción).

OPTIMIZADOR DE CÓDIGO OBJETIVO

En esta fase el compilador intenta mejorar el código objetivo generado por el generador de código. Dichas mejoras incluyen la selección de modos de direccionamiento para mejorar el rendimiento, reemplazando las instrucciones lentas por otras rápidas, y eliminando las operaciones redundantes o innecesarias.

En el código objetivo de muestra dado es posible hacer varias mejoras. Una de ellas es utilizar una instrucción de desplazamiento para reemplazar la multiplicación en la segunda instrucción (la cual por lo regular es costosa en términos del tiempo de ejecución). Otra es emplear un modo de direccionamiento más poderoso, tal como el direccionamiento indizado para realizar el almacenamiento en el arreglo. Con estas dos optimizaciones nuestro código objetivo se convierte en

```
MOV  R0, index     ;; valor de index -> R0
SHL  R0            ;; doble valor en R0
MOV  &a[R0], 6     ;; constante 6 -> dirección a + R0
```

Esto completa nuestra breve descripción de las fases de un compilador. Queremos enfatizar que esta descripción sólo es esquemática y no necesariamente representa la organización real de un compilador trabajando. En su lugar, los compiladores muestran una amplia variación en sus detalles de organización. No obstante, las fases que describimos están presentes de alguna forma en casi todos los compiladores.

También analizamos sólo de manera superficial las estructuras de datos necesarias para mantener la información necesaria en cada fase, tales como el árbol sintáctico, el código intermedio (suponiendo que éstos no sean iguales), la tabla de literales y la tabla de símbolos. Dedicamos la siguiente sección a una breve perspectiva general de las estructuras de datos principales en un compilador.

1.4 ESTRUCTURAS DE DATOS PRINCIPALES EN UN COMPILADOR

La interacción entre los algoritmos utilizados por las fases de un compilador y las estructuras de datos que soportan estas fases es, naturalmente, muy fuerte. El escritor del compilador se esfuerza por implementar estos algoritmos de una manera tan eficaz como sea posible, sin aumentar demasiado la complejidad. De manera ideal, un compilador debería poder compilar un programa en un tiempo proporcional al tamaño del programa, es decir, en $O(n)$ tiempo, donde n es una medida del tamaño del programa (por lo general el número de caracteres). En esta sección señalaremos algunas de las principales estructuras de datos que son necesarias para las fases como parte de su operación y que sirven para comunicar la información entre las fases.

TOKENS

Cuando un rastreador o analizador léxico reúne los caracteres en un token, generalmente representa el token de manera simbólica, es decir, como un valor de un tipo de datos enumerado que representa el conjunto de tokens del lenguaje fuente. En ocasiones también es necesario mantener la cadena de caracteres misma u otra información derivada de ella, tal como el nombre asociado con un token identificador o el valor de un token de número. En la mayoría de los lenguajes el analizador léxico sólo necesita generar un token a la vez (esto se conoce como **búsqueda de símbolo simple**). En este caso se puede utilizar una variable global simple para mantener la información del token. En otros casos (cuyo ejemplo más notable es FORTRAN), puede ser necesario un arreglo de tokens.

ÁRBOL SINTÁCTICO

Si el analizador sintáctico genera un árbol sintáctico, por lo regular se construye como una estructura estándar basada en un apuntador que se asigna de manera dinámica a medida que se efectúa el análisis sintáctico. El árbol entero puede entonces conservarse como una variable simple que apunta al nodo raíz. Cada nodo en la estructura es un registro cuyos campos representan la información recolectada tanto por el analizador sintáctico como, posteriormente, por el analizador semántico. Por ejemplo, el tipo de datos de una expresión puede conservarse como un campo en el nodo del árbol sintáctico para la expresión. En ocasiones, para ahorrar espacio, estos campos se asignan de manera dinámica, o se almacenan en otras estructuras de datos, tales como la tabla de símbolos, que permiten una asignación y desasignación selectivas. En realidad, cada nodo de árbol sintáctico por sí mismo puede requerir de atributos diferentes para ser almacenado, de acuerdo con la clase de estructura del lenguaje que represente (por ejemplo, un nodo de expresión tiene requerimientos diferentes de los de un nodo de sentencia o un nodo de declaración). En este caso, cada nodo en el árbol sintáctico puede estar representado por un registro variable, con cada clase de nodo conteniendo solamente la información necesaria para ese caso.

TABLA DE SÍMBOLOS

Esta estructura de datos mantiene la información asociada con los identificadores: funciones, variables, constantes y tipos de datos. La tabla de símbolos interactúa con casi todas las fases del compilador: el rastreador o analizador léxico, el analizador sintáctico o el analizador semántico puede introducir identificadores dentro de la tabla; el analizador semántico agregará tipos de datos y otra información; y las fases de optimización y generación de código utilizarán la información proporcionada por la tabla de símbolos para efectuar selecciones apropiadas de código objeto. Puesto que la tabla de símbolos tendrá solicitudes de acceso con tanta frecuencia, las operaciones de inserción, eliminación y acceso necesitan ser eficientes, preferiblemente operaciones de tiempo constante. Una estructura de datos estándar para este propósito es la tabla de dispersión o de cálculo de dirección, aunque también se pueden utilizar diversas estructuras de árbol. En ocasiones se utilizan varias tablas y se mantienen en una lista o pila.

TABLA DE LITERALES

La búsqueda y la inserción rápida son esenciales también para la tabla de literales, la cual almacena constantes y cadenas utilizadas en el programa. Sin embargo, una tabla de literales necesita impedir las eliminaciones porque sus datos se aplican globalmente al programa y una constante o cadena aparecerá sólo una vez en esta tabla. La tabla de literales es importante en la reducción del tamaño de un programa en la memoria al permitir la reutilización de constantes y cadenas. También es necesaria para que el generador de código construya direcciones simbólicas para las literales y para introducir definiciones de datos en el archivo de código objetivo.

CÓDIGO INTERMEDIO

De acuerdo con la clase de código intermedio (por ejemplo, código de tres direcciones y código P) y de las clases de optimizaciones realizadas, este código puede conservarse como un arreglo de cadenas de texto, un archivo de texto temporal o bien una lista de estructuras ligadas. En los compiladores que realizan optimizaciones complejas debe ponerse particular atención a la selección de representaciones que permitan una fácil reorganización.

ARCHIVOS TEMPORALES

Al principio las computadoras no poseían suficiente memoria para guardar un programa completo durante la compilación. Este problema se resolvió mediante el uso de archivos temporales para mantener los productos de los pasos intermedios durante la traducción o bien al compilar "al vuelo", es decir, manteniendo sólo la información suficiente de las partes anteriores del programa fuente que permita proceder a la traducción. Las limitantes de memoria son ahora un problema mucho menor, y es posible requerir que una unidad de compilación entera se mantenga en la memoria, en especial si se dispone de la compilación por separado en el lenguaje. Con todo, los compiladores ocasionalmente encuentran útil generar archivos intermedios durante alguna de las etapas del procesamiento. Algo típico de éstos es la necesidad de direcciones de **corrección hacia atrás** durante la generación del código. Por ejemplo, cuando se traduce una sentencia condicional tal como

```
if x = 0 then ... else ...
```

debe generarse un salto desde la prueba para la parte bicondicional "else" antes de conocer la ubicación del código para el "else":

```
CMP X,0
JNE NEXT ;; ubicación del NEXT aún no conocida
<código para la parte "then">
NEXT:
<código para la parte "else">
```

Por lo regular debe dejarse un espacio en blanco para el valor de **NEXT**, el cual se llena una vez que se logra conocer el valor. Esto se consigue fácilmente con el uso de un archivo temporal.

1.5 OTRAS CUESTIONES REFERENTES A LA ESTRUCTURA DEL COMPILADOR

La estructura de un compilador se puede ver desde muchos ángulos distintos. En la sección 1.3 describimos sus fases, las cuales representan la estructura lógica de un compilador. Otros puntos de vista son posibles: la estructura física del compilador, la secuenciación de las operaciones, y así sucesivamente. La persona que escribe el compilador debería estar

familiarizada con tantos puntos de vista de la estructura del compilador como sea posible, ya que la estructura del compilador será determinante para su confiabilidad, eficacia, utilidad y mantenimiento. En esta sección consideraremos otros aspectos de la estructura del compilador y señalaremos cómo se aplica cada punto de vista.

ANÁLISIS Y SÍNTESIS

En esta perspectiva las operaciones del compilador que analizan el programa fuente para calcular sus propiedades se clasifican como la parte de **análisis** del compilador, mientras que las operaciones involucradas en la producción del código traducido se conocen como la parte de **síntesis** del compilador. Como es natural, el análisis léxico, el análisis sintáctico y el análisis semántico pertenecen a la parte de análisis, mientras que la generación del código es la síntesis. Las etapas de optimización pueden involucrar tanto análisis como síntesis. El análisis tiende a ser más matemático y a comprenderse mejor, mientras que la síntesis requiere de técnicas más especializadas. Por consiguiente, es útil separar las etapas del análisis de las etapas de la síntesis, de modo que cada una se pueda modificar de manera independiente respecto a la otra.

ETAPA INICIAL Y ETAPA FINAL

Esta perspectiva considera al compilador separado en aquellas funciones que dependen sólo del lenguaje fuente (la **etapa inicial**) y aquellas operaciones que dependen únicamente del lenguaje objetivo (la **etapa final**). Esto es similar a la división en análisis y síntesis: el analizador léxico, el analizador sintáctico y el analizador semántico son parte de la etapa inicial, mientras que el generador de código es parte de la etapa final. Sin embargo, algo del análisis de optimización puede ser dependiente del objetivo y, por lo tanto, parte de la etapa final, mientras que la síntesis del código intermedio es a menudo independiente del objetivo y, por consiguiente, parte de la etapa inicial. De manera ideal, el compilador estaría estrictamente dividido en estas dos secciones, con la representación intermedia como el medio de comunicación entre ellas:

Esta estructura es especialmente importante para la **portabilidad** del compilador, en la cual el compilador está diseñado con un enfoque hacia la modificación, ya sea del código fuente (lo que involucra volver a escribir la etapa inicial) o del código objetivo (lo que implica reescribir la etapa final). En la práctica esto ha probado ser difícil de conseguir, y los denominados compiladores portátiles todavía tienden a poseer características que dependen tanto del lenguaje fuente como del lenguaje objetivo. Esto puede, en parte, ser culpa de los cambios rápidos y fundamentales tanto en los lenguajes de programación como en las arquitecturas de las máquinas, pero también es difícil retener de manera eficaz a toda la información que uno pudiera necesitar al cambiar a un nuevo lenguaje objetivo o al crear las estructuras de datos adecuadamente generales para permitir un cambio a un nuevo lenguaje fuente. No obstante, una tentativa consistente para separar las etapas inicial y final redundará en beneficios para una portabilidad más fácil.

PASADAS

Un compilador a menudo encuentra conveniente procesar todo el programa fuente varias veces antes de generar el código. Estas repeticiones son conocidas como **pasadas**. Después del paso inicial, donde se construye un árbol sintáctico o un código intermedio a partir de la fuente, una pasada consiste en procesar la representación intermedia,

agregando información a ella, alterando su estructura o produciendo una representación diferente. Las pasadas pueden corresponder o no a las fases, a menudo una pasada consistirá de varias etapas. En realidad, dependiendo del lenguaje, un compilador puede ser de **una pasada**, en el que todas las fases se presentan durante un paso único. Esto resulta en una compilación eficaz pero también en (por lo regular) un código objetivo menos eficiente. Tanto Pascal como C son lenguajes que permiten la compilación de una pasada. (Modula-2 es un lenguaje cuya estructura requiere que un compilador tenga por lo menos dos pasadas.) La mayoría de los compiladores con optimizaciones utilizan más de una pasada; por lo regular se emplea una pasada para análisis léxico y sintáctico, otra pasada para análisis semántico y optimización a nivel del fuente, y una tercera pasada para generación de código y optimización a nivel del objetivo. Los compiladores fuertemente optimizadores pueden emplear incluso más pasadas: cinco, seis o incluso ocho no son algo fuera de lo común.

DEFINICIÓN DE LENGUAJE Y COMPILADORES

Advertimos en la sección 1.1 que las estructuras léxicas y sintácticas de un lenguaje de programación por lo regular son especificadas en términos formales y utilizan expresiones regulares y gramáticas libres de contexto. Sin embargo, la semántica de un lenguaje de programación todavía es comúnmente especificada utilizando descripciones en inglés (u otro lenguaje natural). Estas descripciones (junto con la estructura sintáctica y léxica formales) generalmente son recopiladas en un **manual de referencia del lenguaje**, o **definición de lenguaje**. Con un nuevo lenguaje, una definición de lenguaje y un compilador con frecuencia son desarrollados de manera simultánea, puesto que las técnicas disponibles para el escritor de compiladores pueden tener un impacto fundamental sobre la definición del lenguaje. Similarmente, la manera en la que se define un lenguaje tendrá un impacto fundamental sobre las técnicas que son necesarias para construir el compilador.

Una situación más común para el escritor de compiladores es que el lenguaje que se está implementando es bien conocido y tiene una definición de lenguaje existente. En ocasiones esta definición de lenguaje ha alcanzado el nivel de un **lenguaje estándar** que ha sido aprobado por alguna de las organizaciones de estandarización oficiales, tal como la ANSI (American National Standards Institute) o ISO (International Organization for Standardization). Por ejemplo, FORTRAN, Pascal y C tienen estándares ANSI. Ada tiene un estándar aprobado por el gobierno de Estados Unidos. En este caso, el escritor de compiladores debe interpretar la definición de lenguaje e implementar un compilador acorde con la definición de lenguaje. Esto a menudo no es una tarea fácil, pero en ocasiones se hace más fácil por la existencia de un conjunto de programas de prueba estándar (una **suite** o **serie de pruebas**) contra el cual un compilador puede ser contrastado (una serie de pruebas así existe para Ada). El lenguaje de ejemplo TINY empleado en el texto tiene su estructura léxica, sintáctica y semántica especificadas en las secciones 2.5, 3.7 y 6.5, respectivamente. El apéndice A contiene un manual de referencia del lenguaje mínimo para el lenguaje de proyecto de compilador C-Minus.

Ocasionalmente, un lenguaje tendrá su semántica dada mediante una **definición formal** en términos matemáticos. Varios métodos actualmente en uso son empleados para esto, y ningún método ha conseguido el nivel de un estándar, aunque la denominada **semántica denotacional** se ha convertido en uno de los métodos más comunes, especialmente en la comunidad de programación funcional. Cuando existe una definición formal para un lenguaje, entonces (en teoría) es posible dar una prueba matemática de que un compilador se aviene a la definición. Sin embargo, hacer esto es tan difícil que casi nunca se hace. En cualquier caso, las técnicas para hacerlo así rebasan el alcance de este texto, y las técnicas de semántica formal no se estudiarán aquí.

Un aspecto de la construcción de compiladores que es particularmente afectado por la definición de lenguaje es la estructura y comportamiento del ambiente de ejecución. Los ambientes de ejecución se estudian con detalle en el capítulo 7. Sin embargo, vale la pena notar aquí que la estructura de datos permitida en un lenguaje de programación, y particularmente las clases de llamadas de funciones y valores devueltos permitidos, tienen un efecto decisivo sobre la complejidad del sistema de ejecución. En particular, los tres tipos básicos de ambientes de ejecución, en orden creciente de complejidad, son como se describe a continuación:

En primer lugar, FORTRAN77, sin apuntadores o asignación dinámica y sin llamadas a funciones recursivas, permite un ambiente de ejecución completamente estático, donde toda la asignación de memoria se hace antes de la ejecución. Esto hace la labor de asignación particularmente fácil para la persona que escribe el compilador, en la medida que no necesita generarse código para mantener el ambiente. En segundo lugar, Pascal, C y otros lenguajes provenientes del lenguaje Algol permiten una forma limitada de asignación dinámica y llamadas de funciones recursivas y requieren un ambiente de ejecución "semidinámico" o basado en pilas con una estructura dinámica adicional conocida como *heap*, desde el cual el programador puede organizar la asignación dinámica. Finalmente, los lenguajes funcionales y la mayoría de los lenguajes orientados a objetos, tales como LISP y Smalltalk, requieren un ambiente "completamente dinámico" en el que toda asignación se realiza de manera automática mediante código generado por el compilador. Esto es complicado, porque se requiere que la memoria también sea liberada automáticamente, y esto a su vez requiere complejos algoritmos de "recolección de basura". Examinaremos esos métodos junto con nuestro estudio de los ambientes de ejecución, aunque una completa relación de esta área rebasa el alcance de este libro.

OPCIONES DE COMPILADOR E INTERFACES

Un aspecto importante de la construcción de compiladores es la inclusión de mecanismos para hacer interfaces con el sistema operativo y para proporcionar opciones al usuario para diversos propósitos. Ejemplos de mecanismos de interfaces son el suministro de la entrada y facilidades de salida, además del acceso al sistema de archivos de la máquina objetivo. Ejemplos de opciones de usuario incluyen la especificación de listar características (longitud, mensajes de error, tablas de referencia cruzada) así como opciones de optimización de código (rendimiento de ciertas optimizaciones pero no otras). Tanto la interface como las opciones son calificadas colectivamente como las **pragmáticas** del compilador. En ocasiones una definición de lenguaje especificará que se debe proporcionar cierta pragmática. Por ejemplo, Pascal y C especifican ciertos procedimientos de entrada/salida (en Pascal, son parte del propio lenguaje, mientras que en C son parte de la especificación de una librería estándar). En Ada, varias directivas de compilador, denominadas **pragmas**, son parte de la definición del lenguaje. Por ejemplo, las sentencias de Ada

```
pragma LIST(ON);
...
pragma LIST(OFF);
```

generan un listado de compilador para la parte del programa contenida dentro de los pragmas. En este texto veremos directivas de compilador solamente en el contexto de la generación de un listado con información para propósitos de la depuración del compilador. También, no trataremos cuestiones sobre interfaces de entrada/salida y sistema operativo, puesto que involucran considerables detalles que varían demasiado de un sistema operativo a otro.

MANEJO DE ERRORES

Una de las funciones más importantes de un compilador es su respuesta a los errores en un programa fuente. Los errores pueden ser detectados durante casi cualquier fase de la compilación. Estos **errores estáticos** (o de **tiempo de compilación**) deben ser notificados por un compilador, y es importante que el compilador sea capaz de generar mensajes de error significativos y reanudar la compilación después de cada error. Cada fase de un compilador necesitará una clase ligeramente diferente de manejo de errores, y, por lo tanto, un **manejador de errores** debe contener operaciones diferentes, cada una apropiada para una fase y situación específica. Por consiguiente, las técnicas de manejo de errores para cada fase se estudiarán de manera separada en el capítulo apropiado.

Una definición de lenguaje por lo general requerirá no solamente que los errores estáticos sean detectados por un compilador, sino también ciertos errores de ejecución. Esto requiere que un compilador genere código extra, el cual realizará pruebas de ejecución apropiadas para garantizar que todos esos errores provocarán un evento apropiado durante la ejecución. El más simple de tales eventos será detener la ejecución del programa. Sin embargo, a menudo esto no es adecuado, y una definición de lenguaje puede requerir la presencia de mecanismos para el **manejo de excepciones**. Éstos pueden complicar sustancialmente la administración de un sistema de ejecución, especialmente si un programa puede continuar ejecutándose desde el punto donde ocurrió el error. No consideraremos la implementación de un mecanismo así, pero mostraremos cómo un compilador puede generar código de prueba para asegurar qué errores de ejecución especificados ocasionarán que se detenga la ejecución.

1.6 ARRANQUE AUTOMÁTICO Y PORTABILIDAD

Hemos analizado el lenguaje fuente y el lenguaje objetivo como factores determinantes en la estructura de un compilador y la utilidad de separar cuestiones de lenguaje fuente y objetivo en etapas inicial y final. Pero no hemos mencionado el tercer lenguaje involucrado en el proceso de construcción de compiladores: el lenguaje en el que el compilador mismo está escrito. Para que el compilador se ejecute inmediatamente, este lenguaje de implementación (o **lenguaje anfitrión**) tendría que ser lenguaje de máquina. Así fue en realidad como se escribieron los primeros compiladores, puesto que esencialmente no existían compiladores todavía. Un enfoque más razonable en la actualidad es escribir el compilador en otro lenguaje para el cual ya exista un compilador. Si el compilador existente ya se ejecuta en la máquina objetivo, entonces solamente necesitamos compilar el nuevo compilador utilizando el compilador existente para obtener un programa ejecutable:

Si el compilador existente para el lenguaje B se ejecuta en una máquina diferente de la máquina objetivo, entonces la situación es un poco más complicada. La compilación produce entonces un **compilador cruzado**, es decir, un compilador que genera código objetivo para una máquina diferente de aquella en la que puede ejecutarse. Ésta y otras situaciones más complejas se describen mejor al esquematizar un compilador como un **diagrama T** (llamado así debido a su forma). Un compilador escrito en el lenguaje H (por *language host*

o anfitrión) que traduce lenguaje S (de *source* o fuente) en lenguaje T (por *language Target* u objetivo) se dibuja como el siguiente diagrama T:

Advierta que esto es equivalente a decir que el compilador se ejecuta en la "máquina" H (si H no es código de máquina, entonces la consideraremos como código ejecutable para una máquina hipotética). Típicamente, esperamos que H sea lo mismo que T (es decir, el compilador produce código para la misma máquina que aquella en la que se ejecuta), pero no es necesario que éste sea el caso.

Los diagramas T se pueden combinar en dos maneras. Primero, si tenemos dos compiladores que se ejecutan en la misma máquina H, uno de los cuales traduce lenguaje A al lenguaje B mientras que el otro traduce el lenguaje B al lenguaje C, entonces podemos combinarlos dejando que la salida del primero sea la entrada al segundo. El resultado es un compilador de A a C en la máquina H. Expresamos lo anterior como sigue:

En segundo lugar podemos utilizar un compilador de la "máquina" H a la "máquina" K para traducir el lenguaje de implementación de otro compilador de H a K. Expresamos esto último como sigue:

Ahora el primer escenario que describimos anteriormente, es decir, utilizando un compilador existente para el lenguaje B en la máquina H para traducir un compilador de lenguaje A a H escrito en B, puede verse como el siguiente diagrama, el cual es precisamente un caso especial del diagrama anterior:

El segundo escenario que describimos (donde el compilador de lenguaje B se ejecuta en una máquina diferente, lo cual resulta en un compilador cruzado para A) puede describirse de manera similar como sigue:

Es común escribir un compilador en el mismo lenguaje que está por compilarse:

Mientras que parece ser un enredo de circularidad (puesto que, si todavía no existe compilador para el lenguaje fuente, el compilador mismo no puede ser compilado) existen ventajas importantes que se obtienen de este enfoque.

Considere, por ejemplo, cómo podemos enfocar el problema de la circularidad. Podemos escribir un compilador "rápido e impreciso" en lenguaje ensamblador, traduciendo solamente aquellas características de lenguaje que en realidad sean utilizadas en el compilador (teniendo, naturalmente, limitado nuestro uso de esas características cuando escribamos el compilador "bueno"). Este compilador "rápido e impreciso" también puede producir código muy ineficiente (¡solamente necesita ser correcto!). Una vez que tenemos el compilador "rápido e impreciso" en ejecución, lo utilizamos para compilar el compilador "bueno". Entonces volvemos a compilar el compilador "bueno" para producir la versión final. Este proceso se denomina **arranque automático por transferencia**. Este proceso se ilustra en las figuras 1.2*a* y 1.2*b*.

Después del arranque automático tenemos un compilador tanto en código fuente como en código ejecutable. La ventaja de esto es que cualquier mejoramiento al código fuente del

Figura 1.2a
Primera etapa en un proceso de arranque automático

Figura 1.2b
Segunda etapa en un proceso de arranque automático

compilador puede ser transferido inmediatamente a un compilador que esté trabajando, aplicando el mismo proceso de dos pasos anterior.

Pero existe otra ventaja. Transportar ahora el compilador a una nueva computadora anfitrión solamente requiere que la etapa final del código fuente vuelva a escribirse para generar código para la nueva máquina. Éste se compila entonces utilizando el compilador antiguo para producir un compilador cruzado, y el compilador es nuevamente recompilado mediante el compilador cruzado para producir una versión de trabajo para la nueva máquina. Esto se ilustra en las figuras 1.3*a* y 1.3*b*.

Figura 1.3a
Transportación de un compilador escrito en su propio lenguaje fuente (paso 1)

Figura 1.3b
Transportación de un compilador escrito en su propio lenguaje fuente (paso 2)

1.7 LENGUAJE Y COMPILADOR DE MUESTRA TINY

Un libro acerca de la construcción de compiladores estaría incompleto sin ejemplos para cada paso en el proceso de compilación. En muchos casos ilustraremos técnicas con ejemplos

que son sustraídos de lenguajes existentes, tales como C, C++, Pascal y Ada. No obstante, estos ejemplos no son suficientes para mostrar cómo se conjuntan todas las partes de un compilador. Por eso, también es necesario exhibir un compilador completo y proporcionar un comentario acerca de su funcionamiento.

Este requerimiento (que se ha demostrado en un compilador real) es difícil. Un compilador "real", es decir, uno que esperaríamos utilizar en la programación cotidiana, tiene demasiado detalle y sería algo abrumador para estudiar dentro del marco conceptual de un texto. Por otra parte, un compilador para un lenguaje muy pequeño, cuyo listado pudiera comprimirse en aproximadamente 10 páginas de texto, no podría esperar demostrar de manera adecuada todas las características que un compilador "real" necesita.

Intentaremos satisfacer estos requerimientos al proporcionar código fuente completo en C (ANSI) para un lenguaje pequeño cuyo compilador se pueda comprender fácilmente una vez que las técnicas se hayan entendido. Llamaremos a este lenguaje TINY y lo utilizaremos como un ejemplo ejecutable para las técnicas estudiadas en cada capítulo. El código para su compilador se analizará a medida que se cubran las técnicas. En esta sección proporcionaremos una perspectiva general del lenguaje y su compilador. El código completo del compilador se recopila en el apéndice B.

Un problema adicional es la elección del lenguaje de máquina a utilizar como el lenguaje objetivo del compilador TINY. Nuevamente, la complejidad al utilizar código de máquina real para un procesador existente hace que una selección de esta naturaleza sea difícil. Pero la elección de un procesador específico también tiene el efecto de limitar la ejecución del código objetivo resultante para estas máquinas. En cambio, simplificamos el código objetivo para que sea el lenguaje ensamblador para un procesador hipotético simple, el cual conoceremos como la máquina TM (por las siglas de "TINY Machine"). Daremos un rápido vistazo a esta máquina aquí, pero aplazaremos una descripción más extensa hasta el capítulo 8 (generación de código). Un listado del simulador de TM en C aparece en el apéndice C.

1.7.1 Lenguaje TINY

Un programa en TINY tiene una estructura muy simple: es simplemente una secuencia de sentencias separadas mediante signos de punto y coma en una sintaxis semejante a la de Ada o Pascal. No hay procedimientos ni declaraciones. Todas las variables son variables enteras, y las variables son declaradas simplemente al asignar valores a las mismas (de modo parecido a FORTRAN o BASIC). Existen solamente dos sentencias de control: una sentencia "if" y una sentencia "repeat". Ambas sentencias de control pueden ellas mismas contener secuencias de sentencias. Una sentencia "if" tiene una parte opcional "else" y debe terminarse mediante la palabra clave **end**. También existen sentencias de lectura y escritura que realizan entrada/salida. Los comentarios, que no deben estar anidados, se permiten dentro de llaves tipográficas.

Las expresiones en TINY también se encuentran limitadas a expresiones aritméticas enteras y booleanas. Una expresión booleana se compone de una comparación de dos expresiones aritméticas que utilizan cualesquiera de los dos operadores de comparación < y =. Una expresión aritmética puede involucrar constantes enteras, variables, paréntesis y cualquiera de los cuatro operadores enteros +, -, * y / (división entera), con las propiedades matemáticas habituales. Las expresiones booleanas pueden aparecer solamente como pruebas en sentencias de control: no hay variables booleanas, asignación o E/S (entrada/salida).

La figura 1.4 proporciona un programa de muestra en este lenguaje para la conocida función factorial. Utilizaremos este programa como un ejemplo de ejecución a través del texto.

Figura 1.4
Un programa en lenguaje TINY que proporciona a la salida el factorial de su entrada

```
{ Programa de muestra
  en lenguaje TINY -
  calcula el factorial
}
read x; { introducir un entero }
if x > 0 then { no calcule si x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { salida del factorial de x }
end
```

Aunque TINY carece de muchas características necesarias para los lenguajes de programación reales (procedimientos, arreglos y valores de punto flotante son algunas de las omisiones más serias), todavía es lo suficientemente extenso para ejemplificar la mayoría de las características esenciales de un compilador.

1.7.2 Compilador TINY

El compilador TINY se compone de los siguientes archivos en C, donde enumeramos los archivos de cabecera ("header") (por inclusión) a la izquierda y los archivos de código ("code") a la derecha:

`globals.h`	`main.c`
`util.h`	`util.c`
`scan.h`	`scan.c`
`parse.h`	`parse.c`
`symtab.h`	`symtab.c`
`analyze.h`	`analyze.c`
`code.h`	`code.c`
`cgen.h`	`cgen.c`

El código fuente para estos archivos se encuentra listado en el apéndice B, con números de línea y en el orden dado, excepto que `main.c` está listado antes de `globals.h`. El archivo de cabecera `globals.h` está incluido en todos los archivos de código. Contiene las definiciones de los tipos de datos y variables globales utilizadas a lo largo del compilador. El archivo `main.c` contiene el programa principal que controla el compilador, y asigna e inicializa las variables globales. Los archivos restantes se componen de pares de archivos de cabecera/código, con los prototipos de función disponibles externamente dados en el archivo de cabecera e implementados (posiblemente con funciones locales estáticas adicionales) en el archivo de código asociado. Los archivos `scan`, `parse`, `analyze` y `cgen` corresponden exactamente a las fases del analizador léxico, analizador sintáctico, analizador semántico y generador de código de la figura 1.1. Los archivos `util` contienen funciones de utilerías necesarias para generar la representación interna del código fuente (el árbol sintáctico)

y exhibir la información de error y listado. Los archivos **symtab** contienen una implementación de tabla de cálculo de dirección de una tabla de símbolos adecuada para usarse con TINY. Los archivos **code** contienen utilerías para la generación de código que son dependientes de la máquina objetivo (la máquina TM, descrita en la sección 1.7.3). Los componentes restantes de la figura 1.1 se encuentran ausentes: no hay tabla de literales o manejador de error por separado y no hay fases de optimización. Tampoco hay código intermedio separado del árbol sintáctico. Adicionalmente, la tabla de símbolos interactúa solamente con el analizador semántico y el generador de código (de modo que aplazaremos un análisis de esto hasta el capítulo 6).

Para reducir la interacción entre estos archivos, también hemos hecho el compilador de cuatro pasadas: la primera pasada se compone del analizador léxico y el analizador sintáctico, lo que construye el árbol sintáctico; la segunda y tercera pasadas se encargan del análisis semántico, con la segunda pasada construyendo la tabla de símbolos mientras que la tercera pasada realiza la verificación de tipos; la última pasada es el generador de código. El código en **main.c** que controla estas pasadas es particularmente simple. Ignorando las banderas y la compilación condicional, el código central es como sigue (véanse las líneas 69, 77, 79 y 94 del apéndice B):

```
syntaxTree = parse();
buildSymtab(syntaxTree);
typeCheck(syntaxTree);
codeGen(syntaxTree,codefile);
```

Por flexibilidad, también integramos banderas de compilación condicional que hacen posible construir compiladores parciales. Las banderas, con sus efectos, se describen a continuación:

MARCA O BANDERA	SU EFECTO SI SE ACTIVA	ARCHIVOS NECESARIOS PARA COMPILACIÓN (ACUMULATIVOS)
NO_PARSE	Construye un compilador solamente de análisis léxico.	**globals.h**, **main.c**, **util.h**, **util.c**, **scan.h**, **scan.c**
NO_ANALYZE	Construye un compilador que únicamente realiza análisis sintáctico y léxico.	**parse.h**, **parse.c**
NO_CODE	Construye un compilador que realiza el análisis semántico pero no genera código.	**symtab.h**, **symtab.c**, **analyze.h**, **analyze.c**

Aunque este diseño para el compilador TINY es algo poco realista, tiene la ventaja pedagógica de que los archivos por separado corresponden aproximadamente a las fases, y se pueden analizar (así como compilar y ejecutar) de manera individual en los capítulos que siguen.

El compilador TINY se puede compilar por cualquier compilador C ANSI. Suponiendo que el nombre del archivo ejecutable es **tiny**, se puede utilizar para compilar un programa fuente de TINY en el archivo de texto **sample.tny** al emitir el comando

```
tiny sample.tny
```

(El compilador también agregará la extensión **.tny** si es omitida.) Esto imprimirá un listado de programa en la pantalla (el cual se puede redirigir a un archivo) y (si la generación del

código se activa) también generará el archivo de código objetivo `sample.tm` (para utilizarse con la máquina TM, que se describe más adelante).

Existen diversas opciones para la información en el listado de compilación. Se encuentran disponibles las siguientes banderas:

MARCA O BANDERA	SU EFECTO SI SE ACTIVA
`EchoSource`	Hace eco (duplica) el programa fuente TINY hacia el listado junto con los números de línea.
`TraceScan`	Presenta la información en cada token a medida que el analizador léxico lo reconoce.
`TraceParse`	Presenta el árbol sintáctico en un formato linealizado.
`TraceAnalyze`	Presenta la información resumida acerca de la tabla de símbolos y la verificación de tipos.
`TraceCode`	Imprime comentarios del rastreo de la generación de código hacia el archivo de código.

1.7.3 Máquina TM

Empleamos el lenguaje ensamblador para esta máquina como el lenguaje objetivo para el compilador TINY. La máquina TM tiene instrucciones suficientes para ser un objetivo adecuado para un lenguaje pequeño como TINY. De hecho, TM tiene algunas de las propiedades de las computadoras con conjunto de instrucciones reducido (o RISC, de Reduced Instruction Set Computers), en que toda la aritmética y las pruebas deben tener lugar en registros y los modos de direccionamiento son muy limitados. Para dar alguna idea de la simplicidad de esta máquina traducimos el código para la expresión C

```
a[index] = 6
```

en el lenguaje ensamblador de TM (compare éste con el lenguaje ensamblador hipotético para la misma sentencia en la sección 1.3, página 12):

```
LDC  1,0(0)        carga 0 en registro 1
* la instrucción siguiente
* supone que index está en la localidad 10 en memoria
LD   0,10(1)       carga valor para 10 + R1 en R0
LDC  1,2(0)        carga 2 en registro 1
MUL  0,1,0         pon R1*R0 en R0
LDC  1,0(0)        carga 0 en registro 1
* la instrucción siguiente
* supone que a está en la localidad 20 en memoria
LDA  1,20(1)       carga 20+R1 en R0
ADD  0,1,0         pon R1+R0 en R0
LDC  1,6(0)        carga 6 en registro 1
ST   1,0(0)        almacena R1 en 0+R0
```

Advertimos que existen tres modos de direccionamiento para la operación de carga, todos dados por instrucciones diferentes: `LDC` es "constante de carga", `LD` es "carga desde memoria" y `LDA` es "dirección de carga", todos por sus siglas en inglés. Notamos también que las direcciones siempre deben ser dadas como valores de "registro+desplazamiento", como en `10(1)` (instrucción 2 del código precedente), que establece para la dirección

calculada el agregar el desplazamiento 10 al contenido del registro 1. (Puesto que el 0 fue cargado en el registro 1 en la instrucción previa, esto en realidad se refiere a la localidad absoluta 10.)[1] También advertimos que las instrucciones aritméticas **MUL** y **ADD** pueden tener solamente operandos de registro y son instrucciones de "tres direcciones", en que el registro objetivo del resultado se puede especificar independientemente de los operandos (contraste esto con el código en la sección 1.3, página 12, donde las operaciones fueron de "dos direcciones").

Nuestro simulador para la máquina TM lee el código ensamblador directamente de un archivo y lo ejecuta. De este modo, evitamos la complejidad agregada de traducir el lenguaje ensamblador a código de máquina. Sin embargo, nuestro simulador no es un verdadero ensamblador, en el sentido de que no hay etiquetas o direcciones simbólicas. Así, el compilador TINY todavía debe calcular direcciones absolutas para los saltos. También, para evitar la complejidad extra de vincularse con rutinas externas de entrada/salida, la máquina TM contiene facilidades integradas de E/S para enteros; éstas son leídas desde, y escritas hacia, los dispositivos estándar durante la simulación.

El simulador TM se puede compilar desde el código fuente **tm.c** utilizando cualquier compilador C ANSI. Suponiendo que el archivo ejecutable se llama **tm**, se puede emplear al emitir el comando

```
tm sample.tm
```

donde **sample.tm** es, por ejemplo, el archivo de código producido por el compilador TINY a partir del archivo fuente **sample.tny**. Este comando provoca que el archivo de código sea ensamblado y cargado; entonces el simulador TM se puede ejecutar de manera interactiva. Por ejemplo, si **sample.tny** es el programa de muestra de la figura 1.4, entonces el factorial de 7 se puede calcular con la interacción siguiente:

```
tm sample.tm
TM  simulation (enter h for help)...
Enter command: go
Enter value for IN instruction: 7
OUT instruction prints: 5040
HALT: 0,0,0
Halted
Enter command: quit
Simulation done.
```

1.8 C-MINUS: UN LENGUAJE PARA UN PROYECTO DE COMPILADOR

Un lenguaje más extenso que TINY, adecuado para un proyecto de compilador, se describe en el apéndice A. Éste es un subconjunto considerablemente restringido de C, al

1. El comando **LDC** también requiere un formato de registro+desplazamiento, pero el registro es ignorado y el desplazamiento mismo es cargado como una constante. Esto se debe al formato uniforme simple del ensamblador de TM.

cual llamaremos C-Minus. Contiene enteros, arreglos de enteros y funciones (incluyendo procedimientos, o funciones sin tipo). Tiene declaraciones (estáticas) locales y globales y funciones recursivas (simples). Tiene una sentencia "if" y una sentencia "while". Carece de casi todo lo demás. Un programa se compone de una secuencia de declaraciones de variables y funciones. Una función **main** se debe declarar al último. La ejecución comienza con una llamada a **main**.[2]

Como un ejemplo de un programa en C-Minus, en la figura 1.5 escribimos el programa factorial de la figura 1.4 usando una función recursiva. La entrada/salida en este programa es proporcionada por una función **read** y una función **write** que se pueden definir en términos de las funciones estándar de C **scanf** y **printf**.

C-Minus es un lenguaje más complejo que TINY, particularmente en sus requerimientos de generación de código, pero la máquina TM todavía es un objetivo razonable para su compilador. En el apéndice A proporcionamos una guía sobre cómo modificar y extender el compilador TINY a C-Minus.

Figura 1.5
Un programa de C-Minus que da a la salida el factorial de su entrada

```
int fact( int x )
/* función factorial recursiva */
{ if (x > 1)
    return x * fact(x-1);
  else
    return 1;
}

void main( void )
{ int x;
  x = read();
  if (x > 0) write( fact(x) );
}
```

EJERCICIOS

1.1 Seleccione un compilador conocido que venga empacado con un ambiente de desarrollo, y haga una lista de todos los programas acompañantes que se encuentran disponibles con el compilador junto con una breve descripción de sus funciones.

1.2 Dada la asignación en C

```
a[i+1] = a[i] + 2
```

dibuje un árbol de análisis gramatical y un árbol sintáctico para la expresión utilizando como guía el ejemplo semejante de la sección 1.3.

1.3 Los errores de compilación pueden dividirse aproximadamente en dos categorías: errores sintácticos y errores semánticos. Los errores sintácticos incluyen tokens olvidados o colocados de

2. Para que sea consistente con otras funciones en C-Minus, **main** es declarada como una función **void** ("sin tipo") con una lista de parámetros **void**. Mientras que esto difiere del C ANSI, muchos compiladores de C aceptarán esta anotación.

manera incorrecta, tal como el paréntesis derecho olvidado en la expresión aritmética `(2+3` . Los errores semánticos incluyen tipos incorrectos en expresiones y variables no declaradas (en la mayoría de los lenguajes), tal como la asignación `x = 2`, donde `x` es una variable de tipo arreglo.

a. Proporcione dos ejemplos más de errores de cada clase en un lenguaje de su elección.

b. Seleccione un compilador con el que esté familiarizado y determine si se enumeran todos los errores sintácticos antes de los errores semánticos o si los errores de sintaxis y de semántica están entremezclados. ¿Cómo influye esto en el número de pasadas?

1.4 Esta pregunta supone que usted trabaja con un compilador que tiene una opción para producir salida en lenguaje ensamblador.

a. Determine si su compilador realiza optimizaciones de incorporación de constantes.

b. Una optimización relacionada pero más avanzada es la de propagación de constantes: una variable que actualmente tiene un valor constante es reemplazada por ese valor en las expresiones. Por ejemplo, el código (en sintaxis de C)

```
x = 4;
y = x + 2;
```

se reemplazaría, utilizando propagación de constantes (e incorporación de constantes), por el código

```
x = 4;
y = 6;
```

Determine si su compilador efectuó propagación de constantes.

c. Proporcione tantas razones como pueda de por qué la propagación de constantes es más difícil que la incorporación de constantes.

d. Una situación relacionada con la propagación de constantes y la incorporación de constantes es el uso de las constantes nombradas en un programa. Utilizando una constante nombrada `x` en vez de una variable podemos traducir el ejemplo anterior como el siguiente código en C:

```
const int x = 4;
...
y = x + 2;
...
```

Determine si su compilador realiza propagación/incorporación bajo estas circunstancias. ¿En qué difiere esto del inciso b?

1.5 Si su compilador aceptara la entrada directamente desde el teclado, determine si dicho compilador lee el programa entero antes de generar mensajes de error o bien genera mensajes de error a medida que los encuentra. ¿Qué implicación tiene hacer esto en el número de pasadas?

1.6 Describa las tareas que realizan los programas siguientes y explique en qué se parecen o se relacionan con los compiladores:

a. Un preprocesador de lenguaje **b.** Un módulo de impresión **c.** Un formateador de texto

1.7 Suponga que tiene un traductor de Pascal a C escrito en C y un compilador trabajando para C. Utilice diagramas T para describir los pasos que tomaría para crear un compilador trabajando para Pascal.

1.8 Utilizamos una flecha $\Rightarrow$ para indicar la reducción de un patrón de dos diagramas T a un solo diagrama T. Podemos considerar esta flecha como una "relación de reducción" y formar su cerradura transitiva $\Rightarrow^*$, en la cual permitimos que tenga lugar una secuencia de reducciones.

Dado el siguiente diagrama, en el que las letras establecen lenguajes arbitrarios, determine cuáles lenguajes deben ser iguales para que la reducción sea válida, y muestre los pasos de la reducción simple que la hacen válida:

Proporcione un ejemplo práctico de la reducción que describe este diagrama.

1.9 Una alternativa al método de transportar un compilador descrito en la sección 1.6 y en la figura 1.3 es utilizar un intérprete para el código intermedio producido por el compilador y suprimir una etapa final del todo. Un método así es utilizado por el **sistema P de Pascal**, el cual incluye un compilador de Pascal que produce código P, una clase de código ensamblador para una máquina de pila "genérica" y un intérprete de código P que simula la ejecución del código P. Tanto el compilador de Pascal como el intérprete de código P están escritos en código P.

a. Describa los pasos necesarios para obtener un compilador trabajando para Pascal en una máquina arbitraria, dado un sistema P de Pascal.

b. Describa los pasos necesarios para obtener un compilador trabajando en el código nativo de su sistema, en el inciso a (es decir, un compilador que produce código ejecutable para la máquina anfitrión, en vez de utilizar el intérprete de código P).

1.10 El proceso de transportar un compilador puede ser considerado como dos operaciones distintas: **reasignación del objetivo** (la modificación del compilador para producir código objetivo para una nueva máquina) y **reasignación del anfitrión** (la modificación del compilador para ejecutarse en una nueva máquina). Analice las diferencias de estas dos operaciones en términos de diagramas T.

NOTAS Y REFERENCIAS

La mayoría de los temas mencionados en este capítulo se tratan con más detalle en capítulos subsiguientes, y las notas y referencias de esos capítulos proporcionarán las referencias adecuadas. Por ejemplo, Lex se estudia en el capítulo 2; Yacc en el capítulo 5; la verificación de tipos, tablas de símbolos y análisis de atributos en el capítulo 6; la generación de código, el código en tres direcciones y el código P en el capítulo 8; finalmente, el manejo de errores en los capítulos 4 y 5.

Una referencia estándar completa acerca de los compiladores es Aho [1986], particularmente para la teoría y los algoritmos. Un texto que ofrece muchas sugerencias útiles de implementación es Fischer y LeBlanc [1991]. Pueden hallarse descripciones completas de los compiladores de C en Fraser y Hanson [1995] y Holub [1990]. Un compilador popular de C/C++ cuyo código fuente se encuentra ampliamente disponible a través de Internet es el compilador Gnu. Éste se describe con detalle en Stallman [1994].

Para una visión general de los conceptos de los lenguajes de programación, con información acerca de sus interacciones con los traductores, véase Louden [1993] o Sethi [1996].

Una útil referencia para la teoría de autómatas desde un punto de vista matemático (en oposición al punto de vista práctico adoptado aquí) es la de Hopcroft y Ullman [1979]. Puede encontrarse también ahí información adicional acerca de la jerarquía de Chomsky (así como en el capítulo 3).

Una descripción de los primeros compiladores FORTRAN puede encontrarse en Backus [1957] y Backus [1981]. Una descripción de un antiguo compilador Algol60 puede hallarse en Randell y Russell [1964]. Los compiladores de Pascal se describen en Barron [1981], donde también puede encontrarse una descripción del sistema P de Pascal (Nori [1981]).

El preprocesador Ratfor mencionado en la sección 1.2 se describe en Kernighan [1975]. Los diagramas T de la sección 1.6 fueron introducidos por Bratman [1961].

Este texto se enfoca en las técnicas de traducción estándar útiles para la traducción de la mayoría de los lenguajes. Es posible que se necesiten otras técnicas adicionales para la traducción eficaz de los lenguajes ajenos a la tradición principal de los lenguajes imperativos basados en Algol. En particular, la traducción de los lenguajes funcionales tales como ML y Haskell ha sido la fuente de muchas nuevas técnicas, algunas de las cuales pueden llegar a ser importantes técnicas generales en el futuro. Las descripciones de estas técnicas pueden encontrarse en Appel [1992], Peyton Jones [1992] y Peyton Jones [1987]. Este último contiene una descripción de la verificación de tipos de Hindley-Milner (mencionada en la sección 1.1).

Capítulo 2

Rastreo o análisis léxico

2.1 El proceso del análisis léxico
2.2 Expresiones regulares
2.3 Autómatas finitos
2.4 Desde las expresiones regulares hasta los DFA
2.5 Implementación de un analizador léxico TINY
2.6 Uso de Lex para generar automáticamente un analizador léxico

La fase de rastreo, o **análisis léxico**, de un compilador tiene la tarea de leer el programa fuente como un archivo de caracteres y dividirlo en tokens. Los tokens son como las palabras de un lenguaje natural: cada token es una secuencia de caracteres que representa una unidad de información en el programa fuente. Ejemplos típicos de token son las **palabras reservadas**, como `if` y `while`, las cuales son cadenas fijas de letras; los **identificadores**, que son cadenas definidas por el usuario, compuestas por lo regular de letras y números, y que comienzan con una letra; los **símbolos especiales**, como los símbolos aritméticos + y *; además de algunos símbolos compuestos de múltiples caracteres, tales como > = y <>. En cada caso un token representa cierto patrón de caracteres que el analizador léxico reconoce, o ajusta desde el inicio de los caracteres de entrada restantes.

Como la tarea que realiza el analizador léxico es un caso especial de coincidencia de patrones, necesitamos estudiar métodos de especificación y reconocimiento de patrones en la medida en que se aplican al proceso de análisis léxico. Estos métodos son principalmente los de las **expresiones regulares** y los **autómatas finitos**. Sin embargo, un analizador léxico también es la parte del compilador que maneja la entrada del código fuente, y puesto que esta entrada a menudo involucra un importante gasto de tiempo, el analizador léxico debe funcionar de manera tan eficiente como sea posible. Por lo tanto, también necesitamos poner mucha atención a los detalles prácticos de la estructura del analizador léxico.

Dividiremos el estudio de las cuestiones del analizador léxico como sigue. En primer lugar, daremos una perspectiva general de la función de un analizador léxico y de las estructuras y conceptos involucrados. Enseguida, estudiaremos las expresiones regulares, una notación estándar para representar los patrones en cadenas que forman la estructura léxica de un lenguaje de programación. Continuaremos con el estudio de las máquinas de estados finitos, o autómatas finitos, las cuales representan algoritmos que permiten reconocer los patrones de cadenas dados por las expresiones regulares. También estudiaremos el proceso de construcción de los autómatas finitos fuera de las expresiones

regulares. Después volveremos a los métodos prácticos para escribir programas que implementen los procesos de reconocimiento representados por los autómatas finitos y estudiaremos una implementación completa de un analizador léxico para el lenguaje TINY. Por último, estudiaremos la manera en que se puede automatizar el proceso de producción de un programa analizador léxico por medio de un generador de analizador léxico, y repetiremos la implementación de un analizador léxico para TINY utilizando Lex, que es un generador de analizador léxico estándar disponible para su uso en Unix y otros sistemas.

2.1 EL PROCESO DEL ANÁLISIS LÉXICO

El trabajo del analizador léxico es leer los caracteres del código fuente y formarlos en unidades lógicas para que lo aborden las partes siguientes del compilador (generalmente el analizador sintáctico). Las unidades lógicas que genera el analizador léxico se denominan **tokens**, y formar caracteres en tokens es muy parecido a formar palabras a partir de caracteres en una oración en un lenguaje natural como el inglés o cualquier otro y decidir lo que cada palabra significa. En esto se asemeja a la tarea del deletreo.

Los tokens son entidades lógicas que por lo regular se definen como un tipo enumerado. Por ejemplo, pueden definirse en C como[1]

```
typedef enum
    {IF,THEN,ELSE,PLUS,MINUS,NUM,ID,...}
    TokenType;
```

Los tokens caen en diversas categorías, una de ellas la constituyen las **palabras reservadas**, como `IF` y `THEN`, que representan las cadenas de caracteres "if" y "then". Una segunda categoría es la de los **símbolos especiales**, como los símbolos aritméticos `MÁS` y `MENOS`, los que se representan con los caracteres "+" y "−". Finalmente, existen tokens que pueden representar cadenas de múltiples caracteres. Ejemplos de esto son `NUM` e `ID`, los cuales representan números e identificadores.

Los tokens como entidades lógicas se deben distinguir claramente de las cadenas de caracteres que representan. Por ejemplo, el token de la palabra reservada `IF` se debe distinguir de la cadena de caracteres "if" que representa. Para hacer clara la distinción, la cadena de caracteres representada por un token se denomina en ocasiones su **valor de cadena** o su **lexema**. Algunos tokens tienen sólo un lexema: las palabras reservadas tienen esta propiedad. No obstante, un token puede representar un número infinito de lexemas. Los identificadores, por ejemplo, están todos representados por el token simple `ID`, pero tienen muchos valores de cadena diferentes que representan sus nombres individuales. Estos nombres no se pueden pasar por alto, porque un compilador debe estar al tanto de ellos en una tabla de símbolos. Por consiguiente, un rastreador o analizador léxico también debe construir los valores de cadena de por lo menos algunos de los tokens. Cualquier valor asociado a un token

---

1. En un lenguaje sin tipos enumerados tendríamos que definir los tokens directamente como valores *numéricos simbólicos. Así, al estilo antiguo de C, en ocasiones se vería lo siguiente:*

```
#define IF 256
#define THEN 257
#define ELSE 258
...
```

(Estos números comienzan en 256 para evitar que se confundan con los valores ASCII numéricos.)

se denomina **atributo** del token, y el valor de cadena es un ejemplo de un atributo. Los tokens también pueden tener otros atributos. Por ejemplo, un token **NUM** puede tener un atributo de valor de cadena como "32767", que consta de cinco caracteres numéricos, pero también tendrá un atributo de valor numérico que consiste en el valor real de 32767 calculado a partir de su valor de cadena. En el caso de un token de símbolo especial como **MÁS** (**PLUS**), no sólo se tiene el valor de cadena "+", sino también la operación aritmética real + que está asociada con él mismo. En realidad, el símbolo del token mismo se puede ver simplemente como otro atributo, mientras que el token se puede visualizar como la colección de todos sus atributos.

Un analizador léxico necesita calcular al menos tantos atributos de un token como sean necesarios para permitir el procesamiento siguiente. Por ejemplo, se necesita calcular el valor de cadena de un token **NUM**, pero no es necesario calcular de inmediato su valor numérico, puesto que se puede calcular de su valor de cadena. Por otro lado, si se calcula su valor numérico, entonces se puede descartar su valor de cadena. En ocasiones el mismo analizador léxico puede realizar las operaciones necesarias para registrar un atributo en el lugar apropiado, o puede simplemente pasar el atributo a una fase posterior del compilador. Por ejemplo, un analizador léxico podría utilizar el valor de cadena de un identificador para introducirlo a la tabla de símbolos, o podría pasarlo para introducirlo en una etapa posterior.

Puesto que el analizador léxico posiblemente tendrá que calcular varios atributos para cada token, a menudo es útil recolectar todos los atributos en un solo tipo de datos estructurados, al que podríamos denominar como **registro de token**. Un registro así se podría declarar en C como

```
typedef struct
    { TokenType tokenval;
      char * stringval;
      int numval;
    } TokenRecord;
```

o posiblemente como una unión

```
typedef struct
    { TokenType tokenval;
      union
      { char * stringval;
        int numval;
      } attribute;
    } TokenRecord;
```

(lo que supone que el atributo de valor de cadena sólo es necesario para identificadores y el atributo de valor numérico sólo para números). Un arreglo más común es que el analizador léxico solamente devuelva el valor del token y coloque los otros atributos en variables donde se pueda tener acceso a ellos por otras partes del compilador.

Aunque la tarea del analizador léxico es convertir todo el programa fuente en una secuencia de tokens, pocas veces el analizador hará todo esto de una vez. En realidad, el analizador léxico funcionará bajo el control del analizador sintáctico, devolviendo el siguiente token simple desde la entrada bajo demanda mediante una función que tendrá una declaración similar a la declaración en el lenguaje C

```
TokenType getToken(void);
```

La función **getToken** declarada de esta manera devolverá, cuando se le llame, el siguiente token desde la entrada, y además calculará atributos adicionales, como el valor de cadena del token. La cadena de caracteres de entrada por lo regular no tiene un parámetro para esta función, pero se conserva en un buffer (localidad de memoria intermedia) o se proporciona mediante las facilidades de entrada del sistema.

Como ejemplo del funcionamiento de **getToken** considere la siguiente línea de código fuente en C, que utilizamos como ejemplo en el capítulo 1:

```
a[index] = 4 + 2
```

Suponga que esta línea de código se almacena en un buffer de entrada como sigue, con el siguiente carácter de entrada indicado por la flecha:

Una llamada a **getToken** necesitará ahora saltarse los siguientes cuatro espacios en blanco, reconocer la cadena "a" compuesta del carácter único *a* como el token siguiente, y devolver el valor de token **ID** como el token siguiente, dejando el buffer de entrada como se aprecia a continuación:

De esta manera, una llamada posterior a **getToken** comenzará de nuevo el proceso de reconocimiento con el carácter de corchete izquierdo.

Volveremos ahora al estudio de métodos para definir y reconocer patrones en cadenas de caracteres.

2.2 EXPRESIONES REGULARES

Las expresiones regulares representan patrones de cadenas de caracteres. Una expresión regular *r* se encuentra completamente definida mediante el conjunto de cadenas con las que concuerda. Este conjunto se denomina **lenguaje generado por la expresión regular** y se escribe como $L(r)$. Aquí la palabra *lenguaje* se utiliza sólo para definir "conjunto de cadenas" y no tiene (por lo menos en esta etapa) una relación específica con un lenguaje de programación. Este lenguaje depende, en primer lugar, del conjunto de caracteres que se encuentra disponible. En general, estaremos hablando del conjunto de caracteres ASCII o de algún subconjunto del mismo. En ocasiones el conjunto será más general que el conjunto de caracteres ASCII, en cuyo caso los elementos del conjunto se describirán como **símbolos**. Este conjunto de símbolos legales se conoce como **alfabeto** y por lo general se representa mediante el símbolo griego Σ (sigma).

Una expresión regular *r* también contendrá caracteres del alfabeto, pero esos caracteres tendrán un significado diferente: en una expresión regular todos los símbolos indican *patrones*. En este capítulo distinguiremos el uso de un carácter como patrón escribiendo todo los patrones en negritas. De este modo, **a** es el carácter *a* usado como patrón.

Por último, una expresión regular r puede contener caracteres que tengan significados especiales. Este tipo de caracteres se llaman **metacaracteres** o **metasímbolos**, y por lo general no pueden ser caracteres legales en el alfabeto, porque no podríamos distinguir su uso como metacaracteres de su uso como miembros del alfabeto. Sin embargo, a menudo no es posible requerir tal exclusión, por lo que se debe utilizar una convención para diferenciar los dos usos posibles de un metacarácter. En muchas situaciones esto se realiza mediante el uso de un **carácter de escape** que "desactiva" el significado especial de un metacarácter. Unos caracteres de escape comunes son la diagonal inversa y las comillas. Advierta que los caracteres de escape, si también son caracteres legales en el alfabeto, son por sí mismos metacaracteres.

2.2.1 Definición de expresiones regulares

Ahora estamos en posición de describir el significado de las expresiones regulares al establecer cuáles lenguajes genera cada patrón. Haremos esto en varias etapas. Comenzaremos por describir el conjunto de expresiones regulares básicas, las cuales se componen de símbolos individuales. Continuaremos con la descripción de las operaciones que generan nuevas expresiones regulares a partir de las ya existentes. Esto es similar a la manera en que se construyen las expresiones aritméticas: las expresiones aritméticas básicas son los números, tales como 43 y 2.5. Entonces las operaciones aritméticas, como la suma y la multiplicación, se pueden utilizar para formar nuevas expresiones a partir de las existentes, como en el caso de 43 * 2.5 y 43 * 2.5 + 1.4.

El grupo de expresiones regulares que describiremos aquí es mínimo, ya que sólo contiene los metasímbolos y las operaciones esenciales. Después consideraremos extensiones a este conjunto mínimo.

Expresiones regulares básicas Éstas son precisamente los caracteres simples del alfabeto, los cuales se corresponden a sí mismos. Dado cualquier carácter a del alfabeto Σ, indicamos que la expresión regular **a** corresponde al carácter a escribiendo $L(\mathbf{a}) = \{a\}$. Existen otros dos símbolos que necesitaremos en situaciones especiales. Necesitamos poder indicar una concordancia con la **cadena vacía**, es decir, la cadena que no contiene ningún carácter. Utilizaremos el símbolo ε (épsilon) para denotar la cadena vacía, y definiremos el metasímbolo $\boldsymbol{\varepsilon}$ (ε en negritas) estableciendo que $L(\varepsilon) = \{\varepsilon\}$. También necesitaremos ocasionalmente ser capaces de describir un símbolo que corresponda a la ausencia de cadenas, es decir, cuyo lenguaje sea el **conjunto vacío**, el cual escribiremos como { }. Emplearemos para esto el símbolo $\boldsymbol{\phi}$ y escribiremos $L(\boldsymbol{\phi}) = \{\ \}$. Observe la diferencia entre { } y $\{\varepsilon\}$: el conjunto { } no contiene ninguna cadena, mientras que el conjunto $\{\varepsilon\}$ contiene la cadena simple que no se compone de ningún carácter.

Operaciones de expresiones regulares Existen tres operaciones básicas en las expresiones regulares: 1) selección entre alternativas, la cual se indica mediante el metacarácter **|** (barra vertical); 2) concatenación, que se indica mediante yuxtaposición (sin un metacarácter), y 3) repetición o "cerradura", la cual se indica mediante el metacarácter *****. Analizaremos cada una por turno, proporcionando la construcción del conjunto correspondiente para los lenguajes de cadenas concordantes.

Selección entre alternativas Si r y s son expresiones regulares, entonces $r \mid s$ es una expresión regular que define cualquier cadena que concuerda con r o con s. En términos de lenguajes, el lenguaje de $r \mid s$ es la **unión** de los lenguajes de r y s, o $L(r \mid s) = L(r) \cup L(s)$. Como un ejemplo simple, considere la expresión regular **a|b**: ésta corresponde tanto al

carácter a como al carácter b, es decir, $L(\mathbf{a} \mid \mathbf{b}) = L(\mathbf{a}) \cup L(\mathbf{b}) = \{a\} \cup \{b\} = \{a, b\}$. Como segundo ejemplo, la expresión regular $\mathbf{a} \mid \boldsymbol{\varepsilon}$ corresponde tanto al carácter simple a como a la cadena vacía (que no está compuesta por ningún carácter). En otras palabras, $L(\mathbf{a} \mid \boldsymbol{\varepsilon}) = \{a, \varepsilon\}$.

La selección se puede extender a más de una alternativa, de manera que, por ejemplo, $L(\mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \mid \mathbf{d}) = \{a, b, c, d\}$. En ocasiones también escribiremos largas secuencias de selecciones con puntos, como en **a | b | . . . | z**, que corresponde a cualquiera de las letras minúsculas de la a a la z.

Concatenación La concatenación de dos expresiones regulares r y s se escribe como rs, y corresponde a cualquier cadena que sea la concatenación de dos cadenas, con la primera de ellas correspondiendo a r y la segunda correspondiendo a s. Por ejemplo, la expresión regular **ab** corresponde sólo a la cadena ab, mientras que la expresión regular **(a | b) c** corresponde a las cadenas ac y bc. (El uso de los paréntesis como metacaracteres en esta expresión regular se explicará en breve.)

Podemos describir el efecto de la concatenación en términos de lenguajes generados al definir la concatenación de dos conjuntos de cadenas. Dados dos conjuntos de cadenas S_1 y S_2, el conjunto concatenado de cadenas S_1S_2 es el conjunto de cadenas de S_1 complementado con todas las cadenas de S_2. Por ejemplo, si $S_1 = \{aa, b\}$ y $S_2 = \{a, bb\}$, entonces $S_1S_2 = \{aaa, aabb, ba, bbb\}$. Ahora la operación de concatenación para expresiones regulares se puede definir como sigue: $L(rs) = L(r)L(s)$. De esta manera (utilizando nuestro ejemplo anterior), $L(\mathbf{(a \mid b)\,c}) = L(\mathbf{a} \mid \mathbf{b})L(\mathbf{c}) = \{a, b\}\{c\} = \{ac, bc\}$.

La concatenación también se puede extender a más de dos expresiones regulares: $L(r_1 r_2 \ldots r_n) = L(r_1)L(r_2) \ldots L(r_n)$ = el conjunto de cadenas formado al concatenar todas las cadenas de cada una de las $L(r_1), \ldots, L(r_n)$.

Repetición La operación de repetición de una expresión regular, denominada también en ocasiones **cerradura (de Kleene)**, se escribe r^*, donde r es una expresión regular. La expresión regular r^* corresponde a cualquier concatenación finita de cadenas, cada una de las cuales corresponde a r. Por ejemplo, **a*** corresponde a las cadenas ε, a, aa, aaa, ... (Concuerda con ε porque ε es la concatenación de *ninguna* cadena concordante con **a**.) Podemos definir la operación de repetición en términos de lenguajes generados definiendo, a su vez, una operación similar * para conjuntos de cadenas. Dado un conjunto S de cadenas, sea

$$S^* = \{\varepsilon\} \cup S \cup SS \cup SSS \cup \ldots$$

Ésta es una unión de conjuntos infinita, pero cada uno de sus elementos es una concatenación finita de cadenas de S. En ocasiones el conjunto S^* se escribe como sigue:

$$S^* = \bigcup_{n=0}^{\infty} S^n$$

donde $S^n = S. \,.\, S$ es la concatenación de S por n veces. ($S^0 = \{\varepsilon\}$.)

Ahora podemos definir la operación de repetición para expresiones regulares como sigue:

$$L(r^*) = L(r)^*$$

Considere como ejemplo la expresión regular `(a|bb)*`. (De nueva cuenta, la razón de tener paréntesis como metacaracteres se explicará más adelante.) Esta expresión regular corresponde a cualquiera de las cadenas siguientes: ε, *a, bb, aa, abb, bba, bbbb, aaa, aabb* y así sucesivamente. En términos de lenguajes, $L($`(a|bb)*`$) = L($`a|bb`$)^* = \{a, bb\}^* = \{\varepsilon, a, bb, aa, abb, bba, bbbb, aaa, aabb, abba, abbbb, bbaa, \ldots\}$.

Precedencia de operaciones y el uso de los paréntesis La descripción precedente no toma en cuenta la cuestión de la precedencia de las operaciones de elección, concatenación y repetición. Por ejemplo, dada la expresión regular `a|b*`, ¿deberíamos interpretar esto como `(a|b)*` o como `a|(b*)`? (Existe una diferencia importante, puesto que $L($`(a|b)*`$) = \{\varepsilon, a, b, aa, ab, ba, bb, \ldots\}$, mientras que $L($`a|(b*)`$) = \{\varepsilon, a, b, bb, bbb, \ldots\}$.) La convención estándar es que la repetición debería tener mayor precedencia, por lo tanto, la segunda interpretación es la correcta. En realidad, entre las tres operaciones, se le da al `*` la precedencia más alta, a la concatenación se le da la precedencia que sigue y a la `|` se le otorga la precedencia más baja. De este modo, por ejemplo, `a|bc*` se interpreta como `a|(b(c*))`, mientras que `ab|c*d` se interpreta como `(ab)|((c*)d)`.

Cuando deseemos indicar una precedencia diferente, debemos usar paréntesis para hacerlo. Ésta es la razón por la que tuvimos que escribir `(a|b)c` para indicar que la operación de elección debería tener mayor precedencia que la concatenación, ya que de otro modo `a|bc` se interpretaría como si correspondiera tanto a *a* como a *bc*. De manera similar, `(a|bb)*` se interpretaría sin los paréntesis como `a|bb*`, lo que corresponde a *a, b, bb, bbb*, ... Los paréntesis aquí se usan igual que en aritmética, donde $(3 + 4) * 5 = 35$, pero $3 + 4 * 5 = 23$, ya que se supone que * tiene precedencia más alta que +.

Nombres para expresiones regulares A menudo es útil como una forma de simplificar la notación proporcionar un nombre para una expresión regular larga, de modo que no tengamos que escribir la expresión misma cada vez que deseemos utilizarla. Por ejemplo, si deseáramos desarrollar una expresión regular para una secuencia de uno o más dígitos numéricos, entonces escribiríamos

```
(0|1|2|...|9)(0|1|2|...|9)*
```

o podríamos escribir

```
dígito dígito*
```

donde

```
dígito = 0|1|2|...|9
```

es una **definición regular** del nombre ***dígito***.

El uso de una definición regular es muy conveniente, pero introduce la complicación agregada de que el nombre mismo se convierta en un metasímbolo y se deba encontrar un significado para distinguirlo de la concatenación de sus caracteres. En nuestro caso hicimos esa distinción al utilizar letra cursiva para el nombre. Advierta que no se debe emplear el nombre del término en su propia definición (es decir, de manera recursiva): debemos poder eliminar nombres reemplazándolos sucesivamente con las expresiones regulares para las que se establecieron.

Antes de considerar una serie de ejemplos para elaborar nuestra definición de expresiones regulares, reuniremos todas las piezas de la definición de una expresión regular.

Definición

Una **expresión regular** es una de las siguientes:

1. Una expresión regular **básica** constituida por un solo carácter **a**, donde a proviene de un alfabeto Σ de caracteres legales; el metacarácter $\boldsymbol{\varepsilon}$; o el metacarácter $\boldsymbol{\phi}$. En el primer caso, $L(\mathbf{a}) = \{a\}$; en el segundo, $L(\boldsymbol{\varepsilon}) = \{\varepsilon\}$; en el tercero, $L(\boldsymbol{\phi}) = \{\ \}$.
2. Una expresión de la forma $\boldsymbol{r} \mid \boldsymbol{s}$, donde r y s son expresiones regulares. En este caso, $L(\boldsymbol{r} \mid \boldsymbol{s}) = L(r) \cup L(s)$.
3. Una expresión de la forma rs, donde r y s son expresiones regulares. En este caso, $L(rs) = L(r)L(s)$.
4. Una expresión de la forma r*****, donde r es una expresión regular. En este caso, $L(r\mathbf{*}) = L(r)^*$.
5. Una expresión de la forma **(**r**)**, donde r es una expresión regular. En este caso, $L(\mathbf{(}r\mathbf{)}) = L(r)$. De este modo, los paréntesis no cambian el lenguaje, sólo se utilizan para ajustar la precedencia de las operaciones.

Advertimos que, en esta definición, la precedencia de las operaciones en (2), (3) y (4) está en el orden inverso de su enumeración; es decir, **|** tiene precedencia más baja que la concatenación, y ésta tiene una precedencia más baja que el asterisco *****. También advertimos que esta definición proporciona un significado de metacarácter a los seis símbolos $\boldsymbol{\phi}$, $\boldsymbol{\varepsilon}$, **|**, *****, **(**, **)**.

En lo que resta de esta sección consideraremos una serie de ejemplos diseñados para explicar con más detalles la definición que acabamos de dar. Éstos son algo artificiales, ya que por lo general no aparecen como descripciones de token en un lenguaje de programación. En la sección 2.2.3 consideraremos algunas expresiones regulares comunes que aparecen con frecuencia como tokens en lenguajes de programación.

En los ejemplos siguientes por lo regular se incluye una descripción en idioma coloquial de las cadenas que se harán corresponder, y la tarea es traducir la descripción a una expresión regular. Esta situación, en la que un manual de lenguaje contiene descripciones de los tokens, es la que con más frecuencia enfrentan quienes escriben compiladores. En ocasiones puede ser necesario invertir la dirección, es decir, desplazarse de una expresión regular hacia una descripción en lenguaje coloquial, de modo que también incluiremos algunos ejercicios de esta clase.

Ejemplo 2.1

Consideremos el alfabeto simple constituido por sólo tres caracteres alfabéticos: $\Sigma = \{a, b, c\}$. También el conjunto de todas las cadenas en este alfabeto que contengan exactamente una b. Este conjunto es generado por la expresión regular

```
(a|c)*b(a|c)*
```

Advierta que, aunque **b** aparece en el centro de la expresión regular, la letra b no necesita estar en el centro de la cadena que se desea definir. En realidad, la repetición de a o c antes y después de la b puede presentarse en diferentes números de veces. Por consiguiente, todas las cadenas siguientes están generadas mediante la expresión regular anterior: *b, abc, abaca, baaaac, ccbaca, cccccb.* §

Ejemplo 2.2

Con el mismo alfabeto que antes, considere el conjunto de todas las cadenas que contienen como máximo una *b*. Una expresión regular para este conjunto se puede obtener utilizando la solución al ejemplo anterior como una alternativa (definiendo aquellas cadenas con exactamente una *b*) y la expresión regular `(a|c)*` como la otra alternativa (definiendo los casos sin *b* en todo). De este modo, tenemos la solución siguiente:

```
(a|c)*|(a|c)*b(a|c)*
```

Una solución alternativa sería permitir que *b* o la cadena vacía apareciera entre las dos repeticiones de *a* o *c*:

```
(a|c)*(b|ε)(a|c)*
```

Este ejemplo plantea un punto importante acerca de las expresiones regulares: el mismo lenguaje se puede generar mediante muchas expresiones regulares diferentes. Por lo general, intentamos encontrar una expresión regular tan simple como sea posible para describir un conjunto de cadenas, aunque nunca intentaremos demostrar que encontramos, de hecho, la "más simple": la más breve por ejemplo. Existen dos razones para esto. La primera es que raramente se presenta en situaciones prácticas, donde por lo regular existe una solución estándar "más simple". La segunda es que cuando estudiemos métodos para reconocer expresiones regulares, los algoritmos tendrán que poder simplificar el proceso de reconocimiento sin molestarse en simplificar primero la expresión regular. §

Ejemplo 2.3

Consideremos el conjunto de cadenas *S* sobre el alfabeto $\Sigma = \{a, b\}$ compuesto de una *b* simple rodeada por el mismo número de *a*:

$$S = \{b, aba, aabaa, aaabaaa, \ldots\} = \{a^n b a^n | n \neq 0\}$$

Este conjunto no se puede describir mediante una expresión regular. La razón es que la única operación de repetición que tenemos es la operación de cerradura `*`, la cual permite cualquier número de repeticiones. De modo que si escribimos la expresión `a*ba*` (lo más cercano que podemos obtener en el caso de una expresión regular para *S*), no hay garantía de que el número de *a* antes y después de la *b* será el mismo. Expresamos esto al decir que "las expresiones regulares no pueden contar". Sin embargo, para proporcionar una demostración matemática de este hecho, requeriríamos utilizar un famoso teorema acerca de las expresiones regulares conocido como **lema de la extracción (pumping lemma)**, que se estudia en la teoría de autómatas, pero que aquí ya no volveremos a mencionar.

Es evidente que no todos los conjuntos de cadenas que podemos describir en términos simples se pueden generar mediante expresiones regulares. Por consiguiente, un conjunto de cadenas que *es* el lenguaje para una expresión regular se distingue de otros conjuntos al denominarlo **conjunto regular**. De cuando en cuando aparecen conjuntos no regulares como cadenas en lenguajes de programación que necesitan ser reconocidos por un analizador léxico, los cuales por lo regular son abortados cuando surgen. Retomaremos este tema de manera breve en la sección en que se abordan las consideraciones para el analizador léxico práctico. §

Ejemplo 2.4

Consideremos las cadenas en el alfabeto $\Sigma = \{a, b, c\}$ que no contienen dos *b* consecutivas. De modo que, entre cualesquiera dos *b*, debe haber por lo menos una *a* o una *c*. Construiremos

una expresión regular para este conjunto en varias etapas. En primer lugar, podemos obligar a que una *a* o *c* se presenten *después* de cualquier *b* al escribir

```
(b(a|c))*
```

Podemos combinar esto con la expresión `(a|c)*`, la cual define cadenas que no tienen *b* alguna, y escribimos

```
((a|c)*|(b(a|c))*)*
```

o, advirtiendo que $(r^* \mid s^*)^*$ corresponde con las mismas cadenas que $(r \mid s)^*$

```
((a|c)|(b(a|c)))*
```

o

```
(a|c|ba|bc)*
```

(¡Cuidado! Ésta todavía no es la respuesta correcta.)

El lenguaje generado por esta expresión regular tiene, en realidad, la propiedad que buscamos, a saber, que no haya dos *b* consecutivas (pero aún no es lo bastante correcta). En ocasiones podremos demostrar tales aseveraciones, así que esbozaremos una demostración de que todas las cadenas en $L($`(a|c|ba|bc)*`$)$ no contienen dos *b* consecutivas. La demostración es por inducción sobre la longitud de la cadena (es decir, el número de caracteres en la cadena). Es evidente que esto es verdadero para todas las cadenas de longitud 0, 1 o 2: estas cadenas son precisamente las cadenas ε, *a, c, aa, ac, ca, cc, ba, bc*. Ahora supongamos que es verdadero para todas las cadenas en el lenguaje con una longitud de $i < n$, y sea *s* una cadena en el lenguaje con longitud $n > 2$. Entonces, *s* contiene más de una de las cadenas no-ε anteriormente enumeradas, de modo que $s = s_1 s_2$, donde s_1 y s_2 también se encuentran en el lenguaje y no son ε. Por lo tanto, mediante la hipótesis de inducción, tanto s_1 como s_2 no tienen dos *b* consecutivas. Por consiguiente, la única manera de que *s* misma pudiera tener dos *b* consecutivas sería que s_1 finalizara con una *b* y que s_2 comenzara con una *b*. Pero esto es imposible, porque ninguna cadena en el lenguaje puede finalizar con una *b*.

Este último hecho que utilizamos en el esbozo de la demostración (o sea, que ninguna cadena generada mediante la expresión regular anterior puede finalizar con una *b*) también muestra por qué nuestra solución todavía no es bastante correcta: no genera las cadenas *b*, *ab* y *cb*, que no contienen dos *b* consecutivas. Arreglaremos esto agregando una *b* opcional rezagada de la manera siguiente:

```
(a|c|ba|bc)*(b|ε)
```

Advierta que la imagen especular de esta expresión regular también genera el lenguaje dado

```
(b|ε)(a|c|ab|cb)*
```

También podríamos generar este mismo lenguaje escribiendo

```
(notb|b notb)*(b|ε)
```

donde `notb = a|c`. Éste es un ejemplo del uso de un nombre para una subexpresión. De hecho, esta solución es preferible en los casos en que el alfabeto es grande, puesto que la definición de `notb` se puede ajustar para incluir todos los caracteres, excepto *b*, sin complicar la expresión original. §

Ejemplo 2.5

En este ejemplo proporcionamos la expresión regular y solicitamos determinar una descripción concisa en español del lenguaje que genera. Considere el alfabeto $\Sigma = \{a, b, c\}$ y la expresión regular

```
((b|c)*a(b|c)*a)*(b|c)*
```

Esto genera el lenguaje de todas las cadenas que contengan un número par de *a*. Para ver esto considere la expresión dentro de la repetición exterior izquierda:

```
(b|c)*a(b|c)*a
```

Esto genera aquellas cadenas que finalizan en *a* y que contienen exactamente dos *a* (cualquier número de *b* y de *c* puede aparecer antes o entre las dos *a*). La repetición de estas cadenas nos da todas las cadenas que finalizan en *a* cuyo número de *a* es un múltiplo de 2 (es decir, par). Al añadir la repetición `(b|c)*` al final (como en el ejemplo anterior) obtenemos el resultado deseado.

Advertimos que esta expresión regular también se podría escribir como

```
(nota* a nota* a)* nota*
```

§

2.2.2 Extensiones para las expresiones regulares

Formulamos una definición para las expresiones regulares que emplean un conjunto mínimo de operaciones comunes a todas las aplicaciones, y podríamos limitarnos a utilizar sólo las tres operaciones básicas (junto con paréntesis) en todos nuestros ejemplos. Sin embargo, ya vimos en los ejemplos anteriores que escribir expresiones regulares utilizando sólo estos operadores en ocasiones es poco manejable, ya que se crean expresiones regulares que son más complicadas de lo que serían si se dispusiera de un conjunto de operaciones más expresivo. Por ejemplo, sería útil tener una notación para una coincidencia de cualquier carácter (ahora tenemos que enumerar todos los caracteres en el alfabeto como una larga alternativa). Además, ayudaría tener una expresión regular para una gama de caracteres y una expresión regular para todos los caracteres excepto uno.

En los párrafos siguientes describiremos algunas extensiones a las expresiones regulares estándar ya analizadas, con los correspondientes metasímbolos nuevos, que cubren éstas y situaciones comunes semejantes. En la mayoría de estos casos no existe una terminología común, de modo que utilizaremos una notación similar a la empleada por el generador de analizadores léxicos Lex, el cual se describe más adelante en este capítulo. En realidad, muchas de las situaciones que describiremos aparecerán de nuevo en nuestra descripción de Lex. No obstante, no todas las aplicaciones que utilizan expresiones regulares incluirán estas operaciones, e incluso aunque lo hicieran, se puede emplear una notación diferente.

Ahora pasaremos a nuestra lista de nuevas operaciones.

UNA O MÁS REPETICIONES

Dada una expresión regular *r*, la repetición de *r* se describe utilizando la operación de cerradura estándar, que se escribe *r**. Esto permite que *r* se repita 0 o más veces. Una situación típica que surge es la necesidad de *una* o más repeticiones en lugar de ninguna, lo que garantiza que aparece por lo menos una cadena correspondiente a *r*, y no permite la cadena vacía ε. Un ejemplo es el de un número natural, donde queremos una secuencia de dígitos, pero deseamos que por lo menos aparezca uno. Por ejemplo, si deseamos definir números binarios, podríamos escribir `(0|1)*`, pero esto también

coincidirá con la cadena vacía, la cual no es un número. Por supuesto, podríamos escribir

```
(0|1)(0|1)*
```

pero esta situación se presenta con tanta frecuencia que se desarrolló para ella una notación relativamente estándar en la que se utiliza + en lugar de *: *r*+, que indica una o más repeticiones de *r*. De este modo, nuestra expresión regular anterior para números binarios puede escribirse ahora como

```
(0|1)+
```

CUALQUIER CARÁCTER

Una situación común es la necesidad de generar cualquier carácter en el alfabeto. Sin una operación especial esto requiere que todo carácter en el alfabeto sea enumerado en una alternativa. Un metacarácter típico que se utiliza para expresar una concordancia de cualquier carácter es el punto "`.`", el cual no requiere que el alfabeto se escriba realmente en forma extendida. Con este metacarácter podemos escribir una expresión regular para todas las cadenas que contengan al menos una *b* como se muestra a continuación:

```
.*b.*
```

UN INTERVALO DE CARACTERES

A menudo necesitamos escribir un intervalo de caracteres, como el de todas las letras minúsculas o el de todos los dígitos. Hasta ahora hemos hecho esto utilizando la notación `a|b|...|z` para las letras minúsculas o `0|1|...|9` para los dígitos. Una alternativa es tener una notación especial para esta situación, y una que es común es la de emplear corchetes y un guión, como en `[a-z]` para las letras minúsculas y `[0-9]` para los dígitos. Esto también se puede emplear para alternativas individuales, de modo que `a|b|c` puede escribirse como `[abc]`. También se pueden incluir los intervalos múltiples, de manera que `[a-zA-Z]` representa todas las letras minúsculas y mayúsculas. Esta notación general se conoce como **clases de caracteres**. Advierta que esta notación puede depender del orden subyacente del conjunto de caracteres. Por ejemplo, al escribir `[A-Z]` se supone que los caracteres *B*, *C*, y los demás vienen entre los caracteres *A* y *Z* (una suposición razonable) y que sólo los caracteres en mayúsculas están entre *A* y *Z* (algo que es verdadero para el conjunto de caracteres ASCII). Sin embargo, al escribir `[A-z]` *no* se definirán los mismos caracteres que para `[A-Za-z]`, incluso en el caso del conjunto de caracteres ASCII.

CUALQUIER CARÁCTER QUE NO ESTÉ EN UN CONJUNTO DADO

Como hemos visto, a menudo es de utilidad poder excluir un carácter simple del conjunto de caracteres por generar. Esto se puede conseguir al diseñar un metacarácter para indicar la operación de negación ("not") o complementaria sobre un conjunto de alternativas. Por ejemplo, un carácter estándar que representa la negación en lógica es la "tilde" ~, y podríamos escribir una expresión regular para un carácter en el alfabeto que no sea *a* como `~a` y un carácter que no sea *a*, ni *b*, ni *c*, como

```
~(a|b|c)
```

Una alternativa para esta notación se emplea en Lex, donde el carácter "carat" ^ se utiliza en conjunto con las clases de caracteres que acabamos de describir para la formación de

complementos. Por ejemplo, cualquier carácter que no sea *a* se escribe como `[^a]`, mientras que cualquier carácter que no sea *a*, ni *b* ni *c* se escribe como

```
[^abc]
```

SUBEXPRESIONES OPCIONALES

Por último, un suceso que se presenta comúnmente es el de cadenas que contienen partes opcionales que pueden o no aparecer en cualquier cadena en particular. Por ejemplo, un número puede o no tener un signo inicial, tal como + o -. Podemos emplear alternativas para expresar esto como en las definiciones regulares

```
natural = [0-9]+
naturalconSigno = natural | + natural | - natural
```

Esto se puede convertir rápidamente en algo voluminoso, e introduciremos el metacarácter de signo de interrogación *r*? para indicar que las cadenas que coincidan con *r* son opcionales (o que están presentes 0 o 1 copias de *r*). De este modo, el ejemplo del signo inicial se convierte en

```
natural = [0-9]+
naturalconSigno = (+|-)? natural
```

2.2.3 Expresiones regulares para tokens de lenguajes de programación

Los tokens de lenguajes de programación tienden a caer dentro de varias categorías limitadas que son bastante estandarizadas a través de muchos lenguajes de programación diferentes. Una categoría es la de las **palabras reservadas**, en ocasiones también conocidas como **palabras clave**, que son cadenas fijas de caracteres alfabéticos que tienen un significado especial en el lenguaje. Los ejemplos incluyen `if`, `while` y `do` en lenguajes como Pascal, C y Ada. Otra categoría se compone de los **símbolos especiales**, que incluyen operadores aritméticos, de asignación y de igualdad. Éstos pueden ser un carácter simple, tal como =, o múltiples caracteres o compuestos, tales como := o ++. Una tercera categoría se compone de los **identificadores**, que por lo común se definen como secuencias de letras y dígitos que comienzan con una letra. Una categoría final se compone de **literales** o **constantes**, que incluyen constantes numéricas tales como 42 y 3.14159, literales de cadena como "hola, mundo" y caracteres tales como "a" y "b". Aquí describiremos algunas de estas expresiones regulares típicas y analizaremos algunas otras cuestiones relacionadas con el reconocimiento de tokens. Más detalles acerca de las cuestiones de reconocimiento práctico aparecen posteriormente en el capítulo.

Números Los números pueden ser sólo secuencias de dígitos (números naturales), o números decimales, o números con un exponente (indicado mediante una "e" o "E"). Por ejemplo, 2.71E-2 representa el número .0271. Podemos escribir definiciones regulares para esos números como se ve a continuación:

```
nat = [0-9]+
natconSigno = (+|-)? nat
número = natconSigno("." nat)?(E natconSigno)?
```

Aquí escribimos el punto decimal entre comillas para enfatizar que debería ser generado directamente y no interpretado como un metacarácter.

Identificadores y palabras reservadas Las palabras reservadas son las más simples de escribir como expresiones regulares: están representadas por sus secuencias fijas de caracteres. Si quisiéramos recolectar todas las palabras reservadas en una definición, escribiríamos algo como

```
reservada = if | while | do | ...
```

Los identificadores, por otra parte, son cadenas de caracteres que no son fijas. Un identificador debe comenzar, por lo común, con una letra y contener sólo letras y dígitos. Podemos expresar esto en términos de definiciones regulares como sigue:

```
letra = [a-zA-Z]
dígito = [0-9]
identificador = letra(letra|dígito)*
```

Comentarios Los comentarios por lo regular se ignoran durante el proceso del análisis léxico.[2] No obstante, un analizador léxico debe reconocer los comentarios y descartarlos. Por consiguiente, necesitaremos escribir expresiones regulares para comentarios, aun cuando un analizador léxico pueda no tener un token constante explícito (podríamos llamar a estos **pseudotokens**). Los comentarios pueden tener varias formas diferentes. Suelen ser de formato libre o estar rodeados de delimitadores tales como

```
{éste es un comentario de Pascal}
/* éste es un comentario de C */
```

o comenzar con un carácter o caracteres especificados y continuar hasta el final de la línea, como en

```
; éste es un comentario de Scheme
-- éste es un comentario de Ada
```

No es difícil escribir una expresión regular para comentarios que tenga delimitadores de carácter simple, tal como el comentario de Pascal, o para aquellos que partan de algún(os) carácter(es) especificado(s) hasta el final de la línea. Por ejemplo, el caso del comentario de Pascal puede escribirse como

```
{(~})*}
```

donde escribimos `~}` para indicar "not `}`" y donde partimos del supuesto de que el carácter `}` no tiene significado como un metacarácter. (Una expresión diferente debe escribirse para Lex, la cual analizaremos más adelante en este capítulo.) De manera similar, un comentario en Ada se puede hacer coincidir mediante la expresión regular

```
--(~nuevalínea)*
```

2. En ocasiones pueden contener directivas de compilador.

en la cual suponemos que **nuevalínea** corresponde al final de una línea (lo que se escribe como **\n** en muchos sistemas), que el carácter "-" no tiene significado como un metacarácter, y que el final de la línea no está incluido en el comentario mismo. (Veremos cómo escribir esto en Lex en la sección 2.6.)

Es mucho más difícil escribir una expresión regular para el caso de los delimitadores que tienen más de un carácter de longitud, tal como los comentarios de C. Para ver esto considere el conjunto de cadenas *ba*. . . (ausencia de apariciones de *ab*). . .*ab* (utilizamos *ba*. . .*ab* en lugar de los delimitadores de C /*. . .*/, ya que el asterisco, y en ocasiones la diagonal, es un metacarácter que requiere de un manejo especial). No podemos escribir simplemente

```
ba(~(ab))*ab
```

porque el operador "not" por lo regular está restringido a caracteres simples en lugar de cadenas de caracteres. Podemos intentar escribir una definición para **~(ab)** utilizando **~a**, **~b** y **~(a|b)**, pero esto es no trivial. Una solución es

```
b*(a*~(a|b)b*)*a*
```

pero es difícil de leer (y de demostrar correctamente). De este modo, una expresión regular para los comentarios de C es tan complicada que casi nunca se escribe en la práctica. De hecho, este caso se suele manejar mediante métodos ex profeso en los analizadores léxicos reales, lo que veremos más adelante en este capítulo.

Por último, otra complicación en el reconocimiento de los comentarios es que, en algunos lenguajes de programación, los comentarios pueden estar anidados. Por ejemplo, Modula-2 permite comentarios de la forma

```
(* esto es (* un comentario de *) en Modula-2 *)
```

Los delimitadores de comentario deben estar exactamente pareados en dichos comentarios anidados, de manera que lo que sigue no es un comentario legal de Modula-2:

```
(* esto es (* ilegal en Modula-2 *))
```

La anidación de los comentarios requiere que el analizador léxico cuente el número de los delimitadores. Pero advertimos en el ejemplo 2.3 (sección 2.2.1) que las expresiones regulares no pueden expresar operaciones de conteo. En la práctica utilizamos un esquema de contador simple como una solución adecuada para este problema (véanse los ejercicios).

Ambigüedad, espacios en blanco y búsqueda hacia delante A menudo en la descripción de los tokens de lenguajes de programación utilizando expresiones regulares, algunas cadenas se pueden definir mediante varias expresiones regulares diferentes. Por ejemplo, cadenas tales como **if** y **while** podrían ser identificadores o palabras clave. De manera semejante, la cadena **<>** se podría interpretar como la representación de dos tokens ("menor que" y "mayor que") o como un token simple ("no es igual a"). Una definición de lenguaje de programación debe establecer cuál interpretación se observará, y las expresiones regulares por sí mismas no pueden hacer esto. En realidad, una definición de lenguaje debe proporcionar **reglas de no ambigüedad** que implicarán cuál significado es el conveniente para cada uno de tales casos.

Dos reglas típicas que manejan los ejemplos que se acaban de dar son las siguientes. La primera establece que, cuando una cadena puede ser un identificador o una palabra clave, se prefiere por lo general la interpretación como palabra clave. Esto se da a entender mediante el uso del término **palabra reservada**, lo que quiere decir que es simplemente una palabra clave que no puede ser también un identificador. La segunda establece que, cuando una

cadena puede ser un token simple o una secuencia de varios tokens, por lo común se prefiere la interpretación del token simple. Esta preferencia se conoce a menudo como el **principio de la subcadena más larga**: la cadena más larga de caracteres que podrían constituir un token simple en cualquier punto se supone que representa el siguiente token.[3]

Una cuestión que surge con el uso del principio de la subcadena más larga es la cuestión de los **delimitadores de token**, o caracteres que implican que una cadena más larga en el punto donde aparecen no puede representar un token. Los caracteres que son parte no ambigua de otros tokens son delimitadores. Por ejemplo, en la cadena `xtemp=ytemp`, el signo de igualdad delimita el identificador `xtemp`, porque = no puede aparecer como parte de un identificador. Los espacios en blanco, los retornos de línea y los caracteres de tabulación generalmente también se asumen como delimitadores de token: `while x ...` se interpreta entonces como compuesto de dos tokens que representan la palabra reservada `while` y el identificador de nombre `x`, puesto que un espacio en blanco separa las dos cadenas de caracteres. En esta situación a menudo es útil definir un pseudotoken de espacio en blanco, similar al pseudotoken de comentario, que sólo sirve al analizador léxico de manera interna para distinguir otros tokens. En realidad, los comentarios mismos por lo regular sirven como delimitadores, de manera que, por ejemplo, el fragmento de código en lenguaje C

```
do/**/if
```

representa las dos palabras reservadas `do` e `if` más que el identificador `doif`.

Una definición típica del pseudotoken de espacio en blanco en un lenguaje de programación es

```
espacioenblanco = (nuevalínea|blanco|tabulación|comentario)+
```

donde los identificadores a la derecha representan los caracteres o cadenas apropiados. Advierta que, además de actuar como un delimitador de token, el espacio en blanco por lo regular no se toma en cuenta. Los lenguajes que especifican este comportamiento se denominan de **formato libre**. Las alternativas al formato libre incluyen el formato fijo de unos cuantos lenguajes como FORTRAN y diversos usos de la sangría en el texto, tal como la **regla de fuera de lugar** (véase la sección de notas y referencias). Un analizador léxico para un lenguaje de formato libre debe descartar el espacio en blanco después de verificar cualquier efecto de delimitación del token.

Los delimitadores terminan las cadenas que forman el token pero no son parte del token mismo. De este modo, un analizador léxico se debe ocupar del problema de la **búsqueda hacia delante**: cuando encuentra un delimitador debe arreglar que éste no se elimine del resto de la entrada, ya sea devolviéndolo a la cadena de entrada ("respaldándolo") o mirando hacia delante antes de eliminar el carácter de la entrada. En la mayoría de los casos sólo se necesita hacer esto para un carácter simple ("búsqueda hacia delante de carácter simple"). Por ejemplo, en la cadena `xtemp=ytemp`, el final del identificador `xtemp` se encuentra cuando se halla el =, y el signo = debe permanecer en la entrada, ya que representa el siguiente token a reconocer. Advierta también que es posible que no se necesite la búsqueda hacia delante para reconocer un token. Por ejemplo, el signo de igualdad puede ser el único token que comienza con el carácter =, en cuyo caso se puede reconocer de inmediato sin consultar el carácter siguiente.

En ocasiones un lenguaje puede requerir más que la búsqueda hacia delante de carácter simple, y el analizador léxico debe estar preparado para respaldar posiblemente de manera arbitraria muchos caracteres. En ese caso, el almacenamiento en memoria intermedia de los caracteres de entrada y la marca de lugares para un retroseguimiento se convierten en el diseño de un analizador léxico. (Algunas de estas preguntas se abordan más adelante en este capítulo.)

---

3. En ocasiones esto se denomina principio del "máximo bocado".

FORTRAN es un buen ejemplo de un lenguaje que viola muchos de los principios que acabamos de analizar. Éste es un lenguaje de formato fijo en el cual el espacio en blanco se elimina por medio de un preprocesador antes que comience la traducción. Por consiguiente, la línea en FORTRAN

```
I F ( X 2 . EQ. 0) THE N
```

aparecería para un compilador como

```
IF(X2.EQ.0)THEN
```

de manera que el espacio en blanco ya no funciona como un delimitador. Tampoco hay palabras reservadas en FORTRAN, de modo que todas las palabras clave también pueden ser identificadores, y la posición de la cadena de caracteres en cada línea de entrada es importante para determinar el token que será reconocido. Por ejemplo, la siguiente línea de código es perfectamente correcta en FORTRAN:

```
IF(IF.EQ.0)THENTHEN=1.0
```

Los primeros `IF` y `THEN` son palabras clave, mientras que los segundos `IF` y `THEN` son identificadores que representan variables. El efecto de esto es que un analizador léxico de FORTRAN debe poder retroceder a posiciones arbitrarias dentro de una línea de código. Consideremos, en concreto, el siguiente bien conocido ejemplo:

```
DO99I=1,10
```

Esto inicia un ciclo que comprende el código subsiguiente hasta la línea cuyo número es 99, con el mismo efecto que el Pascal `for i := 1 to 10`. Por otra parte, al cambiar la coma por un punto

```
DO99I=1.10
```

cambia el significado del código por completo: esto asigna el valor 1.1 a la variable con nombre `DO99I`. De este modo, un analizador léxico no puede concluir que el `DO` inicial es una palabra clave hasta que alcance el signo de coma (o el punto), en cuyo caso puede ser obligado a retroceder hacia el principio de la línea y comenzar de nuevo.

2.3 AUTÓMATAS FINITOS

Los autómatas finitos, o máquinas de estados finitos, son una manera matemática para describir clases particulares de algoritmos (o "máquinas"). En particular, los autómatas finitos se pueden utilizar para describir el proceso de reconocimiento de patrones en cadenas de entrada, y de este modo se pueden utilizar para construir analizadores léxicos. Por supuesto, también existe una fuerte relación entre los autómatas finitos y las expresiones regulares, y veremos en la sección siguiente cómo construir un autómata finito a partir de una expresión regular. Sin embargo, antes de comenzar nuestro estudio de los autómatas finitos de manera apropiada, consideraremos un ejemplo explicativo.

El patrón para identificadores como se define comúnmente en los lenguajes de programación está dado por la siguiente definición regular (supondremos que `letra` y `dígito` ya se definieron):

```
identificador = letra(letra|dígito)*
```

Esto representa una cadena que comienza con una letra y continúa con cualquier secuencia de letras y/o dígitos. El proceso de reconocer una cadena así se puede describir mediante el diagrama de la figura 2.1.

Figura 2.1
Un autómata finito para identificadores

En este diagrama los círculos numerados 1 y 2 representan **estados**, que son localidades en el proceso de reconocimiento que registran cuánto del patrón ya se ha visto. Las líneas con flechas representan **transiciones** que registran un cambio de un estado a otro en una coincidencia del carácter o caracteres mediante los cuales son etiquetados. En el diagrama de muestra, el estado 1 es el **estado de inicio**, o el estado en el que comienza el proceso de reconocimiento. Por convención, el estado de inicio se indica dibujando una línea con flechas sin etiqueta que proviene de "de ninguna parte". El estado 2 representa el punto en el cual se ha igualado una sola letra (lo que se indica mediante la transición del estado 1 al estado 2 etiquetada con `letra`). Una vez en el estado 2, cualquier número de letras y/o dígitos se puede ver, y una coincidencia de éstos nos regresa al estado 2. Los estados que representan el fin del proceso de reconocimiento, en los cuales podemos declarar un éxito, se denominan **estados de aceptación**, y se indican dibujando un borde con línea doble alrededor del estado en el diagrama. Puede haber más de uno de éstos. En el diagrama de muestra el estado 2 es un estado de aceptación, lo cual indica que, después que cede una letra, cualquier secuencia de letras y dígitos subsiguiente (incluyendo la ausencia de todas) representa un identificador legal.

El proceso de reconocimiento de una cadena de caracteres real como un identificador ahora se puede indicar al enumerar la secuencia de estados y transiciones en el diagrama que se utiliza en el proceso de reconocimiento. Por ejemplo, el proceso de reconocer `xtemp` como un identificador se puede indicar como sigue:

$$\rightarrow 1 \xrightarrow{\mathtt{x}} 2 \xrightarrow{\mathtt{t}} 2 \xrightarrow{\mathtt{e}} 2 \xrightarrow{\mathtt{m}} 2 \xrightarrow{\mathtt{p}} 2$$

En este diagrama etiquetamos cada transición mediante la letra que se iguala en cada paso.

2.3.1 Definición de los autómatas finitos determinísticos

Los diagramas como el que analizamos son descripciones útiles de los autómatas finitos porque nos permiten visualizar fácilmente las acciones del algoritmo. Sin embargo, en ocasiones es necesario tener una descripción más formal de un autómata finito, y por ello procederemos ahora a proporcionar una definición matemática. La mayor parte del tiempo, no obstante, no necesitaremos una visión tan abstracta como ésta, y describiremos la mayoría de los ejemplos en términos del diagrama solo. Otras descripciones de autómatas finitos también son posibles, particularmente las tablas, y éstas serán útiles para convertir los algoritmos en código de trabajo. Las describiremos a medida que surja la necesidad de utilizarlas.

También deberíamos advertir que lo que hemos estado describiendo son autómatas finitos **determinísticos**: autómatas donde el estado siguiente está dado unívocamente por el estado actual y el carácter de entrada actual. Una generalización útil de esto es el **autómata finito no determinístico**, el cual estudiaremos más adelante en esta sección.

Definición

Un **DFA** (por las siglas del concepto autómata finito determinístico en inglés) M se compone de un alfabeto Σ, un conjunto de estados S, una función de transición T: $S \times \Sigma \rightarrow S$, un estado de inicio $s_0 \in S$ y un conjunto de estados de aceptación $A \subset S$. El lenguaje aceptado por M, escrito como $L(M)$, se define como el conjunto de cadenas de caracteres $c_1c_2 \ldots c_n$ con cada $c_i \in \Sigma$, tal que existen estados $s_1 = T(s_0, c_1)$, $s_2 = T(s_1, c_2), \ldots, s_n = T(s_{n-1}, c_n)$, con s_n como un elemento de A (es decir, un estado de aceptación).

Hacemos las anotaciones siguientes respecto a esta definición. $S \times \Sigma$ se refiere al producto cartesiano o producto cruz de S y Σ: el conjunto de pares (s, c), donde $s \in S$ y $c \in \Sigma$. La función T registra las transiciones: $T(s, c) = s'$ si existe una transición del estado s al estado s' etiquetado mediante c. El segmento correspondiente del diagrama para M tendrá el aspecto siguiente:

La aceptación como la existencia de una secuencia de estados $s_1 = T(s_0, c_1)$, $s_2 = T(s_1, c_2), \ldots,$ $s_n = T(s_{n-1}, c_n)$, con s_n siendo un estado de aceptación, significa entonces lo mismo que el diagrama

$$\rightarrow s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} s_2 \longrightarrow \cdots \longrightarrow s_{n-1} \xrightarrow{c_n} s_n$$

Advertimos un número de diferencias entre la definición de un DFA y el diagrama del ejemplo identificador. En primer lugar, utilizamos los números para los estados en el diagrama del identificador, mientras la definición no restrinja el conjunto de estados a números. En realidad, podemos emplear cualquier sistema de identificación que queramos para los estados, incluyendo nombres. Por ejemplo, podemos escribir un diagrama equivalente al de la figura 2.1 como

donde ahora denominamos a los estados inicio (porque es el estado de inicio) y entrada_id (porque vimos una letra y estará reconociendo un identificador después de letras y números subsiguientes cualesquiera). El conjunto de estados para este diagrama se convierte ahora en {inicio, entrada_id} en lugar de {1, 2}.

Una segunda diferencia entre el diagrama y la definición es que no etiquetamos las transiciones con caracteres sino con nombres que representan un conjunto de caracteres.

Por ejemplo, el nombre `letra` representa cualquier letra del alfabeto de acuerdo con la siguiente definición regular:

```
letra = [a-zA-Z]
```

Ésta es una extensión conveniente de la definición, ya que sería abrumador dibujar 52 transiciones por separado, una para cada letra minúscula y una para cada letra mayúscula. Continuaremos empleando esta extensión de la definición en el resto del capítulo.

Una tercera diferencia fundamental entre la definición y nuestro diagrama es que la definición representa las transiciones como una *función* $T: S \times \Sigma \rightarrow S$. Esto significa que $T(s, c)$ debe tener un valor *para cada s y c*. Pero en el diagrama tenemos T(inicio, c) definida sólo si c es una letra, y T(entrada_id, c) está definida sólo si c es una letra o un dígito. ¿Dónde están las transiciones perdidas? La respuesta es que representan errores; es decir, en el reconocimiento de un identificador no podemos aceptar cualquier otro carácter aparte de letras del estado de inicio y letras o números después de éste.[4] La convención es que estas **transiciones de error** no se dibujan en el diagrama sino que simplemente se supone que siempre existen. Si las dibujáramos, el diagrama para un identificador tendría el aspecto que se ilustra en la figura 2.2.

Figura 2.2
Un autómata finito para identificadores con transiciones de error

En esta figura etiquetamos el nuevo estado error (porque representa una ocurrencia errónea), y etiquetamos las transiciones de error como `otro`. Por convención, `otro` representa cualquier carácter que no aparezca en ninguna otra transición desde el estado donde se origina. Por consiguiente, la definición de `otro` proveniente desde el estado de inicio es

```
otro = ~letra
```

y la definición de `otro` proveniente desde el estado entrada_id es

```
otro = ~(letra|dígito)
```

4. En realidad, estos caracteres no alfanuméricos significan que no hay un identificador en todo (si estamos en el estado de inicio), o que no hemos encontrado un delimitador que finalice el reconocimiento de un identificador (si estamos en un estado de aceptación). Más adelante en esta sección veremos cómo manejar estos casos.

Advierta también que todas las transiciones desde el estado de error van de regreso hacia sí mismo (etiquetamos a estas transiciones como ***cualquiera*** para indicar que cualquier carácter resulta en esta transición). El estado de error también es de no aceptación. De manera que, una vez que ha ocurrido un error, no podemos escapar del estado de error, y nunca aceptaremos la cadena.

Ahora volveremos a una serie de ejemplos sobre los DFA, en forma paralela a los ejemplos de la sección anterior.

Ejemplo 2.6 El conjunto de cadenas que contienen exactamente una *b* es aceptado por el siguiente DFA:

Advierta que no nos hemos preocupado por etiquetar los estados. Omitiremos las etiquetas cuando no sea necesario referirse a los estados por nombre. §

Ejemplo 2.7 El conjunto de cadenas que contienen como máximo una *b* es aceptado por el siguiente DFA:

Advierta cómo este DFA es una modificación del DFA del ejemplo anterior, obtenido al crear el estado de inicio como un segundo estado de aceptación. §

Ejemplo 2.8 En la sección anterior proporcionamos definiciones regulares para constantes numéricas en notación científica como se muestra a continuación:

```
nat = [0-9]+
natconSigno = (+|-)? nat
número = natconSigno("." nat)?(E natconSigno)?
```

Nos gustaría escribir DFA para las cadenas coincidentes con estas definiciones, pero es útil volverlas a escribir primero como se ve en seguida:

```
dígito = [0-9]
nat = dígito+
natconSigno = (+|-)? nat
número = natconSigno("." nat)?(E natconSigno)?
```

Es sencillo escribir un DFA para ***nat*** como sigue (recuerde que **a+** = **aa*** para cualquier **a**):

Un ***naturalconSigno*** es un poco más difícil debido al signo opcional. Sin embargo, podemos observar que un ***naturalconSigno*** comienza con un dígito o con un signo y un dígito, y entonces escribir el siguiente DFA:

También es fácil agregar la parte fraccional opcional como se aprecia a continuación:

Observe que conservamos ambos estados de aceptación para reflejar el hecho de que la parte fraccional es opcional.

Por último, necesitamos agregar la parte exponencial opcional. Para hacer esto observamos que la parte exponencial debe comenzar con la letra *E* y puede presentarse sólo después de que hayamos alcanzado cualquiera de los estados de aceptación anteriores. El diagrama final se ofrece en la figura 2.3.

Figura 2.3 Un autómata finito para números de punto flotante

§

Ejemplo 2.9 Los comentarios no anidados se pueden describir utilizando DFA. Por ejemplo, los comentarios encerrados entre llaves son aceptados por el siguiente DFA:

En este caso, `otro` significa todos los caracteres, excepto el de la llave derecha. Este DFA corresponde a la expresión regular `{(~})*}`, el cual escribimos previamente en la sección 2.2.4.

Observamos en esa sección que era difícil escribir una expresión regular para comentarios que estuvieran delimitados mediante una secuencia de dos caracteres, como ocurre con los comentarios en C, que son de la forma `/*...(no */s)...*/`. En realidad es más fácil escribir un DFA que acepte tales comentarios que escribir una expresión regular para ellos. Un DFA para esos comentarios en C se proporciona en la figura 2.4.

Figura 2.4
Un autómata finito para comentarios tipo C

En esta figura la transición `otro` desde el estado 3 hacia sí mismo se permite para todos los caracteres, excepto el asterisco *, mientras que la transición `otro` del estado 4 al estado 3 se permite para todos los caracteres, excepto el "asterisco" * y la "diagonal" /. Numeramos los estados en este diagrama por simplicidad, pero podríamos haberles dado nombres más significativos, como los siguientes (con los números correspondientes entre paréntesis): inicio (1); preparar_comentario (2); entrada_comentario (3); salida_comentario (4) y final (5). §

2.3.2 Búsqueda hacia delante, retroseguimiento y autómatas no determinísticos

Estudiamos los DFA como una manera de representar algoritmos que aceptan cadenas de caracteres de acuerdo con un patrón. Como tal vez el lector ya haya adivinado, existe una fuerte relación entre una expresión regular para un patrón y un DFA que acepta cadenas de acuerdo con el patrón. Investigaremos esta relación en la siguiente sección. Pero primero necesitamos estudiar más detalladamente los algoritmos precisos que representan los DFA, porque en algún momento tendremos que convertirlos en el código para un analizador léxico.

Ya advertimos que el diagrama de un DFA no representa todo lo que necesita un DFA, sino que sólo proporciona un esbozo de su funcionamiento. En realidad, vimos que la definición matemática implica que un DFA debe tener una transición para cada estado y carácter, y que aquellas transiciones que dan errores como resultado por lo regular se dejan fuera del diagrama para el DFA. Pero incluso la definición matemática no describe todos los aspectos del comportamiento de un algoritmo de DFA. Por ejemplo, no especifica lo que ocurre cuando se presenta un error. Tampoco especifica la acción que tomará un programa al alcanzar un estado de aceptación, o incluso cuando iguale un carácter durante una transición.

Una acción típica que ocurre cuando se hace una transición es mover el carácter de la cadena de entrada a una cadena que acumula los caracteres hasta convertirse en un token simple (el valor de cadena del token o lexema del token). Una acción típica cuando se alcanza un estado de aceptación es devolver el token que se acaba de reconocer, junto con cualquier atributo asociado. Una acción típica cuando se alcanza un estado de error es retroceder hacia la entrada (retroseguimiento) o generar un token de error.

Nuestro ejemplo original de un token de identificador muestra gran parte del comportamiento que deseamos describir aquí, y así regresaremos al diagrama de la figura 2.4. El DFA

de esa figura no muestra el comportamiento que queremos de un analizador léxico por varias razones. En primer lugar, el estado de error no es en realidad un error en absoluto, pero representa el hecho de que un identificador no va a ser reconocido (si venimos desde el estado de inicio), o bien, que se ha detectado un delimitador y ahora deberíamos aceptar y generar un token de identificador. Supongamos por el momento (lo que de hecho es el comportamiento correcto) que existen otras transiciones representando las transiciones sin letra desde el estado de inicio. Entonces podemos indicar que se ha detectado un delimitador desde el estado entrada_id, y que debería generarse un token identificador mediante el diagrama de la figura 2.5:

Figura 2.5
Un autómata finito para un identificador con valor de retorno y delimitador

En el diagrama encerramos la transición *otro* entre corchetes para indicar que el carácter delimitador se debería considerar como búsqueda hacia delante, es decir, que debería ser devuelto a la cadena de entrada y no consumido. Además, el estado de error se ha convertido en el estado de aceptación en este diagrama y no hay transiciones fuera del estado de aceptación. Esto es lo que queremos, puesto que el analizador léxico debería reconocer un token a la vez y comenzar de nuevo en su estado de inicio después de reconocer cada token.

Este nuevo diagrama también expresa el principio de la subcadena más larga descrito en la sección 2.2.4: el DFA continúa igualando letras y dígitos (en estado entrada_id) hasta que se encuentra un delimitador. En contraste, el diagrama antiguo permitiría aceptar al DFA en cualquier punto mientras se lee una cadena de identificador, algo que desde luego no queremos que ocurra.

Ahora volvamos nuestra atención a la cuestión de cómo llegar al estado de inicio en primer lugar. En un lenguaje de programación típico existen muchos tokens, y cada token será reconocido por su propio DFA. Si cada uno de estos tokens comienza con un carácter diferente, entonces es fácil conjuntarlos uniendo simplemente todos sus estados de inicio en un solo estado de inicio. Por ejemplo, considere los tokens dados por las cadenas :=, <= e =. Cada uno de éstos es una cadena fija, y los DFA para ellos se pueden escribir como sigue:

Como cada uno de estos tokens comienza con un carácter diferente, podemos sólo identificar sus estados de inicio para obtener el siguiente DFA:

Sin embargo, supongamos que teníamos varios tokens que comenzaban con el mismo carácter, tales como <, <= y <>. Ahora no podemos limitarnos a escribir el diagrama siguiente, puesto que no es un DFA (dado un estado y un carácter, siempre debe haber una transición única hacia un nuevo estado único):

En vez de eso debemos arreglarlo de manera que sólo quede una transición por hacerse en cada estado, tal como en el diagrama siguiente:

En principio, deberíamos poder combinar todos los tokens en un DFA gigante de esta manera. No obstante, la complejidad de una tarea así se vuelve enorme, especialmente si se hace de una manera no sistemática.

Una solución a este problema es ampliar la definición de un autómata finito para incluir el caso en el que pueda existir más de una transición para un carácter particular, mientras que al mismo tiempo se desarrolla un algoritmo para convertir sistemáticamente estos nuevos autómatas finitos generalizados en DFA. Aquí describiremos estos autómatas generalizados, mientras que pospondremos la descripción del algoritmo de traducción hasta la siguiente sección.

La nueva clase de autómatas finitos se denomina **NFA** por las siglas del término **autómata finito no determinístico** en inglés. Antes de definirlo necesitamos otra generalización que será útil en la aplicación de los autómatas finitos a los analizadores léxicos: el concepto de la transición ε.

Una **transición** ε es una transición que puede ocurrir sin consultar la cadena de entrada (y sin consumir ningún carácter). Se puede ver como una "igualación" de la cadena vacía, la cual antes describimos como ε. En un diagrama una transición ε se describe como si ε fuera en realidad un carácter:

Esto no debe confundirse con una correspondencia del carácter ε en la entrada: si el alfabeto incluye este carácter, se debe distinguir del carácter ε que se usa como metacarácter para representar una transición ε.

Las transiciones ε son hasta cierto punto contraintuitivas, porque pueden ocurrir "espontáneamente", es decir, sin búsqueda hacia delante y sin modificación de la cadena de entrada, pero son útiles de dos maneras. Una es porque permiten expresar una selección de alternativas de una manera que no involucre la combinación de estados. Por ejemplo, la selección de los tokens :=, <= e = se puede expresar al combinar los autómatas para cada token como se muestra a continuación:

Esto tiene la ventaja de que mantiene intacto al autómata original y sólo agrega un nuevo estado de inicio para conectarlos. La segunda ventaja de las transiciones ε es que pueden describir explícitamente una coincidencia de la cadena vacía:

Por supuesto, esto es equivalente al siguiente DFA, el cual expresa que la aceptación ocurriría sin hacer coincidir ningún carácter:

Pero es útil tener la notación explícita anterior.

Ahora definiremos qué es un autómata no determinístico, el cual tiene una definición muy similar a la de un DFA, pero en este caso, según el análisis anterior, necesitamos ampliar el alfabeto Σ para incluir a ε. Haremos esto escribiendo $\Sigma \cup \{\varepsilon\}$ (la unión de Σ y ε), donde antes utilizamos Σ (esto supone que ε no es originalmente un miembro de Σ). También necesitamos ampliar la definición de T (la función de transición) de modo que cada carácter pueda conducir a más de un estado. Hacemos esto permitiendo que el valor de T sea un *conjunto* de estados en lugar de un estado simple. Por ejemplo, dado el diagrama

tenemos $T(1, <) = \{2, 3\}$. En otras palabras, desde el estado 1 podemos moverlos a cualesquiera de los estados 2 o 3 en el carácter de entrada $<$, y T se convierte en una función que mapea pares estado/símbolo hacia *conjuntos de estados*. Por consiguiente, el rango de T es el **conjunto de potencia** del conjunto S de estados (el conjunto de todos los subconjuntos de S); escribimos esto como $\wp(S)$ (p manuscrita de S). Ahora estableceremos la definición.

Definición

Un **NFA** (por las siglas del término autómata finito no determinístico en inglés) M consta de un alfabeto Σ, un conjunto de estados S y una función de transición $T: S \times (\Sigma \cup \{\varepsilon\}) \rightarrow \wp(S)$, así como de un estado de inicio s_0 de S y un conjunto de estados de aceptación A de S. El lenguaje aceptado por M, escrito como $L(M)$, se define como el conjunto de cadenas de caracteres $c_1c_2 \ldots c_n$ con cada c_i de $\Sigma \cup \{\varepsilon\}$ tal que existen estados s_1 en $T(s_0, c_1)$, s_2 en $T(s_1, c_2), \ldots, s_n$ en $T(s_{n-1}, c_n)$, con s_n como un elemento de A.

De nuevo necesitamos advertir algunas cosas sobre esta definición. Cualquiera de las c_i en $c_1c_2 \ldots c_n$ puede ser ε, y la cadena que en realidad es aceptada es la cadena $c_1c_2 \ldots c_n$, con la ε eliminada (porque la concatenación de s con ε es s misma). Por lo tanto, la cadena

$c_1c_2 \ldots c_n$ puede en realidad contener menos de n caracteres. Además, la secuencia de estados $s_1, \ldots, s_n$ se elige de los *conjuntos* de estados $T(s_0, c_1), \ldots, T(s_{n-1}, c_n)$, y esta elección no siempre estará determinada de manera única. Ésta, de hecho, es la razón por la que los autómatas se denominan *no determinísticos*: la secuencia de transiciones que acepta una cadena particular no está determinada en cada paso por el estado y el siguiente carácter de entrada. En realidad, pueden introducirse números arbitrarios de ε en cualquier punto de la cadena, los cuales corresponden a cualquier número de transiciones ε en la NFA. De este modo, una NFA no representa un algoritmo. Sin embargo, se puede simular mediante un algoritmo que haga un retroseguimiento a través de cada elección no determinística, como veremos más adelante en esta sección.

No obstante, consideremos primero un par de ejemplos de NFA.

Ejemplo 2.10 Considere el siguiente diagrama de un NFA.

La cadena **abb** puede ser aceptada por cualquiera de las siguientes secuencias de transiciones:

$$\rightarrow 1 \xrightarrow{a} 2 \xrightarrow{b} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4$$

$$\rightarrow 1 \xrightarrow{a} 3 \xrightarrow{\varepsilon} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4$$

En realidad las transiciones del estado 1 al estado 2 en a, y del estado 2 al estado 4 en b, permiten que la máquina acepte la cadena ab, y entonces, utilizando la transición ε del estado 4 al estado 2, todas las cadenas igualan la expresión regular **ab+**. De manera similar, las transiciones del estado 1 al estado 3 en a, y del estado 3 al estado 4 en ε, permiten la aceptación de todas las cadenas que coinciden con **ab***. Finalmente, siguiendo la transición ε desde el estado 1 hasta el estado 4 se permite la aceptación de todas las cadenas coincidentes con **b***. De este modo, este NFA acepta el mismo lenguaje que la expresión regular **ab+|ab*|b***. Una expresión regular más simple que genera el mismo lenguaje es **(a|ε)b***. El siguiente DFA también acepta este lenguaje:

§

Ejemplo 2.11 Considere el siguiente NFA:

Éste acepta la cadena *acab* al efectuar las transiciones siguientes:

De hecho, no es difícil ver que este NFA acepta el mismo lenguaje que el generado por la expresión regular `(a|c)b*`. §

2.3.3 Implementación de autómatas finitos en código

Existen diversas maneras de traducir un DFA o un NFA en código, y las analizaremos en esta sección. Sin embargo, no todos estos métodos serán útiles para un analizador léxico de compilador, y en las últimas dos secciones de este capítulo se demostrarán los aspectos de la codificación apropiada para analizadores léxicos con más detalle.

Considere de nueva cuenta nuestro ejemplo original de un DFA que acepta identificadores compuestos de una letra seguida por una secuencia de letras y/o dígitos, en su forma corregida que incluye búsqueda hacia delante y el principio de la subcadena más larga (véase la figura 2.5):

La primera y más sencilla manera de simular este DFA es escribir el código de la manera siguiente:

```
{ iniciando en el estado 1 }
if el siguiente carácter es una letra then
  avanzar en la entrada;
  { ahora en el estado 2 }
  while el siguiente carácter es una letra o un dígito do
```

(continúa)

```
      avanza en la entrada; { permanece en el estado 2 }
    end while;
    { ir al estado 3 sin avanzar en la entrada }
    aceptar;
  else
    { error u otros casos }
  end if;
```

Tales códigos utilizan la posición en el código (anidado dentro de pruebas) para mantener el estado implícitamente, como indicamos mediante los comentarios. Esto es razonable si no hay demasiados estados (lo que requeriría muchos niveles de anidación), y si las iteraciones en el DFA son pequeños. Un código como éste se ha utilizado para escribir pequeños analizadores léxicos. Pero existen dos desventajas con este método. La primera es que es de propósito específico, es decir, cada DFA se tiene que tratar de manera un poco diferente, y es difícil establecer un algoritmo que traduzca cada DFA a código de esta manera. La segunda es que la complejidad del código aumenta de manera dramática a medida que el número de estados se eleva o, más específicamente, a medida que se incrementa el número de estados diferentes a lo largo de trayectorias arbitrarias. Como un ejemplo simple de estos problemas, consideremos el DFA del ejemplo 2.9 como se dio en la figura 2.4 (página 53) que acepta comentarios en C, el cual se podría implementar por medio de código de la forma siguiente:

```
{ estado 1 }
if el siguiente carácter es "/" then
  avanzan en la entrada; { estado 2 }
  if el siguiente carácter es "*" then
    adelantar la entrada; { estado 3 }
    hecho := false;
    while not hecho do
      while el siguiente carácter de entrada no es "*" do
        avanza en la entrada;
      end while;
      avanza en la entrada; { estado 4 }
      while el siguiente carácter de entrada es "*" do
        avanza en la entrada;
      end while;
      if el siguiente carácter de entrada es "/" then
        hecho := true;
      end if;
      avanza en la entrada;
    end while;
    aceptar; { estado 5 }
  else { otro procesamiento }
  end if;
else { otro procesamiento }
end if;
```

Advierta el considerable incremento en complejidad, y la necesidad de lidiar con la iteración que involucra los estados 3 y 4 mediante el uso de la variable booleana *hecho*.

Un método de implementación sustancialmente mejor se obtiene al utilizar una variable para mantener el estado actual y escribir las transiciones como una sentencia case doblemente anidada dentro de una iteración, donde la primera sentencia case prueba el estado actual y el segundo nivel anidado prueba el carácter de entrada, lo que da el estado. Por ejemplo, el DFA anterior para identificadores se puede traducir en el esquema de código de la figura 2.6.

Figura 2.6
Implementación del identificador DFA utilizando una variable de estado y pruebas case anidadas

```
estado := 1; { inicio }
while estado = 1 o 2 do
  case estado of
  1:  case carácter de entrada of
      letra: avanza en la entrada;
             estado := 2;
      else estado := . . . { error u otro };
      end case;
  2:  case carácter de entrada of
      letra, dígito: avanza en la entrada;
                     estado := 2; { realmente innecesario }
      else estado := 3;
      end case;
  end case;
end while;
if estado = 3 then aceptar else error ;
```

Observe cómo este código refleja el DFA de manera directa: las transiciones corresponden a la asignación de un nuevo estado a la variable *estado* y avanzan en la entrada (excepto en el caso de la transición "que no consume" del estado 2 al estado 3).

Ahora el DFA para los comentarios en C (figura 2.4) se puede traducir en el esquema de código más legible de la figura 2.7. Una alternativa para esta organización es tener el case exterior basado en el carácter de entrada y los case internos basados en el estado actual (véanse los ejercicios).

En los ejemplos que acabamos de ver, el DFA se "conectó" directamente dentro del código. También es posible expresar el DFA como una estructura de datos y entonces escribir un código "genérico" que tomará sus acciones de la estructura de datos. Una estructura de datos simple que es adecuada para este propósito es una **tabla de transición**, o arreglo bidimensional, indizado por estado y carácter de entrada que expresa los valores de la función de transición T:

	Caracteres en el alfabeto c
Estados s	Estados representando transiciones $T(s, c)$

Como ejemplo, el DFA para identificadores se puede representar como la siguiente tabla de transición:

estado \ carácter de entrada	letra	dígito	otro
1	2		
2	2	2	3
3			

Figura 2.7
Implementación del DFA de la figura 2.4

```
estado := 1; { inicio }
while estado = 1, 2, 3 o 4 do
  case estado of
  1:  case carácter de entrada of
      "/" : avanza en la entrada;
            estado := 2;
      else estado := . . . { error u otro };
      end case;
  2:  case carácter de entrada of
      "*": avanza en la entrada;
            estado := 3;
      else estado := . . . { error u otro };
      end case;
  3:  case carácter de entrada of
      "*": avanza en la entrada;
            estado := 4;
      else advance the input { y permanecer en el estado 3 };
      end case;
  4:  case carácter de entrada of
      "/" avanza en la entrada;
            estado := 5;
      "*": avanza en la entrada; { y permanecer en el estado 4 }
      else avanza en la entrada;
            estado := 3;
      end case;
  end case;
end while;
if estado = 5 then aceptar else error ;
```

En esta tabla las entradas en blanco representan transiciones que no se muestran en el diagrama DFA (es decir, representan transiciones a estados de error u otros procesamientos). También suponemos que el primer estado que se muestra es el estado de inicio. Sin embargo, esta tabla no indica cuáles estados aceptados y cuáles transiciones no consumen sus entradas. Esta información puede conservarse en la misma estructura de datos que representa la tabla o en una estructura de datos por separado. Si agregamos esta información a la tabla de transición anterior (utilizando una columna por separado para indicar estados de aceptación y corchetes para señalar transiciones "no consumidoras de entrada"), obtenemos la tabla siguiente:

estado \ carácter de entrada	letra	dígito	otro	Aceptación
1	2			no
2	2	2	[3]	no
3				sí

Como un segundo ejemplo de una tabla de transición, presentamos la tabla para el DFA correspondiente a los comentarios en C (nuestro segundo ejemplo presentado con anterioridad):

estado \ carácter de entrada	/	*	otro	*Aceptación*
1	2			no
2		3		no
3	3	4	3	no
4	5	4	3	no
5				sí

Ahora podemos escribir del código en una forma que implementará cualquier DFA, dadas las entradas y estructuras de datos apropiadas. El siguiente esquema de código supone que las transiciones se mantienen en un arreglo de transición *T* indizado por estados y carácter de entrada; que las transiciones que avanzan en la entrada (es decir, aquellas que no están marcadas con corchetes en la tabla) están dadas por el arreglo booleano *Avanzar*, indizado también por estados y caracteres de entrada; y que los estados de aceptación están dados por el arreglo booleano *Aceptar*, indizado por estados. El siguiente es el esquema de código:

```
estado := 1;
ch := siguiente carácter de entrada;
while not Aceptar[estado] and not error(estado) do
  nuevoestado := T[estado,ch];
  if Avanzar[estado,ch] then ch := siguiente carácter de entrada;
  estado := nuevoestado;
end while;
if Aceptar[estado] then aceptar;
```

Los métodos algorítmicos como los que acabamos de describir se denominan **controlados por tabla** porque emplean tablas para dirigir el progreso del algoritmo. Los métodos controlados por tabla tienen ciertas ventajas: el tamaño del código se reduce, el mismo código funcionará para muchos problemas diferentes y el código es más fácil de modificar (mantener). La desventaja es que las tablas pueden volverse muy grandes y provocar un importante aumento en el espacio utilizado por el programa. En realidad, gran parte del espacio en los arreglos que acabamos de describir se desperdicia. Por lo tanto, los métodos controlados por tabla a menudo dependen de métodos de compresión de tablas, tales como representaciones de arreglos dispersos, aunque por lo regular existe una penalización en tiempo que se debe pagar por dicha compresión, ya que la búsqueda en tablas se vuelve más lenta. Como los analizadores léxicos deben ser eficientes, rara vez se utilizan estos métodos para ellos, aunque se pueden emplear en programas generadores de analizadores léxicos tales como Lex. No los estudiaremos más aquí.

Por último, advertimos que los NFA se pueden implementar de maneras similares a los DFA, excepto que como los NFA son no determinísticos, tienen muchas secuencias diferentes de transiciones en potencia que se deben probar. Por consiguiente, un programa que simula un NFA debe almacenar transiciones que todavía no se han probado y realizar el retroseguimiento a ellas en caso de falla. Esto es muy similar a los algoritmos que intentan encontrar trayectorias en gráficas dirigidas, sólo que la cadena de entrada guía la búsqueda.

Como los algoritmos que realizan una gran cantidad de retroseguimientos tienden a ser poco eficientes, y un analizador léxico debe ser tan eficiente como sea posible, ya no describiremos tales algoritmos. En su lugar, se puede resolver el problema de simular un NFA por medio del método que estudiaremos en la siguiente sección, el cual convierte un NFA en un DFA. Así que pasaremos a esta sección, donde regresaremos brevemente al problema de simular un NFA.

2.4 DESDE LAS EXPRESIONES REGULARES HASTA LOS DFA

En esta sección estudiaremos un algoritmo para traducir una expresión regular en un DFA. También existe un algoritmo para traducir un DFA en una expresión regular, de manera que las dos nociones son equivalentes. Sin embargo, debido a lo compacto de las expresiones regulares, se suele preferir a los DFA como descripciones de token, y de este modo la generación del analizador léxico comienza comúnmente con expresiones regulares y se sigue a través de la construcción de un DFA hasta un programa de analizador léxico final. Por esta razón, nuestro interés se enfocará sólo en un algoritmo que realice esta dirección de la equivalencia.

El algoritmo más simple para traducir una expresión regular en un DFA pasa por una construcción intermedia, en la cual se deriva un NFA de la expresión regular, y posteriormente se emplea para construir un DFA equivalente. Existen algoritmos que pueden traducir una expresión regular de manera directa en un DFA, pero son más complejos y la construcción intermedia también es de cierto interés. Así que nos concentraremos en la descripción de dos algoritmos, uno que traduce una expresión regular en un NFA y el segundo que traduce un NFA en un DFA. Combinado con uno de los algoritmos para traducir un DFA en un programa descrito en la sección anterior, el proceso de la construcción de un analizador léxico se puede automatizar en tres pasos, como se ilustra mediante la figura que se ve a continuación:

2.4.1 Desde una expresión regular hasta un NFA

La construcción que describiremos se conoce como la **construcción de Thompson**, en honor a su inventor. Utiliza transiciones ε para "pegar" las máquinas de cada segmento de una expresión regular con el fin de formar una máquina que corresponde a la expresión completa. De este modo, la construcción es inductiva, y sigue la estructura de la definición de una expresión regular: mostramos un NFA para cada expresión regular básica y posteriormente mostramos cómo se puede conseguir cada operación de expresión regular al conectar entre sí los NFA de las subexpresiones (suponiendo que éstas ya se han construido).

Expresiones regulares básicas Una expresión regular básica es de la forma **a**, $\boldsymbol{\varepsilon}$ o $\boldsymbol{\phi}$, donde **a** representa una correspondencia con un carácter simple del alfabeto, $\boldsymbol{\varepsilon}$ representa una coincidencia con la cadena vacía y $\boldsymbol{\phi}$ representa la correspondencia con ninguna cadena. Un NFA que es equivalente a la expresión regular **a** (es decir, que acepta precisamente aquellas cadenas en su lenguaje) es

De manera similar, un NFA que es equivalente a ε es

El caso de la expresión regular ϕ (que nunca ocurre en la práctica en un compilador) se deja como ejercicio.

Concatenación Deseamos construir un NFA equivalente a la expresión regular rs, donde r y s son expresiones regulares. Suponemos (de manera inductiva) que los NFA equivalentes a r y s ya se construyeron. Expresamos esto al escribir

para el NFA correspondiente a r, y de manera similar para s. En este dibujo, el círculo a la izquierda dentro del rectángulo redondeado indica el estado de inicio, el círculo doble a la derecha indica el estado de aceptación y los tres puntos indican los estados y transiciones dentro del NFA que no se muestran. Esta figura supone que el NFA correspondiente a r tiene sólo un estado de aceptación. Esta suposición se justificará si todo NFA que construyamos tiene un estado de aceptación. Esto es verdadero para los NFA de expresiones regulares básicas, y será cierto para cada una de las construcciones siguientes.

Ahora podemos construir un NFA correspondiente a rs de la manera siguiente:

Conectamos el estado de aceptación de la máquina de r al estado de inicio de la máquina de s mediante una transición ε. La nueva máquina tiene el estado de inicio de la máquina de r como su estado de inicio y el estado de aceptación de la máquina de s como su estado de aceptación. Evidentemente, esta máquina acepta $L(rs) = L(r)L(s)$ y de este modo corresponde a la expresión regular rs.

Selección entre alternativas Deseamos construir un NFA correspondiente a $r \mid s$ bajo las mismas suposiciones que antes. Hacemos esto como se ve a continuación:

Agregamos un nuevo estado de inicio y un nuevo estado de aceptación y los conectamos como se muestra utilizando transiciones ε. Evidentemente, esta máquina acepta el lenguaje $L(r|s) = L(r) \cup L(s)$.

Repetición Queremos construir una máquina que corresponda a r^*, dada una máquina que corresponda a r. Haremos esto como se muestra enseguida:

Aquí nuevamente agregamos dos nuevos estados, un estado de inicio y un estado de aceptación. La repetición en esta máquina la proporciona la nueva transición ε del estado de aceptación de la máquina de r a su estado de inicio. Esto permite a la máquina de r ser atravesada una o más veces. Para asegurar que la cadena vacía también es aceptada (correspondiente a las cero repeticiones de r), también debemos trazar una transición ε desde el nuevo estado de inicio hasta el nuevo estado de aceptación.

Esto completa la descripción de la construcción de Thompson. Advertimos que esta construcción no es única. En particular, otras construcciones son posibles cuando se traducen operaciones de expresión regular en NFA. Por ejemplo, al expresar la concatenación rs, podríamos haber eliminado la transición ε entre las máquinas de r y s e identificar en su lugar el estado de aceptación de la máquina de r con el estado de inicio de la máquina de s, como se ilustra a continuación:

(Sin embargo, esta simplificación depende del hecho que en las otras construcciones el estado de aceptación no tiene transiciones desde ella misma hacia otros estados: véanse los ejercicios.) Otras simplificaciones son posibles en los otros casos. La razón por la que expresamos las traducciones a medida que las tenemos es que las máquinas se construyen según reglas muy simples. En primer lugar, cada estado tiene como máximo dos transiciones, y si hay dos, ambas deben ser transiciones ε. En segundo lugar, ningún estado se elimina una vez que se construye, y ninguna transición se modifica, excepto para la adición de transiciones desde el estado de aceptación. Estas propiedades hacen muy fácil automatizar el proceso.

Concluimos el análisis de la construcción de Thompson con algunos ejemplos.

Ejemplo 2.12

Traduciremos la expresión regular `ab|a` en un NFA de acuerdo con la construcción de Thompson. Primero formamos las máquinas para las expresiones regulares básicas `a` y `b`:

Entonces formamos la máquina para la concatenación `ab`:

Ahora formamos otra copia de la máquina para `a` y utilizamos la construcción para selección con el fin de obtener el NFA completo para `ab|a` que se muestra en la figura 2.8.

Figura 2.8
NFA para la expresión regular `ab|a` utilizando la construcción de Thompson

§

Ejemplo 2.13

Formamos el NFA de la construcción de Thompson para la expresión regular `letra-(letra|dígito)*`. Como en el ejemplo anterior, formamos las máquinas para las expresiones regulares `letra` y `dígito`:

Después formamos la máquina para la selección `letra|dígito`:

Ahora formamos la NFA para la repetición `(letra|dígito)*` como se ve a continuación:

Por último construimos la máquina para la concatenación de `letra` con `(letra|dígito)*` para obtener el NFA completo, como se ilustra en la figura 2.9.

Figura 2.9
NFA para la expresión regular `letra-(letra|dígito)*` utilizando la construcción de Thompson

§

Como ejemplo final, observamos que el NFA del ejemplo 2.11 (sección 2.3.2) corresponde exactamente a la expresión regular `(a|c)*b` bajo la construcción de Thompson.

2.4.2 Desde un NFA hasta un DFA

Ahora deseamos describir un algoritmo que, dado un NFA arbitrario, construirá un DFA equivalente (es decir, uno que acepte precisamente las mismas cadenas). Para hacerlo necesitaremos algún método con el que se eliminen tanto las transiciones ε como las transiciones múltiples de un estado en un carácter de entrada simple. La eliminación de las transiciones ε implica el construir **cerraduras ε**, las cuales son el conjunto de todos los estados que pueden alcanzar las transiciones ε desde un estado o estados. La eliminación de transiciones múltiples en un carácter de entrada simple implica mantenerse al tanto del conjunto de estados que son alcanzables al igualar un carácter simple. Ambos procesos nos conducen a considerar conjuntos de estados en lugar de estados simples. De este modo, no es ninguna sorpresa que el DFA que construimos tenga como sus estados los *conjuntos de estados* del NFA original. En consecuencia, este algoritmo se denomina **construcción de subconjuntos**. Primero analizaremos la cerradura ε con un poco más de detalle y posteriormente continuaremos con una descripción de la construcción de subconjuntos.

La cerradura ε de un conjunto de estados Definimos la cerradura ε de un estado simple s como el conjunto de estados alcanzables por una serie de cero o más transiciones ε, y escribimos este conjunto como $\bar{s}$. Dejaremos una afirmación más matemática de esta definición para un ejercicio y continuaremos directamente con un ejemplo. Sin embargo, advierta que la cerradura ε de un estado siempre contiene al estado mismo.

Ejemplo 2.14 Considere el NFA siguiente que corresponde a la expresión regular `a*` bajo la construcción de Thompson:

En este NFA tenemos $\bar{1} = \{1, 2, 4\}$, $\bar{2} = \{2\}$, $\bar{3} = \{2, 3, 4\}$ y $\bar{4} = \{4\}$. §

Ahora definiremos la cerradura ε de un conjunto de estados como la unión de las cerraduras ε de cada estado individual. En símbolos, si S es un conjunto de estados, entonces tenemos que

$$\bar{S} = \bigcup_{s \text{ en } S} \bar{s}$$

Por ejemplo, en el NFA del ejemplo 2.14, $\overline{\{1, 3\}} = \overline{1} \cup \overline{3} = \{1, 2, 4\} \cup \{2, 3, 4\} = \{1, 2, 3, 4\}$.

La construcción del subconjunto Ahora estamos en posición de describir el algoritmo para la construcción de un DFA a partir de un NFA M dado, al que denominaremos $\overline{M}$. Primero calculamos la cerradura ε del estado de inicio de M; esto se convierte en el estado de inicio de $\overline{M}$. Para este conjunto, y para cada conjunto subsiguiente, calculamos las transiciones en los caracteres a como sigue. Dado un conjunto S de estados y un carácter a en el alfabeto, calculamos el conjunto $S'_a = \{ t \mid$ para alguna s en S existe una transición de s a t en $a \}$. Después calculamos $\overline{S'_a}$, la cerradura ε de S'_a. Esto define un nuevo estado en la construcción del subconjunto, junto con una nueva transición $S \xrightarrow{a} \overline{S'_a}$. Continuamos con este proceso hasta que no se crean nuevos estados o transiciones. Marcamos como de aceptación aquellos estados construidos de esta manera que contengan un estado de aceptación de M. Éste es el DFA $\overline{M}$. No contiene transiciones ε debido a que todo estado es construido como una cerradura ε. Contiene como máximo una transición desde un estado en un carácter a porque cada nuevo estado se construye de *todos* los estados de M alcanzables por transiciones desde un estado en un carácter simple a.

Ilustraremos la construcción del subconjunto con varios ejemplos.

Ejemplo 2.15 Considere el NFA del ejemplo 2.14. El estado de inicio del DFA correspondiente es $\overline{1} = \{1, 2, 4\}$. Existe una transición desde el estado 2 hasta el estado 3 en a, y no hay transiciones desde los estados 1 o 4 en a, de manera que hay una transición en a desde $\{1, 2, 4\}$ hasta $\overline{\{1, 2, 4\}_a} = \overline{\{3\}} = \{2, 3, 4\}$. Como no hay transiciones adicionales en un carácter desde cualquiera de los estados 1, 2 o 4, volveremos nuestra atención hacia el nuevo estado $\{2, 3, 4\}$. De nueva cuenta existe una transición desde 2 a 3 en a y ninguna transición a desde 3 o 4, de modo que hay una transición desde $\{2, 3, 4\}$ hasta $\overline{\{2, 3, 4\}_a} = \overline{\{3\}} = \{2, 3, 4\}$. De manera que existe una transición a desde $\{2, 3, 4\}$ hacia sí misma. Agotamos los estados a considerar, y así construimos el DFA completo. Sólo resta advertir que el estado 4 del NFA es de aceptación, y puesto que tanto $\{1, 2, 4\}$ como $\{2, 3, 4\}$ contienen al 4, ambos son estados de aceptación del DFA correspondiente. Dibujamos el DFA que construimos como sigue, donde nombramos a los estados mediante sus subconjuntos:

(Una vez que se termina la construcción, podríamos descartar la terminología de subconjuntos si así lo deseamos.) §

Ejemplo 2.16 Consideremos el NFA de la figura 2.8, al cual agregaremos números de estado:

La construcción del subconjunto de DFA tiene como su estado de inicio $\overline{\{1\}} = \{1, 2, 6\}$. Existe una transición en a desde el estado 2 hasta el estado 3, y también desde el estado 6 hasta el estado 7. De este modo, $\overline{\{1, 2, 6\}}_a = \overline{\{3, 7\}} = \{3, 4, 7, 8\}$, y tenemos $\{1, 2, 6\} \xrightarrow{a} \{3, 4, 7, 8\}$. Como no hay otras transiciones de carácter desde 1, 2 o 6, vayamos a $\{3, 4, 7, 8\}$. Existe una transición en b desde 4 hasta 5 y $\overline{\{3, 4, 7, 8\}}_b = \overline{\{5\}} = \{5, 8\}$, y tenemos la transición $\{3, 4, 7, 8\} \xrightarrow{b} \{5, 8\}$. No hay otras transiciones. Así que la construcción del subconjunto produce el siguiente DFA equivalente al anterior NFA:

§

Ejemplo 2.17 Considere el NFA de la figura 2.9 (la construcción de Thompson para la expresión regular `letra(letra|dígito)*`):

La construcción del subconjunto continúa como se explica a continuación. El estado de inicio es $\overline{\{1\}} = \{1\}$. Existe una transición en ***letra*** para $\overline{\{2\}} = \{2, 3, 4, 5, 7, 10\}$. Desde este estado existe una transición en ***letra*** para $\overline{\{6\}} = \{4, 5, 6, 7, 9, 10\}$ y una transición en ***dígito*** para $\overline{\{8\}} = \{4, 5, 7, 8, 9, 10\}$. Finalmente, cada uno de estos estados también tiene transiciones en ***letra*** y ***dígito***, ya sea hacia sí mismos o hacia el otro. El DFA completo se ilustra en la siguiente figura:

§

2.4.3 Simulación de un NFA utilizando la construcción de subconjuntos

En la sección anterior analizamos brevemente la posibilidad de escribir un programa para simular un NFA, una cuestión que requiere tratar con la no determinación, o naturaleza no algorítmica, de la máquina. Una manera de simular un NFA es utilizar la construcción de subconjuntos, pero en lugar de construir todos los estados del DFA asociado, construimos solamente el estado en cada punto que indica el siguiente carácter de entrada. De este modo, construimos sólo aquellos conjuntos de estados que se presentan en realidad en una trayectoria a través del DFA que se toma en la cadena de entrada proporcionada. La ventaja de esto es que podemos no necesitar construir el DFA completo. La desventaja es que, si la trayectoria contiene bucles, un estado se puede construir muchas veces.

Pongamos por caso el ejemplo 2.16, donde si tenemos la cadena de entrada compuesta del carácter simple *a*, construiremos el estado de inicio {1, 2, 6} y luego el segundo estado {3, 4, 7, 8} hacia el cual nos movemos e igualamos la *a*. Entonces, como no hay *b* a continuación, aceptamos sin generar incluso el estado {5, 8}.

Por otra parte, en el ejemplo 2.17, dada la cadena de entrada *r2d2*, tenemos la siguiente secuencia de estados y transiciones:

$$\{1\} \xrightarrow{\mathtt{r}} \{2, 3, 4, 5, 7, 10\} \xrightarrow{\mathtt{2}} \{4, 5, 7, 8, 9, 10\}$$
$$\xrightarrow{\mathtt{d}} \{4, 5, 6, 7, 9, 10\} \xrightarrow{\mathtt{2}} \{4, 5, 7, 8, 9, 10\}$$

Si estos estados se construyen a medida que se presentan las transiciones, entonces todos los estados del DFA están construidos y el estado {4, 5, 7, 8, 9, 10} incluso se construyó dos veces. Así que este proceso es menos eficiente que construir el DFA completo en primer lugar. Es por esto que no se realiza la simulación de los NFA en analizadores léxicos. Queda una opción para igualar patrones en programas de búsqueda y editores, donde el usuario puede dar las expresiones regulares de manera dinámica.

2.4.4 Minimización del número de estados en un DFA

El proceso que describimos para derivar un DFA de manera algorítmica a partir de una expresión regular tiene la desafortunada propiedad de que el DFA resultante puede ser más complejo de lo necesario. Como un caso ilustrativo, en el ejemplo 2.15 derivamos el DFA

para la expresión regular `a*`, mientras que el DFA

también lo hará. Como la eficiencia es muy importante en un analizador léxico, nos gustaría poder construir, si es posible, un DFA que sea mínimo en algún sentido. De hecho, un resultado importante de la teoría de los autómatas establece que, dado cualquier DFA, existe un DFA equivalente que contiene un número mínimo de estados, y que este DFA de estados mínimos o mínimo es único (excepto en el caso de renombrar estados). También es posible obtener directamente este DFA mínimo de cualquier DFA dado, y aquí describiremos brevemente el algoritmo, sin demostrar que en realidad construye el DFA mínimo equivalente (debería ser fácil para el lector convencerse de esto de manera informal al leer el algoritmo).

El algoritmo funciona al crear conjuntos de estados que se unificarán en estados simples. Comienza con la suposición más optimista posible: crea dos conjuntos, uno compuesto de todos los estados de aceptación y el otro compuesto de todos los estados de no aceptación. Dada esta partición de los estados del DFA original, considere las transiciones en cada carácter a del alfabeto. Si todos los estados de aceptación tienen transiciones en a para estados de aceptación, entonces esto define una transición a desde el nuevo estado de aceptación (el conjunto de todos los estados de aceptación antiguos) hacia sí misma. De manera similar, si todos los estados de aceptación tienen transiciones en a hacia estados de no aceptación, entonces esto define una transición a desde el nuevo estado de aceptación hacia el nuevo estado de no aceptación (el conjunto de todos los estados de no aceptación antiguos). Por otra parte, si hay dos estados de aceptación s y t que tengan transiciones en a que caigan en diferentes conjuntos, entonces ninguna transición a se puede definir para este agrupamiento de los estados. Decimos que a **distingue** los estados s y t. En este caso, el conjunto de estados que se está considerando (es decir, el conjunto de todos los estados de aceptación) se debe dividir de acuerdo con el lugar en que caen sus transiciones a. Por supuesto, declaraciones similares se mantienen para cada uno de los otros conjuntos de estados, y una vez que hemos considerado todos los caracteres del alfabeto, debemos movernos a ellos. Por supuesto, si cualquier conjunto adicional se divide, debemos regresar y repetir el proceso desde el principio. Continuamos este proceso de refinar la partición de los estados del DFA original en conjuntos hasta que todos los conjuntos contengan sólo un elemento (en cuyo caso, mostramos que el DFA original es mínimo) o hasta que no se presente ninguna división adicional de conjuntos.

Para que el proceso que acabamos de describir funcione correctamente, también debemos considerar transiciones de error a un estado de error que es de no aceptación. Esto es, si existen estados de aceptación s y t, tales que s tenga una transición a hacia otro estado de aceptación, mientras que t no tenga ninguna transición a (es decir, una transición de error), entonces a distingue s y t. De la misma manera, si un estado de no aceptación s tiene una transición a hacia un estado de aceptación, mientras que otro estado de no aceptación t no tiene transición a, entonces a distingue s y t también en este caso.

Concluiremos nuestro análisis de la minimización de estado con un par de ejemplos.

Ejemplo 2.18

Consideremos el DFA que construimos en el ejemplo anterior, correspondiente a la expresión regular `letra(letra|dígito)*`. Tenía cuatro estados compuestos del estado inicial y tres estados de aceptación. Los tres estados de aceptación tienen transiciones a otros estados de aceptación tanto en `letra` como en `dígito` y ninguna otra transición (sin errores). Por consiguiente, los tres estados de aceptación no se pueden distinguir por ningún carácter, y el algoritmo de minimización da como resultado la combinación de los tres estados de aceptación en uno, dejando el siguiente DFA mínimo (el cual ya vimos al principio de la sección 2.3):

§

Ejemplo 2.19

Considere el siguiente DFA, que dimos en el ejemplo 2.1 (sección 2.3.2) como equivalente a la expresión regular `(a|ε)b*`:

En este caso todos los estados (excepto el estado de error) son de aceptación. Consideremos ahora el carácter *b*. Cada estado de aceptación tiene una transición *b* a otro estado de aceptación, de modo que ninguno de los estados se distingue por *b*. Por otro lado, el estado 1 tiene una transición *a* hacia un estado de aceptación, mientras que los estados 2 y 3 no tienen transición *a* (o, mejor dicho, una transición de error en *a* hacia el estado de no aceptación de error). Así, *a* distingue el estado 1 de los estados 2 y 3, y debemos repartir los estados en los conjuntos {1} y {2, 3}. Ahora comenzamos de nuevo. El conjunto {1} no se puede dividir más, así que ya no lo consideraremos. Ahora los estados 2 y 3 no se pueden distinguir por *a* o *b*. De esta manera, obtenemos el DFA de estado mínimo:

§

2.5 IMPLEMENTACIÓN DE UN ANALIZADOR LÉXICO TINY ("DIMINUTO")

Ahora deseamos desarrollar el código real para un analizador léxico con el fin de ilustrar los conceptos estudiados hasta ahora en este capítulo. Haremos esto para el lenguaje TINY que se presentó de manera informal en el capítulo 1 (sección 1.7). Después analizaremos diversas cuestiones de implementación práctica planteadas por este analizador léxico.

2.5.1 Implementación de un analizador léxico para el lenguaje de muestra TINY

En el capítulo 1 expusimos brevemente y de manera informal el lenguaje TINY. Nuestra tarea aquí es especificar por completo la estructura léxica de TINY, es decir, definir los tokens y sus atributos. Los tokens y las clases de tokens de TINY se resumen en la tabla 2.1.

Los tokens de TINY caen dentro de tres categorías típicas: palabras reservadas, símbolos especiales y "otros" tokens. Existen ocho palabras reservadas, con significados familiares (aunque no necesitamos conocer sus semánticas hasta mucho después). Existen 10 símbolos especiales, que dan las cuatro operaciones aritméticas básicas con enteros, dos operaciones de comparación (igual y menor que), paréntesis, signo de punto y coma y asignación. Todos los símbolos especiales tienen un carácter de longitud, excepto el de asignación, que tiene dos.

Tabla 2.1
Tokens del lenguaje TINY

Palabras reservadas	Símbolos especiales	Otros
`if`	`+`	`número`
`then`	`-`	(1 o más
`else`	`*`	dígitos)
`end`	`/`	
`repeat`	`=`	
`until`	`<`	*`identificador`*
`read`	`(`	(1 o más
`write`	`)`	letras)
	`;`	
	`:=`	

Los otros tokens son números, que son secuencias de uno o más dígitos e identificadores, los cuales (por simplicidad) son secuencias de una o más letras.

Además de los tokens, TINY tiene las siguientes convenciones léxicas. Los comentarios están encerrados entre llaves {...} y no pueden estar anidados; el código es de formato libre; el espacio en blanco se compone de blancos, tabulaciones y retornos de línea; y se sigue el principio de la subcadena más larga en el reconocimiento de tokens.

Al diseñar un analizador léxico para este lenguaje, podríamos comenzar con expresiones regulares y desarrollar NFA y DFA de acuerdo con los algoritmos de la sección anterior. En realidad las expresiones regulares se han dado anteriormente para números, identificadores y comentarios (TINY tiene versiones particularmente simples de éstos). Las expresiones regulares para los otros tokens no son importantes porque todos son cadenas

fijas. En lugar de seguir esta ruta desarrollaremos un DFA para el analizador léxico directamente, ya que los tokens son muy simples. Haremos esto en varios pasos.

En primer lugar, advertimos que todos los símbolos especiales, excepto los de asignación, son caracteres únicos distintos, y un DFA para estos símbolos tendría el aspecto que sigue:

En este diagrama, los diferentes estados de aceptación distinguen el token que el analizador léxico está por devolver. Si empleamos algún otro indicador para el token que será devuelto (digamos, una variable en el código), entonces todos los estados de aceptación se pueden colapsar en un estado que denominaremos **HECHO**. Si combinamos este DFA de dos estados con DFA que acepten números identificadores, obtenemos el DFA siguiente:

Advierta el uso de los corchetes para indicar caracteres de búsqueda hacia delante que no deberán ser consumidos.

Necesitamos agregar comentarios, espacio en blanco y asignación a este DFA. El espacio en blanco es consumido mediante un bucle simple desde el estado de inicio hacia sí mismo. Los comentarios requieren un estado extra, que se alcanza desde el estado de inicio en la llave izquierda y regresa a él en la llave derecha. La asignación también requiere de un estado intermedio, que se alcanza desde el estado de inicio en el signo de punto y coma. Si sigue inmediatamente un signo de igualdad, entonces se genera un token de asignación. De otro modo, el siguiente carácter no debería ser consumido, y se genera un token de error.

Figura 2.10
DFA del analizador léxico de TINY

De hecho, todos los caracteres simples que no están en la lista de símbolos especiales, que no son espacios en blanco o comentarios, y que no son dígitos o letras, se deberán aceptar como errores, y toleraremos éstos con los símbolos de carácter simple. El DFA final para nuestro analizador léxico se puede apreciar en la figura 2.10.

No incluimos palabras reservadas en nuestro análisis o en el DFA de la figura 2.10. Esto es porque es más fácil desde el punto de vista del DFA considerar que las palabras reservadas son lo mismo que los identificadores, y entonces consultar los identificadores en una tabla de palabras reservadas después de la aceptación. En realidad, el principio de la subcadena más larga garantiza que la única acción del analizador léxico que necesita modificarse es el token que es devuelto. De este modo, las palabras reservadas se consideran sólo después de que se ha reconocido un identificador.

Volvamos ahora a un análisis del código para implementar este DFA, el cual está contenido en los archivos **`scan.h`** y **`scan.c`** (véase el apéndice B). El procedimiento principal es **`getToken`** (líneas 674-793), el cual consume caracteres de entrada y devuelve el siguiente token reconocido de acuerdo con el DFA de la figura 2.10. La implementación utiliza el análisis case doblemente anidado que describimos en la sección 2.3.3, con una gran lista case basada en el estado, dentro de la cual se encuentran listas case individuales basadas en el carácter de entrada actual. Los tokens mismos están definidos como un tipo enumerado en **`globals.h`** (líneas 174-186), el cual incluye todos los tokens enumerados en la tabla 2.1, junto con los tokens de administración **`EOF`** (cuando el fin del archivo se alcanza) y **`ERROR`** (cuando se encuentra un carácter erróneo). Los estados del analizador léxico también están definidos como un tipo enumerado, pero dentro del analizador mismo (líneas 612-614).

Un analizador léxico también necesita en general calcular los atributos, si existen, de cada token, y en ocasiones tomar también otras acciones (tales como la inserción de identificadores en una tabla de símbolos). En el caso del analizador léxico de TINY, el único atributo que se calcula es el lexema, o valor de cadena del token reconocido, y éste se coloca en la variable **`tokenString`**. Esta variable, junto con **`getToken`**, son los únicos servicios ofrecidos a otras partes del compilador, y sus definiciones están reunidas en el archivo de cabecera **`scan.h`** (líneas 550-571). Observe que **`tokenString`** se declara con una longitud fija de 41, así que los identificadores, por ejemplo, no pueden tener más de 40 caracteres (más el carácter nulo de terminación). Ésta es una limitación que se analizará posteriormente.

El analizador léxico emplea tres variables globales: las variables de archivo **source** y **listing**, y la variable entera **lineno**, que están declaradas en **globals.h** y asignadas e inicializadas en **main.c**.

La administración adicional efectuada por el procedimiento **getToken** es como se describe a continuación. La tabla **reservedWords** (líneas 649-656) y el procedimiento **reservedLookup** (líneas 658-666) realizan una búsqueda de palabras reservadas después de que un identificador es reconocido por la iteración principal de **getToken**, y el valor de **currentToken** es modificado de acuerdo con esto. Una variable de marca ("flag") denominada **save** se utiliza para indicar si un carácter está por ser agregado a **tokenString**; esto es necesario, porque los espacios en blanco, los comentarios y las búsquedas no consumidas no deberán incluirse.

La entrada de caracteres al analizador léxico se proporciona mediante la función **getNextChar** (líneas 627-642), la cual obtiene los caracteres de **lineBuf**, un buffer interno de 256 caracteres para el analizador léxico. Si el buffer se agota, **getNextChar** lo "refresca" o renueva desde el archivo **fuente** con el procedimiento estándar de C **fgets**, asumiendo cada vez que una nueva línea de código fuente se está alcanzando (e incrementando **lineno**). Aunque esta suposición admite un código más simple, un programa TINY con líneas de más de 255 caracteres no se manejará de manera tan correcta como se necesita. Dejaremos la investigación del comportamiento de **getNextChar** en este caso (y las mejoras a su comportamiento) para los ejercicios.

Finalmente, el reconocimiento de números e identificadores en TINY requiere que las transiciones hacia el estado final desde **ENTRADANUM** y **ENTRADAID** no sean consumidas (véase la figura 2.10). Implementaremos esto al proporcionar un procedimiento **ungetNextChar** (líneas 644-647) que respalde un carácter en el buffer de entrada. De nuevo esto no funciona muy bien para programas con líneas de código fuente muy largas, y las alternativas se exploran en los ejercicios.

Como una ilustración del comportamiento del analizador léxico TINY, considere el programa TINY **sample.tny** de la figura 2.11 (el mismo programa que se dio como ejemplo en el capítulo 1). La figura 2.12 muestra el listado de salida del analizador léxico, dado este programa como la entrada, cuando **TraceScan** y **EchoSource** están establecidos.

El resto de esta sección se dedicará a elaborar algunas de las cuestiones de implementación alcanzadas mediante esta implementación de analizador léxico.

Figura 2.11
Programa de muestra en el lenguaje TINY

```
{ Programa de muestra
  en lenguaje TINY -
  calcula factoriales
}
read x; { se introduce un entero }
if 0 < x then { no se calcula si x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { salida del factorial de x }
end
```

Figura 2.12
Salida del analizador léxico proporcionando al programa TINY de la figura 2.11 como entrada

```
TINY COMPILATION: sample.tny
    1: { Programa de muestra
    2:   en lenguaje TINY -
    3:   calcula factoriales
    4: }
    5: read x; { se introduce un entero }
        5: reserved word: read
        5: ID, name= x
        5: ;
    6: if 0 < x then { no se calcula si x <= 0 }
        6: reserved word: if
        6: NUM, val= 0
        6: <
        6: ID, name= x
        6: reserved word: then
    7:   fact := 1;
        7: ID, name= fact
        7: :=
        7: NUM, val= 1
        7: ;
    8:   repeat
        8: reserved word: repeat
    9:     fact := fact * x;
        9: ID, name= fact
        9: :=
        9: ID, name= fact
        9: *
        9: ID, name= x
        9: ;
   10:     x := x - 1
       10: ID, name= x
       10: :=
       10: ID, name= x
       10: -
       10: NUM, val= 1
   11:   until x = 0;
       11: reserved word: until
       11: ID, name= x
       11: =
       11: NUM, val= 0
       11: ;
   12:   write fact { salida del factorial de x }
       12: reserved word: write
       12: ID, name= fact
   13: end
       13: reserved word: end
       14: EOF
```

2.5.2 Palabras reservadas contra identificadores

Nuestro analizador léxico TINY reconoce palabras reservadas considerándolas primero como identificadores y posteriormente buscándolas en una tabla de palabras reservadas. Esto es una práctica común en analizadores léxicos, pero significa que la eficiencia del analizador dependerá de la eficiencia del proceso de búsqueda en la tabla de palabras reservadas. En nuestro analizador léxico utilizamos un método muy simple (la búsqueda lineal), en el cual la tabla se consulta de manera secuencial de principio a fin. Esto no es un problema para tablas muy pequeñas como las de TINY, que sólo tienen ocho palabras reservadas, pero se vuelve una situación inaceptable en analizadores léxicos para los lenguajes reales, los cuales por lo regular tienen entre 30 y 60 palabras reservadas. En este caso se requiere de una búsqueda más rápida, y esto puede exigir el uso de una mejor estructura de datos que el de una lista lineal. Una posibilidad es una búsqueda binaria, la cual podríamos haber aplicado al escribir la lista de palabras reservadas en orden alfabético. Otra posibilidad es emplear una tabla de cálculo de direcciones (o tabla de dispersión). En este caso nos gustaría utilizar una función de cálculo de dirección que tuviera un número muy pequeño de colisiones. Una función de cálculo de direcciones de esta clase se puede desarrollar en principio, porque las palabras reservadas no van a cambiar (al menos no rápidamente), y sus lugares en la tabla estarán fijos para toda ejecución del compilador. Se realizó cierto esfuerzo de investigación para determinar las **funciones de cálculo de dirección mínimas perfectas** para diversos lenguajes, es decir, funciones que distinguen cada palabra reservada de las otras, y que tienen el número mínimo de valores, de manera que se puede utilizar una tabla de cálculo de dirección no mayor que el número de palabras reservadas. Por ejemplo, si sólo existen ocho palabras reservadas, entonces una función de cálculo de dirección mínima perfecta produciría siempre un valor de 0 a 7, y cada palabra reservada produciría un valor diferente. (Véase la sección de notas y referencias para mayor información.)

Otra opción para tratar con palabras reservadas es emplear la misma tabla que almacena identificadores, es decir, la tabla de símbolos. Antes de comenzar el procesamiento, se introducen todas las palabras reservadas en esta tabla y se marcan como reservadas (de manera que no se permita una redefinición). Esto tiene la ventaja de que se requiere sólo una tabla de búsqueda. Sin embargo, en el analizador léxico de TINY no construimos la tabla de símbolos sino hasta después de la fase de análisis léxico, de manera que esta solución no es apropiada para este diseño en particular.

2.5.3 Asignación de espacio para identificadores

Un defecto adicional en el diseño del analizador léxico TINY es que las cadenas que forman el token sólo pueden tener como máximo 40 caracteres. Esto no es un problema para la mayoría de los tokens, ya que sus tamaños de cadena son fijos, pero es un problema para los identificadores, porque los lenguajes de programación a menudo requieren que identificadores arbitrariamente extensos se permitan en los programas. Aún peor, si asignamos un arreglo de 40 caracteres para cada identificador, entonces se desperdicia gran parte del espacio, debido a que la mayoría de los identificadores son breves. Esto no ocurre en el código del compilador TINY, porque las cadenas que forman el token se copian utilizando la función utilitaria `copyString`, que asigna de manera dinámica sólo el espacio necesario, como veremos en el capítulo 4. Una solución para la limitación de tamaño de `tokenString` sería semejante: asignar espacio sólo con base en las necesidades, utilizando posiblemente la función estándar de C `realloc`. Una alternativa es asignar un arreglo inicial grande para todos los identificadores y después realizar una asignación de memoria "a mano" dentro de este arreglo. (Éste es un caso especial de los esquemas de administración de memoria dinámica estándar que se analizaron en el capítulo 7.)

2.6 USO DE Lex PARA GENERAR AUTOMÁTICAMENTE UN ANALIZADOR LÉXICO

En esta sección repetimos el desarrollo de un analizador léxico para el lenguaje TINY que se realizó en la sección anterior, pero ahora utilizaremos el generador del analizador léxico Lex para generar un analizador a partir de una descripción de los tokens de TINY como expresiones regulares. Como existe una variedad de versiones diferentes de Lex, limitaremos nuestro análisis a aquellas características que son comunes a todas o a la mayoría de las versiones. La versión más popular de Lex se conoce como **flex** (por "Fast Lex"). Se distribuye como parte del **paquete compilador Gnu** que produce la Free Software Foundation, y también se encuentra disponible gratuitamente en muchos sitios de Internet.

Lex es un programa que toma como entrada un archivo de texto que contiene expresiones regulares, junto con las acciones que se tomarán cuando se iguale cada expresión. Este programa produce un archivo de salida que contiene un código fuente en C que define un procedimiento `yylex` que es una implementación controlada por tabla de un DFA correspondiente a las expresiones regulares del archivo de entrada, y que funciona como un procedimiento `getToken`. El archivo de salida de Lex, denominado por lo general `lex.yy.c` o `lexyy.c`, se compila y liga entonces a un programa principal para obtener un programa ejecutable, exactamente como se ligó el archivo `scan.c` con el archivo `tiny.c` en la sección anterior.

En lo que sigue, analizaremos en primer lugar las convenciones de Lex para escribir expresiones regulares y el formato de un archivo de entrada de Lex, para continuar con el análisis del archivo de entrada de Lex del analizador léxico TINY que se dio en el apéndice B.

2.6.1 Convenciones de Lex para expresiones regulares

Las convenciones de Lex son muy similares a las que se analizaron en la sección 2.2.3. En vez de enumerar todos los metacaracteres de Lex y describirlos de manera individual, daremos una perspectiva general y posteriormente proporcionaremos las convenciones de Lex en una tabla.

Lex permite la correspondencia de caracteres simples, o cadenas de caracteres, con sólo escribir los caracteres en secuencia, como lo hicimos en secciones anteriores. Lex también permite que los metacaracteres se igualen como caracteres reales encerrándolos entre comillas. Las comillas también se pueden escribir en torno de caracteres que no son metacaracteres, donde no tienen efecto alguno. De este modo, tiene sentido escribir comillas para encerrar a todos los caracteres que se vayan a igualar directamente, sean o no metacaracteres. Por ejemplo, podemos escribir `if` o `"if"` para igualar la palabra reservada **if** que da inicio a una sentencia `"if"` o condicional. Por otra parte, para igualar un paréntesis izquierdo, debemos escribir `"("`, puesto que es un metacarácter. Una alternativa es utilizar el metacarácter de diagonal invertida \, pero esto sólo funciona para metacaracteres simples: para igualar la secuencia de caracteres `(*` tendríamos que escribir `\(\*`, repitiendo dicha diagonal invertida. Evidentemente, es preferible escribir `"(*"`. También el uso de la diagonal invertida con caracteres regulares puede tener un significado especial. Por ejemplo, `\n` corresponde a un retorno de línea y `\t` corresponde a una tabulación (éstas son convenciones típicas de C, y la mayoría de tales convenciones se llevan a Lex).

Lex interpreta los metacaracteres `*`, `+`, `(`, `)` y `|` de la manera habitual. También utiliza el signo de interrogación como un metacarácter para indicar una parte opcional. Como

ejemplo de la notación de Lex analizada hasta ahora, podemos escribir una expresión regular para el conjunto de cadenas de las *a* y *b* que comienzan con *aa* o *bb* y tienen una *c* opcional al final como

```
(aa|bb)(a|b)*c?
```

o como

```
("aa"|"bb")("a"|"b")*"c"?
```

La convención de Lex para clases de caracteres (conjuntos de caracteres) es escribirlas entre corchetes (paréntesis cuadrados). Por ejemplo, `[abxz]` significa cualquiera de los caracteres *a*, *b*, *x* o *z*, y podríamos escribir la anterior expresión regular en Lex como

```
(aa|bb)[ab]*c?
```

También se pueden escribir intervalos de caracteres en esta forma empleando un guión. De este modo, la expresión `[0-9]` significa en Lex cualquiera de los dígitos desde el cero hasta el nueve. Un punto es un metacarácter que también representa un conjunto de caracteres: representa cualquier carácter, excepto un retorno de línea. También se pueden escribir conjuntos complementarios (es decir, conjuntos que *no* contienen ciertos caracteres) en esta notación utilizando el signo carat `^` como el primer carácter dentro de los corchetes. De esta manera, `[^0-9abc]` significa cualquier carácter que no sea un dígito y no sea una de las letras *a*, *b* o *c*.

Como ejemplo escribamos una expresión regular para el conjunto de números con signo que pueden contener una parte fraccional o un exponente comenzando con la letra *E* (esta expresión se escribió en forma un poco diferente en la sección 2.2.4):

```
("+"|"-")?[0-9]+(."[0-9]+)?(E("+"|"-")?[0-9]+)?
```

Una curiosa característica de Lex es que dentro de los corchetes (que representan una clase de caracteres), la mayoría de los metacaracteres pierden su estatus especial y no necesitan estar entre comillas. Incluso el guión se puede describir como un carácter regular si está enumerado primero. Así, podríamos haber escrito `[-+]` en lugar de `("+"|"-")` en la expresión regular anterior para números (pero no `[+-]` debido al uso del metacarácter de `-` para expresar un intervalo de caracteres). Como otro ejemplo, `[."?]` significa cualquiera de los tres caracteres, punto, comillas o signo de interrogación (los tres caracteres perdieron su significado de metacarácter dentro de los corchetes). Sin embargo, algunos caracteres son todavía metacaracteres incluso dentro de los corchetes, y para obtener el carácter real debemos anteceder al carácter con una diagonal invertida (las comillas no se pueden utilizar porque perdieron su significado de metacarácter). De esta manera, `[\^\\]` significa cualesquiera de los caracteres reales ^ o \.

Una importante convención adicional de metacarácter en Lex es el uso de llaves para denotar nombres de expresiones regulares. Recordemos que a una expresión regular se le puede dar un nombre, y que estos nombres se pueden utilizar en otras expresiones regulares mientras que no haya referencias recursivas. Por ejemplo, definimos ***naturalconSigno*** en la sección 2.2.4 como sigue:

```
nat = [0-9]+
natconSigno = ("+"|"-")? nat
```

En éste y otros ejemplos utilizamos cursivas o itálicas para distinguir los nombres de las secuencias ordinarias de caracteres. Sin embargo, los archivos de Lex son archivos de texto ordinarios, así que no se dispone de letras itálicas. En su lugar, Lex utiliza la convención de que los nombres previamente definidos se encierren mediante llaves. De este modo, el ejemplo anterior aparecería como sigue en Lex (Lex también prescinde del signo de igualdad en la definición de nombres):

```
nat [0-9]+
natconSigno (+|-)?{nat}
```

Advierta que las llaves no aparecen cuando se define un nombre, sólo cuando se utiliza.

La tabla 2.2 contiene una lista resumida de las convenciones de metacaracteres de Lex que hemos analizado. Existen varias otras convenciones de metacaracteres en Lex que no utilizaremos y no analizaremos aquí (véanse las referencias al final del capítulo).

Tabla 2.2
Convenciones de metacaracteres en Lex

Patrón	Significado
`a`	el carácter *a*
`"a"`	el carácter *a*, incluso si *a* es un metacarácter
`\a`	el carácter *a* cuando *a* es un metacarácter
`a*`	cero o más repeticiones de *a*
`a+`	una o más repeticiones de *a*
`a?`	una *a* opcional
`a|b`	*a* o *b*
`(a)`	*a* misma
`[abc]`	cualquiera de los caracteres *a*, *b* o *c*
`[a-d]`	cualquiera de los caracteres *a*, *b*, *c* o *d*
`[^ab]`	cualquier carácter excepto *a* o *b*
`.`	cualquier carácter excepto un retorno de línea
`{xxx}`	la expresión regular que representa el nombre *xxx*

2.6.2 El formato de un archivo de entrada de Lex

Un archivo de entrada de Lex se compone de tres partes, una colección de **definiciones**, una colección de **reglas** y una colección de **rutinas auxiliares** o **rutinas del usuario**. Las tres secciones están separadas por signos de porcentaje dobles que aparecen en líneas separadas comenzando en la primera columna. Por consiguiente, el diseño de un archivo de entrada de Lex es como se muestra a continuación:

```
{definiciones}
%%
{reglas}
%%
{rutinas auxiliares}
```

Para entender adecuadamente cómo interpreta Lex un archivo de entrada de esta clase debemos tener en mente que algunas partes del archivo serán información de expresión regular que Lex utiliza para guiar su construcción del código de salida en C, mientras que otras partes del archivo serán código C real que estamos suministrando a Lex, y que Lex insertará textualmente en el código de salida en la localidad apropiada. Daremos las reglas exactas que Lex emplea para esto después de que hayamos analizado cada una de las tres secciones y proporcionado algunos ejemplos.

La sección de definición se presenta antes del primer `%%`. Contiene dos elementos. En primer lugar, cualquier código C que se deba insertar de manera externa a cualquier función debería aparecer en esta sección entre los delimitadores `%{` y `%}`. (¡Fíjese en el orden de estos caracteres!) En segundo lugar, los nombres para expresiones regulares también se deben definir en esta sección. Un nombre se define al escribirlo en una línea por separado, comenzando en la primera columna y continuando (después de uno o más espacios en blanco) con la expresión regular que representa.

La segunda sección contiene las reglas. Éstas se componen de una secuencia de expresiones regulares seguidas por el código en C que se va a ejecutar cuando se iguale la expresión regular correspondiente.

La tercera y última sección contiene código C para cualquier rutina auxiliar que se invoque en la segunda sección y que no se haya definido en otro sitio. Esta sección también puede contener un programa principal, si queremos compilar la salida de Lex como un programa autónomo. Incluso se puede omitir, en cuyo caso no se necesita describir el segundo `%%`. (El primer `%%` siempre es necesario.)

Daremos una serie de ejemplos para ilustrar el formato de un archivo de entrada de Lex.

Ejemplo 2.20

La siguiente entrada de Lex especifica un analizador léxico que agrega números de línea al texto y envía su salida a la pantalla (o a un archivo, si es redirigido):

```
%{
/* un programa de Lex que agrega números de línea
   a las líneas de texto, imprimiendo el texto nuevo
   en la salida estándar
*/
#include <stdio.h>
int lineno = 1;
%}
line .*\n
%%
{line} { printf("%5d %s",lineno++,yytext); }
%%
main()
{ yylex(); return 0; }
```

Por ejemplo, ejecutar el programa obtenido de Lex en este mismo archivo de entrada proporciona la salida siguiente:

```
1 %{
2 /* un programa de Lex que agrega números de línea
```

```
3     a las líneas de texto, imprimiendo el texto nuevo
4     en la salida estándar
5 */
6 #include <stdio.h>
7 int lineno = 1;
8 %}
9 line .*\n
10 %%
11 {line} { printf("%5d %s",lineno++,yytext); }
12 %%
13 main()
14 { yylex(); return 0; }
```

Haremos comentarios en este archivo de entrada de Lex utilizando estos números de línea. En primer lugar, las líneas 1 hasta la 8 se encuentran entre los delimitadores `%{` y `%}`. Esto provoca que estas líneas se inserten directamente en el código C producido por Lex, externo a cualquier procedimiento. En particular, el comentario en las líneas 2 a la 5 se insertará cerca del principio del programa, y la directiva **`#include`** y la definición de la variable entera **`lineno`** en las líneas 6 y 7 se insertará externamente, de modo que **`lineno`** se convierte en una variable global y su valor se inicializa a 1. La otra definición que aparece antes del primer `%%` es la definición del nombre **`line`**, que se define como la expresión regular **`".*\n"`**, que iguala a 0 o más caracteres (sin incluir un retorno de línea), seguida por un retorno de línea. En otras palabras, la expresión regular definida por **`line`** iguala cada línea de entrada. A continuación del `%%` en la línea 10, la línea 11 comprende la sección de acción del archivo de entrada de Lex. En este caso escribimos una acción simple por realizarse siempre que se iguale una **`line`** (**`line`** está encerrada entre llaves para distinguirla como un nombre, de acuerdo con la convención de Lex). Después de la expresión regular está la **acción**, es decir, el código en C que se va a ejecutar siempre que se iguale la expresión regular. En este ejemplo la acción consiste en una sentencia simple en C, que se encuentra contenida dentro de las llaves de un bloque de C. (No olvide que las llaves que encierran el nombre **`line`** tienen una función por completo diferente de la que tienen las llaves que forman un bloque en el código de C de la siguiente acción.) Esta sentencia en C es para imprimir el número de línea (en campo de cinco espacios, justificado a la derecha) y la cadena **`yytext`**, después de la cual se incrementa **`lineno`**. El nombre **`yytext`** es el nombre interno que Lex le da a la cadena igualada por la expresión regular, que en este caso se compone de cada línea de entrada (incluyendo el retorno de línea).[5] Finalmente se inserta el código en C después del segundo símbolo de porcentaje doble (líneas 13 y 14), como está al final del código C producido por Lex. En este ejemplo el código se compone de la definición de un procedimiento **`main`** que llama a la función **`yylex`**. Esto permite que el código C producido por Lex sea compilado en un programa ejecutable. (**`yylex`** es el nombre que se da al procedimiento construido por Lex que implementa el DFA asociado con las expresiones regulares y acciones dadas en la sección de acción del archivo de entrada.) §

5. Enumeramos los nombres internos de Lex que se estudian en esta sección en una tabla al final de la misma.

Ejemplo 2.21 Considere el siguiente archivo de entrada de Lex:

```
%{
/* un programa de Lex que cambia todos los números
   de notación decimal a notación hexadecimal,
   imprimiendo una estadística de resumen hacia stderr
*/
#include <stdlib.h>
#include <stdio.h>
int count = 0;
%}
digit [0-9]
number {digit}+
%%
{number} { int n = atoi(yytext);
           printf("%x", n);
           if (n > 9) count++; }
%%
main()
{ yylex();
  fprintf(stderr,"número de reemplazos = %d",
                 count);
  return 0;
}
```

Éste es semejante en estructura al ejemplo anterior, excepto que el procedimiento **main** imprime el conteo del número de reemplazos para **stderr** después de llamar a **yylex**. Este ejemplo también es diferente en el sentido de que no todo el texto es igualado. En realidad, sólo los números se igualan en la sección de acción, donde el código C para la acción primero convierte la cadena igualada (**yytext**) a un entero **n**, luego lo imprime en forma hexadecimal (**printf("%x",...)**), y finalmente incrementa **count** si el número es mayor que 9. (Si el número es más pequeño que o igual a 9, entonces no tendrá aspecto diferente en hexadecimal.) Así, la única acción especificada es para cadenas que son secuencias de dígitos. Lex todavía genera un programa que también iguala todos los caracteres no numéricos, y los pasa hasta la salida. Éste es un ejemplo de una **acción predeterminada** por Lex. Si un carácter o cadena de caracteres no se iguala por ninguna de las expresiones regulares en la sección de acción, Lex lo hará de manera predeterminada y duplicará la acción hacia la salida (hará "eco"). (También se puede obligar a Lex a generar un error de ejecución, pero aquí no estudiaremos cómo se hace.) La acción predeterminada también se puede indicar en forma específica mediante la macro **ECHO** de Lex definida internamente. (Estudiaremos este uso en el siguiente ejemplo.) §

Ejemplo 2.22 Considere el siguiente archivo de entrada de Lex:

```
%{
/* Selecciona solamente líneas que finalizan o
   comienzan con la letra 'a'.
   Elimina todo lo demás.
*/
```

```
#include <stdio.h>
%}
finaliza_con_a .*a\n
comienza_con_a a.*\n
%%
{finaliza_con_a} ECHO;
{comienza_con_a} ECHO;
.*\n ;
%%
main()
{ yylex(); return 0; }
```

Esta entrada de Lex provoca que todas las líneas de entrada que comiencen o finalicen con el carácter *a* se escriban hacia la salida. Todas las otras líneas de entrada se suprimen. La eliminación de la entrada es causada por la regla bajo las reglas **ECHO**. En esta regla la acción "vacía" se especifica mediante la expresión regular `.*\n` al escribir un signo de punto y coma para el código de acción en C.

Existe una característica adicional en esta entrada de Lex que vale la pena resaltar. Las reglas enumeradas son **ambiguas** en el sentido de que una cadena puede igualar más de una regla. De hecho, *cualquier* línea de entrada iguala la expresión `.*\n`, sin tener en cuenta si es parte de una línea que comience o finalice con una *a*. Lex tiene un sistema prioritario para resolver tales ambigüedades. En primer lugar, Lex siempre iguala la subcadena más larga posible (de modo que Lex siempre genera un analizador léxico que sigue el principio de la subcadena más larga). Entonces, si la subcadena más larga todavía iguala dos o más reglas, Lex elige la primera regla en el orden en que estén clasificadas en la sección de acción. Es por esto que el archivo de entrada de Lex anterior enumera primero las acciones **ECHO**. Si tuviéramos clasificadas las acciones en el orden siguiente,

```
.*\n ;
{finaliza_con_a} ECHO;
{comienza_con_a} ECHO;
```

entonces el programa producido por Lex no generaría ninguna salida para cualquier archivo, porque toda línea de la entrada se reconocería por medio de la primera regla. §

Ejemplo 2.23

En este ejemplo, Lex genera un programa que convertirá todas las letras mayúsculas, excepto las letras dentro de los comentarios estilo C (es decir, cualquier texto dentro de los delimitadores `/*...*/`):

```
%{
/* Programa en Lex para convertir letras mayúsculas a
   letras minúsculas excepto dentro de los comentarios
*/
#include <stdio.h>
#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
#define TRUE 1
```

```
#endif
%}
%%
[A-Z] {putchar(tolower(yytext[0]));
       /* yytext[0] es un solo carácter
          en letras mayúsculas encontrado */
      }
"/*"  { char c;
        int done = FALSE;
        ECHO;
        do
        { while ((c=input())!='*')
            putchar(c);
          putchar(c);
          while ((c=input())=='*')
            putchar(c);
          putchar(c);
          if (c == '/') done = TRUE;
        } while (!done);
      }
%%
void main(void)
{ yylex();}
```

Este ejemplo muestra cómo se puede escribir el código para esquivar expresiones regulares difíciles e implementar un pequeño DFA directamente como una acción de Lex. Recuerde que en el análisis que se realizó en la sección 2.2.4 se determinó que una expresión regular para un comentario en C es muy compleja de escribir. En su lugar, escribimos una expresión regular sólo para la cadena que comienza un comentario tipo C (es decir, `"/*"`) y después suministramos un código de acción que buscará la cadena de terminación `"*/"`, mientras proporcionamos la acción apropiada para otros caracteres dentro del comentario (en este caso precisamente para duplicarlos sin procesamiento adicional). Hacemos esto al imitar el DFA del ejemplo 2.9 (véase la figura 2.4, página 53). Una vez que hemos reconocido la cadena `"/*"`, estamos en el estado 3, de modo que aquí elige nuestro código el DFA. La primera cosa que hacemos es recorrer los caracteres (duplicándolos hacia la salida) hasta que vemos un asterisco (correspondiente al bucle *otro* en el estado 3), como sigue:

```
while ((c=input())!='*') putchar(c);
```

Aquí todavía utilizamos otro procedimiento interno de Lex denominado `input`. El uso de este procedimiento, más que una entrada directa utilizando `getchar`, asegura que se utilice el buffer de entrada de Lex, y que se conserve la estructura interna de la cadena de entrada. (Advierta, sin embargo, que empleamos un procedimiento de salida directo `putchar`. Esto se analizará con más detalle en la sección 2.6.4.)

El siguiente paso en nuestro código para el DFA corresponde al estado 4. Repetiremos el ciclo de nuevo hasta que *no* veamos un asterisco, y entonces, si el carácter es una diagonal, terminamos; de otra manera, regresamos al estado 3. §

Finalizamos esta subsección con un resumen de las convenciones de Lex que presentamos en los ejemplos.

RESOLUCIÓN DE AMBIGÜEDAD

La salida de Lex siempre iguala en primer lugar la subcadena más larga posible para una regla. Si dos o más reglas provocan que se reconozcan subcadenas de igual longitud, entonces la salida de Lex elegirá la regla clasificada en primer lugar en la sección de acción. Si ninguna regla coincide con cualquier subcadena no vacía, entonces la acción predeterminada copia el siguiente carácter a la salida y continúa.

INSERCIÓN DE CÓDIGO C

1) Cualquier texto escrito entre `%{` y `%}` en la sección de definición se copiará directamente hacia el programa de salida externo para cualquier procedimiento. 2) Cualquier texto en la sección de procedimientos auxiliares se copiará directamente al programa de salida al final del código de Lex. 3) Cualquier código que siga a una expresión regular (por al menos un espacio) en la sección de acción (después del primer `%%`) se insertará en el lugar apropiado en el procedimiento de reconocimiento `yylex` y se ejecutará cuando se presente una coincidencia de la expresión regular correspondiente. El código C que representa una acción puede ser una sentencia simple en C o una sentencia compuesta en C compuesta por cualesquiera declaraciones y sentencias encerradas entre llaves.

NOMBRES INTERNOS

La tabla 2.3 enumera los nombres internos de Lex que analizamos en este capítulo. La mayor parte de ellos ya se estudiaron en los ejemplos anteriores.

Tabla 2.3
Algunos nombres internos en Lex

Nombre interno en Lex	Significado/Uso
`lex.yy.c` o `lexyy.c`	Nombre de archivo de salida de Lex
`yylex`	Rutina de análisis léxico de Lex
`yytext`	Cadena reconocida en la acción actual
`yyin`	Archivo de entrada de Lex (predeterminado: `stdin`)
`yyout`	Archivo de salida de Lex (predeterminado: `stdout`)
`input`	Rutina de entrada con buffer de Lex
`ECHO`	Acción predeterminada de Lex (imprime `yytext` a `yyout`)

Observamos una característica de esta tabla que no se había mencionado, la de que Lex tiene sus propios nombres internos para los archivos, de los cuales toma la entrada y hacia los cuales envía la salida: `yyin` y `yyout`. Por medio de la rutina de entrada estándar de Lex `input` automáticamente tomará la entrada del archivo `yyin`. Sin embargo, en los ejemplos anteriores, evitamos el archivo de salida interno `yyout` y sólo escribimos la salida estándar utilizando `printf` y `putchar`. Una mejor implementación, que permite la asignación de la salida a un archivo arbitrario, reemplazaría estos usos con `fprintf(yyout,...)` y `putc(..., yyout)`.

2.6.3 Un analizador léxico TINY que utiliza Lex

El apéndice B ofrece un listado de un archivo de entrada en Lex `tiny.l` que generará un analizador léxico para el lenguaje TINY, cuyos tokens se describieron en la sección 2.5 (véase la tabla 2.1). A continuación haremos varias observaciones acerca de este archivo de entrada (líneas 3000-3072).

En primer lugar, en la sección de definiciones, el código C que insertamos directamente en la salida de Lex se compone de tres directivas `#include` (`globals.h`, `util.h` y `scan.h`) y la definición del atributo `tokenString`. Esto es necesario para proporcionar la interfaz entre el analizador léxico y el resto del compilador TINY.

El contenido adicional de la sección de definición comprende las definiciones de los nombres para las expresiones regulares que definen los tokens de TINY. Observe que la definición de `number` utiliza el nombre previamente definido `digit`, y la definición de `identifier` utiliza a `letter` definida con anterioridad. Las definiciones también distinguen entre retornos de línea y otros espacios en blanco (espacios y tabulaciones, líneas 3019 y 3020), porque un retorno de línea provocará que `lineno` se incremente.

La sección de acción de la entrada de Lex se compone del listado de los diversos tokens, junto con una sentencia return que devuelve el token apropiado como se define en `globals.h`. En esta definición de Lex enumeramos las reglas para palabras reservadas antes que la regla para un identificador. Si se hubiera enumerado primero la regla del identificador, las reglas de resolución de ambigüedad de Lex hubieran podido ocasionar que siempre se reconociera un identificador en lugar de una palabra reservada. También podríamos escribir código como en el analizador léxico de la sección anterior, en la que sólo se reconocían identificadores, y posteriormente se buscaban las palabras reservadas en una tabla. Esto sería en realidad preferible en un compilador real, puesto que palabras reservadas reconocidas por separado ocasionan que las tablas en el código del analizador léxico generado por Lex crezcan enormemente (y, en consecuencia, la cantidad de memoria utilizada por el analizador).

Una peculiaridad de la entrada de Lex es que tenemos que escribir un código para reconocer comentarios con el fin de asegurarnos de que `lineno` se actualiza de manera correcta, aun cuando la expresión regular para los comentarios de TINY sea fácil de escribir. En realidad, la expresión regular es

```
"{"[^\}]*"}"
```

(Advierta el uso de la diagonal inversa dentro de los corchetes para eliminar el significado de metacarácter de la llave derecha: las comillas no funcionarán aquí.)[6]

Notamos también que no hay código escrito para regresar el token `EOF` al encontrar el final del archivo de entrada. El procedimiento `yylex` de Lex tiene un comportamiento predeterminado al encontrar `EOF`: devuelve el valor 0. A esto se debe que el token `ENDFILE` se haya escrito primero en la definición de `TokenType` en `globals.h` (línea 179), de manera que tendrá el valor 0.

Finalmente, el archivo `tiny.l` contiene una definición del procedimiento `getToken` en la sección de procedimientos auxiliares (líneas 3056-3072). Aunque este código contiene algunas inicializaciones ex profeso de los procedimientos internos de Lex (tales como `yyin` y `yyout`) que sería mejor efectuar directamente en el programa principal,

6. Algunas versiones de Lex tienen una variable `yylineno` definida internamente que se actualiza de manera automática. El uso de esta variable en lugar de `lineno` permitiría eliminar el código especial.

nos permite utilizar de manera directa el analizador léxico generado por Lex, sin cambiar ningún otro archivo en el compilador de TINY. En realidad, después de generar el archivo del analizador léxico de C `lex.yy.c` (o `lexyy.c`), este archivo se puede compilar y ligar directamente con los otros archivos fuente de TINY para producir una versión basada en Lex del compilador. Sin embargo, esta versión del compilador carece de un servicio de la versión anterior, en la que no se proporciona código fuente duplicado con los números de línea (véase el ejercicio 2.35).

EJERCICIOS

2.1 Escriba expresiones regulares para los siguientes conjuntos de caracteres, o explique las razones por las que no se pueden escribir expresiones regulares:

a. Todas las cadenas de letras minúsculas que comiencen y finalicen con *a*.

b. Todas las cadenas de letras minúsculas que o comiencen o finalicen con *a* (o ambos).

c. Todas las cadenas de dígitos que no contengan ceros al principio.

d. Todas las cadenas de dígitos que representen números pares.

e. Todas las cadenas de dígitos tales que todos los números "2" se presenten antes que todos los "9".

f. Todas las cadenas de *a* y *b* que no contengan tres *b* consecutivas.

g. Todas las cadenas de *a* y *b* que contengan un número impar de *a* o un número impar de *b* (o ambos).

h. Todas las cadenas de *a* y *b* que contengan un número par de *a* y un número par de *b*.

i. Todas las cadenas de *a* y *b* que contengan exactamente tantas *a* como *b*.

2.2 Escriba descripciones en español para los lenguajes generados por las siguientes expresiones regulares:

a. `(a|b)*a(a|b|`ε`)`

b. `(A|B|...|Z)(a|b|...|z)*`

c. `(aa|b)*(a|bb)*`

d. `(0|1|...|9|A|B|C|D|E|F)+(x|X)`

2.3 **a.** Muchos sistemas contienen una versión de **grep** (**g**lobal **r**egular **e**xpression **p**rint, impresión de expresión regular global en inglés), un programa de búsqueda de expresiones regulares escrito originalmente para UNIX.[7] Encuentre un documento que describa su grep local, y describa sus convenciones respecto a metasímbolos.

b. Si su editor acepta alguna clase de expresiones regulares para sus búsquedas de cadenas, describa sus convenciones de metasímbolos.

2.4 En la definición de las expresiones regulares describimos la precedencia de las operaciones, pero no su asociatividad. Por ejemplo, no especificamos si `a|b|c` significaba `(a|b)|c` o `a|(b|c)`, y lo mismo para la concatenación. ¿A qué debió esto?

2.5 Demuestre que $L(r^{**}) = L(r^{*})$ para cualquier expresión regular r.

2.6 Al describir los tokens de un lenguaje de programación utilizando expresiones regulares, no es necesario tener los metasímbolos ϕ (para el conjunto vacío) o ε (para la cadena vacía). Explique por qué.

2.7 Dibuje un DFA que corresponda a la expresión regular ϕ.

2.8 Dibuje DFA para cada uno de los conjuntos de caracteres del inciso **a**) al inciso **i**) del ejercicio 2.1, o establezca por qué no existe un DFA.

2.9 Dibuje un DFA que acepte las cuatro palabras reservadas `case`, `char`, `const` y `continue` del lenguaje C.

---

7. Existen actualmente tres versiones de grep disponibles en la mayoría de los sistemas Unix: grep "regular", egrep (grep extendido) y fgrep (grep rápido).

2.10 Vuelva a escribir el pseudocódigo para la implementación del DFA para comentarios en C (sección 2.3.3) utilizando el carácter de entrada como prueba en el case externo y el estado como prueba del case interno. Compare su pseudocódigo con el del texto. ¿En qué circunstancias preferiría usted utilizar esta organización para implementación de código de un DFA?

2.11 Proporcione una definición matemática de la cerradura ε para un conjunto de estados de un NFA.

2.12 **a.** Utilice la construcción de Thompson para convertir la expresión regular `(a|b)*a(a|b|ε)` en un NFA.

b. Convierta el NFA del inciso a en un DFA utilizando la construcción de subconjuntos.

2.13 **a.** Utilice la construcción de Thompson para convertir la expresión regular `(aa|b)*(a|bb)*` en un NFA.

b. Convierta el NFA del inciso a en un DFA utilizando la construcción de subconjuntos.

2.14 Convierta el NFA del ejemplo 2.10 (sección 2.3.2) en un DFA por medio de la construcción de subconjuntos.

2.15 En la sección 2.4.1 se mencionó una simplificación para la construcción de Thompson para la concatenación que elimina la transición ε entre los dos NFA de las expresiones regulares que se están concatenando. También se mencionó que esta simplificación necesitaba el hecho de que no hubiera transiciones fuera del estado de aceptación en los otros pasos de la construcción. Proporcione un ejemplo para demostrar a qué se debe esto. (*Sugerencia*: considere una nueva construcción de NFA por repetición que elimine los nuevos estados de inicio y aceptación, y entonces considere el NFA para r^*s^*.)

2.16 Aplique el algoritmo de minimización de estado de la sección 2.4.4 a los siguientes DFA:

a.

b.

2.17 Los comentarios en Pascal permiten dos convenciones para comentario distintas: pares de llaves `{ ... }` (como en TINY) y pares de paréntesis con asteriscos `(* ... *)`. Escriba un DFA que reconozca ambos estilos de comentario.

2.18 **a.** Escriba una expresión regular para comentarios de C en notación de Lex. (*Sugerencia*: véase el análisis de la sección 2.2.3.)

b. Demuestre que su respuesta al inciso a es correcta.

2.19 La siguiente expresión regular se dio como una definición en Lex de comentarios en C (véase Schreiner y Friedman [1985, p. 25]):

```
"/*"/"*([^*/]|[^*]"/"|"*"[^/])*"*"*"*/"
```

Demuestre que esta expresión es incorrecta. (*Sugerencia*: considere la cadena `/**_/*/`.)

EJERCICIOS DE PROGRAMACIÓN

2.20 Escriba un programa que escriba con letras mayúsculas todos los comentarios en un programa de C.

2.21 Escriba un programa que escriba con letras mayúsculas todas las palabras reservadas fuera de los comentarios en un programa de C. (Una lista de las palabras reservadas de C puede encontrarse en Kernighan y Ritchie [1988, p. 192].)

2.22 Escriba un archivo de entrada de Lex que produzca un programa que escriba con letras mayúsculas todos los comentarios en un programa de C.

2.23 Escriba un archivo de entrada de Lex que produzca un programa que escriba con letras mayúsculas todas las palabras reservadas fuera de los comentarios en un programa de C (véase el ejercicio 2.21).

2.24 Escriba un archivo de entrada de Lex que produzca un programa que cuente caracteres, palabras y líneas en un archivo de texto y que informe de los conteos. Defina una palabra como cualquier secuencia de letras y/o dígitos, sin puntuación o espacios. La puntuación y los espacios en blanco no cuentan como palabras.

2.25 El código en Lex del ejemplo 2.23 (sección 2.6.2) se puede abreviar al usar una marca ("flag") global `inComment` para distinguir el comportamiento dentro de los comentarios respecto al comportamiento en otro sitio. Vuelva a escribir el código del ejemplo para hacer esto.

2.26 Agregue comentarios en C anidados al código en Lex del ejemplo 2.23.

2.27 **a.** Vuelva a escribir el analizador léxico para TINY para emplear búsqueda binaria al examinar palabras reservadas.

b. Vuelva a escribir el analizador léxico para TINY con el fin de utilizar una tabla de cálculo de dirección en la búsqueda de palabras reservadas.

2.28 Elimine el límite de 40 caracteres en los identificadores en el analizador léxico de TINY mediante la asignación dinámica de espacio para `tokenString`.

2.29 **a.** Compruebe el comportamiento del analizador léxico de TINY cuando el programa fuente tiene líneas que exceden el tamaño del buffer del analizador y encuentre tantos problemas como pueda.

b. Vuelva a escribir el analizador léxico de TINY para eliminar los problemas que haya encontrado en el inciso a (o por lo menos mejore su comportamiento). (Esto requerirá volver a escribir los procedimientos `getNextChar` y `ungetNextChar`.)

2.30 Una alternativa al uso del procedimiento `ungetNextChar` en el analizador léxico de TINY para implementar transiciones que no consuman es emplear una marca booleana que señale que el carácter actual está por consumirse, de manera que no se requiera ningún respaldo en la entrada. Vuelva a escribir el analizador léxico de TINY para implementar este método y compárelo con el código existente.

2.31 Agregue comentarios anidados para el analizador léxico de TINY utilizando un contador llamado `nestLevel`.

2.32 Agregue comentarios estilo Ada al analizador léxico de TINY. (Un comentario en Ada comienza con dos guiones y continúa hasta el final de la línea.)

2.33 Agregue la búsqueda de palabras reservadas en una tabla para el analizador léxico de Lex para TINY (puede utilizar búsqueda lineal como en el analizador léxico de TINY escrito a mano o cualquiera de los métodos de búsqueda sugeridos en el ejercicio 2.27).

2.34 Agregue comentarios estilo Ada al código de Lex para el analizador léxico de TINY. (Un comentario en Ada comienza con dos guiones y continúa hasta el final de la línea.)

2.35 Agregue duplicación de línea de código fuente (utilizando la marca `EchoSource`) al código de Lex para el analizador léxico de TINY, de manera que, cuando la marca esté activada, cada línea del código fuente se imprima al archivo del listado con el número de línea. (Esto requiere un conocimiento más extenso de los aspectos internos de Lex de lo que se ha estudiado.)

NOTAS Y REFERENCIAS

La teoría matemática de las expresiones regulares y los autómatas finitos se analiza con detalle en Hopcroft y Ullman [1979], donde se pueden encontrar algunas referencias respecto al desarrollo histórico de la teoría. En particular, se puede hallar una demostración de la equivalencia de los autómatas finitos y las expresiones regulares (empleamos sólo una dirección de la equivalencia en este capítulo). También se puede encontrar ahí un análisis sobre el "lema de la extracción" y sus consecuencias para las limitaciones de las expresiones regulares en la descripción de patrones. Una descripción más detallada del algoritmo de minimización de estados también se puede encontrar allí, junto con una demostración del hecho de que tales DFA son esencialmente únicas. La descripción de una construcción en un paso de un DFA a partir de una expresión regular (en oposición al enfoque de dos pasos descrito aquí) se puede encontrar en Aho, Hopcroft y Ullman [1986]. Un método para compresión de tablas en un analizador léxico controlado por tabla también se proporciona allí. Una descripción de la construcción de Thompson por medio de convenciones bastante diferentes de NFA respecto a las descritas en este capítulo se proporciona en Sedgewick [1990], donde se pueden hallar descripciones de algoritmos de búsqueda de tabla tales como búsqueda binaria y cálculo de dirección para reconocimiento de palabras reservadas. (El cálculo de dirección también se analiza en el capítulo 6.) Las funciones de cálculo de dirección mínimas perfectas, mencionadas en la sección 2.5.2, se comentan en Cichelli [1980] y Sager [1985]. Una utilidad denominada **gperf** se distribuye como parte del paquete de compilador Gnu. Puede generar rápidamente funciones de cálculo de dirección perfectas incluso para conjuntos grandes de palabras reservadas. Aunque aquéllas por lo general no son mínimas, todavía son bastante útiles en la práctica. Gperf se describe en Schmidt [1990].

La descripción original del generador de analizador léxico Lex se encuentra en Lesk [1975], el cual todavía es hasta cierto punto preciso para versiones más recientes. Versiones más recientes, especialmente Flex (Paxson [1990]), han resuelto algunos problemas de eficiencia y son competitivas incluso con analizadores léxicos manuscritos minuciosamente ajustados (Jacobson [1987]). Una descripción útil de Lex también se puede hallar en Schreiner y Friedman [1985], junto con más ejemplos de programas simples en Lex para resolver varias tareas de coincidencia de patrones. Una breve descripción de la familia de grep de reconocedores de patrones (ejercicio 2.3) se puede encontrar en Kernighan y Pike [1984], y un estudio más extenso en Aho [1979]. La regla de fuera de lugar mencionada en la sección 2.2.3 (un uso del espacio en blanco para suministrar formato) se analiza en Landin [1966] y Hutton [1992].

Capítulo 3

Gramáticas libres de contexto y análisis sintáctico

3.1 El proceso del análisis sintáctico
3.2 Gramáticas libres de contexto
3.3 Árboles de análisis gramatical y árboles sintácticos abstractos
3.4 Ambigüedad
3.5 Notaciones extendidas: EBNF y diagramas de sintaxis
3.6 Propiedades formales de los lenguajes libres de contexto
3.7 Sintaxis del lenguaje TINY

El análisis gramatical es la tarea de determinar la sintaxis, o estructura, de un programa. Por esta razón también se le conoce como **análisis sintáctico**. La sintaxis de un lenguaje de programación por lo regular se determina mediante las **reglas gramaticales** de una **gramática libre de contexto**, de manera similar como se determina mediante expresiones regulares la estructura léxica de los tokens reconocida por el analizador léxico. En realidad, una gramática libre de contexto utiliza convenciones para nombrar y operaciones muy similares a las correspondientes en las expresiones regulares. Con la única diferencia de que las reglas de una gramática libre de contexto son **recursivas**. Por ejemplo, la estructura de una sentencia if debe permitir en general que otras sentencias if estén anidadas en ella, lo que no se permite en las expresiones regulares. Este cambio aparentemente elemental para el poder de la representación tiene enormes consecuencias. La clase de estructuras reconocible por las gramáticas libres de contexto se incrementa de manera importante en relación con las de las expresiones regulares. Los algoritmos empleados para reconocer estas estructuras también difieren mucho de los algoritmos de análisis léxico, ya que deben utilizar llamadas recursivas o una pila de análisis sintáctico explícitamente administrada. Las estructuras de datos utilizadas para representar la estructura sintáctica de un lenguaje ahora también deben ser recursivas en lugar de lineales (como lo son para lexemas y tokens). La estructura básica empleada es por lo regular alguna clase de árbol, que se conoce como **árbol de análisis gramatical** o **árbol sintáctico**.

Como en el capítulo anterior, necesitamos estudiar la teoría de las gramáticas libres de contexto antes de abordar los algoritmos de análisis sintáctico y los detalles de los analizadores sintácticos reales que utilizan estos algoritmos. Sin embargo, al contrario de lo que sucede con los analizadores léxicos, donde sólo existe, en esencia un método algorítmico (representado por los autómatas finitos), el análisis sintáctico involucra el tener que elegir entre varios métodos diferentes, cada uno de los cuales tiene distintas propiedades y capacidades. De hecho existen dos categorías generales de algoritmos: de **análisis sintáctico descendente** y de **análisis sintáctico ascendente** (por la manera

en que construyen el árbol de análisis gramatical o árbol sintáctico). Pospondremos un análisis detallado de estos métodos de análisis sintáctico para capítulos subsiguientes. En este capítulo describiremos de manera general el proceso de análisis sintáctico y posteriormente estudiaremos la teoría básica de las gramáticas libres de contexto. En la sección final proporcionaremos la sintaxis del lenguaje TINY en términos de una gramática libre de contexto. El lector familiarizado con la teoría de las gramáticas libres de contexto y los árboles sintácticos puede omitir las secciones medias de este capítulo (o utilizarlas como repaso).

3.1 EL PROCESO DEL ANÁLISIS SINTÁCTICO

La tarea del analizador sintáctico es determinar la estructura sintáctica de un programa a partir de los tokens producidos por el analizador léxico y, ya sea de manera explícita o implícita, construir un árbol de análisis gramatical o árbol sintáctico que represente esta estructura. De este modo, se puede ver el analizador sintáctico como una función que toma como su entrada la secuencia de tokens producidos por el analizador léxico y que produce como su salida el árbol sintáctico

$$\textit{secuencia de tokens} \xrightarrow{\text{analizador sintáctico}} \textit{árbol sintáctico}$$

La secuencia de tokens por lo regular no es un parámetro de entrada explícito, pero el analizador sintáctico llama a un procedimiento del analizador léxico, como **getToken**, para obtener el siguiente token desde la entrada a medida que lo necesite durante el proceso de análisis sintáctico. De este modo, la etapa de análisis sintáctico del compilador se reduce a una llamada al analizador léxico de la manera siguiente:

```
syntaxTree = parse();
```

En un compilador de una sola pasada el analizador sintáctico incorporará todas las otras fases de un compilador, incluyendo la generación del código, y no es necesario construir ningún árbol sintáctico explícito (las mismas etapas del analizador sintáctico representarán de manera implícita al árbol sintáctico), y, por consiguiente, una llamada

```
parse();
```

lo hará. Por lo común, un compilador será de múltiples pasadas, en cuyo caso las pasadas adicionales utilizarán el árbol sintáctico como su entrada.

La estructura del árbol sintáctico depende en gran medida de la estructura sintáctica particular del lenguaje. Este árbol por lo regular se define como una estructura de datos dinámica, en la cual cada nodo se compone de un registro cuyos campos incluyen los atributos necesarios para el resto del proceso de compilación (es decir, no sólo por aquellos que calcula el analizador sintáctico). A menudo la estructura de nodos será un registro variante para ahorrar espacio. Los campos de atributo también pueden ser estructuras que se asignen de manera dinámica a medida que se necesite, como una herramienta adicional para ahorrar espacio.

Un problema más difícil de resolver para el analizador sintáctico que para el analizador léxico es el tratamiento de los errores. En el analizador léxico, si hay un carácter que no puede ser parte de un token legal, entonces es suficientemente simple generar un token de error y consumir el carácter problemático. (En cierto sentido, al generar un token de error, el analizador léxico transfiere la dificultad hacia el analizador sintáctico.) Por otra parte, el analizador sintáctico no sólo debe mostrar un mensaje de error, sino que debe **recuperarse** del error y continuar el análisis sintáctico (para encontrar tantos errores como sea posible). En

ocasiones, un analizador sintáctico puede efectuar **reparación de errores**, en la cual infiere una posible versión de código corregida a partir de la versión incorrecta que se le haya presentado. (Esto por lo regular se hace sólo en casos simples.) Un aspecto particularmente importante de la recuperación de errores es la exhibición de mensajes de errores significativos y la reanudación del análisis sintáctico tan próximo al error real como sea posible. Esto no es fácil, puesto que el analizador sintáctico puede no descubrir un error sino hasta mucho después de que el error real haya ocurrido. Como las técnicas de recuperación de errores dependen del algoritmo de análisis sintáctico que se haya utilizado en particular, se aplazará su estudio hasta capítulos subsiguientes.

3.2 GRAMÁTICAS LIBRES DE CONTEXTO

Una gramática libre de contexto es una especificación para la estructura sintáctica de un lenguaje de programación. Una especificación así es muy similar a la especificación de la estructura léxica de un lenguaje utilizando expresiones regulares, excepto que una gramática libre de contexto involucra reglas de recursividad. Como ejemplo de ejecución utilizaremos expresiones aritméticas simples de enteros con operaciones de suma, resta y multiplicación. Estas expresiones se pueden dar mediante la gramática siguiente:

$$exp \rightarrow exp\ op\ exp \mid (\ exp\ ) \mid \texttt{número}$$
$$op \rightarrow + \mid - \mid *$$

3.2.1 Comparación respecto a la notación de una expresión regular

Consideremos cómo se compara la gramática libre de contexto de muestra anterior con las reglas de la expresión regular dada por **número** del capítulo anterior:

```
número = dígito dígito*
dígito = 0|1|2|3|4|5|6|7|8|9
```

En las reglas de la expresión regular básica tenemos tres operaciones: selección (dada por el metasímbolo de la barra vertical), concatenación (dada sin un metasímbolo) y repetición (dada por el metasímbolo del asterisco). También empleamos el signo de igualdad para representar la definición de un nombre para una expresión regular, y escribimos el nombre en itálicas para distinguirlo de una secuencia de caracteres reales.

Las reglas gramaticales utilizan notaciones semejantes. Los nombres se escriben en cursivas o itálicas (pero ahora con una fuente diferente, de modo que podamos distinguirlas de los nombres para las expresiones regulares). La barra vertical todavía aparece como el metasímbolo para selección. La concatenación también se utiliza como operación estándar. Sin embargo, no hay ningún metasímbolo para la repetición (como el * de las expresiones regulares), un punto al cual regresaremos en breve. Una diferencia adicional en la notación es que ahora utilizamos el símbolo de la flecha → en lugar del de igualdad para expresar las definiciones de los nombres. Esto se debe a que ahora los nombres no pueden simplemente ser reemplazados por sus definiciones, porque está implicado un proceso de definición más complejo, como resultado de la naturaleza recursiva de las definiciones.[1] En nuestro ejemplo, la regla para *exp* es recursiva, en el sentido que el nombre *exp* aparece a la derecha de la flecha.

Advierta también que las reglas gramaticales utilizan expresiones regulares como componentes. En las reglas para *exp* y *op* se tienen en realidad seis expresiones regulares

1. Pero vea más adelante en el capítulo el análisis sobre reglas gramaticales como ecuaciones.

representando tokens en el lenguaje. Cinco de éstas son tokens de carácter simple: +, -, *, (y). Una es el nombre *número*, el nombre de un token representando secuencias de dígitos.

De manera similar a la de este ejemplo, las reglas gramaticales se usaron por primera vez en la descripción del lenguaje Algol60. La notación fue desarrollada por John Backus y adaptada por Peter Naur para el informe Algol60. De este modo, generalmente se dice que las reglas gramaticales en esta forma están en la **forma Backus-Naur**, o **BNF (Backus-Naur Form)**.

3.2.2 Especificación de las reglas de una gramática libre de contexto

Como las expresiones regulares, las reglas gramaticales están definidas sobre un alfabeto, o conjunto de símbolos. En el caso de expresiones regulares, estos símbolos por lo regular son caracteres. En el caso de reglas gramaticales, los símbolos son generalmente tokens que representan cadenas de caracteres. En el último capítulo definimos los tokens en un analizador léxico utilizando un tipo enumerado en C. En este capítulo, para evitar entrar en detalles de la manera en que los tokens se representan en un lenguaje de implementación específica (como C), emplearemos las expresiones regulares en sí mismas para representar los tokens. En el caso en que un token sea un símbolo fijo, como en la palabra reservada `while` o los símbolos especiales como + o :=, escribiremos la cadena en sí en la fuente de código que se utilizó en el capítulo 2. En el caso de tokens tales como identificadores y números, que representan más de una cadena, utilizaremos la fuente de código en itálicas, justo como si el token fuera un nombre para una expresión regular (lo que por lo regular representa). Por ejemplo, representaremos el alfabeto de tokens para el lenguaje TINY como el conjunto

```
{if,then,else,end,repeat,until,read,write,
identificador,número,+,-,*,/,=,<,(,),;,:=}
```

en lugar del conjunto de tokens (como se definieron en el analizador léxico de TINY):

```
{IF,THEN,ELSE,END,REPEAT,UNTIL,READ,WRITE,ID,NUM,
PLUS,MINUS,TIMES,OVER,EQ,LT,LPAREN,RPAREN,SEMI,ASSIGN}
```

Dado un alfabeto, una **regla gramatical libre de contexto en BNF** se compone de una cadena de símbolos. El primer símbolo es un nombre para una estructura. El segundo símbolo es el metasímbolo "→". Este símbolo está seguido por una cadena de símbolos, cada uno de los cuales es un símbolo del alfabeto, un nombre para una estructura o el metasímbolo "|".

En términos informales, una regla gramatical en BNF se interpreta como sigue. La regla define la estructura cuyo nombre está a la izquierda de la flecha. Se define la estructura de manera que incluya una de las selecciones en el lado derecho separada por las barras verticales. Las secuencias de símbolos y nombres de estructura dentro de cada selección definen el diseño de la estructura. Por ejemplo, considere las reglas gramaticales de nuestro ejemplo anterior:

$$exp \rightarrow exp\ op\ exp \mid (\ exp\ ) \mid \textbf{número}$$
$$op \rightarrow + \mid - \mid *$$

La primera regla define una estructura de expresión (con nombre *exp*) compuesta por una expresión seguida por un operador y otra expresión, por una expresión dentro de paréntesis, o bien por un número. La segunda define un operador (con nombre *op*) compuesto de uno de los símbolos +, - o *.

Los metasímbolos y convenciones que usamos aquí son semejantes a los de uso generalizado, pero es necesario advertir que no hay un estándar universal para estas convenciones. En realidad, las alternativas comunes para el metasímbolo de la flecha "→" incluyen "=" (el

signo de igualdad), ":" (el signo de dos puntos) y "::=" (signo de dos puntos doble y el de igualdad). En archivos de texto normales también es necesario hallar un reemplazo para el uso de las itálicas. Esto se hace frecuentemente encerrando los nombres de estructura con "picoparéntesis" < . . . > y escribiendo los nombres de token en itálicas con letras mayúsculas. De este modo, con diferentes convenciones, las reglas gramaticales anteriores podrían aparecer como

```
<exp> ::= <exp> <op> <exp> | ( <exp> ) | NÚMERO
<op> ::= + | - | *
```

Cada autor también tendrá otras variaciones en estas notaciones. Varias de las más importantes (algunas de las cuales utilizaremos ocasionalmente) se analizarán más adelante en esta sección. Sin embargo, vale la pena analizar de inmediato dos pequeñas cuestiones adicionales sobre la notación.

En ocasiones, aunque los paréntesis son útiles para reasignar la precedencia en las expresiones regulares, conviene incluir paréntesis en los metasímbolos de la notación BNF. Por ejemplo, se puede volver a escribir las reglas gramaticales anteriores como una sola regla gramatical de la manera siguiente:

$$exp \rightarrow exp\ (\text{“}\mathbf{+}\text{”} \mid \text{“}\mathbf{-}\text{”} \mid \text{“}\mathbf{*}\text{”})\ exp \mid \text{“}\mathbf{(}\text{”}\ exp\ \text{”}\mathbf{)}\text{”} \mid \textbf{\textit{número}}$$

En esta regla los paréntesis son necesarios para agrupar las opciones de los operadores entre las expresiones en el lado derecho, puesto que la concatenación tiene precedencia sobre la selección (como en las expresiones regulares). De este modo, la regla siguiente tendría un significado diferente (e incorrecto):

$$exp \rightarrow exp\ \text{“}\mathbf{+}\text{”} \mid \text{“}\mathbf{-}\text{”} \mid \text{“}\mathbf{*}\text{”}\ exp \mid \text{“}\mathbf{(}\text{”}\ exp\ \text{”}\mathbf{)}\text{”} \mid \textbf{\textit{número}}$$

Advierta también que, cuando se incluyen los paréntesis como metasímbolo, es necesario distinguir los tokens de paréntesis de los metasímbolos, lo que hicimos poniéndolos entre comillas, como hacíamos en el caso de expresiones regulares. (Para mantener la consistencia también encerramos los símbolos de operador entre comillas.)

Los paréntesis no son absolutamente necesarios como metasímbolos en BNF, ya que siempre es posible separar las partes entre paréntesis en una nueva regla gramatical. De hecho, la operación de selección que da el metasímbolo de la barra vertical tampoco es necesaria en reglas gramaticales, si permitimos que el mismo nombre aparezca cualquier número de veces a la izquierda de la flecha. Por ejemplo, nuestra gramática de expresión simple podría escribirse como se aprecia a continuación:

$$\begin{aligned}
exp &\rightarrow exp\ op\ exp \\
exp &\rightarrow \mathbf{(}\ exp\ \mathbf{)} \\
exp &\rightarrow \textbf{\textit{número}} \\
op &\rightarrow \mathbf{+} \\
op &\rightarrow \mathbf{-} \\
op &\rightarrow \mathbf{*}
\end{aligned}$$

Sin embargo, por lo regular describiremos reglas gramaticales de manera que todas las selecciones para cada estructura estén enumeradas en una sola regla, y cada nombre de estructura aparezca sólo una vez a la izquierda de la flecha.

En ocasiones, por simplicidad, daremos ejemplos de reglas gramaticales en una notación abreviada. En estos casos utilizaremos letras en mayúsculas para nombres de estructura y letras en minúsculas para símbolos de token individuales (que con frecuencia son sólo caracteres simples). De este modo, nuestra gramática de expresión simple podría escribirse en esta forma abreviada de la manera que sigue

$$E \rightarrow E\ O\ E \mid (\ E\ ) \mid \mathbf{n}$$
$$O \rightarrow + \mid - \mid *$$

En ocasiones también simplificaremos la notación cuando estemos utilizando solamente caracteres como tokens y los estemos escribiendo sin utilizar una fuente de código:

$$E \rightarrow E\ O\ E \mid (\ E\ ) \mid a$$
$$O \rightarrow + \mid - \mid *$$

3.2.3 Derivaciones y el lenguaje definido por una gramática

Ahora volvamos a la descripción de cómo las reglas gramaticales determinan un "lenguaje", o conjunto de cadenas legales de tokens.

Las reglas gramaticales libres de contexto determinan el conjunto de cadenas sintácticamente legales de símbolos de token para las estructuras definidas por las reglas. Por ejemplo, la expresión aritmética

```
(34-3)*42
```

corresponde a la cadena legal de siete tokens

```
( número - número ) * número
```

donde los tokens de **número** tienen sus estructuras determinadas por el analizador léxico y la cadena misma es legalmente una expresión porque cada parte corresponde a selecciones determinadas por las reglas gramaticales

$$exp \rightarrow exp\ op\ exp \mid (\ exp\ ) \mid \textbf{número}$$
$$op \rightarrow + \mid - \mid *$$

Por otra parte, la cadena

```
(34-3*42
```

no es una expresión legal, porque se tiene un paréntesis izquierdo que no tiene su correspondiente paréntesis derecho y la segunda selección en la regla gramatical para una expresión *exp* requiere que los paréntesis se generen en pares.

Las reglas gramaticales determinan las cadenas legales de símbolos de token por medio de derivaciones. Una **derivación** es una secuencia de reemplazos de nombres de estructura por selecciones en los lados derechos de las reglas gramaticales. Una derivación comienza con un nombre de estructura simple y termina con una cadena de símbolos de token. En cada etapa de una derivación se hace un reemplazo simple utilizando una selección de una regla gramatical.

Como ejemplo, la figura 3.1 proporciona una derivación para la expresión `(34-3)*42` utilizando las reglas gramaticales como se dan en nuestra expresión gramatical simple. En cada paso se proporciona a la derecha la selección de la regla gramatical utilizada para el reemplazo. (También numeramos los pasos para una referencia posterior.)

Figura 3.1
Una derivación para la expresión aritmética `(34-3)*42`

(1)	*exp* ⇒ *exp op exp*	[*exp* → *exp op exp*]
(2)	⇒ *exp op* **número**	[*exp* → **número**]
(3)	⇒ *exp* * **número**	[*op* → *]
(4)	⇒ (*exp*) * **número**	[*exp* → (*exp*)]
(5)	⇒ (*exp op exp*) * **número**	[*exp* → *exp op exp*]
(6)	⇒ (*exp op* **número**) * **número**	[*exp* → **número**]
(7)	⇒ (*exp* - **número**) * **número**	[*op* → -]
(8)	⇒ (**número** - **número**)*número	[*exp* → **número**]

Advierta que los pasos de derivación utilizan una flecha diferente al metasímbolo de flecha que se emplea en las reglas gramaticales. Esto se debe a que existe una diferencia entre un paso de derivación y una regla gramatical: las reglas gramaticales **definen**, mientras que los pasos de derivación **construyen** mediante reemplazo. En el primer paso de la figura 3.1, la *exp* simple se reemplaza por la cadena *exp op exp* del lado derecho de la regla *exp* → *exp op exp* (la primera selección en la BNF para *exp*). En el segundo paso, la *exp* del extremo derecho en la cadena *exp op exp* se reemplaza por el símbolo **número** del lado derecho de la selección *exp* → **número** para obtener la cadena *exp op* **número**. En ese paso el *op* (operador) se reemplaza por el símbolo * del lado derecho de la regla *op* → * (la tercera de las selecciones en la BNF para *op*) para obtener la cadena *exp* * **número**. Y así sucesivamente.

El conjunto de todas las cadenas de símbolos de token obtenido por derivaciones del símbolo *exp* es el **lenguaje definido por la gramática** de expresiones. Este lenguaje contiene todas las expresiones sintácticamente legales. Podemos escribir esto de manera simbólica como

$$L(G) = \{ s \mid exp \Rightarrow^* s \}$$

donde G representa la gramática de expresión, s representa una cadena arbitraria de símbolos de token (en ocasiones denominada **sentencia**), y los símbolos $\Rightarrow^*$ representan una derivación compuesta de una secuencia de reemplazos como se describieron anteriormente. (El asterisco se utiliza para indicar una secuencia de pasos, así como para indicar repetición en expresiones regulares.) Las reglas gramaticales en ocasiones se conocen como **producciones** porque "producen" las cadenas en $L(G)$ mediante derivaciones.

Cada nombre de estructura en una gramática define su propio lenguaje de cadenas sintácticamente legales de tokens. Por ejemplo, el lenguaje definido por *op* en nuestra gramática de expresión simple define el lenguaje {+, -, *} compuesto sólo de tres símbolos. Por lo regular estamos más interesados en el lenguaje definido por la estructura más general en una gramática. La gramática para un lenguaje de programación a menudo define una estructura denominada *programa*, y el lenguaje de esta estructura es el conjunto de todos los programas sintácticamente legales del lenguaje de programación (advierta que aquí utilizamos la palabra "lenguaje" en dos sentidos diferentes).

Por ejemplo, un BNF para Pascal comenzará con reglas gramaticales tales como

programa → *encabezado-programa* ; *bloque-programa* .
encabezado-programa → . . .
bloque-programa → . . .
. . .

(La primera regla dice que un programa se compone de un encabezado de programa, seguido por un signo de punto y coma, seguido por un bloque de programa, seguido por un punto.) En lenguajes con compilación separada, como C, la estructura más general se conoce a menudo como una *unidad de compilación.* En todo caso, suponemos que la estructura más general se menciona primero en las reglas gramaticales, a menos que lo especifiquemos de otra manera. (En la teoría matemática de las gramáticas libres de contexto esta estructura se denomina **símbolo inicial.**)

Otro segmento de terminología nos permite distinguir con más claridad entre los nombres de estructura y los símbolos del alfabeto (los cuales hemos estado llamando símbolos de token, porque por lo regular son tokens en aplicaciones de compilador). Los nombres de estructuras también se conocen como **no terminales**, porque siempre se debe reemplazar de más en una derivación (no terminan una derivación). En contraste, los símbolos en el alfabeto se denominan **terminales**, porque éstos terminan una derivación. Como los terminales por lo regular son tokens en aplicaciones de compilador, utilizarán ambos nombres, los que, en esencia, son sinónimos. A menudo, se hace referencia tanto a los terminales como a los no terminales como símbolos.

Consideremos ahora algunos ejemplos de lenguajes generados por gramáticas.

Ejemplo 3.1 Considere la gramática G con la regla gramatical simple

$$E \rightarrow (E) \mid a$$

Esta gramática tiene un no terminal E y tres terminales (,) y a. Genera, además, el lenguaje $L(G) = \{a,(a), ((a)), (((a))), \ldots\} = \{(^n\, a)^n \mid n \text{ un entero} \geq 0\}$, es decir, las cadenas compuestas de 0 o más paréntesis izquierdos, seguidos por una a, seguida por el mismo número de paréntesis derechos que de paréntesis izquierdos. Como ejemplo de una derivación para una de estas cadenas ofrecemos una derivación para $((a))$:

$$E \Rightarrow (E) \Rightarrow ((E)) \Rightarrow ((a))$$ §

Ejemplo 3.2 Considere la gramática G con la regla gramatical simple

$$E \rightarrow (E)$$

Ésta resulta ser igual que la gramática del ejemplo anterior, sólo que se perdió la opción $E \rightarrow a$. Esta gramática no genera ninguna cadena, de modo que su lenguaje es vacío: $L(G) = \{\ \}$. La causa es que cualquier derivación que comienza con E genera cadenas que siempre contienen una E. Así, no hay manera de que podamos derivar una cadena compuesta sólo de terminales. En realidad, como ocurre con todos los procesos recursivos (como las demostraciones por inducción o las funciones recursivas), una regla gramatical que define

una estructura de manera recursiva siempre debe tener por lo menos un caso no recursivo (al que podríamos denominar **caso base**). La regla gramatical de este ejemplo no lo tiene, y cualquier derivación potencial está condenada a una recursión infinita. §

Ejemplo 3.3 Considere la gramática G con la regla gramatical simple

$$E \rightarrow E + a \mid a$$

Esta gramática genera todas las cadenas compuestas de a separadas por signos de "más" (+):

$$L(G) = \{ a, a + a, a + a + a, a + a + a + a, \ldots \}$$

Para ver esto (de manera informal), considere el efecto de la regla $E \rightarrow E + a$: esto provoca que la cadena $+ a$ se repita sobre la derecha en una derivación:

$$E \Rightarrow E + a \Rightarrow E + a + a \Rightarrow E + a + a + a \Rightarrow \ldots$$

Finalmente, debemos reemplazar la E a la izquierda utilizando el caso base $E \rightarrow a$.

Podemos demostrar esto más formalmente mediante inducción como sigue. En primer lugar, mostramos que toda cadena $a + a + \cdots + a$ está en $L(G)$ por inducción en el número de las a. La derivación $E \Rightarrow a$ muestra que a está en $L(G)$; suponga ahora que $s = a + a + \cdots + a$, con $n - 1$ a, está en $L(G)$. De este modo, existe una derivación $E \Rightarrow^* s$: ahora la derivación $E \Rightarrow E + a \Rightarrow^* s + a$ muestra que la cadena $s + a$, con $n + a$, está en $L(G)$. A la inversa, también mostramos que cualquier cadena s de $L(G)$ debe ser de la forma $a + a + \cdots + a$. Mostramos esto mediante inducción sobre la longitud de una derivación. Si la derivación tiene longitud 1, entonces es de la forma $E \Rightarrow a$, así que s es de la forma correcta. Ahora, suponga la veracidad de la hipótesis para todas las cadenas con derivaciones de longitud $n - 1$, y sea $E \Rightarrow^* s$ una derivación de longitud $n > 1$. Esta derivación debe comenzar con el reemplazo de E por $E + a$, y de esta manera será de la forma $E \Rightarrow E + a \Rightarrow^* s' + a = s$. Entonces, s' tiene una derivación de longitud $n - 1$, y de este modo será de la forma $a + a + \cdots + a$. Por lo tanto, s misma debe tener esta misma forma. §

Ejemplo 3.4 Considere la siguiente gramática muy simplificada de sentencias:

sentencia → *sent-if* | `otro`
sent-if → `if` (*exp*) *sentencia*
| `if` (*exp*) *sentencia* `else` *sentencia*
exp → `0` | `1`

El lenguaje de esta gramática se compone de sentencias if anidadas de manera semejante al lenguaje C. (Simplificamos las expresiones de prueba lógica ya sea a 0 o a 1, y agrupamos todas las otras sentencias aparte de las sentencias if en el terminal `otro`.) Ejemplos de cadenas en este lenguaje son

```
otro
if (0) otro
if (1) otro
```

```
if (0) otro else otro
if (1) otro else otro
if (0) if (0) otro
if (0) if (1) otro else otro
if (1) otro else if (0) otro else otro
...
```

Advierta cómo la parte else opcional de la sentencia if está indicada por medio de una selección separada en la regla gramatical para *sent-if*. §

Antes advertimos que se conservan las reglas gramaticales en BNF para concatenación y selección, pero no tienen equivalente específico de la operación de repetición para el * de las expresiones regulares. Una operación así de hecho es innecesaria, puesto que la repetición se puede obtener por medio de recursión (como los programadores en lenguajes funcionales lo saben). Por ejemplo, tanto la regla gramatical

$$A \rightarrow A\ a \mid a$$

como la regla gramatical

$$A \rightarrow a\ A \mid a$$

generan el lenguaje $\{a^n \mid n$ un entero $\geq 1\}$ (el conjunto de todas las cadenas con una o más a), el cual es igual al que genera la expresión regular **a+**. Por ejemplo, la cadena *aaaa* puede ser generada por la primera regla gramatical con la derivación

$$A \Rightarrow Aa \Rightarrow Aaa \Rightarrow Aaaa \Rightarrow aaaa$$

Una derivación similar funciona para la segunda regla gramatical. La primera de estas reglas gramaticales es **recursiva por la izquierda**, porque el no terminal A aparece como el primer símbolo en el lado derecho de la regla que define A.[2] La segunda regla gramatical es **recursiva por la derecha**.

El ejemplo 3.3 es otro ejemplo de una regla gramatical recursiva por la izquierda, que ocasiona la repetición de la cadena "+ a". Éste y el ejemplo anterior se puede generalizar como sigue. Considere una regla de la forma

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

donde α y β representan cadenas arbitrarias y β no comienza con A. Esta regla genera todas las cadenas de la forma β, $\beta\alpha$, $\beta\alpha\alpha$, $\beta\alpha\alpha\alpha$, . . . (todas las cadenas comienzan con una β, seguida por 0 o más α). De este modo, esta regla gramatical es equivalente en su efecto a la expresión regular $\beta\alpha^*$. De manera similar, la regla gramatical recursiva por la derecha

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

(donde β no finaliza en A) genera todas las cadenas β, $\alpha\beta$, $\alpha\alpha\beta$, $\alpha\alpha\alpha\beta$, ...

2. Éste es un caso especial de recursión por la izquierda denominado **recursión por la izquierda inmediata**. Casos más generales se analizan en el capítulo siguiente.

Si queremos escribir una gramática que genere el mismo lenguaje que la expresión regular `a*`, entonces debemos tener una notación para una regla gramatical que genere la cadena vacía (porque la expresión regular `a*` incluye la cadena vacía). Una regla gramatical así debe tener un lado derecho vacío. Podemos simplemente no escribir nada en el lado derecho, como ocurre en

$$vacío \rightarrow$$

pero emplearemos más a menudo el metasímbolo épsilon para la cadena vacía (como se usó en las expresiones regulares):

$$vacío \rightarrow \varepsilon$$

Una regla gramatical de esta clase se conoce como **producción ε** (una "producción épsilon"). Una gramática que genera un lenguaje que contenga la cadena vacía debe tener por lo menos una producción ε.

Ahora podemos escribir una gramática que sea equivalente a la expresión regular `a*`, ya sea como

$$A \rightarrow A\ a \mid \varepsilon$$

o como

$$A \rightarrow a\ A \mid \varepsilon$$

Ambas gramáticas generan el lenguaje $\{a^n \mid n \text{ un entero} \geq 0\} = L(\texttt{a*})$. Las producciones ε también son útiles al definir estructuras que son opcionales, como pronto veremos.

Concluiremos esta subsección con algunos otros ejemplos más.

Ejemplo 3.5 Considere la gramática

$$A \rightarrow (\ A\ )\ A \mid \varepsilon$$

Esta gramática genera las cadenas de todos los "paréntesis balanceados". Por ejemplo, la cadena (( )(( ))( ) es generada por la derivación siguiente (la producción ε se utiliza para hacer que desaparezca A cuando sea necesario):

$$\begin{aligned} A &\Rightarrow (\ A\ )\ A \Rightarrow (\ A\ )\ (\ A\ )\ A \Rightarrow (\ A\ )\ (\ A\ ) \Rightarrow (\ A\ )\ (\ ) \Rightarrow (\ (\ A\ )\ A\ )\ (\ ) \\ &\Rightarrow (\ (\ )\ A\ )\ (\ ) \Rightarrow (\ (\ )\ (\ A\ )\ A\ )\ (\ ) \Rightarrow (\ (\ )\ (\ A\ )\ )\ (\ ) \\ &\Rightarrow (\ (\ )\ (\ (\ A\ )\ A\ )\ )\ (\ ) \Rightarrow (\ (\ )\ (\ (\ )\ A\ )\ )\ (\ ) \Rightarrow (\ (\ )\ (\ (\ )\ )\ )\ (\ ) \end{aligned}$$

§

Ejemplo 3.6 La gramática de sentencia del ejemplo 3.4 se puede escribir de la siguiente manera alternativa utilizando una producción ε:

$$\begin{aligned} &sentencia \rightarrow sent\text{-}if \mid \texttt{otro} \\ &sent\text{-}if \rightarrow \texttt{if}\ (\ exp\ )\ sentencia\ parte\text{-}else \\ &parte\text{-}else \rightarrow \texttt{else}\ sentencia \mid \varepsilon \\ &exp \rightarrow \texttt{0} \mid \texttt{1} \end{aligned}$$

Advierta cómo la producción ε indica que la estructura *parte-else* es opcional. §

Ejemplo 3.7 Considere la siguiente gramática G para una secuencia de sentencias:

$$\textit{secuencia-sent} \rightarrow \textit{sent}\ \texttt{;}\ \textit{secuencia-sent} \mid \textit{sent}$$
$$\textit{sent} \rightarrow \texttt{s}$$

Esta gramática genera secuencias de una o más sentencias separadas por signos de punto y coma (las sentencias se extrajeron del terminal simple `s`):

$$L(G) = \{\ \texttt{s},\ \texttt{s;s},\ \texttt{s;s;s}, \ldots\ \}$$

Si queremos permitir que las secuencias de sentencia también sean vacías, podríamos escribir la siguiente gramática G':

$$\textit{secuencia-sent} \rightarrow \textit{sent}\ \texttt{;}\ \textit{secuencia-sent} \mid \varepsilon$$
$$\textit{sent} \rightarrow \texttt{s}$$

pero esto convierte al signo de punto y coma en un **terminador** de sentencia en vez de en un **separador** de sentencia:

$$L(G') = \{\ \varepsilon,\ \texttt{s;},\ \texttt{s;s;},\ \texttt{s;s;s;}, \ldots\ \}$$

Si deseamos permitir que las secuencias de sentencia sean vacías, pero también mantener el signo de punto y coma como un separador de sentencia, debemos escribir la gramática como se muestra a continuación:

$$\textit{secuencia-sent} \rightarrow \textit{secuencia-sent-no vacía} \mid \varepsilon$$
$$\textit{secuencia-sent-no vacía} \rightarrow \textit{sent}\ \texttt{;}\ \textit{secuencia-sent-no vacía} \mid \textit{sent}$$
$$\textit{sent} \rightarrow \texttt{s}$$

Este ejemplo muestra que debe tenerse cuidado en la ubicación de la producción ε cuando se construyan estructuras opcionales. §

3.3 ÁRBOLES DE ANÁLISIS GRAMATICAL Y ÁRBOLES SINTÁCTICOS ABSTRACTOS

3.3.1 Árboles de análisis gramatical

Una derivación proporciona un método para construir una cadena particular de terminales a partir de un no terminal inicial. Pero las derivaciones no sólo representan la estructura de las cadenas que construyen. En general, existen muchas derivaciones para la misma cadena. Por ejemplo, construyamos la cadena de tokens

```
( número - número ) * número
```

a partir de nuestra gramática de expresión simple utilizando la derivación de la figura 3.1. Una segunda derivación para esta cadena se proporciona en la figura 3.2. La única diferencia

entre las dos derivaciones es el orden en el cual se suministran los reemplazos, y ésta es de hecho una diferencia superficial. Para aclarar esto necesitamos una representación para la estructura de una cadena de terminales que abstraiga las características esenciales de una derivación mientras se factorizan las diferencias superficiales de orden. La representación que hace esto es una estructura de árbol, y se conoce como árbol de análisis gramatical.

Figura 3.2
Otra derivación para la expresión `(34-3)*42`

(1)	$exp \Rightarrow exp\ op\ exp$	[$exp \rightarrow exp\ op\ exp$]
(2)	$\Rightarrow$ `(`exp`)` $op\ exp$	[$exp \rightarrow$ `(` exp `)`]
(3)	$\Rightarrow$ `(`$exp\ op\ exp$`)` $op\ exp$	[$exp \rightarrow exp\ op\ exp$]
(4)	$\Rightarrow$ `(`**`número`** $op\ exp$`)` $op\ exp$	[$exp \rightarrow$ **`número`**]
(5)	$\Rightarrow$ `(`**`número`** $-\ exp$`)` $op\ exp$	[$op \rightarrow$ `-`]
(6)	$\Rightarrow$ `(`**`número - número`**`)` $op\ exp$	[$exp \rightarrow$ **`número`**]
(7)	$\Rightarrow$ `(`**`número - número`**`)` `*` exp	[$op \rightarrow$ `*`]
(8)	$\Rightarrow$ `(`**`número - número`**`)` `*` **`número`**	[$exp \rightarrow$ **`número`**]

Un **árbol de análisis gramatical** correspondiente a una derivación es un árbol etiquetado en el cual los nodos interiores están etiquetados por no terminales, los nodos hoja están etiquetados por terminales y los hijos de cada nodo interno representan el reemplazo del no terminal asociado en un paso de la derivación.

Para dar un ejemplo simple, la derivación

$$
\begin{aligned}
exp &\Rightarrow exp\ op\ exp \\
&\Rightarrow \textbf{\texttt{número}}\ op\ exp \\
&\Rightarrow \textbf{\texttt{número}} + exp \\
&\Rightarrow \textbf{\texttt{número}} + \textbf{\texttt{número}}
\end{aligned}
$$

corresponde al árbol de análisis gramatical

El primer paso en la derivación corresponde a los tres hijos del nodo raíz. El segundo paso corresponde al hijo **`número`** de la *exp* en el extremo izquierdo debajo de la raíz, y de manera similar para los dos pasos restantes. Podemos hacer esta correspondencia explícita al numerar los nodos internos del árbol de análisis gramatical mediante el número de paso en el cual se reemplaza su no terminal asociado en una derivación correspondiente. De este modo, si numeramos la derivación anterior como sigue:

(1) $exp \Rightarrow exp\ op\ exp$

(2) $\Rightarrow$ **número** $op\ exp$

(3) $\Rightarrow$ **número** + exp

(4) $\Rightarrow$ **número** + **número**

podemos numerar los nodos internos del árbol de análisis gramatical respectivamente:

Advierta que esta numeración de los nodos internos del árbol de análisis gramatical es en realidad una **numeración preorden**.

Este mismo árbol de análisis gramatical también corresponde a las derivaciones

$$\begin{aligned} exp &\Rightarrow exp\ op\ exp \\ &\Rightarrow exp\ op\ \textbf{número} \\ &\Rightarrow exp + \textbf{número} \\ &\Rightarrow \textbf{número} + \textbf{número} \end{aligned}$$

y

$$\begin{aligned} exp &\Rightarrow exp\ op\ exp \\ &\Rightarrow exp + exp \\ &\Rightarrow \textbf{número} + exp \\ &\Rightarrow \textbf{número} + \textbf{número} \end{aligned}$$

pero se aplicarían diferentes numeraciones de los nodos internos. En realidad la primera de estas dos derivaciones corresponde a la numeración siguiente:

(Dejamos al lector construir la numeración de la otra.) En este caso la numeración es la inversa de una **numeración postorden** de los nodos internos del árbol de análisis gramatical. (Un recorrido postorden visitaría los nodos internos en el orden 4, 3, 2, 1.)

Un árbol de análisis gramatical corresponde en general a muchas derivaciones, que en conjunto representan la misma estructura básica para la cadena de terminales analizada gramaticalmente. Sin embargo, se pueden distinguir derivaciones particulares que están asociadas de manera única con el árbol de análisis gramatical. Una **derivación por la izquierda** es aquella en la cual se reemplaza el no terminal más a la izquierda en cada paso en la deriva-

ción. Por consiguiente, una **derivación por la derecha** es aquella en la cual el no terminal más a la derecha se reemplaza en cada paso de la derivación. Una derivación por la izquierda corresponde a la numeración preorden de los nodos internos de su árbol de análisis gramatical asociado, mientras que una derivación por la derecha corresponde a una numeración postorden en reversa.

En realidad, vimos esta correspondencia en las tres derivaciones y el árbol de análisis gramatical del ejemplo más reciente. La primera de las tres derivaciones que dimos es una derivación por la izquierda, mientras que la segunda es una derivación por la derecha. (La tercera derivación no es por la izquierda ni por la derecha.)

Como un ejemplo más complejo de un árbol de análisis gramatical y de las derivaciones por la izquierda y por la derecha, regresemos a la expresión `(34-3)*42` y las derivaciones que dimos en las figuras 3.1 y 3.2. El árbol de análisis gramatical para esta expresión está dado en la figura 3.3, donde también numeramos los nodos de acuerdo con la derivación de la figura 3.1. Ésta es de hecho una derivación por la derecha, y la numeración correspondiente del árbol de análisis gramatical es una numeración postorden inversa. La derivación de la figura 3.2, por otra parte, es una derivación por la izquierda. (Invitamos al lector a proporcionar una numeración preorden del árbol de análisis gramatical correspondiente a esta derivación.)

Figura 3.3
Árbol de análisis gramatical para la expresión aritmética `(34-3)*42`

3.3.2 Árboles sintácticos abstractos

Un árbol de análisis gramatical es una representación útil de la estructura de una cadena de tokens, ya que los tokens aparecen como las hojas del árbol de análisis gramatical (de izquierda a derecha) y los nodos internos del árbol de análisis gramatical representan los pasos en una derivación (en algún orden). Sin embargo, un árbol de análisis gramatical contiene mucha información más de la que es absolutamente necesaria para que un compilador produzca código ejecutable. Para ver esto, considere el árbol de análisis gramatical para la expresión `3+4` de acuerdo con nuestra gramática de expresión simple:

Éste es el árbol de análisis gramatical de un ejemplo anterior. Aumentamos el árbol para mostrar el valor numérico real de cada uno de los tokens **`número`** (éste es un atributo del token que se calcula mediante el analizador léxico o por medio del analizador gramatical). El **principio de la traducción dirigida por sintaxis** establece que el significado, o semántica, de la cadena **`3+4`** debería relacionarse directamente con su estructura sintáctica como se representa mediante el árbol de análisis gramatical. En este caso el principio de la traducción dirigida por medio de sintaxis significa que el árbol de análisis gramatical debería presuponer que el valor 3 y el valor 4 se van a sumar. En realidad, podemos ver que el árbol da a entender esto de la siguiente manera. La raíz representa la operación de suma de los valores de los dos subárboles *exp* hijos. Cada uno de estos subárboles, por otro lado, representa el valor de su hijo **`número`** único. Sin embargo, existe una manera mucho más simple de representar esta misma información, a saber, mediante el árbol

Aquí, el nodo raíz simplemente se etiqueta por la operación que representa, y los nodos hoja se etiquetan mediante sus valores (en lugar de los tokens **`número`**). De manera similar, la expresión **`(34-3)*42`**, cuyo árbol de análisis gramatical está dado en la figura 3.3, se puede representar en forma más simple por medio del árbol

En este árbol los tokens de paréntesis en realidad han desaparecido, aunque todavía representan precisamente el contenido semántico de restar 3 de 34, para después multiplicarlo por 42.

Tales árboles representan abstracciones de las secuencias de token del código fuente real, y las secuencias de tokens no se pueden recobrar a partir de ellas (a diferencia de los árboles de análisis gramatical). No obstante, contienen toda la información necesaria para traducir de una forma más eficiente que los árboles de análisis gramatical. Tales árboles se conocen como **árboles sintácticos abstractos**, o para abreviar **árboles sintácticos**. Un analizador sintáctico recorrerá todos los pasos representados por un árbol de análisis gramatical, pero por lo regular sólo construirá un árbol sintáctico abstracto (o su equivalente).

Los árboles sintácticos abstractos pueden imaginarse como una representación de árbol de una notación abreviada denominada **sintaxis abstracta**, del mismo modo que un árbol de análisis gramatical es una representación para la estructura de la sintaxis ordinaria (que también se denomina **sintaxis concreta** cuando se le compara con la abstracta). Por ejemplo, la sintaxis abstracta para la expresión **`3+4`** debe escribirse como *OpExp*(*Plus*, *ConstExp*(3),*ConstExp*(4)) y la sintaxis abstracta para la expresión **`(34-3)*42`** se puede escribir como

OpExp(*Times*,*OpExp*(*Minus*,*ConstExp*(34),*ConstExp*(3)),*ConstExp*(42))

En realidad, la sintaxis abstracta puede proporcionar una definición formal utilizando una notación semejante a la BNF, del mismo modo que la sintaxis concreta. Por ejemplo, podríamos escribir las siguientes reglas tipo BNF para la sintaxis abstracta de nuestras expresiones aritméticas simples como

$$exp \rightarrow OpExp(op,exp,exp) \mid ConstExp(integer)$$
$$op \rightarrow Plus \mid Minus \mid Times$$

No investigaremos más al respecto. Nuestro principal interés radica en la estructura de árbol sintáctico real que utilizará el analizador sintáctico, la cual se determinará por medio de una declaración de tipo de datos.[3] Por ejemplo, los árboles sintácticos abstractos para nuestras expresiones aritméticas simples pueden ser determinados mediante las declaraciones de tipo de datos en C

```
typedef enum {Plus,Minus,Times} OpKind;
typedef enum {OpKind,ConstKind} ExpKind;
typedef struct streenode
      { ExpKind kind;
        OpKind op;
        struct streenode *lchild,*rchild;
        int val;
      } STreeNode;
typedef STreeNode *SyntaxTree;
```

Advierta que empleamos tipos enumerados para las dos diferentes clases de nodos de árbol sintáctico (operaciones y constantes enteras), así como para las operaciones (suma, resta, multiplicación) mismas. De hecho, probablemente usaríamos los tokens para representar las operaciones, más que definir un nuevo tipo enumerado. También podríamos haber usado un tipo `union` de C para ahorrar espacio, puesto que un nodo no puede ser al mismo tiempo un nodo de operador y un nodo de constante. Finalmente, observamos que estas tres declaraciones de nodos sólo incluyen los atributos que son directamente necesarios para este ejemplo. En situaciones prácticas habrá muchos más campos para los atributos de tiempo de compilación, tales como tipo de datos, información de tabla de símbolos y así sucesivamente, como se dejará claro con los ejemplos que se presentarán más adelante en este capítulo y en capítulos subsiguientes.

Concluimos esta sección con varios ejemplos de árboles de análisis gramatical y árboles sintácticos utilizando gramáticas que ya consideramos en ejemplos anteriores.

Ejemplo 3.8

Considere la gramática para las sentencias if simplificadas del ejemplo 3.4:

$$sentencia \rightarrow sent\text{-}if \mid \texttt{otro}$$
$$sent\text{-}if \rightarrow \texttt{if (}\ exp\ \texttt{)}\ sentencia$$
$$\mid \texttt{if (}\ exp\ \texttt{)}\ sentencia\ \texttt{else}\ sentencia$$
$$exp \rightarrow \texttt{0} \mid \texttt{1}$$

3. Existen lenguajes para los que la sintaxis abstracta recién dada es esencialmente una declaración de tipo. Véanse los ejercicios.

El árbol de análisis gramatical para la cadena

```
if (0) otro else otro
```

es como se presenta a continuación:

Al utilizar la gramática del ejemplo 3.6,

sentencia → *sent-if* | **otro**
sent-if → **if** (*exp*) *sentencia parte-else*
parte-else → **else** *sentencia* | ε
exp → **0** | **1**

esta misma cadena tiene el siguiente árbol de análisis gramatical.

Un árbol sintáctico abstracto apropiado para las sentencias if desecharía todo, excepto las tres estructuras subordinadas de la sentencia if: la expresión de prueba, la parte de acción (parte "then") y la parte alternativa (parte "else"). De este modo, un árbol sintáctico para la cadena anterior (usando la gramática del ejemplo 3.4 o la del ejemplo 3.6) sería:

Aquí empleamos los tokens restantes `if` y *`otro`* como etiquetas para distinguir la clase del nodo de sentencia en el árbol sintáctico. Esto se efectuaría más apropiadamente empleando un tipo enumerado. Por ejemplo, un conjunto de declaraciones en C apropiado para la estructura de las sentencias y expresiones en este ejemplo sería como el que se muestra a continuación:

```
typedef enum {ExpK,StmtK} NodeKind;
typedef enum {Zero,One} ExpKind;
typedef enum {IfK,OtherK} StmtKind;
typedef struct streenode
  {NodeKind kind;
   ExpKind ekind;
   StmtKind skind;
   struct streenode
      *test,*thenpart,*elsepart;
  } STreeNode;
typedef STreeNode * SyntaxTree;
```

§

Ejemplo 3.9 Considere la gramática de una secuencia de sentencias separadas por signos de punto y coma del ejemplo 3.7:

secuencia-sent → *sent* **;** *secuencia-sent* | *sent*
sent → **s**

La cadena **s;s;s** tiene el siguiente árbol de análisis gramatical respecto a esta gramática:

Un posible árbol sintáctico para esta misma cadena es

En este árbol los nodos de punto y coma son semejantes a los nodos de operador (como los nodos + de expresiones aritméticas), salvo por la diferencia de que éstos sólo "funcionan" al unirse entre sí las sentencias en una secuencia. En su lugar podríamos intentar unir todos los nodos de sentencia en una secuencia con solamente un nodo, de modo que el árbol sintáctico anterior se convertiría en

El problema con esto es que un nodo `seq` puede tener un número arbitrario de hijos y es difícil tomar precauciones al respecto en una declaración de tipo de datos. La solución es utilizar la representación estándar con **hijo de extrema izquierda y hermanos a la derecha** para un árbol (esto se presenta en la mayoría de los textos sobre estructuras de datos). En esta representación el único vínculo físico desde el padre hasta sus hijos es hacia el hijo de la extrema izquierda. Los hijos entonces se vinculan entre sí de izquierda a derecha en una lista de vínculos estándar, los que se denominan vínculos **hermanos** para distinguirlos de los vínculos padre-hijo. El árbol anterior ahora se convierte en el arreglo de hijo de extrema izquierda y hermanos a la derecha:

Con este arreglo también se puede eliminar el nodo `seq` de conexión, y entonces el árbol sintáctico se convierte simplemente en:

Ésta, como es evidente, es la representación más sencilla y más fácil para una secuencia de cosas en un árbol sintáctico. La complicación es que aquí los vínculos son vínculos hermanos que se deben distinguir de los vínculos hijos, y eso requiere un nuevo campo en la declaración del árbol sintáctico. §

3.4 AMBIGÜEDAD

3.4.1 Gramáticas ambiguas

Los árboles de análisis gramatical y los árboles sintácticos expresan de manera única la estructura de la sintaxis, y efectúan derivaciones por la izquierda y por la derecha, pero no derivaciones en general. Desgraciadamente, una gramática puede permitir que una cadena tenga más de un árbol de análisis gramatical. Considere, por ejemplo, la gramática de la aritmética entera simple que hemos estado utilizando como ejemplo estándar

$$exp \rightarrow exp\ op\ exp \mid (\ exp\ ) \mid \texttt{número}$$
$$op \rightarrow + \mid - \mid *$$

y considere la cadena `34-3*42`. Esta cadena tiene dos árboles de análisis gramatical diferentes

y

correspondientes a las dos derivaciones por la izquierda

$exp \Rightarrow exp\ op\ exp$	[$exp \rightarrow exp\ op\ exp$]
$\Rightarrow exp\ op\ exp\ op\ exp$	[$exp \rightarrow exp\ op\ exp$]
$\Rightarrow$ **número** $op\ exp\ op\ exp$	[$exp \rightarrow$ **número**]
$\Rightarrow$ **número** - $exp\ op\ exp$	[$op \rightarrow$ -]
$\Rightarrow$ **número** - **número** $op\ exp$	[$exp \rightarrow$ **número**]
$\Rightarrow$ **número** - **número** * exp	[$op \rightarrow$ *]
$\Rightarrow$ **número** - **número** * **número**	[$exp \rightarrow$ **número**]

y

$exp \Rightarrow exp\ op\ exp$	[$exp \rightarrow exp\ op\ exp$]
$\Rightarrow$ **número** $op\ exp$	[$exp \rightarrow$ **número**]
$\Rightarrow$ **número** - exp	[$op \rightarrow$ -]
$\Rightarrow$ **número** - $exp\ op\ exp$	[$exp \rightarrow exp\ op\ exp$]
$\Rightarrow$ **número** - **número** $op\ exp$	[$exp \rightarrow$ **número**]
$\Rightarrow$ **número** - **número** * exp	[$op \rightarrow$ *]
$\Rightarrow$ **número** - **número** * **número**	[$exp \rightarrow$ **número**]

Los árboles sintácticos asociados son

y

Una gramática que genera una cadena con dos árboles de análisis gramatical distintos se denomina **gramática ambigua**. Una gramática de esta clase representa un serio problema para un analizador sintáctico, ya que no especifica con precisión la estructura sintáctica de un programa (aun cuando las mismas cadenas legales, es decir, los miembros del lenguaje de la gramática, estén completamente determinados). En cierto sentido, una gramática ambigua es como un autómata no determinístico en el que dos rutas por separado pueden aceptar la misma cadena. Sin embargo, la ambigüedad en las gramáticas no se puede eliminar tan fácilmente como el no determinismo en los autómatas finitos, puesto que no hay algoritmo para hacerlo así, a diferencia de la situación en el caso de los autómatas (la construcción del subconjunto analizada en el capítulo anterior).[4]

Una gramática ambigua debe, por lo tanto, considerarse como una especificación incompleta de la sintaxis de un lenguaje, y como tal debería evitarse. Afortunadamente las gramáticas ambiguas nunca pasan las pruebas que presentamos más adelante para los algoritmos estándar de análisis sintáctico, y existe un conjunto de técnicas estándar para tratar con ambigüedades típicas que surgen en lenguajes de programación.

Para tratar con las ambigüedades se utilizan dos métodos básicos. Uno consiste en establecer una regla que especifique en cada caso ambiguo cuál de los árboles de análisis gramatical (o árboles sintácticos) es el correcto. Una regla de esta clase se conoce como **regla de no ambigüedad** o de **eliminación de ambigüedades**. La utilidad de una regla de esta naturaleza es que corrige la ambigüedad sin modificar (con la posible complicación que esto implica) la gramática. La desventaja es que ya no sólo la gramática determina la estructura sintáctica del lenguaje. La alternativa es cambiar la gramática a una forma que obligue a construir el árbol de análisis gramatical correcto, de tal manera que se elimine la ambigüedad. Naturalmente, en cualquier método primero debemos decidir cuál de los árboles es el correcto en un caso ambiguo. Esto involucra de nueva cuenta al principio de la traducción dirigida por sintaxis. El árbol de análisis gramatical (o árbol sintáctico) que buscamos es aquel que refleja correctamente el significado posterior que aplicaremos a la construcción a fin de traducirlo a código objeto.

---

4. La situación en realidad es aún peor, puesto que no hay algoritmo para determinar si una gramática es ambigua en primera instancia. Véase la sección 3.2.7.

¿Cuál de los dos árboles sintácticos antes presentados representan la interpretación correcta de la cadena `34-3*42`? El primer árbol indica, al hacer el nodo de resta un hijo del nodo de multiplicación, que tenemos la intención de evaluar la expresión evaluando primero la resta (34 − 3 = 31) y después la multiplicación (31 * 42 = 1302). El segundo árbol, por otra parte, indica que primero se realizará la multiplicación (3 * 42 = 126) y después la resta (34 −126 = −92). La cuestión de cuál árbol elegiremos depende de cuál de estos cálculos visualicemos como correcto. La convención matemática estándar dicta que la segunda interpretación es la correcta. Esto se debe a que se dice que la multiplicación tiene **precedencia** sobre la resta. Por lo regular, tanto la multiplicación como la división tienen precedencia tanto sobre la suma como sobre la resta.

Para eliminar la ambigüedad dada en nuestra gramática de expresión simple, ahora podríamos simplemente establecer una regla de eliminación de ambigüedad que establezca las precedencias relativas de las tres operaciones representadas. La solución estándar es darle a la suma y a la resta la misma precedencia, y proporcionar a la multiplicación una precedencia más alta.

Desgraciadamente, esta regla aún no elimina por completo la ambigüedad de la gramática. Considere la cadena `34-3-42`. La cadena también tiene dos posibles árboles sintácticos:

y

El primero representa el cálculo (34 − 3) − 42 = −11, mientras el segundo representa el cálculo 34 − (3 − 42) = 73. De nueva cuenta, la cuestión de cuál cálculo es el correcto es un asunto de convención. La matemática estándar dicta que la primera selección es la correcta. Esto se debe a que la resta se considera como **asociativa por la izquierda**; es decir, que se realizan una serie de operaciones de resta de izquierda a derecha.

De este modo, una ambigüedad adicional que requiere de una regla de eliminación de ambigüedades es la asociatividad de cada una de las operaciones de suma, resta y multiplicación. Es común especificar que estas tres operaciones son asociativas por la izquierda. Esto en realidad elimina las ambigüedades restantes en nuestra gramática de expresión simple (aunque no podremos demostrar esto sino hasta más adelante).

También se puede elegir especificar que una operación es **no asociativa**, en el sentido de que una secuencia de más de un operador es una expresión que no está permitida. Por ejemplo, podríamos haber escrito nuestra gramática de expresión simple de la manera que se muestra a continuación:

$$exp \rightarrow factor\ op\ factor \mid factor$$
$$factor \rightarrow \texttt{(}\ exp\ \texttt{)} \mid \textbf{número}$$
$$op \rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*}$$

En este caso cadenas como `34-3-42` e incluso `34-3*42` son ilegales, y en cambio se deben escribir con paréntesis, tal como `(34-3)-42` y `34-(3*42)`. Estas expresiones **completamente entre paréntesis** no necesitan de la especificación de asociatividad o, en realidad, de precedencia. La gramática anterior carece de ambigüedad como está escrita. Naturalmente, no sólo cambiamos la gramática, sino que también cambiamos el lenguaje que se está reconociendo.

Ahora volveremos a métodos para volver a escribir la gramática a fin de eliminar la ambigüedad en vez de establecer reglas de no ambigüedad. Advierta que debemos hallar métodos que no modifiquen las cadenas básicas que se están reconociendo (como se hizo en el ejemplo de las expresiones completamente entre paréntesis).

3.4.2 Precedencia y asociatividad

Para manejar la precedencia de las operaciones en la gramática debemos agrupar los operadores en grupos de igual precedencia, y para cada precedencia debemos escribir una regla diferente. Por ejemplo, la precedencia de la multiplicación sobre la suma y la resta se puede agregar a nuestra gramática de expresión simple como se ve a continuación:

$$exp \rightarrow exp\ opsuma\ exp \mid term$$
$$opsuma \rightarrow \texttt{+} \mid \texttt{-}$$
$$term \rightarrow term\ opmult\ term \mid factor$$
$$opmult \rightarrow \texttt{*}$$
$$factor \rightarrow \texttt{(}\ exp\ \texttt{)} \mid \textbf{número}$$

En esta gramática la multiplicación se agrupa bajo la regla *term*, mientras que la suma y la resta se agrupan bajo la regla *exp*. Como el caso base para una *exp* es un *term*, esto significa que la suma y la resta aparecerán "más altas" (es decir, más cercanas a la raíz) en los árboles de análisis gramatical y sintáctico, de manera que reciben una precedencia más baja. Una agrupación así de operadores en diferentes niveles de precedencia es un método estándar para la especificación sintáctica utilizando BNF. A una agrupación de esta clase se le conoce como **cascada de precedencia**.

Esta última gramática para expresiones aritméticas simples todavía no especifica la asociatividad de los operadores y aún es ambigua. La causa es que la recursión en ambos lados del operador permite que cualquier lado iguale repeticiones del operador en una derivación (y, por lo tanto, en los árboles de análisis gramatical y sintáctico). La solución es reemplazar una de las recursiones con el caso base, forzando las coincidencias repetitivas en el lado en que está la recursión restante. Por consiguiente, reemplazando la regla

$$exp \rightarrow exp\ opsuma\ exp \mid term$$

por

$$exp \rightarrow exp\ opsuma\ term \mid term$$

se hace a la suma y a la resta asociativas por la izquierda, mientras que al escribir

$$exp \rightarrow term\ opsuma\ exp \mid term$$

se hacen asociativas por la derecha. En otras palabras, una regla recursiva por la izquierda hace a sus operadores asociados a la izquierda, mientras que una regla recursiva por la derecha los hace asociados a la derecha.

Para completar la eliminación de ambigüedades en las reglas BNF en el caso de nuestras expresiones aritméticas simples escribimos las reglas que permiten hacer todas las operaciones asociativas por la izquierda:

$$exp \rightarrow exp\ opsuma\ term \mid term$$
$$opsuma \rightarrow \texttt{+} \mid \texttt{-}$$
$$term \rightarrow term\ opmult\ factor \mid factor$$
$$opmult \rightarrow \texttt{*}$$
$$factor \rightarrow \texttt{(}\ exp\ \texttt{)} \mid \textbf{número}$$

Ahora el árbol de análisis gramatical para la expresión `34-3*42` es

y el árbol de análisis gramatical para la expresión `34-3-42` es

Advierta que las cascadas de precedencia provocan que los árboles de análisis gramatical se vuelvan mucho más complejos. Sin embargo, no afectan a los árboles sintácticos.

3.4.3 El problema del else ambiguo

Considere la gramática del ejemplo 3.4 (página 103):

sentencia → *sent-if* | **otro**
sent-if → **if** (*exp*) *sentencia*
| **if** (*exp*) *sentencia* **else** *sentencia*
exp → **0** | **1**

Esta gramática es ambigua como resultado del else opcional. Para ver esto considere la cadena

```
if (0) if (1) otro else otro
```

Esta cadena tiene los dos árboles de análisis gramatical:

y

La cuestión de cuál es el correcto depende de si queremos asociar la parte else con la primera o la segunda sentencia if: el primer árbol de análisis gramatical asocia la parte else con la primera sentencia if; el segundo árbol de análisis gramatical la asocia con la segunda

sentencia if. Esta ambigüedad se denomina **problema del else ambiguo**. Para ver cuál árbol de análisis gramatical es correcto, debemos considerar las implicaciones para el significado de la sentencia if. Para obtener una idea más clara de esto considere el siguiente segmento de código en C

```
if (x != 0)
   if (y == 1/x) ok = TRUE;
   else z = 1/x;
```

En este código, siempre que **x** sea 0, se presentará un error de división entre 0 si la parte else está asociada con la primera sentencia if. De este modo, la implicación de este código (y en realidad la implicación de la sangría de la parte else) es que una parte else siempre debería estar asociada con la sentencia if más cercana que todavía no tenga una parte else asociada. Esta regla de eliminación de ambigüedad se conoce como **regla de la anidación más cercana** para el problema del else ambiguo, e implica que el segundo árbol de análisis gramatical anterior es el correcto. Advierta que, si quisiéramos, *podríamos* asociar la parte else con la primera sentencia if mediante el uso de llaves {...} en C, como en

```
if (x != 0)
  {if (y == 1/x) ok = TRUE;}
else z = 1/x;
```

Una solución para la ambigüedad del else ambiguo en la BNF misma es más difícil que las ambigüedades previas que hemos visto. Una solución es como sigue:

sentencia → *sent-sinigualar* | *sent-igualada*
sent-sinigualar → `if` (*exp*) *sent-sinigualar* `else` *sent-sinigualar* | `otro`
sent-igualada → `if` (*exp*) *sentencia*
| `if` (*exp*) *sent-sinigualar* `else` *sent-igualada*
exp → `0` | `1`

Esto funciona al permitir que llegue solamente una *sentencia-igualada* antes que un `else` en una sentencia if, con lo que se obliga a que todas las partes else se empaten tan pronto como sea posible. Por ejemplo, el árbol de análisis gramatical asociado para nuestra cadena de muestra se convierte ahora en

lo que en realidad asocia la parte else con la segunda sentencia if.

Por lo regular no se emprende la construcción de la regla de la anidación más cercana en la BNF. En su lugar, se prefiere la regla de eliminación de ambigüedad. Una razón es la complejidad agregada de la nueva gramática, pero la razón principal es que los métodos de análisis sintáctico son fáciles de configurar de una manera tal que se obedezca la regla de la anidación más cercana. (La precedencia y la asociatividad son un poco más difíciles de conseguir automáticamente sin volver a escribir la gramática.)

El problema del else ambiguo tiene sus orígenes en la sintaxis de Algol60. La sintaxis se puede diseñar de tal manera que no aparezca el problema del else ambiguo. Una forma es *requerir* la presencia de la parte else, un método que se ha utilizado en LISP y otros lenguajes funcionales (donde siempre se debe devolver un valor). Otra solución es utilizar una **palabra clave de agrupación** para la sentencia if. Los lenguajes que utilizan esta solución incluyen a Algol60 y Ada. En Ada, por ejemplo, el programador escribe

```
if x /= 0 then
  if y = 1/x then ok := true;
  else z := 1/x;
  end if;
end if;
```

para asociar la parte else con la segunda sentencia if. De manera alternativa, el programador escribe

```
if x /= 0 then
  if y = 1/x then ok := true;
  end if;
else z := 1/x;
end if;
```

para asociarla con la primera sentencia if. La correspondiente BNF en Ada (algo simplificada) es

sent-if → **if** *condición* **then** *secuencia-de-sentencias* **end if**
| **if** *condición* **then** *secuencia-de-sentencias*
else *secuencia-de-sentencias* **end if**

De este modo, las dos palabras clave **end if** son la palabra clave de agrupación en Ada. En Algol68 la palabra clave de agrupación es **fi** (**if** escrito al revés).

3.4.4 Ambigüedad no esencial

En ocasiones una gramática puede ser ambigua y aún así producir siempre árboles sintácticos abstractos únicos. Considere, por ejemplo, la gramática de secuencia de sentencias del ejemplo 3.9 (página 113), donde podríamos elegir una simple lista de hermanos como el árbol sintáctico. En este caso una regla gramatical, ya fuera recursiva por la derecha o recursiva por la izquierda, todavía daría como resultado la misma estructura de árbol sintáctico, y podríamos escribir la gramática ambiguamente como

secuencia-sent → *secuencia-sent* **;** *secuencia-sent* | *sent*
sent → **s**

y aún obtener árboles sintácticos únicos. Una ambigüedad tal se podría llamar **ambigüedad no esencial**, puesto que la semántica asociada no depende de cuál regla de eliminación de ambigüedad se emplee. Una situación semejante surge con los operadores binarios, como los de la suma aritmética o la concatenación de cadenas, que representan **operaciones asociativas** (un operador binario $\cdot$ es asociativo si $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ para todo valor a, b y c). En este caso los árboles sintácticos todavía son distintos, pero representan el mismo valor semántico, y podemos no preocuparnos acerca de cuál utilizar. No obstante, un algoritmo de análisis sintáctico necesitará aplicar alguna regla de no ambigüedad que el escritor del compilador puede necesitar suministrar.

3.5 NOTACIONES EXTENDIDAS: EBNF Y DIAGRAMAS DE SINTAXIS

3.5.1 Notación EBNF

Las construcciones repetitivas y opcionales son muy comunes en los lenguajes de programación, lo mismo que las reglas de gramática de la BNF. Por lo tanto, no debería sorprendernos que la notación BNF se extienda en ocasiones con el fin de incluir notaciones especiales para estas dos situaciones. Estas extensiones comprenden una notación que se denomina **BNF extendida**, o **EBNF** (por las siglas del término en inglés).

Considere en primer lugar el caso de la repetición, como el de las secuencias de sentencias. Vimos que la repetición se expresa por la recursión en las reglas gramaticales y que se puede utilizar tanto la recursión por la izquierda como la recursión por la derecha, indicadas por las reglas genéricas

$$A \rightarrow A\ \alpha \mid \beta \quad \text{(recursiva por la izquierda)}$$

y

$$A \rightarrow \alpha\ A \mid \beta \quad \text{(recursiva por la derecha)}$$

donde α y β son cadenas arbitrarias de terminales y no terminales y donde en la primera regla β no comienza con A y en la segunda β no finaliza con A.

Se podría emplear la misma notación para la repetición que la que se utiliza para las expresiones regulares, a saber, el asterisco * (también conocido como cerradura de Kleene en expresiones regulares). Entonces estas dos reglas se escribirían como las reglas no recursivas

$$A \rightarrow \beta\ \alpha^*$$

y

$$A \rightarrow \alpha^*\ \beta$$

En vez de eso EBNF prefiere emplear llaves {. . .} para expresar la repetición (así hace clara la extensión de la cadena que se va a repetir), y escribimos

$$A \rightarrow \beta\ \{\alpha\}$$

y

$$A \rightarrow \{\alpha\}\ \beta$$

para las reglas.

El problema con cualquier notación de repetición es que oculta cómo se construye el árbol de análisis gramatical, pero, como ya se expuso, a menudo no importa. Tomemos por ejemplo el caso de las secuencias de sentencias (ejemplo 3.9). Escribimos la gramática como sigue, en forma recursiva por la derecha:

secuencia-sent → *sent* **;** *secuencia-sent* | *sent*
sent → **s**

Esta regla tiene la forma $A \rightarrow \alpha\ A \mid \beta$, con A = *secuencia-sent*, α = *sent* **;** y β = *sent*. En EBNF esto aparecería como

secuencia-sent → { *sent* **;** } *sent* (forma recursiva por la derecha)

También podríamos haber usado una regla recursiva por la izquierda y obtenido la EBNF

secuencia-sent → *sent* { **;** *sent* } (forma recursiva por la izquierda)

De hecho, la segunda forma es la que generalmente se utiliza (por razones que analizaremos en el siguiente capítulo).

Un problema más importante se presenta cuando se involucra la asociatividad, como ocurre con las operaciones binarias como la resta y la división. Por ejemplo, consideremos la primera regla gramatical en la gramática de expresión simple de la anterior subsección:

exp → *exp opsuma term* | *term*

Esto tiene la forma $A \rightarrow A\ \alpha \mid \beta$, con A = *exp*, α = *opsuma term* y β = *term*. De este modo, escribimos esta regla en EBNF como

exp → *term* { *opsuma term* }

Ahora también debemos suponer que esto implica asociatividad por la izquierda, aunque la regla misma no lo establezca explícitamente. Debemos suponer que una regla asociativa por la derecha estaría implicada al escribir

exp → { *term opsuma* } *term*

pero éste no es el caso. En cambio, una regla recursiva por la derecha como

secuencia-sent → *sent* **;** *secuencia-sent* | *sent*

se visualiza como si fuera una *sentencia* seguida opcionalmente por un signo de punto y coma y una *secuencia-sent*.

Las construcciones opcionales en EBNF se indican encerrándolas entre corchetes [. . .]. Esto es similar en esencia a la convención de la expresión regular de colocar un signo de interrogación después de una parte opcional, pero tiene la ventaja de encerrar la parte opcional sin requerir paréntesis. Por ejemplo, las reglas gramaticales para las sentencias if con partes opcionales else (ejemplos 3.4 y 3.6) se escribirían en EBNF como se describe a continuación:

sentencia → *sent-if* | `otro`
sent-if → `if` (*exp*) *sentencia* [`else` *sentencia*]
exp → `0` | `1`

Una regla recursiva por la derecha tal como

secuencia-sent → *sent* `;` *secuencia-sent* | *sent*

también se escribe como

secuencia-sent → *sent* [`;` *secuencia-sent*]

(compare esto con el uso de las llaves anterior para escribir esta regla en forma recursiva por la izquierda).

Si deseáramos escribir una operación aritmética tal como la suma en forma asociativa derecha, escribiríamos

exp → *term* [*opsuma exp*]

en lugar de utilizar llaves.

3.5.2 Diagramas de sintaxis

Las representaciones gráficas para simbolizar de manera visual las reglas de la EBNF se denominan **diagramas de sintaxis**. Se componen de cajas que representan terminales y no terminales, líneas con flechas que simbolizan secuencias y selecciones, y etiquetas de no terminales para cada diagrama que representan la regla gramatical que define ese no terminal. Para indicar terminales en un diagrama se emplea una caja de forma oval o redonda, mientras que para indicar no terminales se utiliza una caja rectangular o cuadrada.

Considere como ejemplo la regla gramatical

factor → (*exp*) | **`número`**

Ésta se escribe como un diagrama sintáctico de la siguiente manera:

Advierta que *factor* no está situado en una caja, pero se utiliza como una etiqueta para el diagrama de sintaxis, lo cual indica que el diagrama representa la definición de la estructura de ese nombre. Advierta también el uso de las líneas con flechas para indicar la selección y la secuenciación.

Los diagramas de sintaxis se escriben desde la EBNF más que desde la BNF, de modo que necesitamos diagramas que representen construcciones de repetición y opcionales. Dada una repetición tal como

$$A \rightarrow \{ B \}$$

el diagrama de sintaxis correspondiente se dibuja por lo regular como sigue:

Advierta que el diagrama debe tener en cuenta que no aparezca ninguna *B*.

Una construcción opcional tal como

$$A \rightarrow [B]$$

se dibuja como

Concluiremos nuestro análisis de los diagramas de sintaxis con algunos ejemplos en los que se utilizan ejemplos anteriores de EBNF.

Ejemplo 3.10

Considere nuestro ejemplo de ejecución de expresiones aritméticas simples. Éste tiene la BNF (incluyendo asociatividad y precedencia).

exp → *exp opsuma term* | *term*
opsuma → + | -
term → *term opmult factor* | *factor*
opmult → *
factor → (*exp*) | **número**

La EBNF correspondiente es

exp → *term* { *opsuma term* }
opsuma → + | -
term → *factor* { *opmult factor* }
opmult → *
factor → (*exp*) | **número**

Los diagramas de sintaxis correspondientes se proporcionan en la figura 3.4 (el diagrama de sintaxis para *factor* se dio con anterioridad).

Figura 3.4
Diagramas de sintaxis para la gramática del ejemplo 3.10

§

Ejemplo 3.11 Considere la gramática de las sentencias if simplificadas del ejemplo 3.4 (página 103). Éste tiene la BNF

sentencia → *sent-if* | `otro`
sent-if → `if` (*exp*) *sentencia*
| `if` (*exp*) *sentencia* `else` *sentencia*
exp → `0` | `1`

y la EBNF

sentencia → *sent-if* | `otro`
sent-if → `if` (*exp*) *sentencia* [`else` *sentencia*]
exp → `0` | `1`

Los diagramas de sintaxis correspondientes se ofrecen en la figura 3.5.

Figura 3.5
Diagramas de sintaxis para la gramática del ejemplo 3.11

§

3.6 PROPIEDADES FORMALES DE LOS LENGUAJES LIBRES DE CONTEXTO

3.6.1 Una definición formal de los lenguajes libres de contexto

Presentamos aquí de una manera más formal y matemática algo de la terminología y definiciones que presentamos anteriormente en este capítulo. Comenzaremos por establecer una definición formal de una gramática libre de contexto.

Definición

Una **gramática libre de contexto** se compone de lo siguiente:

1. Un conjunto T de **terminales**.
2. Un conjunto N de **no terminales** (disjunto de T).
3. Un conjunto P de **producciones**, o **reglas gramaticales**, de la forma $A \rightarrow \alpha$, donde A es un elemento de N y α es un elemento de $(T \cup N)^*$ (una secuencia posiblemente vacía de terminales y no terminales).
4. Un **símbolo inicial** S del conjunto N.

Sea G una gramática definida del modo anterior, de manera que $G = (T, N, P, S)$. Un **paso** o **etapa de derivación** sobre G es de la forma $\alpha\ A\ \gamma \Rightarrow \alpha\ \beta\ \gamma$, donde α y γ son elementos de $(T \cup N)^*$ y $A \rightarrow \beta$ está en P. (La unión $T \cup N$ de los conjuntos de terminales y

no terminales se conoce en ocasiones como **conjunto de símbolos** de G, y una cadena α en $(T \cup N)^*$ se conoce como **forma de sentencia** u **oracional**.) La relación $\alpha \Rightarrow^* \beta$ se define como la cerradura transitiva de la relación de paso de la derivación $\Rightarrow$; es decir, $\alpha \Rightarrow^* \beta$ si y sólo si, existe una secuencia de 0 o más pasos de derivación ($n \geq 0$)

$$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_{n-1} \Rightarrow \alpha_n$$

tal que $\alpha = \alpha_1$ y $\beta = \alpha_n$. (Si $n = 0$, entonces $\alpha = \beta$.) Una **derivación** sobre la gramática G es de la forma $S \Rightarrow^* w$, donde $w \in T^*$ (es decir, w es una cadena de sólo terminales denominada **sentencia**) y S es el símbolo inicial de G.

El **lenguaje generado por G**, denotado por $L(G)$, se define como el conjunto $L(G) = \{w \in T^* | \text{ existe una derivación } S \Rightarrow^* w \text{ de } G\}$. Es decir, $L(G)$ es el conjunto de sentencias derivable de S.

Una **derivación por la izquierda** $S \Rightarrow^*_{lm} w$ es una derivación en la cual cada paso de derivación $\alpha A \gamma \Rightarrow \alpha \beta \gamma$ es de tal manera que $\alpha \in T^*$; es decir, α se compone sólo de terminales. De manera similar, una **derivación por la derecha** es aquella en la cual cada paso de derivación $\alpha A \gamma \Rightarrow \alpha \beta \gamma$ tiene la propiedad de que $\gamma \in T^*$.

Un **árbol de análisis gramatical** sobre la gramática G es un árbol etiquetado desde la raíz que tiene las siguientes propiedades:

1. Cada nodo está etiquetado con un terminal o un no terminal o ε.
2. El nodo raíz está etiquetado con el símbolo inicial S.
3. Cada nodo hoja está etiquetado con un terminal o con ε.
4. Cada nodo no hoja está etiquetado con un no terminal.
5. Si un nodo con etiqueta $A \in N$ tiene n hijos con etiquetas $X_1, X_2, \ldots, X_n$ (los cuales pueden ser terminales o no terminales), entonces $A \rightarrow X_1 X_2 \ldots X_n \in P$ (una producción de la gramática).

Cada derivación da origen a un árbol de análisis gramatical tal que cada paso $\alpha A \gamma \Rightarrow \alpha \beta \gamma$ en la derivación, con $\beta = X_1 X_2 \ldots X_n$ corresponde a la construcción de n hijos con etiquetas $X_1, X_2, \ldots, X_n$ del nodo con etiqueta A. En general, muchas derivaciones pueden dar origen al mismo árbol de análisis gramatical. Sin embargo, cada árbol de análisis gramatical tiene una única derivación por la izquierda y por la derecha que le da origen. La derivación por la izquierda corresponde a un recorrido preorden del árbol de análisis gramatical, mientras la derivación por la derecha corresponde a un recorrido inverso postorden del árbol de análisis gramatical.

Se dice que un conjunto de cadenas L es un **lenguaje libre de contexto** si existe una gramática G libre de contexto tal que $L = L(G)$. En general, muchas gramáticas diferentes generan el mismo lenguaje libre de contexto, pero las cadenas en el lenguaje tendrán árboles de análisis gramatical diferentes dependiendo de la gramática utilizada.

Una gramática G es **ambigua** si existe una cadena $w \in L(G)$ tal que w tenga dos árboles de análisis gramatical distintos (o derivaciones por la izquierda o por la derecha).

3.6.2 Reglas gramaticales como ecuaciones

Al principio de esta sección advertimos que las reglas gramaticales utilizan el símbolo de la flecha en lugar de un signo de igualdad para representar la definición de los nombres de las estructuras (no terminales), a diferencia de nuestra notación para las expresiones regulares, donde definíamos los nombres de éstas mediante el signo de igualdad. La razón que se dio fue que la naturaleza recursiva de las reglas gramaticales hace la definición de las

relaciones (las reglas gramaticales) menos parecidas a la igualdad, y vimos que en realidad las cadenas definidas por reglas gramaticales resultan de derivaciones, donde se utiliza un método de reemplazo de izquierda a derecha que sigue la dirección de la flecha en la BNF.

Sin embargo, hay un sentido en el que aún se mantiene la igualdad de los lados izquierdo y derecho en una regla gramatical, pero el proceso de definición de lenguaje que este punto de vista da como resultado es diferente. Este punto de vista es importante para la teoría de la semántica de lenguajes de programación y vale la pena estudiarlo brevemente por su penetración en los procesos recursivos, como el análisis sintáctico, aun cuando los algoritmos de análisis sintáctico que estudiamos no se basen en ello.

Considere, por ejemplo, la siguiente regla gramatical, que se extrajo (en forma simplificada) de nuestra gramática de expresión simple:

$$exp \rightarrow exp + exp \mid \texttt{número}$$

Ya vimos que un nombre de no terminal como *exp* define un conjunto de cadenas de terminales (el cual es el lenguaje de la gramática si el no terminal es el símbolo inicial). Suponga que nombramos a este conjunto **E** y que **N** es el conjunto de los números naturales (correspondiente al nombre de la expresión regular **`número`**). Entonces, la regla gramatical dada se puede interpretar como la ecuación de conjuntos

$$\mathbf{E} = (\mathbf{E} + \mathbf{E}) \cup \mathbf{N}$$

donde **E** + **E** es el conjunto de cadenas $\{u + v \mid u,v \in \mathbf{E}\}$. (No estamos *sumando* las cadenas u y v aquí, sino haciendo la concatenación de ellas con el símbolo + entre las mismas.)

Ésta es una ecuación recursiva para el conjunto **E**. Considere cómo puede definir esto a **E**. En primer lugar, el conjunto **N** está contenido en **E** (el caso base), puesto que **E** es la unión de **N** y **E** + **E**. Acto seguido, el hecho de que **E** + **E** también esté contenido en **E** implica que **N** + **N** está contenido en **E**. Pero entonces, como tanto **N** como **N** + **N** están contenidos en **E**, también lo está **N** + **N** + **N**, y así sucesivamente. Podemos visualizar esto como una construcción inductiva de más y más cadenas largas y más largas, y la unión de todos estos conjuntos es el resultado deseado:

$$\mathbf{E} = \mathbf{N} \cup (\mathbf{N} + \mathbf{N}) \cup (\mathbf{N} + \mathbf{N} + \mathbf{N}) \cup (\mathbf{N} + \mathbf{N} + \mathbf{N} + \mathbf{N}) \cup \ldots$$

De hecho, puede demostrarse que esta **E** satisface la ecuación en cuestión. En realidad, **E** es el conjunto *más pequeño* que lo hace. Si observamos el lado derecho de la ecuación para **E** como una función (conjunto) de **E**, de manera que definamos $f(s) = (s + s) \cup \mathbf{N}$, entonces la ecuación para **E** se convierte en $\mathbf{E} = f(\mathbf{E})$. En otras palabras, **E** es un **punto fijo** de la función f y (de acuerdo con nuestro comentario anterior) en realidad es el punto fijo más pequeño. Decimos que **E**, de la manera en que está definido por este método, está proporcionando una **semántica de punto fijo mínimo**.

Puede demostrarse que todas las estructuras de lenguaje de programación definidas recursivamente, como la sintaxis, tipos de datos recursivos y funciones recursivas, tienen semántica de punto fijo mínimo cuando se implementan según los algoritmos habituales que estudiaremos más adelante en este libro. Esto es un punto importante, ya que los métodos como éste se pueden emplear en el futuro para verificar la exactitud de los compiladores. Actualmente es raro que se pruebe si los compiladores son correctos. En su lugar sólo se realizan pruebas del código del compilador para asegurar una aproximación de su exactitud, y a menudo se mantienen errores sustanciales, incluso en compiladores producidos comercialmente.

3.6.3 La jerarquía de Chomsky y los límites de la sintaxis como reglas libres de contexto

Cuando se representa la estructura sintáctica de un lenguaje de programación, las gramáticas libres de contexto en BNF o EBNF son una herramienta útil y poderosa. Pero también es importante saber lo que puede o debería ser representado por la BNF. Ya vimos una situación en la cual la gramática se puede mantener ambigua de manera intencional (el problema del else ambiguo), y de ese modo no expresar la sintaxis completa directamente. Pueden surgir otras situaciones donde podamos intentar expresar demasiado en la gramática, o donde pueda ser imposible expresar un requerimiento en la gramática. En esta sección analizaremos algunos de los casos comunes.

Una cuestión que surge con frecuencia cuando escribimos la BNF para un lenguaje es la extensión para la cual la estructura léxica debería expresarse en la BNF más que en una descripción separada (posiblemente utilizando expresiones regulares). El análisis anterior mostró que las gramáticas libres de contexto pueden expresar concatenación, repetición y selección, exactamente como pueden hacerlo las expresiones regulares. Por lo tanto, podríamos escribir reglas gramaticales para la construcción de todos los tokens de caracteres y prescindir del todo de las expresiones regulares.

Por ejemplo, consideremos la definición de un número como una secuencia de dígitos utilizando expresiones regulares:

```
dígito = 0|1|2|3|4|5|6|7|8|9
número = dígito dígito*
```

En cambio, podemos escribir esta definición utilizando BNF como

$$\begin{aligned} &\textit{dígito} \rightarrow \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \mathtt{3} \mid \mathtt{4} \mid \mathtt{5} \mid \mathtt{6} \mid \mathtt{7} \mid \mathtt{8} \mid \mathtt{9} \\ &\textit{número} \rightarrow \textit{número dígito} \mid \textit{dígito} \end{aligned}$$

Advierta que la recursión en la segunda regla se utiliza sólo para expresar repetición. Se dice que una gramática con esta propiedad es una **gramática regular**, y las gramáticas regulares pueden expresar todo lo que pueden hacer las expresiones regulares. Una consecuencia de esto es que podríamos diseñar un analizador sintáctico que podría aceptar caracteres directamente del archivo fuente de entrada y prescindir por completo del analizador léxico.

¿Por qué no es una buena idea hacer esto? Porque podría quedar comprometida la eficiencia. Un analizador sintáctico es una máquina más poderosa que un analizador léxico, pero es proporcionalmente menos eficiente. Aun así, puede ser razonable y útil incluir una definición de los tokens en la BNF en sí: la gramática expresaría entonces la estructura sintáctica completa, incluyendo la estructura léxica. Naturalmente, el implementador del lenguaje esperaría extraer estas definiciones de la gramática y convertirlas en un analizador léxico.

Una situación diferente se presenta respecto a las **reglas de contexto**, las cuales se presentan frecuentemente en lenguajes de programación. Hemos estado utilizando el término "libre de contexto" sin explicar por qué tales reglas son en efecto "libres de contexto". La razón más simple es que los no terminales aparecen por sí mismos a la izquierda de la flecha en las reglas libres de contexto. De este modo, una regla

$$A \rightarrow \alpha$$

dice que A se puede reemplazar por α *en cualquier sitio*, sin importar dónde se presente. Por otra parte, podríamos definir un **contexto** de manera informal como un par de cadenas (de terminales y no terminales) β, γ, tales que una regla se aplicaría sólo si β se presenta antes y γ se presenta después del no terminal. Podríamos escribir esto como

$$\beta A \gamma \rightarrow \beta\alpha\gamma$$

Una regla de esta clase en la que $\alpha \neq \varepsilon$ se denomina **regla gramatical sensible al contexto**. Las gramáticas sensibles al contexto son más poderosas que las gramáticas libres de contexto, pero son mucho más difíciles de utilizar como la base para un analizador sintáctico.

¿Qué clase de requerimientos en los lenguajes de programación requieren de reglas sensibles al contexto? Los ejemplos típicos involucran el uso de nombres. La regla en C que requiere **declaración antes del uso** es un ejemplo común. Aquí, un nombre debe aparecer en una declaración antes que sea permitido su uso en una sentencia o una expresión:

```
{int x;
 ...
 ...x...
 ...
}
```

Si intentáramos abordar este requerimiento utilizando reglas de BNF, tendríamos que, en primer lugar, incluir las cadenas de nombre mismas en las reglas gramaticales en vez de incluir todos los nombres como tokens de identificador que son indistinguibles. En segundo lugar, tendríamos que escribir para cada nombre una regla estableciendo su declaración antes de un uso potencial. Pero en muchos lenguajes la longitud de un identificador no está restringida, de modo que el número de posibles identificadores es (al menos potencialmente) infinito. Incluso si permitiéramos que los nombres fueran de sólo dos caracteres de longitud, tendríamos el potencial para cientos de nuevas reglas gramaticales. Evidentemente, esta situación es imposible. La solución es semejante a la de una regla de eliminación de ambigüedad: simplemente establecemos una regla (declaración antes del uso) que no está explícita en la gramática. Sin embargo, existe una diferencia: una regla así no se puede imponer por el analizador sintáctico mismo, puesto que está más allá del poder de expresar de las reglas libres de contexto (razonables). En su lugar, esta regla se vuelve parte del análisis semántico, porque depende del uso de la tabla de símbolos (que registra cuáles identificadores se han declarado).

El cuerpo de las reglas de lenguaje, al que se hace referencia como **semántica estática** del lenguaje, está más allá del alcance de verificación del analizador sintáctico, pero todavía puede ser verificado por el compilador. Esto incluye verificación de tipo (en un lenguaje con tipos suministrados estáticamente) y reglas como declaración antes del uso. De ahora en adelante consideraremos como *sintaxis* sólo aquellas reglas que se puedan expresar mediante reglas BNF. Todo lo demás se considerará como semántica.

Existe otra clase de gramática que es incluso más general que las gramáticas sensibles al contexto. Estas gramáticas se denominan **gramáticas sin restricciones** y tienen reglas gramaticales de la forma $\alpha \rightarrow \beta$, donde no hay restricciones en la forma de las cadenas α y β (excepto que α no puede ser ε). Las cuatro clases de gramáticas (sin restricciones, sensibles al contexto, libres de contexto y regulares) también se conocen como gramáticas de tipo 0, tipo 1, tipo 2 y tipo 3, respectivamente. A las clases de lenguaje que se construyen con ellas se les conoce también como **jerarquía de Chomsky**, en honor de Noam Chomsky, quien fue el primero en utilizarlas para describir los lenguajes naturales. Estas gramáticas representan niveles distintos de potencia computacional. En realidad, las gramáticas sin res-

tricciones (o de tipo 0) son equivalentes a las máquinas de Turing, de la misma manera que las gramáticas regulares son equivalentes a los autómatas finitos, y, por consiguiente, representan la clase más general de computación conocida. Las gramáticas libres de contexto también tienen un equivalente de máquina correspondiente, denominada autómata "de pila" (*pushdown*), pero no necesitaremos todo el poder de una máquina así para nuestros algoritmos de análisis sintáctico, así que no los analizaremos más.

También deberíamos ser cuidadosos respecto a que ciertos problemas computacionalmente intratables están asociados con gramáticas y lenguajes libres de contexto. Por ejemplo, al tratar con gramáticas ambiguas, sería excelente que pudiéramos establecer un algoritmo que convirtiera una gramática ambigua en una no ambigua sin modificar el lenguaje subyacente. Desgraciadamente, se sabe que éste es un problema que aún no se ha resuelto, de manera que posiblemente no exista un algoritmo así. De hecho, existen incluso lenguajes libres de contexto para los cuales *no* existe una gramática no ambigua (éstos se denominan **lenguajes inherentemente ambiguos**), e incluso no se ha resuelto el problema de la determinación de si un lenguaje es inherentemente ambiguo.

Por suerte, las complicaciones tales como una ambigüedad inherente no surgen como una regla en los lenguajes de programación, y las técnicas elaboradas ex profeso para la eliminación de la ambigüedad que describimos por lo regular han probado ser adecuadas en casos prácticos.

3.7 SINTAXIS DEL LENGUAJE TINY

3.7.1 Una gramática libre de contexto para TINY

La gramática para TINY está dada en BNF en la figura 3.6. A partir de ahí hacemos las siguientes observaciones. Un programa TINY es simplemente una secuencia de sentencias. Existen cinco clases de sentencias: sentencias if o condicionales, sentencias repeat o de repetición, sentencias read o de lectura, sentencias write o de escritura y sentencias assign o de asignación. Éstas tienen una sintaxis tipo Pascal, sólo que la sentencia if utiliza **end** como palabra clave de agrupación (de manera que no hay ambigüedad de else ambiguo en TINY) y que las sentencias if y las sentencias repeat permiten secuencias de sentencias como cuerpos, de modo que no son necesarios los corchetes o pares de **begin-end** (e incluso

Figura 3.6
Gramática del lenguaje TINY en BNF

programa → *secuencia-sent*
secuencia-sent → *secuencia-sent* **;** *sentencia* | *sentencia*
sentencia → *sent-if* | *sent-repeat* | *sent-assign* | *sent-read* | *sent-write*
sent-if → **if** *exp* **then** *secuencia-sent* **end**
| **if** *exp* **then** *secuencia-sent* **else** *secuencia-sent* **end**
sent-repeat → **repeat** *secuencia-sent* **until** *exp*
sent-assign → **identificador** **:=** *exp*
sent-read → **read identificador**
sent-write → **write** *exp*
exp → *exp-simple op-comparación exp-simple* | *exp-simple*
op-comparación → **<** | **=**
exp-simple → *exp-simple opsuma term* | *term*
addop → **+** | **-**
term → *term opmult factor* | *factor*
opmult → ***** | **/**
factor → **(** *exp* **)** | **número** | **identificador**

begin no es una palabra reservada en TINY). Las sentencias de entrada/salida comienzan mediante las palabras reservadas **read** y **write**. Una sentencia read de lectura puede leer sólo una variable a la vez, y una sentencia write de escritura puede escribir solamente una expresión a la vez.

Las expresiones TINY son de dos variedades: expresiones booleanas que utilizan los operadores de comparación = y < que aparecen en las pruebas de las sentencias if y repeat y expresiones aritméticas (denotadas por medio de *exp-simple* en la gramática) que incluyen los operadores enteros estándar +, -, * y / (este último representa la división entera, en ocasiones conocida como **div**). Las operaciones aritméticas son asociativas por la izquierda y tienen las precedencias habituales. Las operaciones de comparación, en contraste, son no asociativas: sólo se permite una operación de comparación por cada expresión sin paréntesis. Las operaciones de comparación también tienen menor precedencia que cualquiera de las operaciones aritméticas.

Los identificadores en TINY se refieren a variables enteras simples. No hay variables estructuradas, tales como arreglos o registros. Tampoco existen declaraciones de variable en TINY: una variable se declara de manera implícita al aparecer a la izquierda de una asignación. Además, existe sólo un ámbito (global), y no procedimientos o funciones (y, por lo tanto, tampoco llamadas a los mismos).

Una nota final acerca de la gramática de TINY. Las secuencias de sentencias *deben* contener signos de punto y coma separando las sentencias, y un signo de punto y coma después de la sentencia final en una secuencia de sentencias es ilegal. Esto se debe a que no hay sentencias vacías en TINY (a diferencia de Pascal y C). También escribimos la regla BNF para *secuencia-sent* como una regla recursiva por la izquierda, pero en realidad no importa la asociatividad de las secuencias de sentencias, porque el propósito es simplemente que se ejecuten en orden. De este modo, también podríamos simplemente haber escrito la regla *secuencia-sent* recursivamente a la derecha en su lugar. Esta observación también se representará en la estructura del árbol sintáctico de un programa TINY, donde las secuencias de sentencias estarán representadas por listas en lugar de árboles. Ahora volveremos a un análisis de esta estructura.

3.7.2 Estructura del árbol sintáctico para el compilador TINY

En TINY existen dos clases básicas de estructuras: sentencias y expresiones. Existen cinco clases de sentencias (sentencias if, sentencias repeat, sentencias assign, sentencias read y sentencias write) y tres clases de expresiones (expresiones de operador, expresiones de constante y expresiones de identificador). Por lo tanto, un nodo de árbol sintáctico se clasificará principalmente como si fuera una sentencia o expresión y en forma secundaria respecto a la clase de sentencia o expresión. Un nodo de árbol tendrá un máximo de tres estructuras hijo (sólo se necesitan las tres en el caso de una sentencia if con una parte else). Las sentencias serán secuenciadas mediante un campo de hermanos en lugar de utilizar un campo hijo.

Los atributos que deben mantenerse en los nodos de árbol son como sigue (aparte de los campos ya mencionados). Cada clase de nodo de expresión necesita un atributo especial. Un nodo de constante necesita un campo para la constante entera que representa. Un nodo de identificador necesita un campo para contener el nombre del identificador. Y un nodo de operador necesita un campo para contener el nombre del operador. Los nodos de sentencia generalmente no necesitan atributos (aparte del de la clase de nodo que son). Sin embargo, por simplicidad, en el caso de las sentencias assign y read conservaremos el nombre de la variable que se está asignando o leyendo justo en el nodo de sentencia mismo (más que como un nodo hijo de expresión).

La estructura de nodo de árbol que acabamos de describir se puede obtener mediante las declaraciones en C dadas en la figura 3.7, que se repiten del listado correspondiente al archivo `globals.h` en el apéndice B (líneas 198-217). Advierta que utilizamos uniones en estas declaraciones para ayudar a conservar el espacio. Éstas también nos ayudarán a recordar cuáles atributos van con cada clase de nodo. Dos atributos adicionales que aún no hemos mencionado están presentes en la declaración. El primero es el atributo de contabilidad `lineno`; éste nos permite imprimir los números de líneas del código fuente con errores que puedan ocurrir en etapas posteriores de la traducción. El segundo es el campo `type`, el cual después necesitaremos para verificar el tipo de las expresiones (y solamente expresiones). Está declarado para ser del tipo enumerado `ExpType`; esto se estudiará con todo detalle en el capítulo 6.

Figura 3.7
Declaraciones en C para un nodo de árbol sintáctico TINY

```
typedef enum {StmtK,ExpK} NodeKind;
typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK}
              StmtKind;
typedef enum {OpK,ConstK,IdK} ExpKind;

/* ExpType se usa para verificación de tipo */
typedef enum {Void,Integer,Boolean} ExpType;

#define MAXCHILDREN 3

typedef struct treeNode
    { struct treeNode * child[MAXCHILDREN];
      struct treeNode * sibling;
      int lineno;
      NodeKind nodekind;
      union { StmtKind stmt; ExpKind exp;} kind;
      union { TokenType op;
                int val;
                char * name; } attr;
      ExpType type; /* para verificación de tipo de expresiones
*/
    } TreeNode;
```

Ahora queremos dar una descripción visual de la estructura del árbol sintáctico y mostrar visualmente este árbol para un programa de muestra. Para hacer esto utilizamos cajas rectangulares para indicar nodos de sentencia y cajas redondas u ovales para indicar nodos de expresión. La clase de sentencia o expresión se proporcionará como una etiqueta dentro de la caja, mientras que los atributos adicionales también se enumerarán allí entre paréntesis. Los apuntadores hermanos se dibujarán a la derecha de las cajas de nodo, mientras que los apuntadores hijos se dibujarán abajo de las cajas. También indicaremos otras estructuras de árbol que no se especifican en los diagramas mediante triángulos y líneas punteadas para indicar estructuras que puedan o no aparecer. Una secuencia de sentencias conectadas mediante campos de hermanos se vería entonces como se muestra a continuación (los subárboles potenciales están indicados mediante las líneas punteadas y los triángulos).

Una sentencia if (con potencialmente tres hijos) tendrá un aspecto como el siguiente.

Una sentencia repeat tendrá dos hijos. La primera es la secuencia de sentencias que representan su cuerpo; la segunda es la expresión de prueba:

Una sentencia de asignación ("assign") tiene un hijo que representa la expresión cuyo valor se asigna (el nombre de la variable que se asigna se conserva en el nodo de sentencia):

Una sentencia de escritura ("write") también tiene un hijo, que representa la expresión de la que se escribirá el valor:

Una expresión de operador tiene dos hijos, que representan las expresiones de operando izquierdo y derecho:

Todos los otros nodos (sentencias read, expresiones de identificador y expresiones de constante) son nodos hoja.

Finalmente estamos listos para mostrar el árbol de un programa TINY. El programa de muestra del capítulo 1 que calcula el factorial de un entero se repite en la figura 3.8. Su árbol sintáctico se muestra en la figura 3.9.

Figura 3.8
Programa de muestra en el lenguaje TINY

```
{ Programa de muestra
en lenguaje TINY-
calcula el factorial
}
read x; { entrada de un entero }
if 0 < x then { no calcula si x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { salida factorial de x }
end
```

Figura 3.9
Árbol sintáctico para el programa TINY de la figura 3.8

EJERCICIOS

3.1 **a.** Escriba una gramática no ambigua que genere el conjunto de cadenas {`s;`, `s;s;`, `s;s;s;`, . . .}.

b. Dé una derivación por la izquierda y por la derecha para la cadena `s;s;` utilizando su gramática.

3.2 Dada la gramática $A \rightarrow AA \mid (\ A\ ) \mid \varepsilon$,

a. Describa el lenguaje que genera.

b. Muestre que es ambiguo.

3.3 Dada la gramática

$$
\begin{aligned}
&exp \rightarrow exp\ opsuma\ term \mid term\\
&opsuma \rightarrow \texttt{+} \mid \texttt{-}\\
&term \rightarrow term\ opmult\ factor \mid factor\\
&opmult \rightarrow \texttt{*}\\
&factor \rightarrow \texttt{(}\ exp\ \texttt{)} \mid \textbf{\texttt{número}}
\end{aligned}
$$

escriba derivaciones por la izquierda, árboles de análisis gramatical y árboles sintácticos abstractos para las siguientes expresiones:

a. `3+4*5-6` **b.** `3*(4-5+6)` **c.** `3-(4+5*6)`

3.4 La gramática siguiente genera todas las expresiones regulares sobre el alfabeto de letras (utilizamos las comillas para encerrar operadores, puesto que la barra vertical también es un operador además de un metasímbolo):

$$
\begin{aligned}
rexp \rightarrow\ & rexp\ \text{“|”}\ rexp\\
\mid\ & rexp\ rexp\\
\mid\ & rexp\ \text{“*”}\\
\mid\ & \text{“(”}\ rexp\ \text{”)”}\\
\mid\ & \textbf{\texttt{letra}}
\end{aligned}
$$

a. Proporcione una derivación para la expresión regular `(ab|b)*` utilizando esta gramática.

b. Muestre que esta gramática es ambigua.

c. Vuelva a escribir esta gramática para establecer las precedencias correctas para los operadores (véase el capítulo 2).

d. ¿Qué asociatividad da su respuesta en el inciso c para los operadores binarios? Explique su respuesta.

3.5 Escriba una gramática para expresiones booleanas que incluya las constantes `true` y `false`, los operadores `and`, `or` y `not`, además de los paréntesis. Asegúrese de darle a `or` una precedencia más baja que a `and`, y a `and` una precedencia más baja que a `not`, además de permitir la repetición del operador `not` como en la expresión booleana `not not true`. Asegúrese también de que su gramática no sea ambigua.

3.6 Considere la gramática siguiente que representa expresiones simplificadas tipo LISP:

$$
\begin{aligned}
&\textit{lexp} \rightarrow \textit{atom} \mid \textit{list}\\
&\textit{atom} \rightarrow \texttt{número} \mid \texttt{identificador}\\
&\textit{list} \rightarrow \texttt{(}\ \textit{lexp-sec}\ \texttt{)}\\
&\textit{lexp-sec} \rightarrow \textit{lexp-sec}\ \textit{lexp} \mid \textit{lexp}
\end{aligned}
$$

a. Escriba una derivación por la izquierda y por la derecha para la cadena `(a 23 (m x y))`.

b. Dibuje un árbol de análisis gramatical para la cadena del inciso a.

3.7 **a.** Escriba declaraciones tipo C para una estructura de árbol sintáctico abstracto correspondiente a la gramática del ejercicio 3.6.

b. Dibuje el árbol sintáctico para la cadena `(a 23 (m x y))` que resultaría de sus declaraciones del inciso a.

3.8 Dada la gramática siguiente

$$
\begin{aligned}
&\textit{sentencia} \rightarrow \textit{sent-if} \mid \texttt{otro} \mid \varepsilon\\
&\textit{sent-if} \rightarrow \texttt{if}\ \ \texttt{(}\ \textit{exp}\ \texttt{)}\ \textit{sentencia}\ \textit{parte-else}\\
&\textit{parte-else} \rightarrow \texttt{else}\ \textit{sentencia} \mid \varepsilon\\
&\textit{exp} \rightarrow \texttt{0} \mid \texttt{1}
\end{aligned}
$$

a. Dibuje un árbol de análisis gramatical para la cadena

```
if(0) if(1) otro else else otro
```

b. ¿Cuál es el propósito de los dos `else`?

c. ¿Es un código similar permisible en C? Justifique su respuesta.

3.9 (Aho, Sethi y Ullman) Muestre que el siguiente intento para resolver la ambigüedad del else ambiguo todavía es ambiguo (compare con la solución de la página 121):

$$
\begin{aligned}
&\textit{sentencia} \rightarrow \texttt{if}\ \ \texttt{(}\ \textit{exp}\ \texttt{)}\ \textit{sentencia} \mid \textit{sent-igualada}\\
&\textit{sent-igualada} \rightarrow \texttt{if}\ \ \texttt{(}\ \textit{exp}\ \texttt{)}\ \textit{sent-igualada}\ \texttt{else}\ \textit{sentencia} \mid \texttt{otro}\\
&\textit{exp} \rightarrow \texttt{0} \mid \texttt{1}
\end{aligned}
$$

3.10 **a.** Traduzca la gramática del ejercicio 3.6 a EBNF.

b. Dibuje diagramas de sintaxis para la EBNF del inciso a.

3.11 Dada la ecuación de conjuntos $\mathbf{X} = (\mathbf{X} + \mathbf{X}) \cup \mathbf{N}$ ($\mathbf{N}$ es el conjunto de los números naturales; véase la sección 3.6.2, página 129).

a. Muestre que el conjunto $\mathbf{E} = \mathbf{N} \cup (\mathbf{N} + \mathbf{N}) \cup (\mathbf{N} + \mathbf{N} + \mathbf{N}) \cup (\mathbf{N} + \mathbf{N} + \mathbf{N} + \mathbf{N}) \cup \ldots$ satisface la ecuación.

b. Muestre que, dado cualquier conjunto $\mathbf{E}'$ que satisfaga la ecuación, $\mathbf{E} \subset \mathbf{E}'$.

3.12 Signos de menos unitarios se pueden agregar de diversas formas a la gramática de expresión aritmética simple del ejercicio 3.3. Adapte la BNF para cada uno de los casos que siguen de manera que satisfagan la regla establecida.

a. Como máximo se permite un signo de menos unitario en cada expresión, y debe aparecer al principio de una expresión, de manera que `-2-3` es legal[5] (y se evalúa a – 5) y `-2-(-3)` es legal, pero `-2--3` no lo es.

b. Como máximo se permite un signo de menos unitario antes de cierto número o paréntesis izquierdo, de manera que `-2--3` es legal pero `--2` y `-2---3` no lo son.

c. Se permite un número arbitrario de signos de menos antes de números y paréntesis izquierdos, de modo que cualquier caso de los anteriores es legal.

3.13 Considere la siguiente simplificación de la gramática del ejercicio 3.6.

$$
\begin{aligned}
&\textit{lexp} \rightarrow \texttt{número} \mid \texttt{(}\ \textit{op lexp-sec}\ \texttt{)}\\
&\textit{op} \rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*}\\
&\textit{lexp-sec} \rightarrow \textit{lexp-sec lexp} \mid \textit{lexp}
\end{aligned}
$$

Esta gramática puede pensarse como representación de expresiones aritméticas enteras simples en forma de prefijo tipo LISP. Por ejemplo, la expresión `34-3*42` se escribiría en esta gramática como `(- 34 (* 3 42))`.

a. ¿Qué interpretación debería darse a las expresiones legales `(- 2 3 4)` y `(- 2)`? ¿Cuál respecto a las expresiones `(+ 2)` y `(* 2)`?

b. ¿Son un problema la precedencia y la asociatividad con esta gramática? ¿La gramática es ambigua?

3.14 **a.** Escriba declaraciones tipo C para una estructura de árbol sintáctico para la gramática del ejercicio anterior.

b. Dibuje el árbol sintáctico para la expresión `(- 34 (* 3 42))` utilizando su respuesta para el inciso a.

3.15 Se mencionó en la página 111 que la descripción tipo BNF de un árbol sintáctico abstracto

$$
\begin{aligned}
&\textit{exp} \rightarrow \textit{OpExp}(\textit{op,exp,exp}) \mid \textit{ConstExp}(\textit{integer})\\
&\textit{op} \rightarrow \textit{Plus} \mid \textit{Minus} \mid \textit{Times}
\end{aligned}
$$

está cerca de una declaración tipo real en algunos lenguajes. Dos lenguajes de esta clase son ML y Haskell. Este ejercicio es para aquellos lectores que conocen alguno de esos dos lenguajes.

a. Escriba declaraciones de tipo de datos que implementen la sintaxis abstracta anterior.

b. Escriba una expresión sintáctica abstracta para la expresión `(34*(42-3))`.

3.16 Vuelva a escribir el árbol sintáctico `typedef` al principio de la sección 3.3.2 (página 111) para utilizar una `union`.

---

5. Advierta que el segundo signo de menos en esta expresión es un menos *binario*, no un menos unitario.

3.17 Demuestre que la gramática del ejemplo 3.5 (página 105) genera el conjunto de todas las cadenas de paréntesis balanceados, donde w es una cadena de paréntesis balanceados siempre que tengan las siguientes dos propiedades:

1. w contiene exactamente el mismo número de paréntesis izquierdos y derechos.
2. Cada prefijo u de w ($w = ux$ para alguna x) tiene al menos tantos paréntesis izquierdos como paréntesis derechos.

(*Sugerencia:* demuestre por inducción en la longitud de una derivación.)

3.18 **a.** Escriba una gramática sensible al contexto que genere cadenas de la forma xcx, donde x es una cadena de a y b.

b. ¿Es posible escribir una gramática libre de contexto para las cadenas del inciso a? Explique por qué.

3.19 En algunos lenguajes (por ejemplo Modula-2 y Ada), se espera que una declaración de procedimiento se termine mediante sintaxis que incluya el nombre del procedimiento. Por ejemplo, en Modula-2 un procedimiento se declara como se muestra a continuación:

```
PROCEDURE P;
BEGIN
  ...
END P;
```

Advierta el uso del nombre de procedimiento **P** después del **END** de cierre. ¿Se puede verificar un requerimiento así mediante un analizador sintáctico? Justifique su respuesta.

3.20 **a.** Escriba una expresión regular que genere el mismo lenguaje que la gramática siguiente:

$$A \rightarrow aA \mid B \mid \varepsilon$$
$$B \rightarrow bB \mid A$$

b. Escriba una gramática que genere el mismo lenguaje que la expresión regular que se muestra a continuación:

```
(a|c|ba|bc)*(b|ε)
```

3.21 Una **producción simple** es una selección de regla gramatical de la forma $A \rightarrow B$, donde tanto A como B son no terminales.

a. Muestre que las producciones simples pueden ser sistemáticamente eliminadas de una gramática para entregar una gramática sin producciones simples que genere el mismo lenguaje que la gramática original.

b. ¿Usted esperaría que aparecieran producciones simples con frecuencia en la definición de los lenguajes de programación? Justifique su respuesta.

3.22 Una **gramática cíclica** es aquella en la que existe una derivación $A \Rightarrow^* A$ para algún no terminal A.

a. Muestre que una gramática cíclica es ambigua.

b. ¿Usted esperaría que las gramáticas que definen lenguajes de programación fueran a menudo cíclicas? Justifique su respuesta.

3.23 Vuelva a escribir la gramática de TINY de la figura 3.6 en EBNF.

3.24 Para el programa TINY .

```
read x;
x := x+1;
write x
```

a. Dibuje el árbol de análisis gramatical de TINY.

b. Dibuje el árbol sintáctico de TINY.

3.25 Dibuje el árbol sintáctico de TINY para el programa siguiente:

```
read u;
read v; { entrada de dos enteros }
if v = 0 then v := 0 { no hacer nada }
else
  repeat
    temp := v;
    v := u - u/v*v;
    u := temp
  until v = 0
end;
write u { salida gcd de u y v original }
```

NOTAS Y REFERENCIAS

Gran parte de la teoría de las gramáticas libres de contexto se puede encontrar en Hopcroft y Ullman [1979], donde se pueden hallar demostraciones de muchas propiedades, tales como la indeterminación de la ambigüedad inherente. También contiene una relación de la jerarquía de Chomsky. Información adicional está en Ginsburg [1966, 1975]. Chomsky [1956, 1959] fue responsable de gran parte de la teoría inicial, la que aplicó al estudio de los lenguajes naturales. Sólo mucho después se comprendió la importancia del tema para los lenguajes de programación, y el primer uso de las gramáticas libres de contexto llegó en la definición de Algol60 (Naur [1963]). La solución de la página 121 para el problema del else ambiguo utilizando reglas libres de contexto, así como el ejercicio 3.9, se tomó de Aho, Hopcroft y Ullman [1986]. El enfoque de las gramáticas libres de contexto como ecuaciones recursivas (sección 3.6.2) se tomó de la semántica denotacional. Para un estudio de la misma véase Schmidt [1986].

Capítulo 4

Análisis sintáctico descendente

4.1 Análisis sintáctico descendente mediante método descendente recursivo

4.2 Análisis sintáctico LL(1)

4.3 Conjuntos Primero y Siguiente

4.4 Un analizador sintáctico descendente recursivo para el lenguaje TINY

4.5 Recuperación de errores en analizadores sintácticos descendentes

Un algoritmo de análisis sintáctico **descendente** analiza una cadena de tokens de entrada mediante la búsqueda de los pasos en una derivación por la izquierda. Un algoritmo así se denomina descendente debido a que el recorrido implicado del árbol de análisis gramatical es un recorrido de preorden y, de este modo, se presenta desde la raíz hacia las hojas (véase la sección 3.3 del capítulo 3). Los analizadores sintácticos descendentes existen en dos formas: **analizadores sintácticos inversos** o **en reversa** y **analizadores sintácticos predictivos**. Un analizador sintáctico predictivo intenta predecir la siguiente construcción en la cadena de entrada utilizando uno o más tokens de búsqueda por adelantado, mientras que un analizador sintáctico inverso intentará las diferentes posibilidades para un análisis sintáctico de la entrada, respaldando una cantidad arbitraria en la entrada si una posibilidad falla. Aunque los analizadores sintácticos inversos son más poderosos que los predictivos, también son mucho más lentos, ya que requieren una cantidad de tiempo exponencial en general y, por consiguiente, son inadecuados para los compiladores prácticos. Aquí no estudiaremos los analizadores sintácticos inversos (pero revise la sección de notas y referencias y los ejercicios para unos cuantos apuntes y sugerencias sobre este tema).

Las dos clases de algoritmos de análisis sintáctico descendente que estudiaremos aquí se denominan **análisis sintáctico descendente recursivo** y **análisis sintáctico LL(1)**. El análisis sintáctico descendente recursivo además de ser muy versátil, es el método más adecuado para un analizador sintáctico manuscrito, por lo tanto, lo estudiaremos primero. El análisis sintáctico LL(1) se abordará después, aunque ya no se usa a menudo en la práctica, es útil estudiarlo como un esquema simple con una pila explícita, y puede servirnos como un preludio para los algoritmos ascendentes más poderosos (pero también más complejos) del capítulo siguiente. Nos ayudará también al formalizar algunos de los problemas que aparecen en el modo descendente recursivo. El método de análisis sintáctico LL(1) debe su nombre a lo siguiente. La primera "L" se refiere al hecho de que se procesa la entrada de izquierda a derecha, del inglés "Left-right" (algunos analizadores sintácticos antiguos empleaban para procesar la entrada de derecha a izquierda, pero eso es poco habitual en la actualidad). La segunda "L" hace referencia al hecho de que rastrea una derivación por la izquierda para la cadena de entrada. El número 1 entre paréntesis significa que se utiliza sólo un símbolo de entrada para predecir la dirección del análisis sintáctico. (También es posible tener análisis sintáctico LL(k), utilizando k símbolos de búsqueda hacia delante. Estudiaremos esto a grandes rasgos más adelante en el capítulo, pero un símbolo de búsqueda hacia delante es el caso más común.)

Tanto el análisis sintáctico descendente recursivo como el análisis sintáctico LL(1) requieren en general del cálculo de conjuntos de búsqueda hacia delante que se conocen como conjuntos **Primero** y **Siguiente**.[1] Puesto que se pueden construir analizadores sin construir estos conjuntos de manera explícita, pospondremos un análisis de los mismos hasta que hayamos introducido los algoritmos básicos. Entonces continuaremos con un análisis de un analizador sintáctico de TINY construido de manera descendente recursiva y cerraremos el capítulo con una descripción de los métodos de recuperación de errores en el análisis sintáctico descendente.

4.1 ANÁLISIS SINTÁCTICO DESCENDENTE MEDIANTE MÉTODO DESCENDENTE RECURSIVO

4.1.1 El método básico descendente recursivo

La idea del análisis sintáctico descendente recursivo es muy simple. Observamos la regla gramatical para un no terminal *A* como una definición para un procedimiento que reconocerá una *A*. El lado derecho de la regla gramatical para *A* especifica la estructura del código para este procedimiento: la secuencia de terminales y no terminales en una selección corresponde a concordancias de la entrada y llamadas a otros procedimientos, mientras que las selecciones corresponden a las alternativas (sentencias case o if) dentro del código.

Como primer ejemplo consideremos la gramática de expresión del capítulo anterior:

exp → *exp opsuma term* | *term*
opsuma → + | -
term → *term opmult factor* | *factor*
opmult → *
factor → (*exp*) | **número**

y consideremos la regla gramatical para un *factor*. Un procedimiento descendente recursivo que reconoce un *factor* (y al cual llamaremos por el mismo nombre) se puede escribir en pseudocódigo de la manera siguiente:

```
procedure factor ;
begin
  case token of
  ( :   match( ( ) ;
        exp ;
        match( ) ) ;
  number :
        match(number) ;
  else error ;
  end case ;
end factor ;
```

En este pseudocódigo partimos del supuesto de que existe una variable *token* que mantiene el siguiente token actual en la entrada (de manera que este ejemplo utiliza un símbolo de búsqueda hacia delante). También dimos por hecho que existe un procedimiento *match* que compara el token siguiente actual con su parámetro, avanza la entrada si tiene éxito y declara un error si no lo tiene:

1. Estos conjuntos también son necesarios en algunos de los algoritmos para el análisis sintáctico ascendente que se estudian en el capítulo siguiente.

```
procedure match ( expectedToken ) ;
begin
  if token = expectedToken then
    getToken ;
  else
    error ;
  end if ;
end match ;
```

Por el momento dejaremos sin especificar el procedimiento *error* que es llamado tanto por *match* como por *factor*. Se puede suponer que se imprime un mensaje de error y se sale del programa.

Advierta que en las llamadas *match*( ) y *match*(`number`) en *factor*, se sabe que las variables *expectedToken* y *token* son las mismas. Sin embargo, en la llamada *match*( ), no se puede suponer que *token* sea un paréntesis derecho, de manera que es necesaria una prueba. El código para *factor* también supone que se ha definido un procedimiento *exp* que puede llamarlo. En un analizador sintáctico descendente recursivo para la gramática de expresión el procedimiento *exp* llamará a *term*, el procedimiento term llamará a *factor* y el procedimiento *factor* llamará a *exp*, de manera que todos estos procedimientos deben ser capaces de llamarse entre sí. Desgraciadamente, escribir procedimientos descendentes recursivos para las reglas restantes en la gramática de expresión no es tan fácil como para *factor* y requiere el uso de EBNF, como veremos a continuación.

4.1.2 Repetición y selección: el uso de EBNF

Considere como un segundo ejemplo la regla gramatical (simplificada) para una sentencia if:

sent-if → `if` (*exp*) *sentencia*
| `if` (*exp*) *sentencia* `else` *sentencia*

Esto se puede traducir en el procedimiento

```
procedure ifStmt ;
begin
  match (if) ;
  match ( ( ) ;
  exp ;
  match ( ) ) ;
  statement ;
  if token = else then
    match (else) ;
    statement ;
  end if ;
end ifStmt ;
```

En este ejemplo podríamos no distinguir de inmediato las dos selecciones en el lado derecho de la regla gramatical (ambas comienzan con el token `if`). En su lugar, debemos aplazar la decisión acerca de reconocer la parte-else opcional hasta que veamos el token `else` en la entrada. De este modo, el código para la sentencia if corresponde más a la EBNF

sent-if → `if` (*exp*) *sentencia* [`else` *sentencia*]

que a la BNF, donde los corchetes de la EBNF se traducen en una prueba en el código para *Sent-if*. De hecho, la notación EBNF está diseñada para reflejar muy de cerca el código real de un analizador sintáctico descendente recursivo, de modo que una gramática debería siempre traducirse en EBNF si se está utilizando el modo descendente recursivo. Advierta también que, aun cuando esta gramática es ambigua (véase el capítulo anterior), es natural escribir un analizador sintáctico que haga concordar cada token `else` tan pronto como se encuentre en la entrada. Esto corresponde precisamente a la regla de eliminación de ambigüedad más cercanamente anidada.

Considere ahora el caso de una *exp* en la gramática para expresiones aritméticas simples en BNF:

$$exp \rightarrow exp\ opsuma\ term \mid term$$

Si estamos intentando convertir esto en un procedimiento *exp* recursivo de acuerdo con nuestro plan, lo primero que podemos intentar hacer es llamar a *exp* mismo, y esto conduciría a un inmediato ciclo recursivo infinito. Tratar de probar cuál selección usar (*exp* → *exp opsuma term*, o bien, *exp* → *term*) es realmente problemático, ya que tanto *exp* como *term* pueden comenzar con los mismos tokens (un número o paréntesis izquierdo).

La solución es utilizar la regla de EBNF

$$exp \rightarrow term\ \{\ opsuma\ term\ \}$$

en su lugar. Las llaves expresan la repetición que se puede traducir al código para un ciclo o bucle de la manera siguiente:

```
procedure exp ;
begin
  term ;
  while token = + or token = - do
    match (token) ;
    term ;
  end while ;
end exp ;
```

De manera similar, la regla EBNF para *term*:

$$term \rightarrow factor\ \{\ opmult\ factor\ \}$$

se convierte en el código

```
procedure term ;
begin
  factor ;
  while token = * do
    match (token) ;
    factor ;
  end while ;
end term ;
```

Aquí eliminamos los no terminales *opsuma* y *opmult* como procedimientos separados, ya que su única función es igualar a los operadores:

$$opsuma \rightarrow + \mid -$$
$$opmult \rightarrow *$$

En su lugar haremos esta concordancia dentro de *exp* y *term*.

Una cuestión que surge con este código es si la asociatividad izquierda implicada por las llaves (y explícita en la BNF original) todavía puede mantenerse. Suponga, por ejemplo, que deseamos escribir una calculadora descendente recursiva para la aritmética entera simple de nuestra gramática. Podemos asegurarnos de que las operaciones son asociativas por la izquierda realizando las operaciones a medida que circulamos a través del ciclo o bucle (suponemos ahora que los procedimientos de análisis sintáctico son funciones que devuelven un resultado entero):

```
function exp : integer ;
var temp : integer ;
begin
  temp := term ;
  while token = + or token = - do
    case token of
     + : match (+) ;
         temp := temp + term ;
     - : match (-) ;
         temp := temp - term ;
    end case ;
  end while ;
  return temp ;
end exp ;
```

y de manera similar para *term*. Empleamos estas ideas para crear una calculadora simple funcionando, cuyo código en C está dado en la figura 4.1 (páginas 148-149), donde en lugar de escribir un analizador léxico completo, optamos por utilizar llamadas a `getchar` y `scanf` en lugar de un procedimiento `getToken`.

Este método de convertir reglas gramaticales en EBNF a código es muy poderoso, y lo utilizaremos para proporcionar un analizador sintáctico completo para el lenguaje TINY de la sección 4.4. Sin embargo, existen algunas dificultades y debemos tener cuidado al programar las acciones dentro del código. Un ejemplo de esto está en el pseudocódigo anterior para *exp*, donde tiene que ocurrir una concordancia de las operaciones antes de las llamadas repetidas a *term* (de otro modo *term* vería una operación como su primer token, lo que generaría un error). En realidad, el protocolo siguiente para conservar la variable global *token* actual debe estar rígidamente adherido: *token* debe estar establecido antes que comience el análisis sintáctico y *getToken* (o su equivalente) debe ser llamado precisamente después de haber probado con éxito un token (ponemos esto en el procedimiento *match* en el pseudocódigo).

También debe tenerse el mismo cuidado al programar las acciones durante la construcción de un árbol sintáctico. Vimos que la asociatividad por la izquierda en una EBNF con repetición puede mantenerse para propósitos de cálculo al realizar éste a medida que se ejecuta el ciclo.

Figura 4.1 (inicio)
Una calculadora descendente recursiva para aritmética entera simple

```
/* Calculadora de aritmética entera simple según el EBNF:

    <exp> -> <term> { <opsuma> <term> }
    <opsuma> -> + | -
    <term> -> <factor> { <opmult> <factor> }
    <opmult> -> *
    <factor> -> ( <exp> ) | Número

    Introduce una línea de texto desde stdin
    Sale "Error" o el resultado.
*/

#include <stdio.h>
#include <stdlib.h>

char token; /* variable token global */

/* prototipos de función para llamadas recursivas */
int exp(void);
int term(void);
int factor(void);

void error(void)
{ fprintf(stderr,"Error\n");
  exit(1);
}

void match( char expectedToken)
{ if (token==expectedToken) token = getchar();
  else error();
}

main()
{ int result;
  token = getchar(); /* carga token con el primer carácter
                        para búsqueda hacia delante */
  result = exp();
  if (token=='\n') /* verifica el fin de línea */
    printf("Result = %d\n",result);
  else error(); /* caracteres extraños en línea */
  return 0;
}
```

Figura 4.1 (conclusión)
Una calentadora descendente recursiva para aritmética entera simple

```
int exp(void)
{ int temp = term();
  while ((token=='+')||(token=='-'))
    switch (token) {
    case '+': match('+');
              temp+=term();
              break;
    case '-': match('-');
              temp-=term();
              break;
    }
  return temp;
}

int term(void)
{ int temp = factor();
  while (token=='*') {
    match('*');
    temp*=factor();
  }
  return temp;
}

int factor(void)
{ int temp;
  if (token=='(') {
    match('(');
    temp = exp();
    match(')');
  }
  else if (isdigit(token)) {
    ungetc(token,stdin);
    scanf("%d",&temp);
    token = getchar();
  }
  else error();
  return temp;
}
```

Sin embargo, esto ya no corresponde a una construcción descendente del árbol gramatical o sintáctico. En realidad, si consideramos la expresión **3+4+5**, con árbol sintáctico

el nodo que representa la suma de 3 y 4 debe ser creado (o procesado) antes del nodo raíz (el nodo que representa su suma con 5). Al traducir esto a la construcción de árbol sintáctico real, tenemos el siguiente pseudocódigo para el procedimiento *exp*:

```
function exp : syntaxTree ;
var temp, newtemp : syntaxTree ;
begin
  temp := term ;
  while token = + or token = - do
    case token of
     + : match (+) ;
         newtemp := makeOpNode(+) ;
         leftChild(newtemp) := temp ;
         rightChild(newtemp) := term ;
         temp := newtemp ;
     - : match (-) ;
         newtemp := makeOpNode(-) ;
         leftChild(newtemp) := temp ;
         rightChild(newtemp) := term ;
         temp := newtemp ;
    end case ;
  end while ;
  return temp ;
end exp ;
```

o la versión más simple

```
function exp : syntaxTree ;
var temp, newtemp : syntaxTree ;
begin
  temp := term ;
  while token = + or token = - do
    newtemp := makeOpNode(token) ;
    match (token) ;
    leftChild(newtemp) := temp ;
    rightChild(newtemp) := term ;
    temp := newtemp ;
  end while ;
  return temp ;
end exp ;
```

En este código utilizamos una nueva función *makeOpNode*, que recibe un token de operador como un parámetro y devuelve un nodo de árbol sintáctico recién construido. También indicamos la asignación de un árbol sintáctico *p* como un hijo a la izquierda o a la derecha de un árbol sintáctico *t* al escribir *leftChild*(*t*) := *p* o *rightChild*(*t*) := *p*. Con este pseudocódigo el procedimiento *exp* en realidad construye el árbol sintáctico y no el árbol de análisis gramatical. Esto se debe a que una llamada a *exp* no construye invariablemente un nuevo nodo del árbol; si no hay operadores, *exp* simplemente pasa de regreso al árbol recibido desde la llamada inicial a *term* como su propio valor. También se puede escribir el pseudocódigo correspondiente para *term* y *factor* (véanse los ejercicios).

En contraste, el árbol sintáctico para una sentencia if se puede construir de manera estrictamente descendente por medio de un analizador sintáctico descendente recursivo:

```
function ifStatement : syntaxTree ;
var temp : syntaxTree ;
begin
  match (if) ;
  match ( ( ) ;
  temp := makeStmtNode(if) ;
  testChild(temp) := exp ;
  match ( ) ) ;
  thenChild(temp) := statement ;
  if token = else then
    match (else) ;
    elseChild(temp) := statement ;
  else
    elseChild(temp) := nil ;
  end if ;
end ifStatement ;
```

La flexibilidad del análisis sintáctico descendente recursivo, al permitir al programador ajustar la programación de las acciones, lo hace el método de elección para analizadores sintácticos generados a mano.

4.3 Otros problemas de decisión

El método descendente recursivo que describimos es muy poderoso, pero todavía es de propósito específico. Con un lenguaje pequeño cuidadosamente diseñado (tal como TINY o incluso C), estos métodos son adecuados para construir un analizador sintáctico completo. Deberíamos estar conscientes de que pueden ser necesarios más métodos formales en situaciones complejas. Pueden surgir diversos problemas. En primer lugar, puede ser difícil convertir una gramática originalmente escrita en BNF en forma EBNF. Una alternativa para el uso de la EBNF se estudia en la siguiente sección, donde se construye una BNF transformada que en esencia es equivalente a la EBNF. En segundo lugar, cuando se formula una prueba para distinguir dos o más opciones de regla gramatical

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

puede ser difícil decidir cuándo utilizar la selección $A \rightarrow \alpha$ y cuándo emplear la selección $A \rightarrow \beta$, si tanto α como β comienzan con no terminales. Un problema de decisión de esta naturaleza requiere el cálculo de los conjuntos **Primero** de α y β: el conjunto de tokens que pueden iniciar legalmente cada cadena. Formalizaremos los detalles de este cálculo en la sección 4.3. En tercer lugar, al escribir el código para una producción ε

$$A \rightarrow \varepsilon$$

puede ser necesario conocer cuáles tokens pueden aparecer legalmente después del no terminal A, puesto que tales tokens indican que A puede desaparecer de manera apropiada en este punto del análisis sintáctico. Este conjunto se llama conjunto **Siguiente** de A. El cálculo de este conjunto también se precisará en la sección 4.3.

Una inquietud final que puede requerir el cálculo de los conjuntos Primero y Siguiente es la detección temprana de errores. Considere, por ejemplo, el programa calculadora de la figura 4.1. Dada la entrada `)3-2)`, el analizador sintáctico descenderá desde `exp` hasta `term` hasta `factor` antes de que se informe de un error, mientras que se podría declarar el error ya en `exp`, puesto que un paréntesis derecho no es un primer carácter legal en una expresión. El conjunto Primero de *exp* nos diría esto, lo que permitiría una detección más temprana del error. (La detección y recuperación de errores se analiza con más detalle al final del capítulo.)

4.2 ANÁLISIS SINTÁCTICO LL(1)

4.2.1 El método básico del análisis sintáctico LL(1)

El análisis sintáctico LL(1) utiliza una pila explícita en vez de llamadas recursivas para efectuar un análisis sintáctico. Es útil representar esta pila de manera estándar, de manera que las acciones de un analizador sintáctico LL(1) se puedan visualizar rápida y fácilmente. En este análisis a manera de introducción usaremos la gramática simple que genera cadenas de paréntesis balanceados:

$$S \rightarrow (\ S\ )\ S \mid \varepsilon$$

(véase el ejemplo 3.5 del capítulo anterior).

La tabla 4.1 muestra las acciones de un analizador sintáctico descendente dada esta gramática y la cadena `()`. En esta tabla hay cuatro columnas. La primera columna numera los pasos para una referencia posterior, en tanto que la segunda muestra el contenido de la pila de análisis sintáctico, con la parte inferior de ésta hacia la izquierda y la parte superior hacia la derecha. Marcamos la parte inferior de la pila con un signo monetario ($). De este modo, una pila que contenga el no terminal S en la parte superior aparece como

$$\$\ S$$

y los elementos adicionales de la pila son empujados a la derecha. La tercera columna de la tabla 4.1 muestra la entrada. Los símbolos de entrada están enumerados de izquierda a derecha. Se utiliza un signo monetario para marcar el final de la entrada [esto corresponde a un token de EOF (por sus siglas en inglés) o fin de archivo, generado por un analizador léxico]. La cuarta columna de la tabla proporciona una descripción abreviada de la acción tomada por el analizador sintáctico, lo cual cambiará la pila y (posiblemente) la entrada, como se indica en el siguiente renglón de la tabla.

Tabla 4.1
Acciones de análisis sintáctico de un analizador sintáctico descendente

	Pila de análisis sintáctico	Entrada	Acción
1	$\$\ S$	$(\ )\ \$$	$S \rightarrow (\ S\ )S$
2	$\$\ S\ )\ S\ ($	$(\ )\ \$$	match (concordar)
3	$\$\ S\ )\ S$	$)\ \$$	$S \rightarrow \varepsilon$
4	$\$\ S\ )$	$)\ \$$	match (concordar)
5	$\$\ S$	$\$$	$S \rightarrow \varepsilon$
6	$\$$	$\$$	aceptar

Un analizador sintáctico descendente comienza al insertar el símbolo inicial sobre la pila. Acepta una cadena de entrada si, después de una serie de acciones, la pila y la entrada se quedan vacías. De este modo, un esquema general para realizar con éxito un análisis sintáctico descendente es

$ *SímboloInicial* *CadenaEntrada* $
. . . . . .
. . . . . .
$ $ aceptar

En el caso de nuestro ejemplo, el símbolo inicial es S, y la cadena de entrada es ( ).

Un analizador sintáctico descendente realiza su análisis al reemplazar un no terminal en la parte superior de la pila por una de las selecciones en la regla gramatical (en BNF) para ese no terminal. Realiza esto con miras a producir el token de entrada actual en la parte superior de la pila de análisis sintáctico, de donde ha reconocido el token de entrada y puede descartarlo tanto de la pila como de la entrada. Estas dos acciones,

1. Reemplazar un no terminal A en la parte superior de la pila mediante una cadena α utilizando la selección de regla gramatical $A \rightarrow \alpha$, y
2. Hacer concordar un token en la parte superior de la pila con el siguiente token de entrada,

son las dos acciones básicas en un analizador sintáctico descendente. La primera acción, que podría ser llamada **generar**, la indicamos al escribir la selección de BNF utilizada en el reemplazo (cuyo lado izquierdo debe ser el no terminal que actualmente está en la parte superior de la pila). La segunda acción iguala o hace concordar un token en la parte superior de la pila con el siguiente token en la entrada (y los desecha a ambos sacándolos de la pila y avanzando a la entrada); indicamos esta acción escribiendo la palabra *match* (*concordar*). Es importante advertir que en la acción *generar*, la cadena de reemplazo α de la BNF se debe insertar *en reversa* en la pila (ya que eso asegura que la cadena α arribará a la parte superior de la pila en el orden de izquierda a derecha).

Por ejemplo, en el paso 1 del análisis sintáctico en la tabla 4.1, la pila y la entrada son

$ S ( ) $

y la regla que usamos para reemplazar S en la parte superior de la pila es $S \rightarrow$ (S) S, de manera que la cadena S) S (se inserta en la pila para obtener

$ S) S (( ) $

Ahora generamos la siguiente terminal de entrada, a saber, un paréntesis izquierdo, en la parte superior de la pila, y efectuamos una acción de *concordar* o *match* para obtener la situación siguiente:

$ S) S) $

La lista de acciones generadas en la tabla 4.1 corresponde precisamente a los pasos de una derivación por la izquierda de la cadena ( ):

$S \Rightarrow (S)S$ $[S \rightarrow (\ S\ )\ S]$
$\Rightarrow ()S$ $[S \rightarrow \varepsilon]$
$\Rightarrow ()$ $[S \rightarrow \varepsilon]$

Esto es característico del análisis sintáctico descendente. Si queremos construir un árbol de análisis gramatical a medida que avanza el análisis sintáctico, podemos agregar acciones de construcción de nodo a medida que cada no terminal o terminal se inserta en la pila. De este modo, el nodo raíz del árbol de análisis gramatical (correspondiente al símbolo inicial) se construye al principio del análisis sintáctico. Y también en el paso 2 de la tabla 4.1, cuando

S se reemplaza por (S) S, los nodos para cada uno de los cuatro símbolos de reemplazo se construyen a medida que se insertan los símbolos en la pila y se conectan como hijos respecto al nodo de S que reemplazan en la pila. Para hacer esto efectivo tenemos que modificar la pila de manera que contenga apuntadores para estos nodos construidos, en vez de simplemente los no terminales o terminales mismos. Veremos también cómo se puede modificar este proceso para producir una construcción del árbol sintáctico en lugar del árbol de análisis gramatical.

4.2.2 El algoritmo y tabla del análisis sintáctico LL(1)

Al utilizar el método de análisis sintáctico apenas descrito, cuando un no terminal A está en la parte superior de la pila de análisis sintáctico, debe tomarse una decisión, basada en el token de entrada actual (la búsqueda hacia adelante), que selecciona la regla gramatical que se va a utilizar para A cuando se reemplaza A en la pila. En contraste, no es necesario tomar una decisión cuando un token está en la parte superior de la pila, puesto que es el mismo que el token de entrada actual, y se presenta una concordancia, o no lo es, y se presenta un error.

Podemos expresar las selecciones posibles construyendo una **tabla de análisis sintáctico LL(1)**. Una tabla de esa naturaleza es esencialmente un arreglo bidimensional indizado por no terminales y terminales que contienen opciones de producción a emplear en el paso apropiado del análisis sintáctico (incluyendo el signo monetario $ para representar el final de la entrada). Llamamos a esta tabla $M[N, T]$. Aquí N es el conjunto de no terminales de la gramática, T es el conjunto de terminales o tokens (por conveniencia, eliminamos el hecho de que se debe agregar $ a T) y M se puede imaginar como la tabla de "movimientos". Suponemos que la tabla $M[N, T]$ inicia con todas sus entradas vacías. Cualquier entrada que permanezca vacía después de la construcción representa errores potenciales que se pueden presentar durante un análisis sintáctico.

Agregamos selecciones u opciones de producción a esta tabla de acuerdo con las reglas siguientes:

1. Si $A \rightarrow \alpha$ es una opción de producción, y existe una derivación $\alpha \Rightarrow^* a\ \beta$, donde a es un token, entonces se agrega $A \rightarrow \alpha$ a la entrada de tabla $M[A, a]$.
2. Si $A \rightarrow \alpha$ es una opción de producción, y existen derivaciones $\alpha \Rightarrow^* \varepsilon$ y $S\ \$ \Rightarrow^* \beta\ A\ a\ \gamma$, donde S es el símbolo inicial y a es un token (o $), entonces se agrega $A \rightarrow \alpha$ a la entrada de tabla $M[A, a]$.

La idea dentro de estas reglas es la siguiente. En la regla 1, dado un token a en la entrada, deseamos seleccionar una regla $A \rightarrow \alpha$ si α puede producir una a para comparar. En la regla 2, si A deriva la cadena vacía (vía $A \rightarrow \alpha$), y si a es un token que puede venir legalmente después de A en una derivación, entonces deseamos seleccionar $A \rightarrow \alpha$ para hacer que A desaparezca. Advierta que un caso especial de la regla 2 ocurre cuando $\alpha = \varepsilon$.

Estas reglas son difíciles de implementar de manera directa, pero en la siguiente sección desarrollaremos algoritmos que permitan hacerlo así, involucrando a los conjuntos Primero y Siguiente ya mencionados. Sin embargo, en casos muy simples, estas reglas se pueden realizar a mano.

Considere como primer ejemplo la gramática para paréntesis balanceados utilizada en la subsección anterior, hay un no terminal (S), tres tokens (paréntesis izquierdo, paréntesis derecho y $) y dos opciones de producción. Puesto que hay sólo una producción no vacía para S, a saber, $S \rightarrow$ (S) S, cada cadena derivable de S debe ser vacía, o bien, comenzar con un paréntesis izquierdo, y esta opción de producción se agrega a la entrada $M[S, (]$ (y sólo allí). Esto completa todos los casos bajo la regla 1. Ya que $S \Rightarrow$ (S) S, la regla 2 se aplica con $\alpha = \varepsilon$, $\beta =$ (, $A = S$, $a =$) y $\gamma = S\ \$$, de manera que $S \rightarrow \varepsilon$ se agrega a $M[S,)]$. Puesto que $S\ \$ \Rightarrow^* S\ \$$ (la derivación vacía), $S \rightarrow \varepsilon$ también se agrega a $M[S, \$]$. Esto completa la

construcción de la tabla de análisis sintáctico LL(1), la que podemos escribir de la manera siguiente:

$M[N, T]$	(	)	$
S	$S \rightarrow (\ S\ )\ S$	$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$

Para completar el algoritmo de análisis sintáctico LL(1), esta tabla debe proporcionar opciones únicas para cada par de no terminal y token. De este modo, establecemos la siguiente

Definición

Una gramática es una **gramática LL(1)** si la tabla de análisis sintáctico LL(1) asociada tiene como máximo una producción en cada entrada de la tabla.

Una gramática LL(1) no puede ser ambigua, porque la definición implica que un analizador sintáctico no ambiguo se puede construir empleando la tabla de análisis sintáctico LL(1). En realidad, dada una gramática LL(1), en la figura 4.2 se proporciona un algoritmo de análisis sintáctico que utiliza la tabla de análisis sintáctico LL(1). Este algoritmo produce precisamente las acciones descritas en el ejemplo de la subsección anterior.

Mientras que el algoritmo de la figura 4.2 requiere que las entradas de la tabla de análisis sintáctico tengan como máximo una producción cada una, se pueden construir reglas de no ambigüedad en la construcción de la tabla para casos ambiguos simples tales como el problema del else ambiguo, de una manera similar a la descendente recursiva.

Figura 4.2
Algoritmo de análisis sintáctico LL(1) basado en tabla

```
(* supone que $ marca la parte inferior de la pila y el final de la entrada *)
inserta el símbolo inicial en la parte superior o tope de la pila de análisis sintáctico ;
while el tope de la pila de análisis sintáctico ≠ $ and el siguiente token de entrada
≠ $ do
  if el tope de la pila de análisis sintáctico es el terminal a
        and el siguiente token de entrada = a
  then (* concuerda *)
     extraer de la pila de análisis sintáctico ;
     avanzar la entrada ;
  else if el tope de análisis sintáctico es el no terminal A
        and el siguiente token de entrada es el terminal a
        and la entrada de la tabla de análisis sintáctico M[A, a] contiene
              la producción A → X1X2 . . . Xn
  then (* generar *)
     extraer de la pila de análisis sintáctico ;
     for i := n downto 1 do
        insertar Xi en la pila de análisis sintáctico ;
  else error ;
if el tope de la pila de análisis sintáctico = $
        and el siguiente token de entrada = $
then aceptar
else error ;
```

Considere, por ejemplo, la gramática simplificada de las sentencias if (véase el ejemplo 3.6 del capítulo 3):

sentencia → *sent-if* | `otro`
sent-if → `if` (*exp*) *sentencia parte-else*
parte-else → `else` *sentencia* | ε
exp → `0` | `1`

La construcción de la tabla de análisis sintáctico LL(1) da el resultado que se muestra en la tabla 4.2, donde no enumeramos los terminales de paréntesis (o) en la tabla, ya que no producen ninguna acción. (La construcción de esta tabla se explicará en forma detallada en la siguiente sección.)

Tabla 4.2
Tabla de análisis sintáctico LL(1) para sentencias if (ambiguas)

M[*N*, *T*]	`if`	`otro`	`else`	0	1	$
sentencia	*sentencia* → *sent-if*	*sentencia* → `otro`				
sent-if	*sent-if* → `if` (*exp*) *sentencia parte-else*					
parte-else			*parte-else* → `else` *sentencia* *parte else* → ε			*parte-else* → ε
exp				*exp* → 0	*exp* → 1	

En la tabla 4.2, la entrada *M*[*parte-else*, `else`] contiene dos entradas, correspondientes a la ambigüedad del else. Como en el caso descendente recursivo, al construir esta tabla podríamos aplicar una regla de eliminación de ambigüedad que siempre preferiría la regla que genera el token de búsqueda hacia delante actual sobre cualquier otra, y, por consiguiente, se preferiría la producción

parte-else → `else` *sentencia*

en vez de la producción *parte-else* → ε. Esto de hecho corresponde a la regla de eliminación de ambigüedad más cercanamente anidada. Con esta modificación la tabla 4.2 se hace no ambigua, y la gramática se puede analizar sintácticamente como si fuera una gramática LL(1). Por ejemplo, la tabla 4.3 muestra las acciones de análisis sintáctico del algoritmo de análisis sintáctico LL(1), dada la cadena

```
if(0) if(1) otro else otro
```

(Para ser más concisos utilizamos las siguientes abreviaturas en esa figura: *sentencia* = *S*, *sent-if* = *I*, *parte-else* = *L*, *exp* = *E*, `if` = `i`, `else` = `e`, `otro` = `o`.)

Tabla 4.3
Acciones de análisis sintáctico LL(1) para sentencias if utilizando la regla de eliminación anidada de ambigüedad más próxima

Pila de análisis sintáctico	Entrada	Acción
$ S	i(0)i(1)o e o$	$S \rightarrow I$
$ I	i(0)i(1)o e o$	$I \rightarrow$ i(E) S L
$ L S) E (i	i(0)i(1)o e o$	concuerda (match)
$ L S) E (	(0)i(1)o e o$	concuerda (match)
$ L S) E	0)i(1)o e o$	$E \rightarrow$ 0
$ L S) 0	0)i(1)o e o$	concuerda (match)
$ L S)	)i(1)o e o$	concuerda (match)
$ L S	i(1)o e o$	$S \rightarrow I$
$ L I	i(1)o e o$	$I \rightarrow$ i(E) S L
$ L L S) E (i	i(1)o e o$	$I \rightarrow$ i(E) S L
$ L L S) E (i	i(1)o e o$	concuerda (match)
$ L L S) E (	(1)o e o$	concuerda (match)
$ L L S) E	1)o e o$	$E \rightarrow$ 1
$ L L S)	)o e o$	concuerda (match)
$ L L S	o e o$	$S \rightarrow$ o
$ L L o	o e o$	concuerda (match)
$ L L	e o$	$L \rightarrow$ e S
$ L S e	e o$	concuerda (match)
$ L S	o$	$S \rightarrow$ o
$ L o	o$	concuerda (match)
$ L	$	$L \rightarrow \varepsilon$
$	$	aceptar (accept)

4.2.3 Eliminación de recursión por la izquierda y factorización por la izquierda

En la repetición y selección en el análisis sintáctico LL(1) se presentan problemas similares a los que surgen en el análisis sintáctico descendente recursivo, por esa razón aún no hemos podido dar una tabla de análisis sintáctico LL(1) para la gramática de expresión aritmética simple de secciones anteriores. Resolvimos estos problemas para descendente recursivo utilizando notación EBNF. No podemos aplicar las mismas ideas al análisis sintáctico LL(1); en vez de eso debemos volver a escribir la gramática dentro de la notación BNF en una forma que el algoritmo de análisis sintáctico LL(1) pueda aceptar. Las dos técnicas estándar que aplicamos son la **eliminación de recursión por la izquierda** y la **factorización por la izquierda**. Consideraremos cada una de estas técnicas. Debe hacerse énfasis en que nada garantiza que la aplicación de estas técnicas convertirá una gramática en una gramática LL(1), del mismo modo que EBNF no garantizaba resolver todos los problemas al escribir un analizador sintáctico descendente recursivo. No obstante, son muy útiles en la mayoría de las situaciones prácticas y tienen la ventaja de que sus aplicaciones se pueden automatizar, de manera que, suponiendo un resultado con éxito, se pueden utilizar para generar automáticamente un analizador sintáctico LL(1) (véase la sección de notas y referencias).

Eliminación de recursión por la izquierda La recursión por la izquierda se utiliza por lo regular para hacer operaciones asociativas por la izquierda, como en la gramática de expresión simple, donde

$$exp \rightarrow exp\ opsuma\ term \mid term$$

hace las operaciones representadas por *opsuma* asociativas por la izquierda. Éste es el caso más simple de recursión por la izquierda, donde hay sólo una opción de producción recursiva por la izquierda simple. Otro caso ligeramente más complejo es aquel en que más de una opción es recursiva por la izquierda, lo que pasa si escribimos *opsuma*:

$$exp \rightarrow exp + term \mid exp - term \mid term$$

Estos dos casos involucran la **recursión por la izquierda inmediata**, donde la recursión se presenta sólo dentro de la producción de un no terminal simple (como *exp*). Un caso más difícil es la recursión por la izquierda *indirecta*, tal como en las reglas

$$A \rightarrow B\ b \mid \ldots$$
$$B \rightarrow A\ a \mid \ldots$$

Tales reglas casi nunca se presentan en gramáticas de lenguajes de programación reales, pero incluiremos una solución para este caso con el fin de completar la exposición. Consideraremos primero la recursión por la izquierda inmediata.

CASO 1: Recursión por la izquierda inmediata simple

En este caso la recursión por la izquierda se presenta sólo en reglas gramaticales de la forma

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

donde α y β son cadenas de terminales y no terminales y β no comienza con A. En la sección 3.2.3 vimos que estas reglas gramaticales generan cadenas de la forma $\beta\alpha^n$, para $n \geq 0$. La selección $A \rightarrow \beta$ es el caso base, mientras que $A \rightarrow A\ \alpha$ es el caso recursivo.

Para eliminar la recursión por la izquierda volvemos a escribir estas reglas gramaticales divididas en dos: una que primero genera β y otra que genera las repeticiones de α utilizando recursión por la derecha en vez de recursión por la izquierda:

$$A \rightarrow \beta\ A'$$
$$A' \rightarrow \alpha\ A' \mid \varepsilon$$

Ejemplo 4.1

Considere de nueva cuenta la regla recursiva izquierda de la gramática de expresión simple:

$$exp \rightarrow exp\ opsuma\ term \mid term$$

La forma de esta regla es $A \rightarrow A\ \alpha \mid \beta$, con $A = exp$, $\alpha = opsuma\ term$ y $\beta = term$. Al volverla a escribir para eliminar la recursión por la izquierda obtenemos que

$$exp \rightarrow term\ exp'$$
$$exp' \rightarrow opsuma\ term\ exp' \mid \varepsilon$$

§

CASO 2: Recursión por la izquierda inmediata general

Éste es el caso en que tenemos producciones de la forma

$$A \rightarrow A\ \alpha_1 \mid A\ \alpha_2 \mid \ldots \mid A\ \alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

donde ninguna de las $\beta_1, \ldots, \beta_m$ comienzan con A. En este caso la solución es semejante a la del caso simple, con las opciones extendidas en consecuencia:

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$$

Ejemplo 4.2

Considere la regla gramatical

$$exp \rightarrow exp + term \mid exp - term \mid term$$

Eliminemos la recursión por la izquierda de la manera siguiente:

$$exp \rightarrow term\ exp'$$
$$exp' \rightarrow + term\ exp' \mid - term\ exp' \mid \varepsilon$$

§

CASO 3: Recursión por la izquierda general

El algoritmo que aquí describimos está garantizado para funcionar sólo en el caso de gramáticas sin producciones ε y sin ciclos, donde un **ciclo** es una derivación de por lo menos un paso que comienza y finaliza con el mismo no terminal: $A \Rightarrow \alpha \Rightarrow^* A$. Es casi seguro que un ciclo ocasionará que un analizador sintáctico entre en un ciclo infinito, y las gramáticas con ciclos nunca aparecen como gramáticas de lenguajes de programación. Estas gramáticas tienen producciones ε, pero por lo regular en formas muy restringidas, de modo que este algoritmo casi siempre funcionará también para estas gramáticas.

El algoritmo funciona al elegir un orden arbitrario para todas los no terminales del lenguaje, digamos, $A_1, \ldots, A_m$, y posteriormente eliminar todas las recursiones izquierdas que no incrementen el índice de las A_i. Esto elimina todas las reglas de la forma $A_i \rightarrow A_j\ \gamma$ con $j \leq i$. Si hacemos esto para cada i desde 1 hasta m, entonces ningún ciclo recursivo puede ser izquierdo, puesto que cada paso en un ciclo así sólo incrementaría el índice, y de este modo no se podría volver a alcanzar el índice original. El algoritmo se proporciona detalladamente en la figura 4.3.

Figura 4.3
Algoritmo para eliminación de recursión por la izquierda general

for i := 1 **to** m **do**
  for j := 1 **to** $i-1$ **do**
    reemplazar cada selección de regla gramatical de la forma $A_i \rightarrow A_j\ \beta$ *por la regla* $A_i \rightarrow \alpha_1\ \beta \mid \alpha_2\ \beta \mid \ldots \mid \alpha_k\ \beta$, *donde* $A_j \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$ *es la regla actual para* A_j

Ejemplo 4.3

Considere la gramática siguiente:

$$A \rightarrow B\ a \mid A\ a \mid c$$
$$B \rightarrow B\ b \mid A\ b \mid d$$

(Esta gramática es completamente artificial, ya que esta situación no se presenta en ningún lenguaje de programación estándar.)

Consideraremos que B tiene un número más alto que A para propósitos del algoritmo (es decir, $A_1 = A$ y $A_2 = B$). Como $n = 2$, el ciclo externo del algoritmo de la figura 4.3 se ejecuta dos veces, una vez para $i = 1$ y una vez para $i = 2$. Cuando $i = 1$, el ciclo interno (con índice j) no se ejecuta, de modo que la única acción es eliminar la recursión por la izquierda inmediata de A. La gramática resultante es

$$\begin{aligned} A &\rightarrow B\,a\,A' \mid c\,A' \\ A' &\rightarrow a\,A' \mid \varepsilon \\ B &\rightarrow B\,b \mid A\,b \mid d \end{aligned}$$

Ahora el ciclo externo se ejecuta para $i = 2$, y el ciclo interno se ejecuta una vez, con $j = 1$. En este caso eliminamos la regla $B \rightarrow A\,b$ reemplazando A con sus opciones de la primera regla. De este modo obtenemos la gramática

$$\begin{aligned} A &\rightarrow B\,a\,A' \mid c\,A' \\ A' &\rightarrow a\,A' \mid \varepsilon \\ B &\rightarrow B\,b \mid B\,a\,A'\,b \mid c\,A'\,b \mid d \end{aligned}$$

Finalmente, eliminamos la recursión por la izquierda inmediata de B para obtener

$$\begin{aligned} A &\rightarrow B\,a\,A' \mid c\,A' \\ A' &\rightarrow a\,A' \mid \varepsilon \\ B &\rightarrow c\,A'\,b\,B' \mid d\,B' \\ B' &\rightarrow b\,B' \mid a\,A'\,b\,B' \mid \varepsilon \end{aligned}$$

Esta gramática no tiene recursión por la izquierda. §

La eliminación de la recursión por la izquierda no cambia el lenguaje que se está reconociendo, pero modifica la gramática y, en consecuencia, también los árboles de análisis gramatical. En realidad, este cambio ocasiona una complicación para el analizador sintáctico (y para el diseñador del analizador). Considere, por ejemplo, la gramática de expresión simple que hemos estado utilizando como ejemplo estándar. Como vimos, la gramática es recursiva por la izquierda para expresar la asociatividad por la izquierda de las operaciones. Si eliminamos la recursión por la izquierda inmediata como en el ejemplo 4.1, obtenemos la gramática dada en la figura 4.4.

Figura 4.4
Gramática de expresión aritmética simple con recursión por la izquierda eliminada

$$\begin{aligned} &exp \rightarrow term\ exp' \\ &exp' \rightarrow opsuma\ term\ exp' \mid \varepsilon \\ &opsuma \rightarrow \texttt{+} \mid \texttt{-} \\ &term \rightarrow factor\ term' \\ &term' \rightarrow opmult\ factor\ term' \mid \varepsilon \\ &opmult \rightarrow \texttt{*} \\ &factor \rightarrow \texttt{(}\ exp\ \texttt{)} \mid \texttt{número} \end{aligned}$$

Ahora considere el árbol de análisis gramatical para la expresión `3+4+5`:

Este árbol ya no expresa la asociatividad izquierda de la resta. No obstante, un analizador sintáctico todavía construiría el árbol sintáctico apropiado asociativo por la izquierda:

No es completamente trivial hacer esto utilizando la nueva gramática. Para ver cómo, considere en su lugar la cuestión algo más simple de calcular el valor de la expresión utilizando el árbol de análisis gramatical dado. Para hacer esto, se debe pasar el valor 3 desde el nodo raíz *exp* hasta su hijo a la derecha *exp'*. Este nodo *exp'* debe entonces restar 4 y pasar el nuevo valor -1 hacia *su* hijo de extrema derecha (otro *exp'*). Este nodo *exp'* a su vez debe restar 5 y pasar el valor -6 hacia el nodo *exp'* final. Este nodo tiene únicamente un hijo ε y simplemente pasa el valor -6 de regreso. Este valor se devuelve entonces hacia la parte superior del árbol, al nodo raíz *exp* y es el valor final de la expresión.

Considere cómo podría funcionar esto en un analizador sintáctico descendente recursivo. La gramática con esta recursión por la izquierda eliminada daría origen a los procedimientos *exp* y *exp'* de la manera siguiente:

```
procedure exp ;
begin
   term ;
   exp' ;
end exp ;
```

```
procedure exp' ;
begin
  case token of
   + : match (+) ;
       term ;
       exp' ;
   - : match (-) ;
       term ;
       exp' ;
  end case ;
end exp' ;
```

Para obtener esos procedimientos que realmente permiten calcular el valor de la expresión volveríamos a escribirlos de la manera siguiente:

```
function exp : integer ;
var temp : integer ;
begin
  temp := term ;
  return exp'(temp) ;
end exp ;


function exp' ( valsofar : integer ) : integer ;
begin
  if token = + or token = - then
     case token of
      + : match (+) ;
          valsofar := valsofar + term ;
      - : match (-) ;
          valsofar := valsofar - term ;
     end case ;
     return exp'(valsofar) ;
  else return valsofar ;
end exp' ;
```

Advierta cómo el procedimiento *exp'* ahora necesita un parámetro pasado desde el procedimiento *exp*. Una situación semejante se presenta si estos procedimientos están por devolver un árbol sintáctico (asociativo por la izquierda). El código que dimos en la sección 4.1 empleaba una solución más simple basada en EBNF, los cuales no requieren el parámetro extra.

Finalmente, notamos que la nueva gramática de expresión de la figura 4.4 es en realidad una gramática LL(1). La tabla de análisis sintáctico LL(1) se proporciona en la tabla 4.4. Como con las tablas anteriores, regresaremos a su construcción en la sección siguiente.

Factorización por la izquierda La factorización por la izquierda se requiere cuando dos o más opciones de reglas gramaticales comparten una cadena de prefijo común, como en la regla

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

Tabla 4.4
Tabla de análisis sintáctico LL(1) para la gramática de la figura 4.4

M[N, T]	(	número	)	+	-	*	$
exp	*exp* → *term exp'*	*exp* → *term exp'*					
exp'			*exp'* → ε	*exp'* → *opsuma term exp'*	*exp'* → *opsuma term exp'*		*exp'* → ε
opsuma				*opsuma* → +	*opsuma* → -		
term	*term* → *factor term'*	*term* → *factor term'*					
term'			*term'* → ε	*term'* ε	*term'* → ε	*term'* → *opmult factor term'*	*term'* → ε
opmult						*opmult* → *	
factor	*factor* → (*exp*)	*factor* → **número**					

Ejemplos de esto son las reglas recursivas por la derecha para secuencias de sentencias (ejemplo 3.7 del capítulo 3):

secuencia-sent → *sent* **;** *secuencia-sent* | *sent*
sent → **s**

y la siguiente versión de la sentencia if:

sent-if → **if** (*exp*) *sentencia*
| **if** (*exp*) *sentencia* **else** *sentencia*

Obviamente, un analizador sintáctico LL(1) no puede distinguir entre las opciones de producción en una situación de esta clase. La solución en este caso simple es "factorizar" la α por la izquierda y volver a escribir la regla como dos reglas

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta \mid \gamma$$

(Si queremos utilizar paréntesis como metasímbolos en reglas gramaticales, también podríamos escribir $A \rightarrow \alpha\ (\beta \mid \gamma)$, lo que se parece mucho a la factorización en aritmética.) Para que la factorización por la izquierda trabaje adecuadamente, debemos estar seguros de que α es de hecho la cadena más larga compartida por los lados derechos. Esto también es posible cuando hay más de dos opciones que compartan un prefijo. Establecemos el algoritmo general en la figura 4.5 y entonces lo aplicamos a varios ejemplos. Advierta que a medida que el algoritmo continúa, el número de opciones de producción para cada no terminal que

puede compartir un prefijo se reduce en al menos uno en cada paso, de modo que el algoritmo garantiza llegar al final.

Figura 4.5
Algoritmo para factorización por la izquierda de una gramática

while *existan cambios en el lenguaje* **do**
  for *cada no terminal A* **do**
    sea α *un prefijo de longitud máxima que se comparte*
      por dos o más opciones de producción para A
  if $\alpha \neq \varepsilon$ **then**
    sean $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ *todas las seleciones de producción para A*
      y supongamos que $\alpha_1, \ldots, \alpha_k$ *comparten* α, *de manera que*
      $A \rightarrow \alpha\,\beta_1 \mid \ldots \mid \alpha\,\beta_k \mid \alpha_{k+1} \mid \ldots \mid \alpha_n$, *las* β_j *no comparten*
      prefijos comunes y las $\alpha_{k+1}, \ldots, \alpha_n$ *no comparten a* α
    reemplace la regla $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ *por las reglas*
      $A \rightarrow \alpha\,A' \mid \alpha_{k+1} \mid \ldots \mid \alpha_n$
      $A' \rightarrow \beta_1 \mid \ldots \mid \beta_k$

Ejemplo 4.4

Considere la gramática para las secuencias de sentencias escrita en forma recursiva por la derecha:

secuencia-sent → *sent* **;** *secuencia-sent* | *sent*
sent → **s**

La regla gramatical para *secuencia-sent* tiene un prefijo compartido que se puede factorizar por la izquierda de la manera que se muestra a continuación:

secuencia-sent → *sent sec-sent'*
sec-sent' → **;** *secuencia-sent* | ε

Advierta que si habíamos escrito la regla *secuencia-sent* recursivamente por la izquierda en lugar de recursivamente por la derecha,

secuencia-sent → *secuencia-sent* **;** *sent* | *sent*

entonces la eliminación de la recursión por la izquierda inmediata daría como resultado las reglas

secuencia-sent → *sent sec-sent'*
sec-sent' → **;** *sent sec-sent'* | ε

Esto es casi igual al resultado que obtuvimos de la factorización por la izquierda, y, efectivamente, al sustituir la *secuencia-sent* por *sent sec-sent'* en la última regla se hace que los dos resultados sean idénticos. §

Ejemplo 4.5

Considere la siguiente gramática (parcial) para sentencias if:

sent-if → `if` (*exp*) *sentencia*
  | `if` (*exp*) *sentencia* `else` *sentencia*

La forma factorizada por la izquierda de esta gramática es

$$\textit{sent-if} \rightarrow \texttt{if}\ (\ \textit{exp}\ )\ \textit{sentencia parte-else}$$
$$\textit{parte-else} \rightarrow \texttt{else}\ \textit{sentencia} \mid \varepsilon$$

Ésta es precisamente la forma en la que la utilizamos en la sección 4.2.2 (véase la tabla 4.2). §

Ejemplo 4.6

Suponga que escribimos una gramática de expresión aritmética en la que damos una asociatividad por la derecha de operación aritmética en lugar de una asociatividad por la izquierda (aquí usamos + precisamente para tener un ejemplo concreto):

$$\textit{exp} \rightarrow \textit{term} + \textit{exp} \mid \textit{term}$$

Esta gramática necesita ser factorizada por la izquierda, y obtenemos las reglas

$$\textit{exp} \rightarrow \textit{term exp}'$$
$$\textit{exp}' \rightarrow + \textit{exp} \mid \varepsilon$$

Ahora, continuando como en el ejemplo 4.4, suponga que sustituimos *term exp'* en lugar de *exp* en la segunda regla (esto es legal, porque, de cualquier manera, esta expansión tendría lugar en el siguiente paso de una derivación). Entonces obtenemos

$$\textit{exp} \rightarrow \textit{term exp}'$$
$$\textit{exp}' \rightarrow + \textit{term exp}' \mid \varepsilon$$

Esto es idéntico a la gramática obtenida de la regla recursiva por la izquierda mediante la eliminación de la recursión por la izquierda. Por consiguiente, tanto la factorización por la izquierda como la eliminación de recursión por la izquierda pueden oscurecer la semántica de la estructura del lenguaje (en este caso, ambas oscurecen la asociatividad).

Si, por ejemplo, deseamos conservar la asociatividad derecha de la operación de las reglas gramaticales anteriores (en cualquier forma), entonces debemos arreglar que cada operación + se aplique al final en vez de al principio. Dejamos al lector escribir esto para procedimientos descendentes recursivos. §

Ejemplo 4.7

Aquí tenemos un caso típico donde una gramática de lenguaje de programación deja de ser LL(1), puesto que tanto las asignaciones como las llamadas de procedimientos comienzan con un identificador. Escribimos la representación siguiente de este problema:

$$\textit{sentencia} \rightarrow \textit{sent-asignación} \mid \textit{sent-llamada} \mid \texttt{otro}$$
$$\textit{sent-asignación} \rightarrow \texttt{identificador} := \textit{exp}$$
$$\textit{sent-llamada} \rightarrow \texttt{identificador}\ (\ \textit{exp-list}\ )$$

Esta gramática no es LL(1) porque el `identificador` es compartido como el primer token tanto de *sent-asignación* como de *sent-llamada*, de modo que podría ser el token de búsqueda hacia delante para cualquiera de ellas. Desgraciadamente, la gramática no está en una forma que se pueda factorizar por la izquierda. Lo que debemos hacer es reemplazar primero *sent-asignación* y *sent-llamada* por los lados derechos de sus producciones definidas de la manera que se presenta a continuación:

$$
\begin{aligned}
\textit{sentencia} \rightarrow\ & \texttt{identificador} := \textit{exp} \\
\mid\ & \texttt{identificador}\ (\ \textit{exp-list}\ ) \\
\mid\ & \texttt{otro}
\end{aligned}
$$

Entonces factorizamos por la izquierda para obtener

$$
\begin{aligned}
\textit{sentencia} \rightarrow\ & \texttt{identificador}\ \textit{sentencia}' \\
\mid\ & \texttt{otro} \\
\textit{sentencia}' \rightarrow\ & := \textit{exp} \mid (\ \textit{exp-list}\ )
\end{aligned}
$$

Advierta cómo oscurece esto la semántica de la llamada y la asignación al separar el identificador (la variable que se asignará o el procedimiento que se llamará) de la acción de asignación o llamada real (representadas por *sentencia'*). Un analizador sintáctico LL(1) debe arreglar esto haciendo que el identificador esté disponible para la llamada o para la asignación de alguna manera (por ejemplo, un parámetro) o bien, ajustando el árbol sintáctico. §

Finalmente, advertimos que todos los ejemplos a los que hemos aplicado factorización por la izquierda en realidad se convierten en gramáticas LL(1) después de la transformación. Construiremos las tablas de análisis sintáctico LL(1) para algunos de ellos en la siguiente sección. Las demás se dejan para los ejercicios.

4.2.4 Construcción del árbol sintáctico en análisis sintáctico LL(1)

Resta comentar cómo se puede adaptar el análisis sintáctico LL(1) para construir árboles sintácticos en vez de árboles de análisis gramatical. (En la sección 4.2.1 describimos cómo se pueden construir los árboles de análisis gramatical utilizando la pila de análisis sintáctico.) Vimos en la sección en la que se trató el análisis sintáctico descendente recursivo que la adaptación de ese método a la construcción del árbol sintáctico era relativamente fácil. En contraste, los analizadores sintácticos LL(1) son más difíciles de adaptar. Esto en parte se debe a que, como ya hemos visto, la estructura del árbol sintáctico (tal como la asociatividad por la izquierda) puede ser oscurecida mediante la factorización por la izquierda y la eliminación de la recursión por la izquierda. Sin embargo, se debe principalmente al hecho de que la pila de análisis sintáctico representa sólo estructuras pronosticadas, no estructuras que se hayan visto en realidad. Por consiguiente, se debe posponer la construcción de nodos de árbol sintáctico hasta el punto en que se eliminen las estructuras de la pila de análisis sintáctico, en vez de hacerlo cuando se inserten por primera vez. En general, esto requiere que se utilice una pila extra para seguir la pista de los nodos del árbol sintáctico y que se coloquen marcadores de "acción" en la pila de análisis sintáctico que indiquen cuándo y qué acciones deberían presentarse en la pila del árbol. Los analizadores sintácticos ascendentes (el capítulo siguiente) son más fáciles de adaptar a la construcción de árboles sintácticos utilizando una pila de análisis sintáctico y, por lo tanto, son los que se prefiere usar como métodos de análisis sintáctico basados en pilas y controlados por tabla. Así, proporcionamos sólo un breve ejemplo de los detalles de cómo se puede hacer esto para el análisis sintáctico LL(1).

Ejemplo 4.8

Utilizaremos una gramática de expresiones muy sencillas con sólo la operación de suma o adición. La BNF es

$$E \rightarrow E + \mathtt{n} \mid \mathtt{n}$$

Esto provoca que la suma se aplique asociativamente por la izquierda. La gramática LL(1) correspondiente con recursión por la izquierda eliminada es

$$E \rightarrow \mathbf{n}\, E'$$
$$E' \rightarrow \mathbf{+}\, \mathbf{n}\, E' \mid \varepsilon$$

Mostramos ahora cómo se puede utilizar esta gramática para calcular el valor aritmético de la expresión. La construcción de un árbol sintáctico es semejante.

Para calcular un valor del resultado de una expresión, utilizaremos una pila por separado en la que se pueda almacenar los valores intermedios del cálculo, a la cual llamaremos **pila de valores**. Debemos clasificar dos operaciones en esa pila. La primera operación consiste en insertar un número cuando es igualado en la entrada. La segunda es sumar dos números en la pila. La primera se puede realizar mediante el procedimiento *match* (basada en el token que está en concordancia). La segunda necesita ser organizada en la pila de análisis sintáctico. Efectuaremos esto insertando un símbolo especial en la pila de análisis sintáctico, el cual, cuando se extraiga, indicará que se va a realizar una suma. El símbolo que utilizaremos para este propósito es el signo de número (#). Este símbolo ahora se convierte en un nuevo símbolo de pila y también se debe agregar a la regla gramatical que iguala a un +, a saber, la regla para E':

$$E' \rightarrow \mathbf{+}\, \mathbf{n}\, \#\, E' \mid \varepsilon$$

Advierta que la suma es programada precisamente *después* del número siguiente, pero antes de que se procese cualquier otro no terminal E'. Esto garantiza la asociatividad por la izquierda. Ahora veremos cómo tiene lugar el cálculo del valor para la expresión **3+4+5**. Indicamos la pila de análisis sintáctico, entrada y acción como antes, pero además presentamos la pila de valores a la derecha (que crece hacia la izquierda). Las acciones del analizador sintáctico se proporcionan en la tabla 4.5.

Tabla 4.5

Pila de análisis sintáctico con acciones de la pila de valores para el ejemplo 4.8

Pila de análisis sintáctico	Entrada	Acción	Pila de valores
$ E	3 + 4 + 5 $	$E \rightarrow \mathbf{n}\, E'$	$
$ E' **n**	3 + 4 + 5 $	concordar/insertar	$
$ E'	+ 4 + 5 $	$E' \rightarrow \mathbf{+}\, \mathbf{n}\, \#\, E'$	3 $
$ E' # **n** +	+ 4 + 5 $	concordar	3 $
$ E' # **n**	4 + 5 $	concordar/insertar	3 $
$ E' #	+ 5 $	agregar a pila	4 3 $
$ E'	+ 5 $	$E' \rightarrow \mathbf{+}\, \mathbf{n}\, \#\, E'$	7 $
$ E' # **n** +	+ 5 $	concordar	7 $
$ E' # **n**	5 $	concordar/insertar	7 $
$ E' #	$	agregar a pila	5 7 $
$ E'	$	$E' \rightarrow \varepsilon$	12 $
$	$	aceptar	12 $

Observe que cuando tiene lugar una suma, los operandos están en orden inverso en la pila de valores. Esto es típico de tales esquemas de evaluación basados en pila. §

4.3 CONJUNTOS PRIMERO Y SIGUIENTE

Para completar el algoritmo de análisis sintáctico LL(1) desarrollamos un algoritmo que permite construir la tabla de análisis sintáctico LL(1). Como ya lo indicamos en varios puntos, esto involucra el cálculo de los conjuntos Primero y Siguiente, y más adelante en esta sección volveremos a la definición y construcción de estos conjuntos, después de lo cual proporcionaremos una descripción precisa de la construcción de una tabla de análisis sintáctico LL(1). Al final de la sección consideraremos de manera breve cómo se puede extender la construcción a más de un símbolo de búsqueda hacia delante.

4.3.1 Conjuntos Primero

Definición

Si X es un símbolo de la gramática (un terminal o no terminal) o ε, entonces el conjunto **Primero(X)**, compuesto de terminales, y posiblemente de ε, se define de la manera siguiente:

1. Si X es un terminal o ε, entonces Primero(X) = $\{X\}$.
2. Si X es no terminal, entonces para cada selección de producción $X \rightarrow X_1 X_2 \ldots X_n$, Primero($X$) contiene Primero($X_1$) $- \{\varepsilon\}$. Si también para cualquier $i < n$, todos los conjuntos Primero(X_1), . . . , Primero(X_i) contienen ε, entonces Primero(X) contiene Primero(X_{i+1}) $- \{\varepsilon\}$. Si todos los conjuntos Primero(X_1), . . . , Primero(X_n) contienen ε, entonces Primero(X) también contiene ε.

Ahora definamos **Primero(α)**, para cualquier cadena $\alpha = X_1X_2 \ldots X_n$ (una cadena de terminales y no terminales), de la manera siguiente. Primero(α) contiene Primero(X_1) $- \{\varepsilon\}$. Para cada $i = 2, \ldots, n$, si Primero(X_k) contiene ε para toda $k = 1, \ldots, i - 1$, entonces Primero(α) contiene Primero(X_i) $- \{\varepsilon\}$. Finalmente, si para toda $i = 1, \ldots, n$, Primero(X_i) contiene ε, entonces Primero(α) contiene ε.

Esta definición se puede convertir fácilmente en un algoritmo. En realidad, el único caso difícil se presenta cuando se calcula Primero(A) para todo no terminal A, ya que el conjunto Primero de un terminal no es importante, y el conjunto primero de una cadena α se construye a partir de los conjuntos primero de los símbolos individuales en n pasos como máximo, donde n es el número de símbolos en α. Por consiguiente, establecemos el pseudocódigo para el algoritmo sólo en el caso de no terminales, el cual se presenta en la figura 4.6.

Figura 4.6
Algoritmo para calcular Primero(A) para todos los no terminales A

```
for todo no terminal A do Primero(A) := { };
while existan cambios a cualquier Primero(A) do
  for cada selección de producción A → X1X2 . . . Xn do
    k := 1 ; Continuar := verdadero ;
    while Continuar = verdadero and k <= n do
      agregar Primero(Xk) − {ε} a primero(A);
      if ε no esta en Primero(Xk) then Continuar := falso ;
      k := k + 1 ;
    if Continuar = verdadero then agregar ε a Primero(A) ;
```

También es fácil ver cómo se puede interpretar esta definición cuando no hay producciones ε: sólo se necesita mantener sumando Primero(X_1) a Primero(A) para cada no terminal A y opción de producción $A \rightarrow X_1 \ldots$ hasta que no tengan lugar sumas adicionales. En otras palabras, considere sólo el caso $k = 1$ de la figura 4.6, y el ciclo while interno no es necesario. Establecemos este algoritmo de manera separada en la figura 4.7. Si hay producciones ε, la situación se vuelve más complicada, ya que también debemos preguntarnos si ε está en Primero(X_1), y si es así, continuar realizando el mismo proceso para X_2, y así sucesivamente. Aun así, el proceso se terminará después de un número finito de pasos. En realidad, este proceso no sólo calcula los terminales que pueden aparecer como los primeros símbolos en una cadena derivada de un no terminal, sino que también determina si un no terminal puede derivar la cadena vacía (es decir, desaparecer). Tales no terminales también se denominan anulables:

Figura 4.7
Algoritmo simplificado de la figura 4.6 en ausencia de producciones ε

for *todos los no terminales A* **do** *Primero(A)* := {};
while *existan cambios para cualquier Primero(A)* **do**
  for *cada selección de producción* $A \rightarrow X_1X_2 \ldots X_n$ **do**
    agregue Primero(X_1) *a Primero*(A);

Definición

Un no terminal A es **anulable** si existe una derivación $A \Rightarrow^* \varepsilon$.

Ahora mostraremos el teorema siguiente.

Teorema

Un no terminal A es anulable si y sólo si Primero(A) contiene a ε.

Demostración: Mostramos que si A es anulable, entonces Primero(A) contiene a ε. La inversa se puede demostrar de manera similar. Emplearemos la inducción sobre la longitud de una derivación. Si $A \Rightarrow \varepsilon$, entonces debe haber una producción $A \rightarrow \varepsilon$ y, por definición, Primero(A) contiene Primero(ε) = $\{\varepsilon\}$. Suponga ahora la certeza de la sentencia para derivaciones de longitud $< n$, y sea $A \Rightarrow X_1 \ldots X_k \Rightarrow^* \varepsilon$ una derivación de longitud n (utilizando una selección de producción $A \rightarrow X_1 \ldots X_k$). Si cualquiera de las X_i es terminal, no puede derivar a ε, así que todas las X_i deben ser no terminales. En realidad, la existencia de la derivación anterior, de la cual son una parte, implica que cada $X_i \Rightarrow^* \varepsilon$, y en menos de n pasos. De este modo, mediante la hipótesis de inducción, para cada i, Primero(X_i) contiene ε. Finalmente, por definición, Primero(A) debe contener a ε.

Ofrecemos varios ejemplos del cálculo de conjuntos Primero para no terminales.

Ejemplo 4.9

Considere nuestra gramática de expresión entera simple:[2]

exp → *exp opsuma term* | *term*
opsuma → + | -
term → *term opmult factor* | *factor*
opmult → *
factor → (*exp*) | **número**

Escribimos cada opción por separado de manera que podamos considerarlas en orden (también las numeraremos como referencia):

(1) *exp* → *exp opsuma term*
(2) *exp* → *term*
(3) *opsuma* → +
(4) *opsuma* → -
(5) *term* → *term opmult factor*
(6) *term* → *factor*
(7) *opmult* → *
(8) *factor* → (*exp*)
(9) *factor* → **número**

Como esta gramática no contiene producciones ε, podemos utilizar el algoritmo simplificado que se muestra en la figura 4.7. También notamos que las reglas recursivas por la izquierda 1 y 5 no agregarán nada al cálculo de los conjuntos Primero.[3] Por ejemplo, la regla gramatical 1 establece sólo que Primero(*exp*) debería sumarse a Primero(*exp*). De este modo, podríamos eliminar estas producciones del cálculo. Sin embargo, en este ejemplo las mantendremos en la lista para mayor claridad.

Ahora aplicamos el algoritmo de la figura 4.7, considerando las producciones en el orden que acabamos de dar. La producción 1 no provoca cambios. La producción 2 agrega el contenido de Primero(*term*) a Primero(*exp*). Pero Primero(*term*) actualmente está vacío, de modo que esto tampoco cambia nada. Las reglas 3 y 4 agregan + y - a Primero(*opsuma*), respectivamente, de manera que Primero(*opsuma*) = {+, -}. La regla 5 no agrega nada. La regla 6 agrega Primero(*factor*) a Primero(*term*), pero Primero(*factor*) continúa vacío, por lo tanto, no ocurre ningún cambio. La regla 6 agrega * a Primero(*opmult*), de manera que Primero(*opmult*) = {*}. La regla 8 agrega (a Primero(*factor*), mientras que la regla 9 agrega **number** a Primero(*factor*), de modo que Primero(*factor*) = {(, **number**}. Ahora volvemos a comenzar con la regla 1, puesto que ha habido cambios. En este caso no se provocan cambios desde la regla 1 hasta la regla 5 (Primero(*term*) todavía está vacío). La regla 6 agrega Primero(*factor*) a Primero(*term*) y Primero(*factor*) = {(, **number**}, de modo que ahora también Primero(*term*) = {(, **number**}. Las reglas 8 y 9 no producen más cambios. Nuevamente, debemos comenzar con la regla 1, puesto que un conjunto ha cambiado. La regla 2 al final agregará el nuevo contenido de Primero(*term*) a Primero(*exp*), por consiguiente, Primero(*exp*) = {(, **number**}. Es necesario un paso más a través de las reglas gramaticales, y no se presentan más cambios, así que después de cuatro pasos calculamos los siguientes conjuntos Primero:

---

2. Esta gramática tiene recursión por la izquierda y no es LL(1), de modo que no podemos construir una tabla de análisis sintáctico LL(1) para ella. Sin embargo, todavía es un ejemplo útil de cómo calcular conjuntos Primero.

3. En la presencia de producciones, las reglas recursivas por la izquierda pueden contribuir a los conjuntos Primero.

Primero(*exp*) = { (, **número** }
Primero(*term*) = { (, **número** }
Primero(*factor*) = { (, **número** }
Primero(*opsuma*) = {+, -}
Primero(*opmult*) = {*}

(Advierta que si hubiéramos enumerado las reglas gramaticales para *factor* en primer lugar en vez de al último, podríamos haber reducido el número de pasos de cuatro a dos.) Indicamos este cálculo en la tabla 4.6. En esa tabla sólo se registran los cambios en el cuadro apropiado donde ocurren. Las entradas en blanco indican que ningún conjunto ha cambiado en ese paso. También eliminamos el último paso, puesto que no ocurre ningún cambio.

Tabla 4.6
Cálculo de conjuntos Primero para la gramática del ejemplo 4.9

Regla gramatical	Paso 1	Paso 2	Paso 3
exp → *exp* *opsuma term*			
exp → *term*			Primero(*exp*) = { (, **número** }
opsuma → +	Primero(*opsuma*) = {+}		
opsuma → -	Primero(*opsuma*) = {+, -}		
term → *term* *opmult factor*			
term → *factor*		Primero(*term*) = { (, **número** }	
opmult → *	Primero(*opmult*) = {*}		
factor → (*exp*)	Primero(*factor*) = { (}		
factor → **número**	Primero(*factor*) = { (, **número** }		

§

Ejemplo 4.10 Considere la gramática (factorizada por la izquierda) de las sentencias if (ejemplo 4.5):

sentencia → *sent-if* | **otro**
sent-if → **if** (*exp*) *sentencia parte-else*
parte-else → **else** *sentencia* | ε
exp → **0** | **1**

Esta gramática tiene una producción ε, pero lo único que se puede anular es el no terminal *parte-else*, de modo que pueden surgir muy pocas complicaciones durante el cálculo. En realidad, sólo necesitaremos agregar ε en un paso, lo que no afectará a ninguno de los otros pasos,

puesto que ninguno comienza con un no terminal cuyo conjunto Primero contenga ε. Esto es típico en las gramáticas de lenguajes de programación reales, donde las producciones ε casi siempre están muy limitadas y rara vez exhiben la complejidad del caso general.

Como antes, escribimos las opciones de regla gramatical por separado y las numeramos:

(1) *sentencia* → *sent-if*
(2) *sentencia* → `otro`
(3) *sent-if* → `if` (*exp*) *sentencia parte-else*
(4) *parte-else* → `else` *sentencia*
(5) *parte-else* → ε
(6) *exp* → `0`
(7) *exp* → `1`

De nueva cuenta, recorremos paso a paso las opciones de producción, haciendo una nueva pasada siempre que un conjunto Primero haya cambiado en la pasada anterior. La regla gramatical 1 comienza sin hacer cambios, puesto que Primero(*sent-if*) todavía está vacío. La regla dos agrega el terminal `otro` a Primero(*sentencia*), de modo que Primero(*sentencia*) = { `otro`}. La regla tres agrega `if` a Primero(*sent-if*), de tal modo que Primero(*sent-if*) = {`if`}. La regla 4 agrega `else` a Primero(*parte-else*), de manera que Primero(*parte-else*) = {`else`}. La regla 5 agrega ε a Primero(*parte-else*), de tal suerte que Primero(*parte-else*) = {`else`, ε}. Las reglas 6 y 7 agregan `0` y `1` a Primero(*exp*), de modo que Primero(*exp*) = {`0`, `1`}. Ahora efectuamos una segunda pasada, comenzando con la regla 1. Esta regla ahora agrega `if` a Primero(*sentencia*), puesto que Primero(*sent-if*) contiene este terminal. De este modo, Primero(*sentencia*) = {`if`, `otro`}. Ningún otro cambio se presenta durante la segunda pasada, y una tercera pasada no da como resultado ningún cambio. De este modo, calculamos los siguientes conjuntos Primero:

Primero(*sentencia*) = {`if`, `otro`}
Primero(*sent-if*) = {`if`}
Primero(*parte-else*) = {`else`, ε}
Primero(*exp*) = {`0`, `1`}

La tabla 4.7 exhibe este cálculo de manera semejante a la tabla 4.6. Como antes, la tabla sólo muestra cambios y no exhibe la pasada final (donde no se presentan cambios).

Tabla 4.7
Cálculo de conjuntos Primero para la gramática del ejemplo 4.10

Regla gramatical	Paso 1	Paso 2
sentencia → *sent-if*		Primero(*sentencia*) = {`if`, `otro`}
sentencia → `otro`	Primero(*sentencia*) = {`otro`}	
sent-if → `if` (*exp*) *sentencia parte-else*	Primero(*sent-if*) = {`if`}	
parte-else → `else` *sentencia*	Primero(*parte-else*) = {`else`}	
parte-else → ε	Primero(*parte-else*) = {`else`, ε}	
exp → `0`	Primero(*exp*) = {`0`}	
exp → `1`	Primero(*exp*) = {`0`, `1`}	

§

Ejemplo 4.11 Considere la gramática siguiente para secuencias de sentencias (véase el ejemplo 4.4):

secuencia-sent → *sent sec-sent'*
sec-sent' → ; *secuencia-sent* | ε
sent → s

De nueva cuenta, enumeramos de manera individual las opciones de producción:

(1) *secuencia sent* → *sent sec-sent'*
(2) *sec-sent'* → ; *secuencia sent*
(3) *sec-sent'* → ε
(4) *sent* → s

En la primera pasada la regla 1 no agrega nada. Las reglas 2 y 3 producen Primero(*sec-sent'*) = {;, ε}. La regla 4 produce Primero(*sentencia*) = {s}. En la segunda pasada, la regla 1 ahora produce Primero(*secuencia-sent*) = Primero(*sent*) = {s}. No se hace ningún otro cambio, y una tercera pasada tampoco produce cambio alguno. Calculamos los siguientes conjuntos Primero:

Primero(*secuencia-sent*) = {s}
Primero(*sent*) = {s}
Primero(*sec-sent'*) = {;, ε}

Dejamos al lector la construcción de una tabla semejante a las tablas 4.6 y 4.7. §

4.3.2 Conjuntos Siguiente

Definición

Dado un no terminal A, el conjunto **Siguiente(A)**, compuesto de terminales, y posiblemente \$, se define de la manera siguiente.

1. Si A es el símbolo inicial, entonces \$ está en Siguiente(A).
2. Si hay una producción $B \rightarrow \alpha\ A\ \gamma$, entonces Primero($\gamma$) − {$\varepsilon$} está en Siguiente($A$).
3. Si hay una producción $B \rightarrow \alpha\ A\ \gamma$ tal que ε está en Primero(γ), entonces Siguiente(A) contiene Siguiente(B).

Primero examinamos el contenido de esta definición y después escribimos el algoritmo para el cálculo de los conjuntos Siguiente que resultan de ello. La primera cosa que advertimos es que el signo monetario \$, utilizado para marcar el fin de la entrada, se comporta como si fuera un token en el cálculo del conjunto Siguiente. Sin él, no tendríamos un símbolo para seguir la cadena entera a ser comparada. Puesto que una cadena así se genera mediante el símbolo inicial de la gramática, siempre se debe agregar el símbolo \$ al conjunto Siguiente del símbolo inicial. (Será el único miembro del conjunto Siguiente del símbolo inicial si este símbolo nunca aparece en el lado derecho de una producción.)

La segunda cosa que observamos es que el "pseudotoken" vacío ε nunca es un elemento de un conjunto Siguiente. Esto tiene sentido porque ε se utilizaba en conjuntos Primero sólo para marcar las cadenas que pudieran desaparecer. En realidad no es posible reconocerlo en la entrada. Los símbolos siguientes, por otra parte, siempre se compararán con los

tokens de entrada existentes (incluyendo el símbolo $, que se concordará con un EOF generado por el analizador léxico).

También notamos que los conjuntos Siguiente están definidos sólo para no terminales, mientras que los conjuntos Primero también están definidos para terminales y para cadenas de terminales y no terminales. *Podríamos* extender la definición de conjuntos Siguiente a cadenas de símbolos, pero no es necesario, puesto que al construir tablas de análisis sintáctico LL(1) sólo necesitaremos los conjuntos Siguiente de no terminales.

Finalmente, advertimos que la definición de conjuntos Siguiente funciona "por la derecha" en producciones, mientras que la definición de los conjuntos Primero funciona "por la izquierda". Con esto queremos decir que una producción $A \rightarrow \alpha$ *no* tiene información acerca del conjunto Siguiente de A si α no contiene A. Sólo las ocurrencias de A en los lados derechos de las producciones contribuirán a Siguiente(A). Por lo tanto, en general cada selección de regla gramatical hará contribuciones a los conjuntos Siguiente *de cada* no terminal que aparezca en el lado derecho, a diferencia del caso de los conjuntos Primero, donde cada selección de regla gramatical se agrega al conjunto Primero de únicamente un no terminal (el que está a la izquierda).

Además, dada una regla gramatical $A \rightarrow \alpha\ B$, Siguiente(B) incluirá a Siguiente(A) por el caso 3 de la definición. Esto se debe a que, en cualquier cadena que contenga A, A se podría reemplazar por $\alpha\ B$ (es decir, "libertad de contexto" en acción). Esta propiedad es, en cierto sentido, lo opuesto de la situación para los conjuntos Primero, donde si $A \rightarrow B\ \alpha$, entonces Primero(A) incluye Primero(B) (excepto posiblemente para ε).

En la figura 4.8 damos el algoritmo para el cálculo de los conjuntos Siguiente que resultan de su correspondiente definición. Con este algoritmo calculamos conjuntos Siguiente para las mismas tres gramáticas, para las cuales antes calculamos los conjuntos Primero. (Como en los conjuntos Primero, cuando no hay producciones ε, el algoritmo se simplifica, simplificación que dejamos al lector.)

Figura 4.8
Algoritmo para el cálculo de conjuntos Siguiente

Siguiente(símbolo-inicial) := {$} ;
for *todos los no terminales* $A \neq$ *símbolo-inicial* **do** *Siguiente*(A) := { } ;
while *existan cambios a cualquier conjunto Siguiente* **do**
  for *cada producción* $A \rightarrow X_1X_2 \ldots X_n$ **do**
    for each X_i *que sea un no terminal* **do**
      agregue Primero($X_{i+1}X_{i+2} \ldots X_n$) − {$\varepsilon$} *a Siguiente*($X_i$)
      (* *Nota: si* $i=n$, *entonces* $X_{i+1}X_{i+2} \ldots X_n = \varepsilon$ *)
      if ε *está en Primero*($X_{i+1}X_{i+2} \ldots X_n$) **then**
        agregue Siguiente(A) *a Siguiente*(X_i)

Ejemplo 4.12

Considere otra vez la gramática de expresión simple, cuyos conjuntos Primero calculamos en el ejemplo 4.9 de la manera siguiente:

Primero(*exp*) = { (, **número** }
Primero(*term*) = { (, **número** }
Primero(*factor*) = { (, **número** }
Primero(*opsuma*) = {+, -}
Primero(*opmult*) = {*}

Volvemos a escribir las selecciones u opciones de producción con numeración:

(1) *exp* → *exp opsuma term*
(2) *exp* → *term*
(3) *opsuma* → +
(4) *opsuma* → -
(5) *term* → *term opmult factor*
(6) *term* → *factor*
(7) *opmult* → *
(8) *factor* → (*exp*)
(9) *factor* → **número**

Las reglas 3, 4, 7 y 9 no tienen no terminales en sus lados derechos, de modo que no agregan nada al cálculo de los conjuntos Siguiente. Consideramos las otras reglas en orden. Antes de comenzar, establezcamos Siguiente(*exp*) = {$}, todos los otros conjuntos Siguiente se inicializan como vacíos.

La regla 1 afecta a los conjuntos Siguiente de tres no terminales: *exp*, *opsuma* y *term*. Primero(*opsuma*) se agrega a Siguiente(*exp*), de modo que ahora Siguiente(*exp*) = {$, +, -}. A continuación, Primero(*term*) se agrega a Siguiente(*opsuma*), de manera que Siguiente(*opsuma*) = { (, **number**}. Finalmente, Siguiente(*exp*) se agrega a Siguiente(*term*), de modo que Siguiente(*term*) = {$, +, -}.

La regla 2 vuelve a causar que Siguiente(*exp*) se agregue a Siguiente(*term*), pero esto se acaba de realizar mediante la regla 1, de manera que no ocurre un cambio a los conjuntos Siguiente.

La regla 5 tiene tres efectos. Se agrega Primero(*opmult*) a Siguiente(*term*), de manera que Siguiente(*term*) = {$, +, -, *}. Entonces Primero(*factor*) se agrega a Siguiente(*opmult*), de modo que Siguiente(*opmult*) = { (, **number**}. Finalmente, Siguiente(*term*) se agrega a Siguiente(*factor*), de modo que Siguiente(*factor*) = {$, +, -, *}.

La regla 6 tiene el mismo efecto que el último paso para la regla 5 y, por lo tanto, no se hacen cambios.

Finalmente, la regla 8 agrega Primero( )) = {) } a Siguiente(*exp*) de modo que Siguiente(exp) = {$, +, -,)}.

En la segunda pasada la regla 1 agrega) a Siguiente(*term*) (de manera que Siguiente(*term*) = {$, +, -, *,) }), y la regla 5 agrega) a Siguiente(*factor*) (así que Siguiente(*factor*) = {$, +, -, *,) }). Una tercera pasada no produce más cambios, y el algoritmo termina. Hemos calculado los siguientes conjuntos Siguiente:

Siguiente(*exp*) = {$, +, -,) }
Siguiente(*opsuma*) = { (, **número**}
Siguiente(*term*) = {$, +, -, *,) }
Siguiente(*opmult*) = { (, **número** }
Siguiente(*factor*) = {$, +, -, *,) }

Como en el cálculo de los conjuntos Primero, exhibimos el progreso de este cálculo en la tabla 4.8. Como antes, omitimos el paso de conclusión en esta tabla y sólo indicamos los cambios a los conjuntos Siguiente cuando ocurren. Omitimos también las cuatro selecciones de regla gramatical que no pueden afectar el cálculo. (Incluimos las dos reglas *exp* → *term* y *term* → *factor* porque pueden tener un efecto, aun cuando no lo tengan en realidad.)

Tabla 4.8
Cálculo de conjuntos Siguiente para la gramática del ejemplo 4.12

Regla gramatical	Paso 1	Paso 2
exp → *exp opsuma term*	Siguiente(*exp*) = {$, +, -} Siguiente(*opsuma*) = {(, **número**} Siguiente(*term*) = {$, +, -}	Siguiente(*term*) = {$, +, -, *,)}
exp → *term*		
term → *term opmult factor*	Siguiente(*term*) = {$, +, -, *} Siguiente(*opmult*) = {(, **número**} Siguiente(*factor*) = {$, +, -, *}	Siguiente(*factor*) = {$, +, -, *,)}
term → *factor*		
factor → (*exp*)	Siguiente(*exp*) = {$, +, -,)}	

§

Ejemplo 4.13

Considere de nuevo la gramática simplificada de las sentencias if, cuyos conjuntos Primero calculamos en el ejemplo 4.10, de la manera siguiente:

Primero(*sentencia*) = {**if**, **otro**}
Primero(*sent-if*) = {**if**}
Primero(*parte-else*) = {**else**, ε}
Primero(*exp*) = {**0**, **1**}

Repetimos aquí las opciones de producción con numeración:

(1) *sentencia* → *sent-if*
(2) *sentencia* → **otro**
(3) *sent-if* → **if** (*exp*) *sentencia parte-else*
(4) *parte-else* → **else** *sentencia*
(5) *parte-else* → ε
(6) *exp* → **0**
(7) *exp* → **1**

Las reglas 2, 5, 6 y 7 no tienen efecto sobre el cálculo de los conjuntos Siguiente, de modo que los ignoraremos.

Comencemos estableciendo que Siguiente(*sentencia*) = {$} e inicializando los Siguiente de los otros no terminales como vacíos. La regla 1 agrega ahora Siguiente(*sentencia*) a Siguiente(*sent-if*), de modo que Siguiente(*sent-if*) = {$}. La regla 3 afecta los conjuntos Siguiente de *exp*, *sentencia* y *parte-else*. En primer lugar, Siguiente(*exp*) implica Primero()) = {)}, de modo que Siguiente(*exp*) = {)}. Entonces Siguiente(*sentencia*) implica Primero(*parte-else*) = {ε}, de manera que Siguiente(*sentencia*) = {$, **else**}. Finalmente, Siguiente(*sent-if*) se agrega tanto a Siguiente(*parte-else*) como a Siguiente(*sentencia*)

(esto se debe a que *sent-if* puede desaparecer). La primera adición nos proporciona Siguiente(*parte-else*) = {$}, pero la segunda no produce cambios. Finalmente, la regla 4 provoca que Siguiente(*parte-else*) se agregue a Siguiente(*sentencia*), lo que tampoco produce ningún cambio.

En la segunda pasada, la regla 1 vuelve a agregar Siguiente(*sentencia*) a Siguiente(*sent-if*), lo que resulta en Siguiente(*sent-if*) = {$, **else**}. La regla 3 ahora provoca que el terminal **else** se agregue a Siguiente(*parte-else*), de modo que Siguiente(*parte-else*) = {$, **else**}. Finalmente, la regla 4 no ocasiona más cambios. La tercera pasada tampoco produce cambios adicionales, y hemos calculado los siguientes conjuntos Siguiente:

Siguiente(*sentencia*) = {$, **else**}
Siguiente(*sent-if*) = {$, **else**}
Siguiente(*parte-else*) = {$, **else**}
Siguiente(*exp*) = {) }

Dejamos al lector el construir una tabla para este cálculo como en el ejemplo anterior. §

Ejemplo 4.14

Calculemos los conjuntos Siguiente para la gramática simplificada de secuencia de sentencias del ejemplo 4.11, con las opciones de regla gramatical:

(1) *secuencia-sent* → *sent sec-sent'*
(2) *sec-sent'* → **;** *secuencia-sent*
(3) *sec-sent'* → ε
(4) *sent* → **s**

En el ejemplo 4.11 calculamos los siguientes conjuntos Primero:

Primero(*secuencia-sent*) = {**s**}
Primero(*sent*) = {**s**}
Primero(*sec-sent'*) = {**;**, ε}

Las reglas gramaticales 3 y 4 no contribuyen al cálculo del conjunto Siguiente. Comenzamos con Siguiente(*secuencia-sent*) = {$} y los otros conjuntos Siguiente como vacíos. La regla 1 produce Siguiente(*sent*) = {**;**} y Siguiente(*sec-sent'*) = {$}. La regla 2 no tiene efectos. Una segunda pasada no produce cambios adicionales. De este modo, calculamos los conjuntos Siguiente

Siguiente(*secuencia-sent*) = {$}
Siguiente(*sent*) = {**;**}
Siguiente(*sec-sent'*) = {$} §

4.3.3 Construcción de tablas de análisis sintáctico LL(1)

Considere ahora la construcción original de las entradas de la tabla de análisis sintáctico LL(1), como se dieron en la sección 4.2.2:

1. Si $A \rightarrow \alpha$ es una opción de producción, y existe una derivación $\alpha \Rightarrow^* a\ \beta$, donde a es un token, entonces agregue $A \rightarrow \alpha$ a la entrada de tabla $M[A, a]$.
2. Si $A \rightarrow \varepsilon$ es una producción ε y existe una derivación $S\ \$ \Rightarrow^* \alpha\ A\ a\ \beta$, donde S es el símbolo inicial y a es un token (o signo de \$), entonces agregue $A \rightarrow \varepsilon$ a la entrada de tabla $M[A, a]$.

Evidentemente, el token a en la regla 1 está en Primero(α), y el token a de la regla 2 está en Siguiente(A). De este modo llegamos a la siguiente construcción algorítmica de la tabla de análisis sintáctico LL(1):

Construcción de la tabla de análisis sintáctico LL(1) $M[N, T]$

Repita los siguientes dos pasos para cada no terminal A y opción de producción $A \rightarrow \alpha$.

1. Para cada token a en Primero(α), agregue $A \rightarrow \alpha$ a la entrada $M[A, a]$.
2. Si ε está en Primero(α), para cada elemento a de Siguiente(A) (un token o signo de \$), agregue $A \rightarrow \alpha$ a $M[A, a]$.

El teorema siguiente es fundamentalmente una consecuencia directa de la definición de una gramática LL(1) y la construcción de la tabla de análisis sintáctico recién dada, y dejamos su demostración para los ejercicios:

Teorema

Una gramática en BNF es **LL(1)** si se satisfacen las condiciones siguientes.

1. Para toda producción $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$, Primero($\alpha_i$) $\cap$ Primero(α_j) es vacío para toda i y j, $1 \leq i, j \leq n$, $i \neq j$.
2. Para todo no terminal A tal que Primero(A) contenga a ε, Primero(A) $\cap$ Siguiente(A) es vacío.

Ahora examinaremos algunos ejemplos de tablas de análisis sintáctico para gramáticas que ya vimos con anterioridad.

Ejemplo 4.15

Considere la gramática de expresión simple que hemos estado utilizando como ejemplo estándar a lo largo de este capítulo. Esta gramática, como fue dada originalmente (véase el ejemplo 4.9 anterior) es recursiva por la izquierda. En la última sección escribimos una gramática equivalente utilizando la eliminación de recursión por la izquierda de la manera siguiente:

$$
\begin{aligned}
&exp \rightarrow term\ exp' \\
&exp' \rightarrow opsuma\ term\ exp' \mid \varepsilon \\
&opsuma \rightarrow \texttt{+} \mid \texttt{-} \\
&term \rightarrow factor\ term' \\
&term' \rightarrow opmult\ factor\ term' \mid \varepsilon \\
&opmult \rightarrow \texttt{*} \\
&factor \rightarrow \texttt{(}\ exp\ \texttt{)} \mid \textbf{\texttt{número}}
\end{aligned}
$$

Debemos calcular los conjuntos Primero y Siguiente para los no terminales de esta gramática. Dejamos este cálculo para los ejercicios y simplemente establecemos el resultado:

Primero(*exp*) = { (, **número**}	Siguiente(*exp*) = {$,) }
Primero(*exp'*) = {+, -, ε}	Siguiente(*exp'*) = {$,) }
Primero(*opsuma*) = {+, -}	Siguiente(*opsuma*) = { (, **número**}
Primero(*term*) = { (, **número**}	Siguiente(*term*) = {$,), +, -}
Primero(*term'*) = {*, ε}	Siguiente(*term'*) = {$,), +, -}
Primero(*opmult*) = {*}	Siguiente(*opmult*) = { (, **número**}
Primero(*factor*) = { (, **numéro**}	Siguiente(*factor*) = {$,), +, -, *}

Al aplicar la construcción de la tabla de análisis sintáctico LL(1) que acabamos de describir, obtenemos una tabla como la 4.4 de la página 163. §

Ejemplo 4.16 Considere la gramática simplificada de las sentencias if

sentencia → *sent-if* | **otro**
sent-if → **if** (*exp*) *sentencia parte-else*
parte-else → **else** *sentencia* | ε
exp → **0** | **1**

Los conjuntos Primero de esta gramática se calcularon en el ejemplo 4.10, mientras que los conjuntos Siguiente en el ejemplo 4.13. Mostramos estos conjuntos de nueva cuenta aquí:

Primero(*sentencia*) = {**if**, **otro**}	Siguiente(*sentencia*) = {$, **else**}
Primero(*sent-if*) = {**if**}	Siguiente(*sent-if*) = {$, **else**}
Primero(*parte-else*) = {**else**, ε}	Siguiente(*parte-else*) = {$, **else**}
Primero(*exp*) = {**0**, **1**}	Siguiente(*exp*) = {) }

La construcción de la tabla de análisis sintáctico LL(1) nos proporciona la tabla 4.3 de la página 156. §

Ejemplo 4.17 Considere la gramática del ejemplo 4.4 (en la que se aplicó factorización por la izquierda):

secuencia-sent → *sent sec-sent'*
sec-sent' → **;** *secuencia-sent* | ε
sent → **s**

Esta gramática tiene los siguientes conjuntos Primero y Siguiente

Primero(*secuencia-sent*) = {**s**}	Siguiente(*secuencia-sent*) = {$}
Primero(*sent*) = {**s**}	Siguiente(*sent*) = {**;**, $}
Primero(*sec-sent'*) = {**;**, ε}	Siguiente(*sec-sent'*) = {$}

y la tabla de análisis sintáctico LL(1) en la parte superior de la página siguiente.

M[N, T]	s	;	$
secuencia-sent	*secuencia-sent* → *sent sec-sent'*		
sent	*sent* → s		
sec-sent'		*sec-sent'* → ; *secuencia-sent*	*sec-sent'* → ε

§

4.3.4 Extensión de la búsqueda hacia delante. Analizadores sintácticos LL(k)

El trabajo previo se puede extender a k símbolos de búsqueda hacia delante. Por ejemplo, podemos definir Primero$_k(\alpha) = \{ w_k \mid \alpha \Rightarrow^* w \}$, donde w es una cadena de tokens y w_k = los primeros k tokens de w (o w, si w tiene menos de k tokens). De manera similar, podemos definir Siguiente$_k(A) = \{w_k \mid S\$ \Rightarrow^* \alpha A w\}$. Aunque estas definiciones son menos "algorítmicas" que nuestras definiciones para el caso $k = 1$, pueden desarrollarse algoritmos que permitan calcular estos conjuntos, y construir como antes la tabla de análisis sintáctico LL(k).

Sin embargo, en el análisis sintáctico LL(k) se presentan diversas complicaciones. En primer lugar, la tabla de análisis sintáctico se vuelve mucho más extensa, puesto que el número de columnas se incrementa de manera exponencial con k. (Esto se puede contrarrestar hasta cierto punto utilizando métodos de compresión de tablas.) En segundo lugar, la tabla misma de análisis sintáctico no expresa la potencia completa del análisis sintáctico LL(k), sobre todo porque no todas las cadenas Siguiente se presentan en todos los contextos. De este modo, el análisis sintáctico por medio de la tabla tal como la construimos se distingue del análisis sintáctico LL(k) al nombrarlo **análisis sintáctico LL(k) fuerte** o **análisis sintáctico SLL(k)** [por las siglas de **Strong LL(k)**]. Remitimos al lector a la sección de notas y referencias para más información.

Los análisis sintácticos LL(k) y SLL(k) no son comunes, en parte debido a la complejidad agregada, pero sobre todo porque si una gramática falla en ser LL(1) probablemente en la práctica no sea LL(k) para ninguna k. Por ejemplo, una gramática con recursión por la izquierda nunca es LL(k), sin importar cuán grande sea k. Sin embargo, los analizadores sintácticos descendentes recursivos pueden utilizar de manera selectiva búsquedas hacia delante más grandes cuando es necesario, e incluso, como hemos visto, emplear métodos ex profeso para analizar sintácticamente gramáticas que no sean LL(k) para ninguna k.

4.4 UN ANALIZADOR SINTÁCTICO DESCENDENTE RECURSIVO PARA EL LENGUAJE TINY

En esta sección estudiaremos el analizador sintáctico descendente recursivo completo para el lenguaje TINY que se muestra en el apéndice B. El analizador sintáctico construye un árbol sintáctico como se describió en la sección 3.7 del capítulo anterior y, adicionalmente, imprime una representación del árbol sintáctico para el archivo de listado. El analizador sintáctico utiliza la EBNF tal como se proporciona en la figura 4.9, correspondiente al BNF de la figura 3.6 del capítulo 3.

Figura 4.9
Gramática del lenguaje TINY en EBNF

programa → *secuencia-sent*
secuencia-sent → *sentencia* { ; *sentencia* }
sentencia → *sent-if* | *sent-repeat* | *sent-assign* | *sent-read* | *sent-write*
sent-if → **if** *exp* **then** *secuencia-sent* [**else** *secuencia-sent*] **end**
sent-repeat → **repeat** *secuencia-sent* **until** *exp*
sent-assign → **identificador** := *exp*
sent-read → **read identificador**
sent-write → **write** *exp*
exp → *exp-simple* [*op-comparación exp-simple*]
op-comparación → < | =
exp-simple → *term* { *opsuma term* }
opsuma → + | -
term → *factor* { *opmult factor* }
opmult → * | /
factor → (*exp*) | **número** | **identificador**

El analizador sintáctico de TINY sigue de cerca el esquema del análisis sintáctico descendente recursivo que se dio en la sección 4.1. El analizador sintáctico se compone de dos archivos de código, **parse.h** y **parse.c**. El archivo **parse.h** (apéndice B, líneas 850-865) es muy simple: consta de una declaración simple

```
TreeNode * parse(void);
```

que define la rutina de análisis sintáctico principal **parse**, la cual devuelve un apuntador al árbol sintáctico construido por el analizador sintáctico. El archivo **parse.c** se proporciona en el apéndice B, líneas 900-1114. Se compone de 11 procedimientos mutuamente recursivos que corresponden directamente a la gramática EBNF de la figura 4.9: uno para *secuencia-sent* (*stmt-sequence*), uno para *sentencia* (*statement*), uno para cada una de las cinco clases diferentes de sentencias y cuatro para los diferentes niveles de precedencia de las expresiones. Los no terminales del operador no se incluyen como procedimientos, pero se reconocen como parte de sus expresiones asociadas. Tampoco hay correspondencia de procedimiento para *programa* (*program*), puesto que un programa sólo es una secuencia de sentencias, de manera que la rutina **parse** (**"análisis sintáctico"**) simplemente llama a (**"secuencia-sent"**) **stmt_sequence**.

El código del analizador sintáctico también incluye una variable estática **token** que mantiene el token de búsqueda hacia delante, un procedimiento **match** que busca un token específico, al que llama **getToken** si lo encuentra y declara un error en caso contrario, y un procedimiento **syntaxError** que imprime un mensaje de error en el archivo del listado. El procedimiento principal **parse** inicializa **token** para el primer token en la entrada, llama a **stmt_sequence** y posteriormente verifica si está el final del archivo fuente antes de regresar al árbol construido por **stmt_sequence**. (Si después de que regresa **stmt_sequence** hay más tokens, significa que existe un error.)

El contenido de cada procedimiento recursivo debería ser relativamente autoexplicativo, con la posible excepción de **stmt_sequence**, el cual se escribió en una forma algo más compleja para mejorar el manejo de errores; esto se explicará en el análisis del manejo de errores que en breve se realizará. Los procedimientos de análisis sintáctico recursivo utilizan tres procedimientos utilitarios, los cuales se reúnen por simplicidad en el archivo **util.c**

(apéndice B, líneas 350-526), con la interfaz **util.h** (apéndice B, líneas 300-335). Estos *procedimientos son*

1. **newStmtNode** (líneas 405-421), el cual toma un parámetro indicando la clase de sentencia, y asigna un nuevo nodo de sentencia de esa clase, devolviendo un apuntador al nodo recién asignado;
2. **newExpNode** (líneas 423-440), el cual toma un parámetro indicando la clase de expresión, y asigna un nuevo nodo de expresión de esa clase, devolviendo un apuntador al nodo recién asignado; y
3. **copyString** (líneas 442-455), el cual toma un parámetro de cadena, asigna espacio suficiente para una copia, y copia la cadena, devolviendo un apuntador a la copia recién asignada.

El procedimiento **copyString** es necesario porque el lenguaje C no asigna de manera automática espacio para cadenas y porque el analizador léxico vuelve a utilizar el mismo espacio para los valores de cadena (o lexemas) de los tokens que reconoce.

También se incluye en **util.c** un procedimiento **printTree** (líneas 473-506) que escribe una versión lineal del árbol sintáctico para el listado, de manera que podamos observar el resultado de un análisis sintáctico. Este procedimiento se llama desde el programa principal, bajo el control de la variable global **traceParse**.

El procedimiento **printTree** funciona al imprimir información del nodo y luego efectuar una sangría para imprimir la información del hijo. El árbol real se puede reconstruir a partir de estas sangrías. **traceParse** imprime el árbol sintáctico del programa TINY de muestra (véase el capítulo 3, figuras 3.8 y 3.9, páginas 137-138) en el archivo del listado como se muestra en la figura 4.10.

Figura 4.10
Exhibición de un árbol sintáctico de TINY mediante el procedimiento printTree

```
Read: x
If
  Op: <
    const: 0
    Id: x
  Assign to: fact
    const: 1
  Repeat
    Assign to: fact
      Op: *
        Id: fact
        Id: x
    Assign to: x
      Op: -
        Id: x
        const: 1
    Op: =
      Id: x
      const: 0
  Write
    Id: fact
```

4.5 RECUPERACIÓN DE ERRORES EN ANALIZADORES SINTÁCTICOS DESCENDENTES

La respuesta de un analizador sintáctico a los errores de sintaxis a menudo es un factor crítico en la utilidad de un compilador. Un analizador sintáctico debe determinar por lo menos si un programa es sintácticamente correcto o no. Un analizador sintáctico que sólo realiza esta tarea se denomina **reconocedor**, puesto que se limita a reconocer cadenas en el lenguaje libre de contexto generado por la gramática del lenguaje de programación en cuestión. Vale la pena establecer que cualquier analizador sintáctico debe comportarse por lo menos como un reconocedor; es decir, si un programa contiene un error de sintaxis, el analizador sintáctico debe indicar que existe *algún* error y, a la inversa, si un programa no contiene errores de sintaxis, entonces el analizador sintáctico no debe afirmar que existe alguno.

Más allá de este comportamiento mínimo, un analizador sintáctico puede mostrar muchos niveles diferentes de respuestas a los errores. Habitualmente, un analizador sintáctico intentará proporcionar un mensaje de error importante, cuando menos para el primer error que encuentre, y también intentar determinar de manera tan exacta como sea posible la ubicación en la que haya ocurrido el error. Algunos analizadores sintácticos pueden ir tan lejos como para intentar alguna forma de **corrección de errores** (o, para decirlo de manera quizás más apropiada, **reparación de errores**), donde el analizador sintáctico intenta deducir un programa correcto a partir del incorrecto que se le proporciona. Si intenta esto, la mayor parte de las veces se limitará sólo a casos fáciles, como la falta de un signo de puntuación. Existe un grupo de algoritmos que se pueden aplicar para encontrar un programa correcto que en cierto sentido se parezca mucho al proporcionado (habitualmente en términos del número de tokens que deben insertarse, eliminarse o modificarse). Esta **corrección de errores de distancia mínima** es por lo regular muy ineficiente como para aplicarse a cualquier error y, en cualquier caso, la reparación de errores que da como resultado a menudo está muy lejos de lo que el programador pretendía. Por consiguiente, es raro que se vea en analizadores sintácticos reales. A los escritores de compiladores les es muy difícil generar mensajes de error significativos sin intentar hacer corrección de errores.

La mayoría de las técnicas para la recuperación de errores son de propósito específico, ya que se aplican a lenguajes específicos y a algoritmos de análisis sintáctico específico, con muchos casos especiales para situaciones particulares. Los principios generales son difíciles de obtener. Algunas consideraciones importantes que se aplican son las siguientes.

1. Un analizador sintáctico debería intentar determinar que ha ocurrido un error *tan pronto como fuera posible*. Esperar demasiado tiempo antes de la declaración del error significa que la ubicación del error real puede haberse perdido.
2. Después de que se ha presentado un error, el analizador sintáctico debe seleccionar un lugar probable para reanudar el análisis. Un analizador sintáctico siempre debería intentar analizar tanto código como fuera posible, a fin de encontrar tantos errores reales como sea posible durante una traducción simple.
3. Un analizador sintáctico debería intentar evitar el **problema de cascada de errores**, en la cual un error genera una larga secuencia de mensajes de error falsos.
4. Un analizador sintáctico debe evitar bucles infinitos en los errores, en los que se genera una cascada sin fin de mensajes de error sin consumir ninguna entrada.

Algunos de estos objetivos entran en conflicto entre sí, de tal manera que un escritor de compiladores tiene que efectuar "convenios" durante la construcción de un manejador de errores. Por ejemplo, el evitar los problemas de cascada de errores y bucle infinito puede ocasionar que el analizador sintáctico omita algo de la entrada, con lo que compromete el objetivo de procesar tanta información de la entrada como sea posible.

4.5.1 Recuperación de errores en analizadores sintácticos descendentes recursivos

Una forma estándar de recuperación de errores en los analizadores sintácticos descendentes recursivos se denomina **modo de alarma**. El nombre se deriva del hecho que, en situaciones

complejas, el manejador de errores consumirá un número posiblemente grande de tokens en un intento de hallar un lugar para reanudar el análisis sintáctico (en el peor de los casos, incluso puede consumir todo el resto del programa, lo que no es mejor que simplemente salir después del error). Sin embargo, cuando se implementa con cuidado, éste puede ser un método para la recuperación de errores mucho mejor que lo que implica su nombre.[4] Este modo de alarma tiene, además, la ventaja de que virtualmente asegura que el analizador sintáctico no caiga en un bucle infinito durante la recuperación de errores.

El mecanismo básico del modo de alarma es proporcionar a cada procedimiento recursivo un parámetro extra compuesto de un conjunto de **tokens de sincronización**. A medida que se efectúa el análisis sintáctico, los tokens que pueden funcionar como tokens de sincronización se agregan a este conjunto conforme se presenta cada llamada. Si se encuentra un error, el analizador sintáctico **explora hacia delante**, desechando los tokens hasta que ve en la entrada uno del conjunto de tokens de sincronización, en donde se reanuda el análisis sintáctico. Las cascadas de errores se evitan (hasta cierto punto) al no generar nuevos mensajes de error mientras tiene lugar esta exploración adelantada.

Las decisiones importantes que se tienen que tomar en este método de recuperación de errores consisten en determinar qué tokens agregar al conjunto de sincronización en cada punto del análisis sintáctico. Por lo general los conjuntos Siguiente son candidatos importantes para tales tokens de sincronización. Los conjuntos Primero también se pueden utilizar para evitar que el manejador de errores omita tokens importantes que inicien nuevas construcciones principales (como sentencias o expresiones). Los conjuntos Primero también son importantes, puesto que permiten que un analizador sintáctico descendente recursivo detecte pronto los errores en el análisis sintáctico, lo que siempre es útil en cualquier recuperación de errores. Es importante darse cuenta que el modo de alarma funciona mejor cuando el compilador "sabe" cuándo *no* alarmarse. Por ejemplo, los símbolos de puntuación perdidos, tales como los de punto y coma, y las comas, e incluso los paréntesis derechos olvidados, no siempre deberían provocar que un manejador de errores consuma tokens. Naturalmente, debe tenerse cuidado para asegurar que no se presente un bucle infinito.

Ilustraremos la recuperación de errores en modo de alarma esquematizando en pseudocódigo su implementación en la calculadora descendente recursiva de la sección 4.1.2 (véase también la figura 4.1). Además de los procedimientos *match* y *error*, que en esencia permanecen iguales (excepto que *error* ya no sale de inmediato), tenemos dos procedimientos más, *checkinput*, que realiza la verificación temprana de búsqueda hacia delante, y *scanto*, que es el token consumidor en modo de alarma propiamente dicho:

```
procedure scanto ( synchset ) ;
begin
  while not ( token in synchset ∪ { $ }) do
    getToken ;
end scanto ;

procedure checkinput ( firstset, followset ) ;
begin
  if not ( token in firstset ) then
    error ;
    scanto ( firstset ∪ followset ) ;
  end if ;
end;
```

Aquí el signo $ se refiere al fin de la entrada (EOF).

4. Wirth [1976], de hecho, llama al modo de alarma la regla de "no alarma", supuestamente en un intento de mejorar su imagen.

Estos procedimientos se utilizan como sigue en los procedimientos *exp* y *factor* (que ahora tienen un parámetro *synchset*):

```
procedure exp ( synchset ) ;
begin
  checkinput ( { (, number }, synchset ) ;
  if not ( token in synchset ) then
    term ( synchset ) ;
    while token = + or token = - do
      match (token) ;
      term ( synchset ) ;
    end while ;
    checkinput ( synchset, { (, number }) ;
 end if;
end exp ;

procedure factor ( synchset ) ;
begin
 checkinput ( { (, number }, synchset ) ;
 if not ( token in synchset ) then
  case token of
  ( : match( ( ) ;
      exp ( { ) } ) ;
      match( ) ) ;
  number :
      match(number) ;
  else error ;
  end case ;
  checkinput ( synchset, { (, number }) ;
 end if ;
end factor ;
```

Advierta cómo *checkinput* es llamado dos veces en cada procedimiento: una vez para verificar que un token en el conjunto Primero sea el token siguiente en la entrada y una segunda vez para verificar que un token en el conjunto Siguiente (o *synchset*) sea el token siguiente en la salida.

Esta forma de modo de alarma generará errores razonables (mensajes de error útiles que también se pueden agregar como parámetros a *checkinput* y *error*). Por ejemplo, la cadena de entrada `(2+-3)*4-+5` generará exactamente dos mensajes de error (uno para el primer signo de menos y otro para el segundo signo de más).

Observamos que, en general, *synchset* se pasa en las llamadas recursivas, con nuevos tokens de sincronización agregados de manera apropiada. En el caso de *factor*, se hace una excepción después de que se ve un paréntesis izquierdo: se llama a *exp* con paréntesis derecho sólo como su conjunto siguiente (*synchset* se descarta). Esto es típico de la clase de análisis de propósito específico que acompaña a la recuperación de errores en modo de alarma. (Hicimos esto de modo que, por ejemplo, la expresión `(2+*)` no generase un falso mensaje de error para el paréntesis derecho.) Dejamos un análisis del comportamiento de este código, así como su implementación en C, para los ejercicios. Por desgracia, para obtener los mejores mensajes de error y recuperación de errores, virtualmente toda prueba de token se debe examinar por la posibilidad de que una prueba más general, o una prueba más temprana, mejore el comportamiento del error.

4.5.2 Recuperación de errores en analizadores sintácticos LL(1)

La recuperación de errores en modo de alarma se puede implementar en analizadores sintácticos LL(1) de manera semejante a como se implementaron en el análisis sintáctico descendente recursivo. Como el algoritmo es no recursivo, se requiere de una nueva pila para mantener los parámetros *synchset*, y debe planearse una llamada a *checkinput* en el algoritmo antes de cada acción generada por el algoritmo (cuando un no terminal está en la parte superior de la pila).[5] Advierta que la situación de error primario ocurre con un no terminal A en la parte superior de la pila y que el token de entrada actual no esté en Primero(A) (o Siguiente(A), si ε está en Primero(A)). El caso en que un token está en la parte superior de la pila, y no es el mismo que el token de entrada actual, normalmente no ocurre, porque los tokens, en general, sólo se insertan en la pila cuando se ven, de hecho, en la entrada (los métodos de compresión de tabla pueden comprometer esto ligeramente). Dejamos las modificaciones para el algoritmo de análisis sintáctico de la figura 4.2, página 155, para los ejercicios.

Una alternativa para el uso de una pila extra es construir de manera estática los conjuntos de tokens de sincronización directamente en la tabla de análisis sintáctico LL(1), junto con las acciones correspondientes que tomaría *checkinput*. Dado un no terminal A en la parte superior de la pila y un token de entrada que no esté en Primero(A) (o Siguiente(A), si ε está en Primero(A)), existen tres alternativas posibles:

1. Extraer A de la pila.
2. Extraer de manera sucesiva tokens de la entrada hasta que se vea un token para el cual se pueda reiniciar el análisis sintáctico.
3. Insertar un nuevo no terminal en la pila.

Seleccionamos la alternativa 1 si el token de entrada actual es $ o está en Siguiente(A) y la alternativa 2 si el token de entrada actual no es $ y no está en Primero(A) $\cup$ Siguiente(A). La opción 3 ocasionalmente es útil en situaciones especiales, pero rara vez es apropiada (analizaremos un caso más adelante). Indicamos la primera acción en la tabla de análisis sintáctico mediante la notación *extraer* y la segunda mediante la notación *explorar*. (Advierta que una acción *extraer* es equivalente a una reducción mediante una producción ε.)

Con estas convenciones la tabla de análisis sintáctico LL(1) (tabla 4.4) se verá como en la tabla 4.9. El comportamiento de un analizador sintáctico LL(1) que utilice esta tabla, dada la cadena `(2+*)`, se muestra en la tabla 4.10. En esa tabla el análisis sintáctico se muestra sólo a partir del primer error (de manera que el prefijo `(2+` ya se ha reconocido con éxito). También utilizamos las abreviaturas E para *exp*, E' para *exp'*, y así sucesivamente. Observe que hay dos movimientos de error adyacentes antes que se reanude con éxito el análisis sintáctico. Podemos arreglar suprimir un mensaje de error en el segundo movimiento de error exigiendo, después del primer error, que el analizador sintáctico haga uno o más movimientos con éxito antes de generar cualquier mensaje de error nuevo. De este modo, se evitarían las cascadas de mensajes de error.

Existe (por lo menos) un problema en este método de recuperación de errores que pide una acción especial. Como muchas acciones de error extraen un no terminal desde la pila, es posible que ésta se vuelva vacía, todavía con alguna entrada para el análisis sintáctico. Un caso simple de esto en el ejemplo que acabamos de dar es cualquier inicio de cadena con un paréntesis derecho: esto causará que E se extraiga de inmediato y la pila se quede vacía con toda la entrada esperando ser consumida.

5. Las llamadas a *checkinput* al final de una comparación, como en el código en un recursivo descendente, se pueden planear también por medio de una pila especial de símbolos de manera similar al esquema del cálculo de valores de la sección 4.2.4, página 167.

Una acción que puede tomar el analizador sintáctico en esta situación es insertar el símbolo inicial en la pila y explorar hacia delante en la entrada hasta que se vea un símbolo en el conjunto Primero del símbolo inicial.

Tabla 4.9

Tabla de análisis sintáctico LL(1) (tabla 4.4) con entradas de recuperación de errores

M[N, T]	(	número	)	+	-	*	$
exp	*exp* → *term exp'*	*exp* → *term exp'*	extraer	explorar	explorar	explorar	extraer
exp'	explorar	explorar	*exp'* → ε	*exp'* → *opsuma term exp'*	*exp'* → *opsuma term exp'*	explorar	*exp'* → ε
opsuma	extraer	extraer	explorar	*opsuma* → +	*opsuma* → -	explorar	extraer
term	*term* → *factor term'*	*term* → *factor term'*	extraer	extraer	extraer	explorar	extraer
term'	explorar	explorar	*term'* → ε	*term'* → ε	*term'* → ε	*term'* → *opmult factor term'*	*term'* → ε
opmult	extraer	extraer	explorar	explorar	explorar	*opmult* → *	extraer
factor	*factor* → (*exp*)	*factor* → **número**	extraer	extraer	extraer	extraer	extraer

Tabla 4.10

Movimientos de un analizador sintáctico LL(1) utilizando la tabla 4.9

Pila de análisis sintáctico	Entrada	Acción
$ *E' T'*) *E' T*	*)$	explorar (error)
$ *E' T'*) *E' T*	)$	extraer (error)
$ *E' T'*) *E'*	)$	*E'* → ε
$ *E' T'*)	)$	concordar
$ *E' T'*	$	*T'* → ε
$ *E'*	$	*E'* → ε
$	$	aceptar

4.5.3 Recuperación de errores en el analizador sintáctico de TINY

El manejo de errores del analizador sintáctico de TINY, como se da en el apéndice B, es muy rudimentario: sólo está implementada una forma muy primitiva de recuperación en modo de alarma, sin los conjuntos de sincronización. El procedimiento `match` simplemente declara error, estableciendo cuál token que no esperaba encontró. Además, los procedimientos `statement` y `factor` declaran error cuando no se encuentra una selección correcta. El procedimiento `parse` también declara error si se encuentra un token distinto al fin de archivo después de que termina el análisis sintáctico. El principal mensaje de error que se genera

es "unexpected token" ("token inesperado"), el cual puede no ser muy útil para el usuario. Además, el analizador sintáctico no hace un intento por evitar cascadas de error. Por ejemplo, el programa de muestra con un signo de punto y coma agregado después de la sentencia write

```
...
 5: read x ;
 6: if 0 < x then
 7:   fact := 1;
 8:   repeat
 9:     fact := fact * x;
10:     x := x - 1
11:   until x = 0;
12:   write fact; {<-- ¡SIGNO DE PUNTO Y COMA ERRÓNEO! }
13: end
14:
```

provoca que se generen los siguientes *dos* mensajes de error (cuando únicamente ha ocurrido un error):

```
Syntax error at line 13: unexpected token -> reserved word: end
Syntax error at line 14: unexpected token -> EOF
```

Y el mismo programa con la comparación < eliminada en la segunda línea de código

```
...
 5: read x ;
 6: if 0 x then { <-- ¡SIGNO DE COMPARACIÓN PERDIDO AQUÍ! }
 7:   fact := 1;
 8:   repeat
 9:     fact := fact * x;
10:     x := x - 1
11:   until x = 0;
12:   write fact
13: end
14:
```

ocasiona que se impriman *cuatro* mensajes de error en el listado:

```
Syntax error at line 6: unexpected token -> ID, name = x
Syntax error at line 6: unexpected token -> reserved word: then
Syntax error at line 6: unexpected token -> reserved word: then
Syntax error at line 7: unexpected token -> ID, name = fact
```

Por otra parte, algo del comportamiento del analizador sintáctico de TINY es razonable. Por ejemplo, un signo de punto y coma perdido (más que uno sobrante) generará sólo un mensaje de error, y el analizador sintáctico seguirá construyendo el árbol sintáctico correcto como si el signo de punto y coma hubiera estado allí desde el principio, con lo que está realizando una forma rudimentaria de corrección de error en este único caso. Este comportamiento resulta de dos hechos de la codificación. El primero es que el procedimiento `match` no consume un token, lo que da como resultado un comportamiento que es idéntico al de insertar

un token perdido. El segundo es que el procedimiento **stmt_sequence** se escribió de manera que conecte tanto del árbol sintáctico como sea posible en el caso de un error. En particular, se debe tener cuidado para asegurar que los apuntadores hermanos estén conectados dondequiera que se encuentre un apuntador no nulo (los procedimientos del analizador sintáctico están diseñados para devolver un apuntador de árbol sintáctico nulo si se encuentra un error). También, la manera obvia de escribir el cuerpo de **stmt_sequence** basado en el EBNF

```
statement();
while (token==SEMI)
{ match(SEMI);
  statement();
}
```

se puede escribir con una prueba de bucle más complicada:

```
statement();
while ((token!=ENDFILE) && (token!=END) &&
        (token!=ELSE) && (token!=UNTIL))
{ match(SEMI);
  statement();
}
```

El lector puede advertir que los cuatro tokens en esta prueba negativa comprenden el conjunto Siguiente para *stmt-sequence* (*secuencia-sent*). Esto no es un accidente, ya que una prueba puede buscar un token en el conjunto Primero [como lo hacen los procedimientos para *statement* (*sentencia*) y *factor*], o bien, buscar un token que *no* esté en el conjunto Siguiente. Esto último es particularmente efectivo en la recuperación de errores, puesto que si se pierde un símbolo Primero, el análisis sintáctico se detendría. Dejamos para los ejercicios un esbozo del comportamiento del programa para mostrar que un signo de punto y coma perdido en realidad provocaría que el resto del programa se omitiera si **stmt_sequence** se escribiera en la primera forma dada.

Finalmente, advertimos que el analizador sintáctico también se escribió de una manera tal que no puede caer en un bucle infinito cuando encuentra errores (el lector debería haberse preocupado por esto cuando advirtió que **match** no consume un token no esperado). Esto se debe a que, en una ruta arbitraria a través de los procedimientos de análisis sintáctico, con el tiempo se debe encontrar el caso predeterminado de **statement** o **factor**, y ambos *consumen* un token mientras generan un mensaje de error.

EJERCICIOS

4.1 Escriba el pseudocódigo para *term* y *factor* correspondiente al pseudocódigo para *exp* de la sección 4.1.2 (página 150) que construye un árbol sintáctico para expresiones aritméticas simples.

4.2 Dada la gramática $A \rightarrow (A)\ A \mid \varepsilon$ escriba el pseudocódigo para el análisis sintáctico de esta gramática mediante el método descendente recursivo.

4.3 Dada la gramática

sentencia → *sent-assign* | *sent-call* | `otro`
sent-assign → `identificador` := *exp*
sent-call → `identificador` (*exp-list*)

escriba el pseudocódigo para analizar sintácticamente esta gramática mediante el método descendente recursivo.

4.4 Dada la gramática

$$
\begin{aligned}
&\textit{lexp} \rightarrow \texttt{número} \mid \texttt{(}\ \textit{op lexp-seq}\ \texttt{)}\\
&\textit{op} \rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*}\\
&\textit{lexp-seq} \rightarrow \textit{lexp-seq lexp} \mid \textit{lexp}
\end{aligned}
$$

escriba el pseudocódigo para calcular el valor numérico de una *lexp* mediante el método descendente recursivo (véase el ejercicio 3.13 del capítulo 3).

4.5 Muestre las acciones de un analizador sintáctico LL(1) que utiliza la tabla 4.4 (página 163) para reconocer las siguientes expresiones aritméticas:

a. `3+4*5-6` **b.** `3*(4-5+6)` **c.** `3-(4+5*6)`

4.6 Muestre las acciones de un analizador sintáctico LL(1) que utiliza la tabla de la sección 4.2.2 (página 155) para reconocer las siguientes cadenas de paréntesis balanceados:

a. `(())()` **b.** `(()())` **c.** `()(())`

4.7 Dada la gramática $A \rightarrow (\ A\ )\ A \mid \varepsilon$,

a. Construya los conjuntos Primero y Siguiente para el no terminal A.

b. Muestre que esta gramática es LL(1).

4.8 Considere la gramática

$$
\begin{aligned}
&\textit{lexp} \rightarrow \textit{atom} \mid \textit{list}\\
&\textit{atom} \rightarrow \texttt{número} \mid \texttt{identificador}\\
&\textit{list} \rightarrow \texttt{(}\ \textit{lexp-seq}\ \texttt{)}\\
&\textit{lexp-seq} \rightarrow \textit{lexp-seq lexp} \mid \textit{lexp}
\end{aligned}
$$

a. Elimine la recursión por la izquierda.

b. Construya los conjuntos Primero y Siguiente para los no terminales de la gramática resultante.

c. Muestre que la gramática resultante es LL(1).

d. Construya la tabla de análisis sintáctico LL(1) para la gramática resultante.

e. Muestre las acciones del analizador sintáctico LL(1) correspondiente, dada la cadena de entrada `(a (b (2)) (c))`.

4.9 Considere la gramática siguiente (similar, pero no idéntica a la gramática del ejercicio 4.8):

$$
\begin{aligned}
&\textit{lexp} \rightarrow \textit{atom} \mid \textit{list}\\
&\textit{atom} \rightarrow \texttt{número} \mid \texttt{identificador}\\
&\textit{list} \rightarrow \texttt{(}\ \textit{lexp-seq}\ \texttt{)}\\
&\textit{lexp-seq} \rightarrow \textit{lexp}\ \texttt{,}\ \textit{lexp-seq} \mid \textit{lexp}
\end{aligned}
$$

a. Factorice por la izquierda esta gramática.

b. Construya los conjuntos Primero y Siguiente para los no terminales de la gramática resultante.

c. Muestre que la gramática resultante es LL(1).

d. Construya la tabla de análisis sintáctico LL(1) para la gramática resultante.

e. Muestre las acciones del analizador sintáctico LL(1) correspondiente, dada la cadena de entrada `(a,(b,(2)),(c))`.

4.10 Considere la gramática siguiente de declaraciones simplificadas en C:

declaración → *tipo var-list*
tipo → `int` | `float`
var-list → `identificador`, *var-list* | `identificador`

a. Factorice por la izquierda esta gramática.
b. Construya los conjuntos Primero y Siguiente para los no terminales de la gramática resultante.
c. Muestre que la gramática resultante es LL(1).
d. Construya la tabla de análisis sintáctico LL(1) para la gramática resultante.
e. Muestre las acciones del analizador sintáctico LL(1) correspondiente, dada la cadena de entrada `int x,y,z`.

4.11 Una tabla de análisis sintáctico LL(1), tal como la de la tabla 4.4 (página 163), generalmente tiene muchas entradas en blanco que representan errores. En muchos casos todas las entradas en blanco en un renglón se pueden reemplazar por una sola **entrada predeterminada**, con lo que se disminuye el tamaño de la tabla de manera considerable. Se presentan posibles entradas predeterminadas en renglones de no terminal cuando un no terminal tiene una opción de producción simple o cuando tiene una producción ε. Aplique estas ideas a la tabla 4.4. ¿Cuáles son las desventajas de un esquema de esta naturaleza, si es que existen?

4.12 **a.** ¿Puede ser ambigua una gramática LL(1)? Justifique su respuesta.
b. ¿Puede una gramática ambigua ser LL(1)? Justifique su respuesta.
c. ¿Una gramática ambigua debe ser LL(1)? Justifique su respuesta.

4.13 Muestre que una gramática recursiva por la izquierda no puede ser LL(1).

4.14 Demuestre el teorema de la página 178 que concluye que una gramática es LL(1) de dos condiciones en sus conjuntos Primero y Siguiente.

4.15 Defina el operador $\oplus$ sobre dos conjuntos de cadenas de token S_1 y S_2 como se describe a continuación: $S_1 \oplus S_2 = \{\text{Primero}(xy) \mid x \in S_1, y \in S_2\}$.
a. Muestre que, para cualesquiera dos no terminales A y B, Primero(AB) = Primero(A) $\oplus$ Primero(B).
b. Muestre que las dos condiciones del teorema en la página 178 se pueden reemplazar por la condición simple: si $A \rightarrow \alpha$ y $A \rightarrow \beta$, entonces (Primero(α) $\oplus$ Siguiente(A)) $\cap$ (Primero(β) $\oplus$ Siguiente(A)) es vacío.

4.16 Un no terminal A es **inútil** si no hay derivación del símbolo inicial para una cadena de tokens en la que aparezca A.
a. Proporcione una formulación matemática de esta propiedad.
b. ¿Es probable que una gramática de lenguaje de programación tenga un símbolo inútil? Explique por qué.
c. Muestre que, si una gramática tiene un símbolo inútil, el cálculo de los conjuntos Primero y Siguiente como se dio en este capítulo puede producir conjuntos que sean demasiado grandes para construir con precisión una tabla de análisis sintáctico LL(1).

4.17 Proporcione un algoritmo que elimine no terminales inútiles (y producciones asociadas) de una gramática sin cambiar el lenguaje reconocido. (Véase el ejercicio anterior.)

4.18 Muestre que si una gramática carece de no terminales inútiles, lo opuesto al teorema de la página 178 es verdadero (véase el ejercicio 4.16).

4.19 Dé los detalles del cálculo de los conjuntos Primero y Siguiente que se exhiben en el ejemplo 4.15 (página 178).

4.20 **a.** Construya los conjuntos Primero y Siguiente para los no terminales de la gramática factorizada por la izquierda del ejemplo 4.7 (página 165).
b. Construya la tabla de análisis sintáctico LL(1) utilizando los resultados que obtuvo en el inciso a.

4.21 Dada la gramática $A \rightarrow a\ A\ a \mid \varepsilon$,

a. Muestre que esta gramática no es LL(1).

b. Un intento de escribir un analizador sintáctico descendente recursivo para esta gramática está representado por el pseudocódigo siguiente.

```
procedure A ;
begin
  if token = a then
     getToken ;
     A ;
     if token = a then getToken ;
     else error ;
  else if token <> $ then error ;
end A ;
```

Muestre que este procedimiento no funcionará correctamente.

c. Se *puede* escribir un analizador sintáctico descendente recursivo **en reversa** para este lenguaje, pero se necesita usar un procedimiento *unGetToken* que tome un token como parámetro y devuelva ese token al frente del flujo de tokens de entrada. También es necesario que el procedimiento A se escriba como una función booleana que devuelva un éxito o un fracaso, de manera que cuando A se llame a sí misma, pueda probar si hubo un éxito antes de consumir otro token, y de modo que, si la selección $A \rightarrow a\ A\ a$ fracasa, el código pueda continuar para intentar la alternativa $A \rightarrow \varepsilon$. Vuelva a escribir el pseudocódigo del inciso b, según esta fórmula, y explore su operación en la cadena *aaaa*$.

4.22 En la gramática TINY de la figura 4.9 (página 181) no se hace una distinción clara entre expresiones booleanas y aritméticas. Por ejemplo, a continuación tenemos un programa TINY sintácticamente legal:

```
if 0 then write 1>0 else x := (x<1)+1 end
```

Vuelva a escribir la gramática TINY de manera que sólo se permitan expresiones booleanas (expresiones que contengan operadores de comparación) como la prueba de una sentencia if o repeat, y sólo se permitan expresiones aritméticas en sentencias write o assign o como operandos de cualquiera de los operadores.

4.23 Agregue los operadores booleanos **and**, **or** y **not** a la gramática TINY de la figura 4.9 (página 181). Asígneles las propiedades descritas en el ejercicio 3.5 (página 139), así como también una precedencia más baja que la de todos los operadores aritméticos. Asegúrese de que cualquier expresión pueda ser booleana o entera.

4.24 Los cambios a la sintaxis de TINY en el ejercicio 4.23 empeoraron el problema que se describió en el ejercicio 4.22. Vuelva a escribir la respuesta al ejercicio 4.23 para distinguir rigurosamente entre expresiones booleanas y aritméticas, e incorpore esto a la solución del ejercicio 4.22.

4.25 La recuperación de errores en modo de alarma para la gramática de expresión aritmética simple, como se describió en la sección 4.5.1, todavía tiene algunas desventajas. Una es que los ciclos o bucles while que prueban operadores deberían continuar funcionando en ciertas circunstancias. Por ejemplo, la expresión `(2) (3)` perdió su operador entre los factores, pero el manejador de errores consume el segundo factor sin reiniciar el análisis sintáctico. Vuelva a escribir el pseudocódigo para mejorar su comportamiento en este caso.

4.26 Explore las acciones de un analizador sintáctico LL(1) en la entrada `(2+-3)*4-+5`, utilizando la recuperación de errores dada en la tabla 4.9 (página 187).

4.27 Vuelva a escribir el algoritmo de análisis sintáctico LL(1) de la figura 4.2 (página 155) para implementar la recuperación de errores en modo de alarma completo, y examine su comportamiento en la entrada `(2+-3)*4-+5`.

4.28 **a.** Explore la operación del procedimiento `stmt_sequence` en el analizador sintáctico TINY para verificar que se construya el árbol sintáctico correcto para el siguiente programa en TINY, a pesar del signo de punto y coma perdido:

```
x := 2
y := x + 2
```

b. ¿Qué árbol sintáctico se construye para el siguiente programa (incorrecto)?:

```
x 2
y := x + 2
```

c. Suponga que en su lugar se escribió el procedimiento `stmt_sequence` utilizando el código más simple esbozado en la página 189:

```
statement();
while (token==SEMI)
{ match(SEMI);
  statement();
}
```

¿Qué árboles sintácticos se construyen para los programas en los incisos a y b utilizando esta versión del procedimiento `stmt_sequence`?

EJERCICIOS DE PROGRAMACIÓN

4.29 Agregue lo siguiente a la calculadora simple descendente recursiva de aritmética entera de la figura 4.1, páginas 148-149 (asegúrese de que tengan precedencia y asociatividad correctas):

a. División entera con el símbolo `/`.

b. Módulo entero con el símbolo `%`.

c. Exponenciación entera con el símbolo `^`. (Advertencia: este operador tiene mayor precedencia que la multiplicación y es asociativo por la derecha.)

d. Menos unitario con el símbolo `-`. (Véase el ejercicio 3.12, página 140.)

4.30 Vuelva a escribir la calculadora descendente recursiva de la figura 4.1 (páginas 148-149) de manera que calcule con números de punto flotante en lugar de enteros.

4.31 Vuelva a escribir la calculadora descendente recursiva de la figura 4.1 de manera que *distinga* entre valores de punto flotante y valores enteros, en vez de simplemente calcular todo como números enteros o de punto flotante. (Sugerencia: un "valor" es ahora un registro con una marca que indica si es entero o de punto flotante.)

4.32 **a.** Vuelva a escribir la calculadora descendente recursiva de la figura 4.1 de manera que devuelva un árbol sintáctico de acuerdo con las declaraciones de la sección 3.3.2 (página 111).

b. Escriba una función que tome como parámetro el árbol sintáctico producido por su código del inciso a y devuelva el valor calculado al recorrer el árbol.

4.33 Escriba una calculadora descendente recursiva para aritmética entera simple semejante al de la figura 4.1, pero utilice la gramática de la figura 4.4 (página 160).

4.34 Considere la gramática siguiente:

$$lexp \rightarrow \texttt{número} \mid \texttt{(}\ op\ lexp\text{-}seq\ \texttt{)}$$
$$op \rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*}$$
$$lexp\text{-}seq \rightarrow lexp\text{-}seq\ lexp \mid lexp$$

Esta gramática puede imaginarse como representación de expresiones aritméticas enteras simples en forma prefija tipo LISP. Por ejemplo, la expresión `34-3*42` se escribiría en esta gramática como `(- 34 (* 3 42))`.

Escriba una calculadora descendente recursiva para las expresiones dadas mediante esta gramática.

4.35 **a.** Invente una estructura de árbol sintáctico para la gramática del ejercicio anterior y escriba un analizador sintáctico descendente recursivo para la misma que devuelva un árbol sintáctico.

b. Escriba una función que tome como parámetro el árbol sintáctico producido por su código del inciso a y devuelva el valor calculado al recorrer el árbol.

4.36 **a.** Utilice la gramática para expresiones regulares del ejercicio 3.4, página 138 (con una eliminación apropiada de ambigüedades) para construir un analizador sintáctico descendente recursivo que lea una expresión regular y realice la construcción de Thompson de un NFA (véase el capítulo 2).

b. Escriba un procedimiento que tome una estructura de datos NFA producida por el analizador sintáctico del inciso a y construya un DFA equivalente según la construcción de subconjunto.

c. Escriba un procedimiento que tome la estructura de datos producida mediante su procedimiento del inciso b y encuentre concordancias de subcadenas más largas en un archivo de texto según el DFA que representa. (¡Su programa ahora se ha convertido en una versión "compilada" de grep!)

4.37 Agregue los operadores de comparación `<=` (menor o igual que), `>` (mayor que), `>=` (mayor o igual que) y `<>` (distinto de) al analizador sintáctico de TINY. (Esto requerirá agregar estos tokens y modificar también el analizador léxico, pero no debería requerir un cambio en el árbol sintáctico.)

4.38 Incorpore sus cambios gramaticales del ejercicio 4.22 al analizador sintáctico de TINY.

4.39 Incorpore sus cambios gramaticales del ejercicio 4.23 al analizador sintáctico de TINY.

4.40 **a.** Vuelva a escribir el programa de la calculadora descendente recursiva de la figura 4.1 (páginas 148-149) para implementar la recuperación de errores en modo de alarma como se esquematizó en la sección 4.5.1.

b. Agregue mensajes de error útiles a su manejador de errores del inciso a.

4.41 Una de las razones por las que el analizador sintáctico de TINY produce mensajes de error deficientes es que el procedimiento **`match`** está limitado a imprimir sólo el token actual cuando se presenta un error, en vez de imprimir tanto el token actual como el token esperado, y que ningún mensaje de error especial se pasa al procedimiento **`match`** desde sus ubicaciones de llamada. Vuelva a escribir el procedimiento **`match`** de manera que imprima el token esperado además del token actual y que también pase un mensaje de error al procedimiento **`syntaxError`**. Esto requerirá volver a escribir el procedimiento **`syntaxError`**, así como los cambios a las llamadas de **`match`** para incluir un mensaje de error apropiado.

4.42 Agregue conjuntos de sincronización de tokens y recuperación de errores en modo de alarma al analizador sintáctico de TINY, de la manera que se describió en la página 184.

4.43 (De Wirth [1976]) Las estructuras de datos basadas en diagramas sintácticos pueden ser utilizadas por un analizador sintáctico descendente recursivo "genérico" que analizará sintácticamente

cualquier conjunto de reglas gramaticales LL(1). Una estructura de datos adecuada está dada por las siguientes declaraciones en C:

```
typedef struct rulerec
      { struct rulerec *next,*other;
        int isToken;
        union
        { Token name;
          struct rulerec *rule;
        } attr;
      } Rulerec;
```

El campo **next** (**"siguiente"**) se utiliza para apuntar al siguiente elemento en la regla gramatical, mientras que el campo **other** (**"otro"**) se emplea para apuntar a las alternativas dadas por el metasímbolo |. De este modo, la estructura de datos para la regla gramatical

$$factor \rightarrow (\ exp\ ) \mid \textbf{número}$$

tendría el aspecto siguiente

donde los campos de la estructura de registro se muestran de la manera siguiente

isToken	nombre/regla
otro	siguiente

a. Dibuje las estructuras de datos para el resto de las reglas gramaticales de la figura 4.4, página 160. (*Sugerencia*: necesitará un token especial para representar internamente ε en la estructura de datos.)

b. Escriba un procedimiento de análisis sintáctico genérico que utilice estas estructuras de datos para reconocer una cadena de entrada.

c. Escriba un generador de analizador sintáctico que lea reglas BNF (ya sea desde un archivo o desde la entrada estándar) y genere las estructuras de datos precedentes.

NOTAS Y REFERENCIAS

El análisis sintáctico descendente recursivo ha sido un método estándar para la construcción de analizadores sintácticos desde principios de la década de los sesenta y la introducción de reglas BNF en el informe de Algol60 [Naur, 1963]. Para una descripción inicial del método, véase Hoare [1962]. Los analizadores sintácticos descendentes recursivos en reversa se han vuelto populares en lenguajes funcionales lentos fuertemente tipificados, tales como Haskell y Miranda, donde esta forma de análisis sintáctico descendente recursivo se denomina análisis sintáctico de combinación. Para una descripción de este método véase a Peyton Jones y Lester [1992] o Hutton [1992]. El uso de EBNF junto con el análisis sintáctico descendente recursivo ha sido popularizado por Wirth [1976].

El análisis sintáctico LL(1) fue ampliamente estudiado en la década de los sesenta y principios de los setenta. Para una descripción inicial véase Lewis y Stearns [1968]. Una investigación sobre el análisis sintáctico LL(k) se puede encontrar en Fischer y LeBlanc [1991], donde puede hallarse un ejemplo de una gramática LL(2) que no es SLL(2). Para usos prácticos del análisis sintáctico LL(k), véase Parr, Dietz y Cohen [1992].

Existen, por supuesto, muchos métodos más de análisis sintáctico descendente aparte de los dos que acabamos de estudiar en este capítulo. Para un ejemplo de otro método más general véase Graham, Harrison y Ruzzo [1980].

La recuperación de errores en modo de alarma se estudia en Wirth [1976] y Stirling [1985]. La recuperación de errores en analizadores sintácticos LL(k) se estudia en Burke y Fischer [1987]. Métodos de reparación de errores más sofisticados se estudian en Fischer y LeBlanc [1991] y Lyon [1974].

En este capítulo no se analizaron herramientas automáticas para la generación de analizadores sintácticos descendentes, principalmente porque la herramienta más ampliamente utilizada, Yacc, se estudiará en el siguiente capítulo. Sin embargo, existen buenos generadores de analizadores sintácticos descendentes. Uno de ellos se llama Antlr y es parte del conjunto de herramientas de construcción de compiladores de Purdue (PCCTS, Purdue Compiler Construction Tool Set). Véase una descripción en Parr, Dietz y Cohen [1992]. Antlr genera un analizador sintáctico descendente recursivo desde una descripción EBNF. Tiene diversas características útiles, incluyendo un mecanismo integrado para construir árboles sintácticos. Para una revisión de un generador de analizador sintáctico LL(1) denominado LLGen, véase Fischer y LeBlanc [1991].

Capítulo 5

Análisis sintáctico ascendente

5.1 Perspectiva general del análisis sintáctico ascendente

5.2 Autómatas finitos de elementos LR(0) y análisis sintáctico LR(0)

5.3 Análisis sintáctico SLR(1)

5.4 Análisis sintáctico LALR(1) y LR(1) general

5.5 Yacc: un generador de analizadores sintácticos LALR(1)

5.6 Generación de un analizador sintáctico TINY utilizando Yacc

5.7 Recuperación de errores en analizadores sintácticos ascendentes

En el capítulo anterior contemplamos los algoritmos de análisis sintáctico descendente básicos del análisis sintáctico predictivo y descendente recursivo. En este capítulo describiremos las principales técnicas de análisis sintáctico ascendente y las construcciones que se asocian con ellas. Como con el análisis sintáctico descendente, nos limitaremos a estudiar aquellos algoritmos de análisis sintáctico que utilizan como máximo un símbolo de búsqueda hacia adelante, pero incluiremos algunos comentarios sobre cómo extender los algoritmos.

Con una terminología similar a la de los analizadores sintácticos LL(1), el algoritmo ascendente más general se denomina **análisis sintáctico LR(1)** (la L indica que la entrada se procesa de izquierda a derecha, "Left-to-right", la R indica que se produce una derivación por la derecha, "Rightmost derivation", mientras que el número 1 indica que se utiliza un símbolo de búsqueda hacia adelante). Una consecuencia de la potencia del análisis sintáctico ascendente es el hecho de que también es significativo hablar de **análisis sintáctico LR(0)**, donde no se consulta *ninguna* búsqueda hacia adelante al tomar decisiones de análisis sintáctico. (Esto es posible porque un token de búsqueda hacia adelante se puede examinar *después* de que aparece en la pila del análisis sintáctico, y si esto ocurre no cuenta como búsqueda hacia adelante.) Una mejora en el análisis sintáctico LR(0) que hace algún uso de la búsqueda hacia adelante se conoce como **análisis sintáctico SLR(1)** (por las siglas en inglés de análisis sintáctico LR(1) *simple*). Un método que es un poco más potente que el análisis sintáctico SLR(1) pero menos complejo que el análisis sintáctico LR(1) general se denomina **análisis sintáctico LALR(1)** (por las siglas de *"lookahead* LR(1) parsing", o sea, "análisis sintáctico LR(1) de *búsqueda hacia adelante"*).

Abordaremos las construcciones necesarias para cada uno de estos métodos de análisis sintáctico en las secciones siguientes. Esto incluirá la construcción de los DFA de elementos LR(0) y LR(1), descripciones de los algoritmos de análisis sintáctico SLR(1), LR(1) y LALR(1), así como la construcción de las tablas de análisis sintáctico que se asocian con ellos. También describiremos el uso de Yacc, un generador de analizador sintáctico LALR(1) y con él generaremos un analizador sintáctico para el lenguaje TINY que cons-

truye los mismos árboles sintácticos que el analizador sintáctico descendente recursivo que desarrollamos en el capítulo anterior.

Los algoritmos de análisis sintácticos ascendentes son en general más poderosos que los métodos descendentes. (Por ejemplo, la recursión por la izquierda no es un problema en el análisis sintáctico ascendente.) Como es de esperarse, las construcciones involucradas en estos algoritmos también son más complejas. Por consiguiente, tendremos que describirlas con cuidado y presentarlas por medio de ejemplos de gramáticas muy simples. Al principio del capítulo daremos dos ejemplos de esta clase, los cuales utilizaremos a través del resto del capítulo. También continuaremos con el uso de algunos de los ejemplos recurrentes del capítulo anterior (expresiones aritméticas enteras, sentencias if, etc.). Sin embargo, no realizaremos a mano *ninguno* de los algoritmos de análisis sintáctico ascendente para el lenguaje TINY completo, ya que esto sería demasiado complejo. De hecho, todos los métodos ascendentes importantes son realmente muy complejos para hacer a mano la codificación, pero son muy adecuados para generadores de analizadores sintácticos como Yacc. Sin embargo, es importante comprender el funcionamiento de los métodos, de manera que el escritor del compilador pueda analizar apropiadamente el comportamiento de un generador de analizadores sintácticos. El diseñador de un lenguaje de programación también puede beneficiarse de esta información, puesto que un generador de analizadores sintácticos puede identificar problemas potenciales con una sintaxis de lenguaje propuesta en BNF.

Los temas vistos con anterioridad que serán necesarios para comprender el funcionamiento de los algoritmos de análisis sintáctico ascendente incluyen las propiedades de los autómatas finitos y la construcción del subconjunto de un DFA a partir de un NFA (capítulo 2, secciones 2.3 y 2.4), así como las propiedades generales de las gramáticas libres de contexto, derivaciones y árboles de análisis gramatical (capítulo 3, secciones 3.2 y 3.3). Ocasionalmente también serán necesarios los conjuntos Siguiente (capítulo 4, sección 4.3). Comenzamos este capítulo con una perspectiva general del análisis sintáctico ascendente.

5.1 PERSPECTIVA GENERAL DEL ANÁLISIS SINTÁCTICO ASCENDENTE

Un analizador sintáctico ascendente utiliza una pila explícita para realizar un análisis sintáctico, de manera semejante a como lo hace un analizador sintáctico descendente no recursivo. La pila de análisis sintáctico contendrá tanto tokens como no terminales, y también alguna información de estado adicional que analizaremos posteriormente. La pila está vacía al principio de un análisis sintáctico ascendente y al final de un análisis sintáctico exitoso contendrá el símbolo inicial. Un esquema para el análisis sintáctico ascendente es, por lo tanto,

$	*CadenaEntrada* $		
...	...		
...	...		
$ *SímboloInicial*		$	aceptar

donde la pila de análisis sintáctico está a la izquierda, la entrada está en el centro y las acciones del analizador sintáctico están a la derecha (en este caso, "aceptar" es la única acción indicada).

Un analizador sintáctico ascendente tiene dos posibles acciones (aparte de "aceptar"):

1. **Desplazar** o transferir un terminal de la parte frontal de la entrada hasta la parte superior de la pila.
2. **Reducir** una cadena α en la parte superior de la pila a un no terminal A, dada la selección BNF $A \rightarrow \alpha$.

Por esta razón en ocasiones un analizador sintáctico ascendente se conoce como analizador sintáctico de **reducción por desplazamiento ("shift-reduce")**.[1] Las acciones de desplazamiento se indican al escribir la palabra *shift* o *desplazamiento*. Las acciones de reducción se indican al escribir la palabra *reduce* o *reducción* y proporcionar la selección BNF utilizada en la reducción.[2] Otra característica de los analizadores sintácticos ascendentes es que, por razones técnicas que se comentarán más adelante, las gramáticas siempre se **aumentan** con un nuevo símbolo inicial. Esto significa que si S es el símbolo inicial, se agrega a la gramática un nuevo símbolo inicial S', con una producción simple correspondiente al símbolo inicial anterior:

$$S' \rightarrow S$$

Continuaremos de inmediato con dos ejemplos, y después comentaremos algunas de las propiedades del análisis sintáctico ascendente que se muestran mediante estos ejemplos.

Ejemplo 5.1

Considere la siguiente gramática aumentada para paréntesis balanceados:

$$S' \rightarrow S$$
$$S \rightarrow (\ S\ )\ \ S \mid \varepsilon$$

Un análisis sintáctico ascendente de la cadena ( ) en el que se utiliza esta gramática se proporciona en la tabla 5.1.

Tabla 5.1
Acciones de análisis sintáctico de un analizador sintáctico ascendente para la gramática del ejemplo 5.1

	Pila de análisis sintáctico	Entrada	Acción
1	$	( ) $	desplazamiento
2	$ (	) $	reducción $S \rightarrow \varepsilon$
3	$ (S	) $	desplazamiento
4	$ (S)	$	reducción $S \rightarrow \varepsilon$
5	$ (S) S	$	reducción $S \rightarrow (\ S\ )\ S$
6	$ S	$	reducción $S' \rightarrow S$
7	$ S'	$	aceptar

§

Ejemplo 5.2

Considere la siguiente gramática aumentada para expresiones aritméticas rudimentarias (sin paréntesis y con una operación):

$$E' \rightarrow E$$
$$E \rightarrow E + \mathbf{n} \mid \mathbf{n}$$

Un análisis sintáctico ascendente de la cadena $\mathbf{n} + \mathbf{n}$ en el que se utiliza esta gramática se proporciona en la tabla 5.2.

1. Por la misma razón, los analizadores sintácticos descendentes se podrían llamar analizadores de generación por concordancia, pero esto no se acostumbra.

2. En el caso de una reducción, podríamos escribir simplemente la selección BNF por sí misma, como hicimos para una acción de *generación* en el análisis sintáctico descendente, pero se acostumbra agregar la *reducción*.

Tabla 5.2
Acciones de análisis sintáctico de un analizador sintáctico ascendente para la gramática del ejemplo 5.2

	Pila de análisis sintáctico	Entrada	Acción
1	$	n + n $	desplazamiento
2	$ n	+ n $	reducción $E \rightarrow n$
3	$ E	+ n $	desplazamiento
4	$ E +	n $	desplazamiento
5	$ E + n	$	reducción $E \rightarrow E + n$
6	$ E	$	reducción $E' \rightarrow E$
7	$ E'	$	aceptar

§

Los analizadores sintácticos ascendentes tienen menos dificultades que los descendentes con la búsqueda hacia adelante. De hecho, un analizador ascendente puede transferir símbolos de entrada en la pila hasta que determine qué acción realizar (suponiendo que se puede determinar una acción que no requiera los símbolos para ser transferida de regreso a la entrada). Sin embargo, un analizador sintáctico ascendente puede necesitar examinar más allá de la parte superior de la pila para determinar qué acción realizar. Por ejemplo, en la tabla 5.1 la línea 5 tiene S en la parte superior de la pila, y el analizador sintáctico realiza una reducción mediante la producción $S \rightarrow (S) S$, mientras que la línea 6 también tiene S en la parte superior de la pila, pero el analizador sintáctico efectúa una reducción mediante $S' \rightarrow S$. Para poder saber que $S \rightarrow (S) S$ es una reducción válida para el paso 5, debemos saber que la pila en realidad contiene la cadena $(S) S$ en ese punto. De este modo, el análisis sintáctico ascendente requiere de una "pila de búsqueda hacia adelante" arbitraria. Esto no es ni con mucho tan serio como la entrada de búsqueda hacia adelante, puesto que el analizador sintáctico mismo construye la pila y puede colocar la información apropiada para que esté disponible. El mecanismo que efectuará esto es un autómata finito determinístico de "elementos" que se describirá en la siguiente sección.

Como es natural, no basta con hacer el seguimiento del contenido de la pila para poder determinar de manera única el siguiente paso en un análisis sintáctico de reducción por desplazamiento, también puede ser necesario consultar el token siguiente en la entrada como una búsqueda hacia adelante. Por ejemplo, en la línea 3 de la tabla 5.2, E está en la pila y ocurre un desplazamiento, mientras que en la línea 6, E está nuevamente en la pila pero ocurre una reducción mediante $E' \rightarrow E$. La diferencia es que en la línea 3 el token siguiente en la entrada es +, mientras que en la línea 6 el siguiente token de entrada es $. De este modo, cualquier algoritmo que realiza ese análisis sintáctico debe utilizar el siguiente token de entrada (la búsqueda hacia adelante) para determinar la acción apropiada. Diferentes métodos de análisis sintáctico de reducción por desplazamiento utilizan la búsqueda hacia adelante de distintas maneras, y esto da como resultado analizadores sintácticos con una complejidad y potencia variadas. Antes de examinar los algoritmos individuales haremos algunas observaciones generales acerca de cómo pueden caracterizarse las etapas intermedias en un análisis sintáctico ascendente.

En primer lugar, observemos de nuevo que un analizador sintáctico de reducción por desplazamiento sigue una derivación por la derecha de la cadena de entrada, pero los pasos de la derivación se presentan en orden inverso. En la tabla 5.1 tenemos cuatro reducciones, correspondientes en modo inverso a los cuatro pasos de la derivación por la derecha:

$$S' \Rightarrow S \Rightarrow (S) S \Rightarrow (S) \Rightarrow ( )$$

En la tabla 5.2 la derivación correspondiente es

$$E' \Rightarrow E \Rightarrow E + n \Rightarrow n + n$$

Cada una de las cadenas intermedias de terminales y no terminales en una derivación así se conoce como **forma sentencial derecha**. Cada una de esas formas sentenciales se divide entre la entrada y la pila de análisis sintáctico durante un análisis sintáctico de reducción por desplazamiento. Por ejemplo, la forma sentencial derecha $E + n$, que se presenta

en el tercer paso de la derivación anterior, aparece en los pasos 3, 4 y 5 de la tabla 5.2. Si indicamos dónde está la parte superior de la pila en cada momento mediante el símbolo ‖ (es decir, dónde se presenta la división entre la pila y la entrada), entonces el paso 3 de la tabla 5.2 lo indica la expresión $E \| + n$ y el paso 4 la expresión $E + \| n$. En cada caso la secuencia de símbolos en la pila de análisis sintáctico se llama **prefijo viable** de la forma sentencial derecha. Por consiguiente, E, $E +$ y $E + n$ son todos prefijos viables de la forma sentencial derecha $E + n$, mientras que la forma sentencial derecha $n + n$ tiene ε y n como sus prefijos viables (pasos 1 y 2 de la tabla 5.2). Advierta que $n +$ no es un prefijo viable de $n + n$.

Un analizador sintáctico de reducción por desplazamiento transferirá terminales de la entrada a la pila hasta que sea posible realizar una reducción para obtener la siguiente forma sentencial derecha. Esto ocurrirá cuando la cadena de símbolos en la parte superior de la pila concuerde con el lado derecho de la producción que se utiliza en la reducción siguiente. Esta cadena, junto con la posición en la forma sentencial derecha donde se presenta, y la producción utilizada para reducirla, se denomina **controlador (handle)** de la forma sentencial derecha.[3] Por ejemplo, en la forma sentencial derecha $n + n$, el controlador es la cadena consistente de un solo token n que se encuentra más a la izquierda junto con la producción $E \rightarrow n$ que se utiliza para reducirla a fin de producir la nueva forma sentencial derecha $E + n$. El manejo de esta nueva forma sentencial es la cadena entera $E + n$ (un prefijo viable), junto con la producción $E \rightarrow E + n$. En ocasiones, por abuso de notación, nos referiremos a la cadena en sí como el controlador.

La determinación del controlador siguiente en un análisis sintáctico es la tarea principal de un analizador sintáctico de reducción por desplazamiento. Advierta que la cadena de un controlador siempre forma un lado derecho completo para su producción (la producción utilizada en la siguiente reducción), y que la posición de extremo derecho de la cadena del controlador corresponderá a la parte superior de la pila cuando se produzca la reducción. De este modo, parece plausible que un analizador sintáctico de reducción por desplazamiento querrá determinar sus acciones basado en las posiciones en los lados derechos de las producciones. Cuando estas posiciones alcanzan el extremo derecho de una producción, entonces esta misma producción es candidata para una reducción, y es posible que el controlador esté en la parte superior de la pila. Sin embargo, para ser el controlador, a la cadena de la parte superior de la pila no le es suficiente concordar con el lado derecho de una producción. En realidad, si una producción ε está disponible para reducción, como en el ejemplo 5.1, entonces su lado derecho (la cadena vacía) siempre está en la parte superior de la pila. Las reducciones se presentan sólo cuando la cadena resultante es realmente una forma sentencial derecha. Por ejemplo, en el paso 3 de la tabla 5.1 podría realizarse una reducción por medio de $S \rightarrow \varepsilon$, pero la cadena es realmente una forma sentencial derecha. Por ejemplo, en el paso 3 de la tabla 5.1 podría realizarse una reducción por medio de $S \rightarrow \varepsilon$, pero la cadena resultante $(S\ S)$ no es una forma sentencial derecha, por lo tanto, ε no es el controlador en esta posición en la forma sentencial (S).

5.2 AUTÓMATAS FINITOS DE ELEMENTOS LR(0) Y ANÁLISIS SINTÁCTICO LR(0)

5.2.1 Elementos LR(0)

Un **elemento LR(0)** (o simplemente **elemento** para abreviar) de una gramática libre de contexto es una regla de producción con una posición distinguida en su lado derecho. Indicaremos esta posición distinguida mediante un punto (el cual, por supuesto, se convierte en un

3. Si la gramática es ambigua, de manera que puede existir más de una derivación, entonces puede haber más de un controlador de una forma sentencial derecha. Si la gramática no es ambigua, entonces los controladores son únicos.

metasímbolo para no confundirlo con un token real). De este modo, si $A \rightarrow \alpha$ es una regla de producción, y si β y γ son dos cadenas cualesquiera de símbolos (incluyendo la cadena vacía ε), tales como $\beta\gamma = \alpha$, entonces $A \rightarrow \beta \,.\, \gamma$ es un elemento LR(0). Éstos se conocen como elementos LR(0), porque no contienen referencia explícita a la búsqueda hacia delante.

Ejemplo 5.3 Considere la gramática del ejemplo 5.1:

$$S' \rightarrow S$$
$$S \rightarrow (\ S\ )\ S \mid \varepsilon$$

Esta gramática tiene tres reglas de producción y ocho elementos:

$$S' \rightarrow .S$$
$$S' \rightarrow S.$$
$$S \rightarrow .(S)S$$
$$S \rightarrow (.S)S$$
$$S \rightarrow (S.)S$$
$$S \rightarrow (S).S$$
$$S \rightarrow (S)S.$$
$$S \rightarrow .$$

§

Ejemplo 5.4 La gramática del ejemplo 5.2 tiene los siguientes ocho elementos:

$$E' \rightarrow .E$$
$$E' \rightarrow E.$$
$$E \rightarrow .E + \mathbf{n}$$
$$E \rightarrow E.+ \mathbf{n}$$
$$E \rightarrow E + .\mathbf{n}$$
$$E \rightarrow E + \mathbf{n}.$$
$$E \rightarrow .\mathbf{n}$$
$$E \rightarrow \mathbf{n}.$$

§

La idea detrás del concepto de un elemento es que un elemento registra un paso intermedio en el reconocimiento del lado derecho de una opción de regla gramatical específica. En particular, el elemento $A \rightarrow \beta \,.\, \gamma$ construido de la regla gramatical $A \rightarrow \alpha$ (con $\alpha = \beta\gamma$) significa que ya se ha visto β y que se pueden derivar los siguientes tokens de entrada de γ. En términos de la pila de análisis sintáctico, esto significa que β deberá aparecer en la parte superior de la pila. Un elemento $A \rightarrow .\alpha$ significa que podemos estar cerca de reconocer una A mediante el uso de la selección de regla gramatical $A \rightarrow \alpha$ (a los elementos de esta clase los denominamos **elementos iniciales**). Un elemento $A \rightarrow \alpha.$ significa que α reside ahora en la parte superior de la pila de análisis sintáctico y puede ser el controlador, si se va a utilizar $A \rightarrow \alpha$ para la siguiente reducción (tales elementos se conocen como **elementos completos**).

5.2.2 Autómatas finitos de elementos

Los elementos LR(0) se pueden emplear como los estados de un autómata finito que mantienen información acerca de la pila de análisis sintáctico y del progreso de un análisis de reducción por desplazamiento. Esto comenzará como un autómata finito no determinístico.

A partir de este NFA de elementos LR(0) podemos construir el DFA de conjuntos de elementos LR(0) utilizando la construcción del subconjunto del capítulo 2. Como veremos, también es fácil construir el DFA de conjuntos de elementos LR(0) de manera directa.

¿Cuáles son las transiciones del NFA de elementos LR(0)? Considere el elemento $A \rightarrow \alpha \,.\, \gamma$ y suponga que γ comienza con el símbolo X, el cual puede ser un token o un no terminal, de manera que el elemento se pueda escribir como $A \rightarrow \alpha \,.\, X\eta$. Entonces existe una transición en el símbolo X desde el estado representado por este elemento hasta el estado representado por el elemento $A \rightarrow \alpha\, X \,.\, \eta$. En forma gráfica esto se escribe como

Si X es un token, entonces esta transición corresponde a un desplazamiento de X desde la entrada hacia la parte superior de la pila durante un análisis sintáctico. Por otra parte, si X es un no terminal, entonces la interpretación de esta transición es más compleja, puesto que X nunca aparecerá como un símbolo de entrada. De hecho, una transición de esta clase todavía corresponderá a la inserción de X en la pila durante un análisis sintáctico, pero esto sólo puede ocurrir durante una reducción en la que intervenga una producción $X \rightarrow \beta$. Ahora, como una reducción de esta clase debe estar precedida por el reconocimiento de una β, y el estado proporcionado por el elemento inicial $X \rightarrow .\beta$ representa el comienzo de este proceso (el punto indica que estamos cerca de reconocer a β), entonces para cada elemento $A \rightarrow \alpha \,.\, X\eta$ debemos agregar una transición ε

en cada regla de producción $X \rightarrow \beta$ de X, indicando que se puede producir X mediante el reconocimiento de cualquiera de los lados derechos de sus producciones.

Estos dos casos representan las únicas transiciones en el NFA de elementos LR(0). Queda por analizar la selección del estado inicial y los estados de aceptación del NFA. El estado inicial del NFA debería corresponder al estado inicial del analizador sintáctico: la pila está vacía, y estamos por reconocer un S, donde S es el símbolo inicial de la gramática. De este modo, cualquier elemento inicial $S \rightarrow .\alpha$ construido a partir de una regla de producción para S podría servir como estado inicial. Desgraciadamente, puede haber muchas reglas de producción de esta clase para S. ¿Cómo podemos decir cuál utilizar? De hecho, no podemos. La solución es **aumentar** la gramática mediante una producción simple $S' \rightarrow S$, donde S' es un nuevo no terminal. Entonces S' se convierte en el estado inicial de la **gramática aumentada**, y el elemento inicial $S' \rightarrow .S$ se convierte en el estado inicial del NFA. Ésta es la razón por la que aumentamos las gramáticas de los ejemplos anteriores.

¿Cuáles estados deberían convertirse en estados de aceptación en este NFA? Aquí debemos recordar que el propósito del NFA es mantener un seguimiento del estado de un análisis sintáctico, no reconocer cadenas directamente, como están diseñados para hacerlo los autómatas del capítulo 2. De este modo, el analizador sintáctico mismo decidirá cuándo aceptar, y el NFA no necesita contener esa información, de manera que el NFA de hecho no tendrá estados de aceptación en absoluto. (El NFA *tendrá* alguna información acerca de la aceptación, pero no en la forma de un estado de aceptación. Comentaremos esto cuando describamos los algoritmos de análisis sintáctico que usa el autómata.)

Esto completa la descripción del NFA de elementos LR(0). Ahora volveremos a las gramáticas simples de los dos ejemplos anteriores y construiremos sus NFA de elementos LR(0).

Ejemplo 5.5

En el ejemplo 5.3 enumeramos los ocho elementos LR(0) de la gramática del ejemplo 5.1. Por lo tanto, el NFA tiene ocho estados, como se muestra en la figura 5.1. Advierta que cada elemento de la figura con un punto antes que el no terminal S tiene una transición ε para todo elemento inicial de S.

Figura 5.1
El NFA de elementos LR(0) para la gramática del ejemplo 5.5

§

Ejemplo 5.6

En el ejemplo 5.4 enumeramos los elementos LR(0) asociados con la gramática del ejemplo 5.2. El NFA de los elementos aparece en la figura 5.2. Advierta que el elemento inicial $E \rightarrow . E + n$ tiene una transición ε hacia sí mismo. (Esta situación se presentará en todas las gramáticas con recursión por la izquierda inmediata.)

Figura 5.2
El NFA de elementos LR(0) para la gramática del ejemplo 5.6

§

Para completar la descripción del uso de elementos en el seguimiento del estado de análisis sintáctico, debemos construir el DFA de conjuntos de elementos correspondientes al NFA de los elementos de acuerdo con la construcción de subconjuntos del capítulo 2. Entonces podremos establecer el algoritmo de análisis sintáctico LR(0). Realizaremos la construcción del subconjunto para los dos ejemplos de NFA que acabamos de dar.

Ejemplo 5.7

Considere el NFA de la figura 5.1. El estado inicial del DFA asociado es la cerradura ε del conjunto compuesto del elemento $S' \rightarrow .S$, y éste es el conjunto de tres elementos

$\{S' \rightarrow .S, S \rightarrow .\ (\ S\ )\ S, S \rightarrow .\}$. Puesto que existe una transición desde $S' \rightarrow .S$ hasta $S' \rightarrow S\ .$ en S, existe una transición correspondiente del estado inicial al estado DFA $\{\ S' \rightarrow S\ .\ \}$ (sin que haya transiciones ε desde $S' \rightarrow S\ .$ a ningún otro elemento). También existe una transición en (del estado inicial al estado DFA $\{\ S \rightarrow (.\ S\ )\ S, S \rightarrow .(\ S\ )\ S, S \rightarrow .\}$ (la cerradura ε de $\{\ S \rightarrow (\ .\ S\ )\ S\ \}$). El estado DFA $\{\ S \rightarrow (.\ S\ )\ S, S \rightarrow .(\ S\ )\ S, S \rightarrow .\}$ tiene transiciones hacía sí mismo en (y a $\{\ S \rightarrow (\ S\ .)\ S\ \}$ en S. Este estado tiene una transición al estado $\{\ S \rightarrow (\ S\ ).\ S, S \rightarrow .(\ S\ )\ S, S \rightarrow .\}$ en). Finalmente, este último estado tiene transiciones en (al estado antes construido $\{S \rightarrow (.\ S\ )\ S, S \rightarrow .(\ S\ )\ S, S \rightarrow .\}$ y una transición en S al estado $\{\ S \rightarrow (\ S\ )\ S\ .\}$. El DFA completo se proporciona en la figura 5.3, donde numeramos los estados como referencia (el estado 0 se asigna tradicionalmente al estado inicial).

Figura 5.3
El DFA de conjuntos de elementos correspondientes al NFA de la figura 5.1

§

Ejemplo 5.8

Considere el NFA de la figura 5.2. El estado inicial del DFA asociado se compone del conjunto de los tres elementos $\{E' \rightarrow .E\ ,\ E \rightarrow .E + \mathbf{n}\ ,\ E \rightarrow\ .\mathbf{n}\}$. Existe una transición del elemento $E' \rightarrow .E$ al elemento $E' \rightarrow E\ .$ en E, pero también hay una transición en E del elemento $E \rightarrow .E + \mathbf{n}$ al elemento $E \rightarrow E\ .+\ \mathbf{n}$. De este modo, existe una transición en E del estado inicial del DFA a la cerradura del conjunto $\{E' \rightarrow E\ .,\ E \rightarrow E\ .+\ \mathbf{n}\}$. Puesto que no hay transiciones ε desde cualquiera de estos elementos, este conjunto es su propia cerradura ε y forma un estado DFA completo. Existe otra transición del estado inicial, correspondiente a la transición en el símbolo $\mathbf{n}$ de $E \rightarrow\ .\mathbf{n}$ a $E \rightarrow \mathbf{n}.$. Puesto que no se tienen transiciones ε del elemento $E \rightarrow \mathbf{n}.$, este elemento es su propia cerradura ε y forma el estado DFA $\{E \rightarrow \mathbf{n}.\}$. No hay transiciones desde este estado, de modo que las únicas transiciones por ser calculadas todavía vienen del conjunto $\{E' \rightarrow E\ .,\ E \rightarrow E\ .+\ \mathbf{n}\}$. Sólo hay una transición de este conjunto, correspondiente a la transición del elemento $E \rightarrow E\ .+\ \mathbf{n}$ al elemento $E \rightarrow E\ +.\mathbf{n}$ con el símbolo +. El elemento $E \rightarrow E\ +.\mathbf{n}$ tampoco tiene alguna transición ε, por lo tanto, forma un conjunto con un solo elemento en el DFA. Finalmente, existe una transición en $\mathbf{n}$ del conjunto $\{E \rightarrow E\ +.\mathbf{n}\}$ al conjunto $\{E \rightarrow E + \mathbf{n}.\}$. El DFA completo se proporciona en la figura 5.4.

Figura 5.4
El DFA de conjuntos de elementos correspondientes al NFA de la figura 5.2

§

Al construir el DFA de los conjuntos de elementos LR(0), en ocasiones se distinguen los elementos que se agregan a un estado durante la etapa de la cerradura ε de los que lo originan como objetivo de transiciones no ε. Los primeros se denominan **elementos de cerradura**, mientras que los últimos se conocen como **elementos de núcleo**. En el estado 0 de la figura 5.4, $E' \rightarrow .E$ es un elemento de núcleo (el único), mientras que $E \rightarrow .E + n$ y $E \rightarrow .n$ son elementos de cerradura. En el estado 2 de la figura 5.3, $S \rightarrow (.S)$ S es un elemento de núcleo, mientras que $S \rightarrow .(S)$ S y $S \rightarrow .$ son elementos de cerradura. De acuerdo con la definición de las transiciones ε de los NFA de elementos, todos los elementos de cerradura son elementos iniciales.

La importancia de distinguir entre elementos de núcleo y de cerradura es que, dada la gramática, los elementos de núcleo determinan unívocamente el estado y sus transiciones. De este modo, sólo es necesario especificar los elementos de núcleo para caracterizar completamente el DFA de conjuntos de elementos. Los generadores de analizadores sintácticos que construyen el DFA pueden, por lo tanto, informar sólo de los elementos de núcleo (esto es cierto para Yacc, por ejemplo).

Si en vez de calcular primero el NFA de los elementos y posteriormente aplicar la construcción de subconjuntos, se calcula directamente el DFA de los conjuntos de elementos, se presenta una simplificación adicional. En realidad, es fácil determinar de inmediato a partir de un conjunto de elementos cuáles son las transiciones ε y a cuáles elementos iniciales apuntan. Por consiguiente, los generadores de analizadores sintácticos como Yacc siempre calculan el DFA directamente desde la gramática, lo que haremos también en lo que resta del capítulo.

5.2.3 El algoritmo de análisis sintáctico LR(0)

Ahora estamos listos para establecer el algoritmo de análisis sintáctico LR(0). Como el algoritmo depende del seguimiento del estado actual en el DFA de conjuntos de elementos, debemos modificar la pila de análisis sintáctico para no sólo almacenar símbolos sino también números de estado. Haremos esto mediante la inserción del nuevo número de estado en la pila del análisis sintáctico después de cada inserción de un símbolo. De hecho, los estados mismos contienen toda la información acerca de los símbolos, así que podríamos prescindir del todo de los símbolos y mantener sólo los números de estado en la pila de análisis sintáctico. Sin embargo, mantendremos los símbolos en la pila por conveniencia y claridad.

Para comenzar un análisis sintáctico insertamos el marcador inferior \$ y el estado inicial 0 en la pila, de manera que al principio del análisis la situación se puede representar como

Pila de análisis sintáctico	Entrada
\$ 0	*CadenaEntrada* \$

Supongamos ahora que el siguiente paso es desplazar un token *n* a la pila y vayamos al estado 2 (esto ocurrirá en realidad cuando el DFA esté como en la figura 5.4 y *n* sea el token siguiente en la entrada). Esto se representa de la manera siguiente:

Pila de análisis sintáctico	Entrada
$ 0 *n* 2	*Resto de CadenaEntrada* $

El algoritmo de análisis sintáctico LR(0) elige una acción basado en el estado DFA actual, que es siempre el estado que aparece en la parte superior de la pila.

Definición

El algoritmo de análisis sintáctico LR(0). Sea s el estado actual (en la parte superior de la pila de análisis sintáctico). Entonces las acciones se definen como sigue:

1. Si el estado s contiene cualquier elemento de la forma $A \rightarrow \alpha . X \beta$, donde X es un terminal, entonces la acción es desplazar el token de entrada actual a la pila. Si este token es X, y el estado s contiene el elemento $A \rightarrow \alpha . X \beta$, entonces el nuevo estado que se insertará en la pila es el estado que contiene el elemento $A \rightarrow \alpha X . \beta$. Si este token no es X para algún elemento en el estado s de la forma que se acaba de describir, se declara un error.
2. Si el estado s contiene cualquier elemento completo (un elemento de la forma $A \rightarrow \alpha .$), entonces la acción es reducir mediante la regla $A \rightarrow \alpha$. Una reducción mediante la regla $S' \rightarrow S$, donde S es el estado inicial, es equivalente a la aceptación, siempre que la entrada esté vacía, y a un error si no es así. En todos los otros casos el nuevo estado se calcula como sigue. Elimine la cadena α y todos sus estados correspondientes de la pila de análisis sintáctico (la cadena α debe estar en la parte superior de la pila, de acuerdo con la manera en que esté construido el DFA). De la misma manera, retroceda en el DFA hacia el estado en el cual comenzó la construcción de α (éste debe ser el estado descubierto por la eliminación de α). Nuevamente, mediante la construcción del DFA, este estado debe contener un elemento de la forma $B \rightarrow \alpha . A \beta$. Inserte A en la pila, e inserte (como el nuevo estado) el estado que contiene el elemento $B \rightarrow \alpha A . \beta$. (Advierta que esto corresponde a seguir la transición en A en el DFA, lo cual es en realidad razonable, puesto que estamos insertando A en la pila.)

Se dice que una gramática es una **gramática LR(0)** si las reglas anteriores son no ambiguas. Esto significa que si un estado contiene un elemento completo $A \rightarrow \alpha .$, no puede contener otros. En efecto, si un estado así también contiene un elemento "desplazado" $A \rightarrow \alpha . X \beta$ (donde X es un terminal), entonces surge una ambigüedad, ya sea que se realice una acción 1) o una acción 2). Esta situación se conoce como **conflicto en reducción por desplazamiento**. De manera similar, si un estado de esta clase contiene otro elemento completo $B \rightarrow \beta .$, surge una ambigüedad al decidir qué producción utilizar para la reducción ($A \rightarrow \alpha$ o $B \rightarrow \beta$). Esta situación se conoce como **conflicto de reducción-reducción**. De este modo, una gramática es LR(0) si y sólo si cada estado es un estado de desplazamiento (un estado que contiene sólo elementos de "desplazamiento") o un estado de reducción que contiene sólo un elemento completo.

Advertimos que ninguna de las dos gramáticas que hemos estado utilizando como nuestros ejemplos son gramáticas LR(0). En la figura 5.3 los estados 0, 2 y 4 contienen todos conflictos de reducción por desplazamiento para el algoritmo de análisis sintáctico LR(0), mientras que en el DFA de la figura 5.4 el estado 1 contiene un conflicto de reducción por desplazamiento. Esto no es de sorprender, puesto que casi todas las gramáticas "reales" no son LR(0). Sin embargo, presentamos el ejemplo siguiente de una gramática que es LR(0).

Ejemplo 5.9

Considere la gramática

$$A \rightarrow (\ A\ ) \mid \mathtt{a}$$

La gramática aumentada tiene el DFA de los conjuntos de elementos que se muestran en la figura 5.5, que es LR(0). Para ver cómo funciona el algoritmo de análisis sintáctico LR(0) consideremos la cadena `((a))`. Un análisis sintáctico de esta cadena de acuerdo con el algoritmo de análisis sintáctico LR(0) se da mediante los pasos de la tabla 5.3. El análisis comienza en el estado 0, el cual es un estado de desplazamiento, de modo que el primer token `(` se desplaza a la pila. Luego, como el DFA indica una transición del estado 0 al estado 3 en el símbolo `(`, el estado 3 se inserta en la pila. Este estado también es un estado de desplazamiento, de modo que el siguiente `(` se desplaza a la pila, y la transición con `(` regresa al estado 3. Al desplazar de nueva cuenta se pone a **a** en la pila, y la transición con **a** del estado 3 se va al estado 2. Ahora nos encontramos en el paso 4 de la tabla 5.3, y alcanzamos el primer estado de reducción. Aquí el estado 2 y el símbolo **a** se extraen de la pila y en el proceso se regresa al estado 3. Entonces se inserta A en la pila y se lleva a cabo la transición de A del estado 3 al estado 4. El estado 4 es un estado de desplazamiento, de modo que `)` se desplaza a la pila y la transición con `)` lleva el análisis sintáctico al estado 5. Aquí se presenta una reducción mediante la regla $A \rightarrow (A)$, al extraer los estados 5, 4, 3 y los símbolos `)`, A, `(` de la pila. El análisis sintáctico está ahora en el estado 3, y nuevamente A y el estado 4 se insertan en la pila. De nuevo, `)` se desplaza a la pila y se inserta el estado 5. Otra reducción mediante $A \rightarrow (A)$ elimina la cadena `(` 3 A 4 `)` 5 (hacia atrás) de la pila, lo que deja el análisis en el estado 0. Ahora se inserta A y se lleva a cabo la transición de A del estado 0 al estado 1. El estado 1 es el estado de aceptación. Como ahora la entrada está vacía, acepta el algoritmo de análisis.

Figura 5.5
El DFA de conjuntos de elementos para el ejemplo 5.9

Tabla 5.3
Acciones de análisis sintáctico para el ejemplo 5.9

	Pila de análisis sintáctico	Entrada	Acción
1	$ 0	((a)) $	desplazamiento
2	$ 0 (3	(a)) $	desplazamiento
3	$ 0 (3 (3	a)) $	desplazamiento
4	$ 0 (3 (3 a 2	)) $	reducción $A \rightarrow$ a
5	$ 0 (3 (3 A 4	)) $	desplazamiento
6	$ 0 (3 (3 A 4) 5	) $	reducción $A \rightarrow (A)$
7	$ 0 (3 A 4	) $	desplazamiento
8	$ 0 (3 A 4) 5	$	reducción $A \rightarrow (A)$
9	$ 0 A 1	$	aceptar

§

El DFA de los conjuntos de elementos y las acciones especificadas por el algoritmo de análisis sintáctico LR(0) se pueden combinar en una tabla de análisis sintáctico, de manera que el análisis sintáctico LR(0) se convierta en un método de análisis sintáctico controlado por tabla. Una organización típica es que los renglones de la tabla sean etiquetados con los estados del DFA, y las columnas se etiqueten como sigue. Puesto que los estados del análisis sintáctico LR(0) son estados "de desplazamiento" o estados "de reducción", se reserva una columna para indicar esto en cada estado. En el caso de un estado "de reducción" se utiliza otra columna para indicar la regla gramatical que se usará en la reducción. En el caso de un estado de desplazamiento el símbolo por ser desplazado determina el estado siguiente (mediante el DFA), y así debe haber una columna para cada token, cuyas entradas son los nuevos estados por introducirse en un desplazamiento de ese token. Las transiciones de no terminales (que se insertan durante una reducción) representan un caso especial, puesto que no se ven realmente en la entrada, aunque el analizador sintáctico actúa como si estuvieran desplazados. De este modo, en un estado "de desplazamiento" también se necesita tener una columna para cada no terminal, y tradicionalmente estas columnas se enlistan en una parte separada de la tabla denominada sección **Ir a** (**Goto**).

Un ejemplo de una tabla de análisis sintáctico de esta naturaleza es la tabla 5.4, que es la tabla para la gramática del ejemplo 5.9. Se alienta al lector para que verifique que ésta conduce a acciones de análisis sintáctico para ese ejemplo como las dadas en la tabla 5.3.

Tabla 5.4
Tabla de análisis sintáctico para la gramática del ejemplo 5.9

Estado	**Acción**	**Regla**	**Entrada**			**Goto**
			(	a	)	A
0	desplazamiento		3	2		1
1	reducción	$A' \rightarrow A$				
2	reducción	$A \rightarrow$ a				
3	desplazamiento		3	2		4
4	desplazamiento				5	
5	reducción	$A \rightarrow (A)$				

Advertimos que las entradas vacías en una tabla de análisis sintáctico de esta clase representan errores. En situaciones prácticas donde la recuperación de errores es necesaria

necesitaremos especificar con precisión qué acción toma el analizador sintáctico para cada una de estas entradas en blanco. Pospondremos este análisis hasta una sección posterior.

5.3 ANÁLISIS SINTÁCTICO SLR(1)

5.3.1 El algoritmo de análisis sintáctico SLR(1)

El análisis sintáctico LR(1) simple, o SLR(1) utiliza el DFA de conjuntos de elementos LR(0) como se construyeron en la sección anterior. Sin embargo, incrementa de manera importante la potencia del análisis sintáctico LR(0) al utilizar el token siguiente en la cadena de entrada para dirigir sus acciones. Lo hace de dos maneras. La primera consulta el token de entrada *antes* de un desplazamiento para asegurarse de que existe una transición DFA apropiada. La segunda utiliza el conjunto Siguiente de un no terminal, como se construyó en la sección 4.3, para decidir si debería efectuarse una reducción. Sorprende que este simple uso de la búsqueda hacia delante sea tan poderoso que permita analizar casi toda construcción de un lenguaje que se presenta comúnmente.

Definición

El algoritmo de análisis sintáctico SLR(1). Sea s el estado actual (en la parte superior de la pila de análisis sintáctico). Entonces las acciones se definen como sigue:

1. Si el estado s contiene cualquier elemento de la forma $A \rightarrow \alpha . X\ \beta$, donde X es un terminal, y X es el siguiente token en la cadena de entrada, entonces la acción es desplazar el token de entrada actual a la pila, y el nuevo estado que se insertará en la pila es el estado que contiene el elemento $A \rightarrow \alpha X\ .\ \beta$.
2. Si el estado s contiene el elemento completo $A \rightarrow \gamma.$, y el token siguiente en la cadena de entrada está en una Siguiente(A), entonces la acción es reducir mediante la regla $A \rightarrow \gamma$. Una reducción mediante la regla $S' \rightarrow S$, donde S es el estado inicial, es equivalente a la aceptación; esto ocurrirá sólo si el siguiente token de entrada es \$.[4] En todos los otros casos, el nuevo estado se calcula como sigue. Elimine la cadena α y todos sus estados correspondientes de la pila de análisis sintáctico. De la misma manera, retroceda en el DFA hacia el estado en el cual comenzó la construcción de α. Por construcción, este estado debe contener un elemento de la forma $B \rightarrow \gamma . A\ \beta$. Inserte A en la pila, e inserte el estado que contiene el elemento $B \rightarrow \alpha A\ .\ \beta$.
3. Si el siguiente token de entrada es de naturaleza tal que no se aplique ninguno de los dos casos anteriores, se declara un error.

Decimos que una gramática es una **gramática SLR(1)** si la aplicación de las anteriores reglas de análisis sintáctico SLR(1) no producen una ambigüedad. En particular, una gramática es SLR(1) si y sólo si, para cualquier estado s, se satisfacen las siguientes dos condiciones:

4. En realidad, el conjunto Siguiente para el estado inicial aumentado S' de cualquier gramática siempre es el conjunto compuesto sólo de \$, aparece únicamente en la regla gramatical $S' \rightarrow S$.

1. Para cualquier elemento $A \rightarrow \alpha . X \beta$ en s con un terminal X, no hay elemento completo $B \rightarrow \gamma .$ en s con X en Siguiente(B).
2. Para cualesquiera dos elementos completos $A \rightarrow \alpha .$ y $B \rightarrow \beta .$ en s, Siguiente(A) $\cap$ Siguiente(B) es vacía.

Una violación de la primera de estas condiciones representa un **conflicto de reducción por desplazamiento**. Una violación de la segunda de estas condiciones representa un **conflicto de reducción-reducción**.

Estas dos condiciones son semejantes en esencia a las dos condiciones para el análisis sintáctico LL(1) establecidas en el capítulo anterior, excepto que, como con todos los métodos de análisis sintáctico de reducción por desplazamiento, las decisiones sobre cuál regla gramatical utilizar se pueden posponer hasta el último momento posible, lo que da como resultado un analizador sintáctico más poderoso.

Una tabla de análisis para el análisis sintáctico SLR(1) también puede construirse de manera similar a la del análisis sintáctico LR(0) descrita en la sección anterior. Las diferencias son las siguientes. Como un estado puede tener tanto desplazamientos como reducciones en un analizador sintáctico SLR(1) (dependiendo de la búsqueda hacia delante), cada ingreso en la sección de entrada debe tener ahora una etiqueta de "desplazamiento" o "reducción", y las selecciones de regla gramatical también deben ser colocadas en entradas marcadas "reducción". Eso también provoca que las columnas de acción y reglas sean innecesarias. Puesto que el símbolo $ de fin de entrada también puede ser uno de búsqueda hacia delante legal, se debe crear una nueva columna para este símbolo en la sección de entrada. Demostraremos la construcción de la tabla de análisis sintáctico SLR(1) junto con nuestro primer ejemplo de análisis SLR(1).

Ejemplo 5.10

Consideremos la gramática del ejemplo 5.8, cuyo DFA de conjuntos de elementos está dado en la figura 5.4. Como establecimos, esta gramática no es LR(0), pero es SLR(1). Los conjuntos Siguiente para los no terminales son Siguiente(E') = {$} y Siguiente($E$) = {$, +}. La tabla de análisis sintáctico SLR(1) se proporciona en la tabla 5.5. En la tabla se indica un desplazamiento mediante la letra s en la entrada, y una reducción mediante la letra r. De este modo, en el estado 1 en la entrada + se indica un desplazamiento, junto con una transición al estado 3. En el estado 2 en la entrada +, por otra parte, se indica una reducción mediante la producción $E \rightarrow n$. También escribimos la acción "aceptar" en el estado 1 en la entrada $ en lugar de r ($E' \rightarrow E$).

Tabla 5.5
Tabla de análisis sintáctico SLR(1) para el ejemplo 5.10

Estado	Entrada			Ir a
	n	+	$	E
0	s2			1
1		s3	aceptar	
2		r ($E \rightarrow n$)	r ($E \rightarrow n$)	
3	s4			
4		r ($E \rightarrow E + n$)	r ($E \rightarrow E + n$)	

Finalizamos este ejemplo con un análisis sintáctico de la cadena $n + n + n$. Los pasos del análisis sintáctico se proporcionan en la tabla 5.6. El paso 1 de esta figura comienza en el estado 0 en el token de entrada n, y la tabla de análisis sintáctico indica la acción "s2",

es decir, desplazar el token a la pila e ir al estado 2. Indicamos esto en la tabla 5.6 con la frase "desplazar 2". En el paso 2 de la figura, el analizador sintáctico está en el estado 2 con el token de entrada +, y la tabla indica una reducción mediante la regla $E \rightarrow \mathbf{n}$. En este caso el estado 2 y el token **n** son extraídos de la pila, exponiendo al estado 0. Se inserta el símbolo E y se toma la acción Ir a para E del estado 0 al estado 1. En el paso 3 el analizador está en el estado 1 con el token de entrada +, y la tabla indica un desplazamiento y una transición al estado 3. En el estado 3 en la entrada **n** la tabla también indica un desplazamiento y una transición al estado 4. En el estado 4 sobre la entrada + la tabla indica una reducción mediante la regla $E \rightarrow E + \mathbf{n}$. Esta reducción se cumple al extraer de la pila la cadena $E + \mathbf{n}$ y sus estados asociados, exponiendo nuevamente el estado 0, insertando E y llevando Ir a al estado 1. Los pasos restantes en el análisis son semejantes.

Tabla 5.6
Acciones de análisis sintáctico para el ejemplo 5.10

	Pila de análisis sintáctico	Entrada	Acción
1	$ 0	**n** + **n** + **n** $	desplazamiento 2
2	$ 0 **n** 2	+ **n** + **n** $	reducción $E \rightarrow \mathbf{n}$
3	$ 0 E 1	+ **n** + **n** $	desplazamiento 3
4	$ 0 E 1 + 3	**n** + **n** $	desplazamiento 4
5	$ 0 E 1 + 3 **n** 4	+ **n** $	reducción $E \rightarrow E + \mathbf{n}$
6	$ 0 E 1	+ **n** $	desplazamiento 3
7	$ 0 E 1 + 3	**n** $	desplazamiento 4
8	$ 0 E 1 + 3 **n** 4	$	reducción $E \rightarrow E + \mathbf{n}$
9	$ 0 E 1	$	aceptar

§

Ejemplo 5.11

Considere la gramática de los paréntesis balanceados, cuyo DFA de elementos LR(0) está dado en la figura 5.3 (página 205). Un cálculo directo produce Siguiente(S') = {$} y Siguiente($S$) = {$,) }. La tabla de análisis sintáctico SLR(1) se proporcionan en la tabla 5.7. Advierta cómo los estados no-LR(0) 0, 2 y 4 tienen tanto desplazamientos como reducciones mediante la producción ε $S \rightarrow \varepsilon$. La tabla 5.8 proporciona los pasos que el algoritmo de análisis sintáctico SLR(1) lleva a cabo para analizar la cadena ( ) ( ). Observe cómo la pila continúa creciendo hasta las reducciones finales. Esto es característico de los analizadores sintácticos ascendentes en presencia de reglas recursivas por la derecha tales como $S \rightarrow (S) S$. De esta manera, la recursión por la derecha puede ocasionar la sobrecarga de la pila, y esto se debe evitar si es posible.

Tabla 5.7
Tabla de análisis sintáctico SLR(1) para el ejemplo 5.11

Estado	Entrada			Ir a
	(	)	$	S
0	s2	r ($S \rightarrow \varepsilon$)	r ($S \rightarrow \varepsilon$)	1
1			aceptar	
2	s2	r ($S \rightarrow \varepsilon$)	r ($S \rightarrow \varepsilon$)	3
3		s4		
4	s2	r ($S \rightarrow \varepsilon$)	r ($S \rightarrow \varepsilon$)	5
5		r ($S \rightarrow (S) S$)	r ($S \rightarrow (S) S$)	

Tabla 5.8
Acciones de análisis sintáctico para el ejemplo 5.11

	Pila de análisis sintáctico	Entrada	Acción
1	$ 0	( ) ( ) $	desplazamiento 2
2	$ 0 (2	) ( ) $	reducción $S \rightarrow \varepsilon$
3	$ 0 (2 S 3	) ( ) $	desplazamiento 4
4	$ 0 (2 S 3) 4	( ) $	desplazamiento 2
5	$ 0 (2 S 3) 4 (2	) $	reducción $S \rightarrow \varepsilon$
6	$ 0 (2 S 3) 4 (2 S 3	) $	desplazamiento 4
7	$ 0 (2 S 3) 4 (2 S 3) 4	$	reducción $S \rightarrow \varepsilon$
8	$ 0 (2 S 3) 4 (2 S 3) 4 S 5	$	reducción $S \rightarrow (S)S$
9	$ 0 (2 S 3) 4 S 5	$	reducción $S \rightarrow (S)S$
10	$ 0 S 1	$	aceptar

§

5.3.2 Reglas de eliminación de ambigüedad para conflictos de análisis sintáctico

Los conflictos de análisis en el análisis sintáctico SLR(1), como con todos los métodos de análisis sintáctico de reducción por desplazamiento, pueden ser de dos clases: conflictos de reducción por desplazamiento y conflictos de reducción-reducción. En el caso de conflictos de reducción por desplazamiento existe una regla de eliminación de ambigüedad natural, que consiste en preferir siempre el desplazamiento en vez de la reducción. La mayoría de los analizadores sintácticos de reducción por desplazamiento resuelven, por lo tanto, de manera automática los conflictos de reducción por desplazamiento prefiriendo el desplazamiento a la reducción. El caso de los conflictos de reducción-reducción es más difícil: tales conflictos con frecuencia (pero no siempre) indican un error en el diseño de la gramática. (Más adelante se proporcionan ejemplos de este tipo de conflictos.) Preferir el desplazamiento a la reducción en un conflicto de reducción por desplazamiento incorpora automáticamente la regla de anidación más próxima para la ambigüedad del else en sentencias if, como se muestra en el siguiente ejemplo. Esto es una razón para dejar que la ambigüedad permanezca en una gramática de lenguaje de programación.

Ejemplo 5.12

Considere la gramática de sentencias if simplificadas utilizada en los capítulos anteriores (véase, por ejemplo, el capítulo 3, sección 3.4.3):

$$
\begin{aligned}
&\textit{sentencia} \rightarrow \textit{sent-if} \mid \texttt{otro} \\
&\textit{sent-if} \rightarrow \texttt{if}\ (\ \textit{exp}\ )\ \textit{sentencia} \\
&\qquad\qquad \mid \texttt{if}\ (\ \textit{exp}\ )\ \textit{sentencia}\ \texttt{else}\ \textit{sentencia} \\
&\textit{exp} \rightarrow \texttt{0} \mid \texttt{1}
\end{aligned}
$$

Puesto que ésta es una gramática ambigua, debe haber un conflicto de análisis en alguna parte en cualquier algoritmo de análisis sintáctico. Para ver esto en un analizador SLR(1) simplificamos aún más la gramática, lo que hace más manejable la construcción del DFA de conjuntos de elementos. Incluso eliminamos del todo la expresión de prueba y escribimos la gramática como sigue (todavía con la ambigüedad del else):

$$
\begin{aligned}
&S \rightarrow I \mid \texttt{otro} \\
&I \rightarrow \texttt{if}\ S \mid \texttt{if}\ S\ \texttt{else}\ S
\end{aligned}
$$

El DFA de conjuntos de elementos se muestra en la figura 5.6. Para construir las acciones del análisis sintáctico SLR(1) necesitamos los conjuntos Siguiente para S e I. Éstos son

$$\text{Siguiente}(S) = \text{Siguiente}(I) = \{\$, \texttt{else}\}$$

Ahora podemos ver el conflicto de análisis provocado por el problema del else ambiguo. Se presenta en el estado 5 del DFA, donde el elemento completo $I \rightarrow$ **if** S. indica que va a tener lugar una reducción mediante la regla $I \rightarrow$ **if** S en las entradas **else** y \$, mientras que el elemento $I \rightarrow$ **if** S. **else** S indica que va a ocurrir un desplazamiento del token de entrada en **else**. De este modo, el else ambiguo resultará en un conflicto de reducción por desplazamiento en la tabla de análisis sintáctico SLR(1). Evidentemente, una regla de eliminación de ambigüedad que prefiera el desplazamiento a la reducción eliminará el conflicto y analizará según la regla de anidación más próxima. (En realidad, si se prefiriera la reducción sobre el desplazamiento, no habría manera de introducir los estados 6 o 7 en el DFA, lo que resultaría en errores espurios de análisis sintáctico.)

Figura 5.6
DFA de conjuntos de elementos LR(0) para el ejemplo 5.12

La tabla de análisis sintáctico SLR(1) que resulta de esta gramática se muestra en la tabla 5.9. En esa tabla utilizamos un esquema de numeración para las selecciones de regla gramatical en las acciones de reducción en lugar de escribir las reglas mismas. La numeración es

(1) $S \rightarrow I$
(2) $S \rightarrow$ *otro*
(3) $I \rightarrow$ **if** S
(4) $I \rightarrow$ **if** S **else** S

Advierta que no es necesario numerar la producción de aumento $S' \rightarrow S$, puesto que una reducción mediante esta regla corresponde a la aceptación y se escribe como "aceptar" en la tabla.

Advertimos al lector que los números de producción utilizados en entradas de reducción son fáciles de confundir con los números de estado empleados en las entradas para desplazamiento y direccionamiento (Ir a). Por ejemplo, en la tabla 5.9, en el estado 5 la entrada bajo el encabezado de entrada `else` es s6, lo que indica un desplazamiento y una transición al estado 6, mientras que la entrada bajo el encabezado de entrada $ es r3, lo que indica una reducción mediante el número de producción 3 (es decir, $I \rightarrow$ `if` S).

En la tabla 5.9 también eliminamos el conflicto de reducción por desplazamiento en la tabla en favor del desplazamiento. Sombreamos la entrada en la tabla para mostrar dónde se habría presentado el conflicto.

Tabla 5.9
Tabla de análisis sintáctico SLR(1) para el ejemplo 5.12 (con el conflicto de análisis sintáctico eliminado)

Estado	Entrada				Ir a	
	`if`	`else`	`otro`	$	S	I
0	s4		s3		1	2
1				aceptar		
2		r1		r1		
3		r2		r2		
4	s4		s3		5	2
5		s6		r3		
6	s4		s3		7	2
7		r4		r4		

§

5.3.3 Límites de la potencia del análisis sintáctico SLR(1)

El análisis sintáctico SLR(1) es una extensión simple y efectiva del análisis LR(0) que es tan potente que puede controlar casi cualquier estructura práctica de lenguaje. Por desgracia, existen algunas situaciones en las que el análisis sintáctico SLR(1) no es lo suficientemente poderoso, esto nos conducirá a estudiar el análisis sintáctico LALR(1) y LR(1) general con más poder. El ejemplo siguiente es una situación típica en la que falla el análisis sintáctico SLR(1).

Ejemplo 5.13

Considere las siguientes reglas gramaticales para sentencias, extraídas y simplificadas de Pascal (una situación semejante ocurre en C):

sent → *llamad-sent* | *sent-asignac*
llamad-sent → `identificador`
sent-asignac → *var* `:=` *exp*
var → *var* `[` *exp* `]` | `identificador`
exp → *var* | `número`

Esta gramática modela sentencias que pueden hacer llamadas a procedimientos sin parámetros, o bien asignaciones o expresiones a variables. Advierta que tanto las asignaciones como las llamadas a procedimiento comienzan con un identificador. No es sino hasta que se detecta el fin de la sentencia o el token `:=` que un analizador sintáctico puede decidir si la

sentencia es una asignación o una llamada. Simplificamos esta situación para la gramática siguiente, donde eliminamos las opciones alternativas para una variable, e hicimos más simples las opciones de sentencias, sin modificar la situación básica:

$$S \rightarrow \texttt{id} \mid V \texttt{:=} E$$
$$V \rightarrow \texttt{id}$$
$$E \rightarrow V \mid \texttt{n}$$

Para mostrar cómo esta gramática produce un conflicto de análisis en el análisis sintáctico SLR(1) considere el estado inicial del DFA de los conjuntos de elementos:

$$S' \rightarrow .\, S$$
$$S \rightarrow .\, \texttt{id}$$
$$S \rightarrow .\, V \texttt{:=} E$$
$$V \rightarrow .\, \texttt{id}$$

Este estado tiene una transición por desplazamiento sobre **id** al estado

$$S \rightarrow \texttt{id}.$$
$$V \rightarrow \texttt{id}.$$

Ahora, Siguiente(S) = {\$} y Siguiente(V) = { **:=**, \$} (**:=** debido a la regla $V \rightarrow V$ **:=** E, y \$ porque un E puede ser un V). De este modo, el algoritmo de análisis sintáctico SLR(1) pide una reducción en este estado tanto por la regla $S \rightarrow$ **id** como por la regla $V \rightarrow$ **id** bajo el símbolo de entrada \$. (Éste es un conflicto de reducción-reducción.) Este conflicto de análisis sintáctico es en realidad un problema "falso" ocasionado por la debilidad del método SLR(1). Efectivamente, la reducción mediante $V \rightarrow$ **id** *nunca* debería hacerse en este estado cuando la entrada es \$, puesto que una variable nunca puede presentarse al final de una sentencia hasta después que el token **:=** se haya visto y desplazado. §

En las dos secciones siguientes mostraremos cómo se puede eliminar este problema de análisis mediante el uso de métodos de análisis sintácticos más poderosos.

5.3.4 Gramáticas SLR(k)

Como con otros algoritmos de análisis, el algoritmo de análisis sintáctico SLR(1) se puede extender al análisis sintáctico SLR(k) donde las acciones de análisis están basadas en $k \geq 1$ símbolos de búsqueda hacia delante. Utilizando los conjuntos Primero$_k$ y Siguiente$_k$ como se definieron en el capítulo anterior, un analizador sintáctico SLR(k) utiliza las siguientes dos reglas:

1. Si el estado s contiene un elemento de la forma $A \rightarrow \alpha . X \beta$, (donde X es un token) y $Xw \in$ Primero$_k(X \beta)$ son los siguientes tokens k en la cadena de entrada, entonces la acción es desplazar el token de entrada actual a la pila, y el nuevo estado que se insertará en la pila es el estado que contiene el elemento $A \rightarrow \alpha X . \beta$.
2. Si el estado s contiene el elemento completo $A \rightarrow \alpha .$, y $w \in$ Siguiente$_k(A)$ son los siguientes k tokens en la cadena de entrada, entonces la acción es reducir mediante la regla $A \rightarrow \alpha$.

El análisis sintáctico SLR(k) es más poderoso que el análisis SLR(1) cuando $k > 1$, pero con un costo sustancial en complejidad, ya que la tabla de análisis sintáctico aumenta exponencialmente su tamaño respecto a k. Las construcciones de lenguaje típicas que no son

SLR(1) se controlan mejor utilizando un analizador sintáctico LALR(1), empleando reglas estándar de eliminación de ambigüedad, o volviendo a escribir la gramática. Mientras que es verdad que la gramática simple no-SLR(1) del ejemplo 5.13 parece ser SLR(2), el problema del lenguaje de programación que viene de allí no es SLR(k) para ninguna k.

5.4 ANÁLISIS SINTÁCTICO LALR(1) Y LR(1) GENERAL

En esta sección estudiaremos la forma más general de análisis sintáctico LR(1), en ocasiones conocida como análisis sintáctico LR(1) **canónico**. Este método supera el problema con el análisis sintáctico SLR(1) descrito al final de la sección anterior, pero con el costo de incrementar de manera sustancial la complejidad. En realidad, el análisis sintáctico LR(1) general por lo regular se considera demasiado complejo para utilizarlo en la construcción de analizadores sintácticos en la mayoría de las situaciones. Afortunadamente, una modificación del análisis sintáctico general LR(1), denominado LALR(1) (por las siglas en inglés del análisis "Lookahead" LR), conserva la mayor parte del beneficio del análisis LR(1) general, mientras que mantiene la eficiencia del método SLR(1). El método LALR(1) se ha vuelto el método de elección para generadores de analizadores sintácticos tales como Yacc, por lo que lo estudiaremos posteriormente en esta sección. Sin embargo, para comprender este método primero debemos estudiar el método general.

5.4.1 Autómatas finitos de elementos LR(1)

La dificultad con el método SLR(1) es que aplica las búsquedas hacia delante *después* de construir el DFA de elementos LR(0), una construcción que ignora búsquedas hacia delante. La potencia del método LR(1) general se debe a que utiliza un nuevo DFA que tiene las búsquedas integradas en su construcción desde el inicio. Este DFA utiliza elementos que son una extensión de los elementos LR(0). Se conocen como **elementos LR(1)** porque incluyen un token de búsqueda hacia delante simple en cada elemento. Más exactamente, un elemento LR(1) es un par compuesto de un elemento LR(0) y un token de búsqueda hacia delante. Escribimos elementos LR(1) utilizando corchetes como

$$[A \rightarrow \alpha.\beta, a]$$

donde $A \rightarrow \alpha.\beta$ es un elemento LR(0) y a es un token (la búsqueda hacia delante).

Para completar la definición del autómata utilizado para el análisis sintáctico LR(1) general necesitamos definir las transiciones entre elementos LR(1). Éstas son semejantes a las transiciones LR(0), excepto porque también se mantienen al tanto de las búsquedas hacia delante. Como con los elementos LR(0) que incluyen transiciones ε, es necesario construir un DFA cuyos estados sean conjuntos de elementos que sean cerraduras ε. La diferencia principal entre los autómatas LR(0) y LR(1) viene en la definición de las transiciones ε. Daremos primero la definición del caso más fácil (las transiciones no ε), que son esencialmente idénticas a las correspondientes al caso LR(0).

Definición

Definición de transiciones LR(1) (parte 1). Dado un elemento LR(1) $[A \rightarrow \alpha . X\gamma, a]$, donde X es cualquier símbolo (terminal o no terminal), existe una transición con X al elemento $[A \rightarrow \alpha X . \gamma, a]$.

Observe que en este caso la misma a de búsqueda hacia delante aparece en ambos elementos. De este modo, estas transiciones no provocan la aparición de nuevas búsquedas hacia delante. Sólo las transiciones ε "crean" nuevas búsquedas hacia delante de la manera siguiente.

Definición

Definición de transiciones LR(1) (parte 2). Dado un elemento LR(1) $[A \rightarrow \alpha . B\ \gamma, a]$, donde B es un no terminal, existen transiciones ε para elementos $[B \rightarrow .\beta, b]$ para cada producción $B \rightarrow \beta$ y *para cada token* b en Primero(γa).

Advierta cómo estas transiciones ε se mantienen al tanto del contexto en el cual se necesita reconocer la estructura B. Efectivamente, el elemento $[A \rightarrow \alpha . B\ \gamma, a]$ dice que en este punto del análisis sintáctico podemos querer reconocer una B, pero *sólo* si esta B está seguida por una cadena derivable de la cadena γa, y tales cadenas deben comenzar con un token en Primero(γa). Puesto que la cadena γ sigue a B en la producción $A \rightarrow \alpha B \gamma$, si a se construye para estar en Siguiente(A), entonces Primero(γa) $\subset$ Siguiente(B), y las b en los elementos $[B \rightarrow .\beta, b]$ siempre estarán en Siguiente(B). La potencia del método LR(1) general reside en el hecho de que el conjunto Primero(γa) puede ser un subconjunto *propio* de Siguiente(B). (Un analizador SLR(1) toma esencialmente las búsquedas hacia delante b del conjunto Siguiente completo.) Advierta también que la búsqueda hacia delante original a aparece como una de las b sólo si γ puede derivar la cadena vacía. Gran parte de las veces (especialmente en situaciones prácticas), esto ocurrirá sólo si γ es ε misma, si esto sucede obtenemos el caso especial de una transición ε de $[A \rightarrow \alpha . B, a]$ a $[B \rightarrow .\beta, a]$.

Para completar la descripción de la construcción del DFA de los conjuntos de elementos LR(1) resta especificar el estado inicial. Como en el caso LR(0), hacemos esto aumentando la gramática con un nuevo símbolo inicial S' y una nueva producción $S' \rightarrow S$ (donde S es el símbolo inicial original). Entonces el símbolo inicial del NFA de elementos LR(1) se convierte en el elemento $[S' \rightarrow .S, \$]$, donde el signo monetario \$ representa el marcador de terminación (y es el único símbolo en Siguiente(S')). En efecto, esto nos dice que comenzamos reconociendo una cadena derivable de S, seguida por el símbolo \$.

Ahora examinaremos varios ejemplos de la construcción del DFA de elementos LR(1).

Ejemplo 5.14 Considere la gramática

$$A \rightarrow (\ A\ ) \mid \mathtt{a}$$

del ejemplo 5.9. Iniciamos la construcción de este DFA de conjuntos de elementos LR(1) aumentando la gramática y formando el elemento LR(1) inicial $[A' \rightarrow .A, \$]$. La cerradura ε de este elemento es el estado inicial del DFA. Como A no está seguida por ningún símbolo en este elemento (en la terminología del análisis anterior acerca de transiciones, la cadena γ es ε), existen transiciones ε para los elementos $[A \rightarrow .(\ A\ ), \$]$ y $[A \rightarrow .\mathtt{a}, \$]$ (es decir, Primero($\gamma\$$) = $\{\$\}$, en la terminología del análisis anterior). El estado inicial (estado 0) es entonces el conjunto de estos tres elementos:

$$\begin{array}{ll} \text{Estado 0:} & [A' \rightarrow .A, \$] \\ & [A \rightarrow .(\ A\ ), \$] \\ & [A \rightarrow .\mathtt{a}, \$] \end{array}$$

Desde este estado hay una transición con A a la cerradura del conjunto que contiene e elemento $[A' \rightarrow A., \$]$, y puesto que no hay transiciones desde elementos completos, est estado contiene sólo el elemento simple $[A' \rightarrow A., \$]$. No hay transiciones fuera de este es tado, al que numeraremos como estado 1:

Estado 1: $[A' \rightarrow A., \$]$

(Éste es el estado desde el cual el algoritmo de análisis sintáctico LR(1) generará las acciones de aceptación.)

Al regresar al estado 0, existe también una transición con el token (para la cerradura del conjunto que está compuesta del elemento $[A \rightarrow (\ .\ A\ ), \$]$. Debido a que existen transiciones ε desde este elemento, esta cerradura es no trivial. En efecto, existen transiciones ε desde este elemento a los elementos $[A \rightarrow .(\ A\ ),\ )]$ y $[A \rightarrow .\mathtt{a},\ )]$. Esto se debe a que, en el elemento $[A \rightarrow (.\ A\ ), \$]$, A se está reconociendo en el lado derecho *en el contexto de los paréntesis*. Es decir, el conjunto Siguiente de la A del lado derecho es Primero( ) $) = \{)\}$. De este modo, obtuvimos un nuevo token de búsqueda hacia delante en esta situación, y el nuevo estado DFA se compone de los elementos siguientes:

Estado 2: $[A \rightarrow (\ .\ A\ ), \$]$
$[A \rightarrow .(\ A\ ),\ )]$
$[A \rightarrow .\mathtt{a},\ )]$

Regresamos una vez más al estado 0, donde encontramos una última transición al estado generado por el elemento $[A \rightarrow \mathtt{a}., \$]$. Y ya que esto es un elemento completo, es un estado de un elemento:

Estado 3: $[A \rightarrow \mathtt{a}., \$]$

Ahora volvemos al estado 2. Desde este estado existe una transición con A para la cerradura ε de $[A \rightarrow (\ A\ .), \$]$, que es un estado con un elemento:

Estado 4: $[A \rightarrow (\ A\ .), \$]$

También existe una transición con (a la cerradura ε de $[A \rightarrow (.\ A\ ),\ )]$. Aquí también generamos los elementos de cerradura $[A \rightarrow .(\ A\ ),\ )]$ y $[A \rightarrow .\mathtt{a},\ )]$, por las mismas razones que en la construcción del estado 2. Por consiguiente, obtenemos el nuevo estado:

Estado 5: $[A \rightarrow (\ .\ A\ ),\ )]$
$[A \rightarrow .(\ A\ ),\ )]$
$[A \rightarrow .\mathtt{a},\ )]$

Advierta que este estado es el mismo que el estado 2, excepto por la búsqueda hacia delante del primer elemento.

Finalmente, tenemos una transición con el token a del estado 2 al estado 6:

Estado 6: $[A \rightarrow \mathtt{a}.,\ )]$

Observe de nueva cuenta que esto es casi lo mismo que el estado tres, excepto por la búsqueda hacia delante.

El siguiente estado que tiene una transición es el estado 4, que tiene una transición con el token) al estado

Estado 7: $[A \rightarrow (\ A\ ).,\ \$]$

Al regresar al estado 5, tenemos una transición de este estado hacia sí mismo con (, una transición con A al estado

$$\text{Estado 8:} \qquad [A \rightarrow (\ A\ .\ ),\ )]$$

y una transición al estado 6 ya construido con **a**.

Finalmente, existe una transición en) del estado 8 a

$$\text{Estado 9:} \qquad [A \rightarrow (\ A\ )\ .,\ )].$$

De este modo, el DFA de elementos LR(1) para esta gramática tiene diez estados. El DFA completo se muestra en la figura 5.7. Al comparar esto con el DFA de los conjuntos de elementos LR(0) para la misma gramática (véase la figura 5.5, página 208), observamos que el DFA de elementos LR(1) es casi el doble de extenso. Esto no es inusual. En realidad, el número de estados LR(1) puede exceder el número de estados LR(0) por un factor de 10 en situaciones complejas con muchos tokens de búsqueda hacia delante.

Figura 5.7
El DFA de conjuntos de elementos LR(1) para el ejemplo 5.14

§

5.4.2 El algoritmo de análisis sintáctico LR(1)

Antes de considerar ejemplos adicionales necesitamos completar la exposición acerca del *análisis sintáctico LR(1) mediante el reestablecimiento del algoritmo de análisis sintáctico* basado en la nueva construcción DFA. Esto es fácil de hacer, puesto que sólo es un reestablecimiento del algoritmo de análisis sintáctico SLR(1), excepto porque utiliza los tokens de búsqueda hacia delante en los elementos LR(1) en lugar de los conjuntos Siguiente.

El algoritmo de análisis sintáctico LR(1) general Sea s el estado actual (en la parte superior de la pila de análisis sintáctico). Entonces las acciones se definen de la manera siguiente:

1. Si el estado s contiene cualquier elemento LR(1) de la forma $[A \rightarrow \alpha.X\ \beta, a]$, donde X es un terminal, y X es el token siguiente en la cadena de entrada, entonces la acción es desplazar el token de entrada actual a la pila, y el nuevo estado que se insertará en la pila es el estado que contiene el elemento LR(1) $[A \rightarrow \alpha X.\beta, a]$.
2. Si el estado s contiene el elemento completo LR(1) $[A \rightarrow \alpha., a]$, y el token siguiente en la cadena de entrada es a, entonces la acción es reducir mediante la regla $A \rightarrow \alpha$. Una reducción mediante la regla $S' \rightarrow S$, donde S es el estado inicial, es equivalente a la aceptación. (Esto ocurrirá sólo si el siguiente token de entrada es \$.) En los otros casos el nuevo estado se calcula del modo que se describe a continuación. Elimine la cadena α y todos sus estados correspondientes de la pila de análisis sintáctico. De la misma manera, retroceda en el DFA hasta el estado en el que comenzó la construcción de α. Por construcción, este estado debe contener un elemento LR(1) de la forma $[B \rightarrow \alpha.A\ \beta, b]$. Inserte A en la pila, e inserte el estado que contiene el elemento $[B \rightarrow \alpha A.\beta, b]$.
3. Si el siguiente token de entrada es de tal naturaleza que ninguno de los dos casos anteriores se aplique, se declara un error.

Como con los métodos anteriores, decimos que una gramática es una **gramática LR(1)** si la aplicación de las anteriores reglas de análisis sintáctico LR(1) general no producen ambigüedad. En particular, una gramática es LR(1) si y sólo si, para cualquier estado s, se satisfacen las siguientes dos condiciones:

1. Para cualquier elemento $[A \rightarrow \alpha.X\ \beta, a]$ en s, donde X es un terminal, no hay elemento en s de la forma $[B \rightarrow \beta., X]$ (de otro modo existe un conflicto de reducción por desplazamiento).
2. No existen dos elementos en s de la forma $[A \rightarrow \alpha., a]$ y $[B \rightarrow \beta., a]$ (de otra manera, existe un conflicto de reducción-reducción).

También resaltamos que se puede construir una tabla de análisis sintáctico a partir del DFA de conjuntos de elementos LR(1) que expresan el algoritmo de análisis sintáctico LR(1) general. Esta tabla tiene exactamente la misma forma que la tabla para los analizadores SLR(1), como se muestra en el ejemplo siguiente.

Ejemplo 5.15

En la tabla 5.10 damos la tabla de análisis sintáctico LR(1) general para la gramática del ejemplo 5.14, la cual puede construirse fácilmente a partir del DFA de la figura 5.7. En la tabla empleamos la siguiente numeración para las reglas gramaticales en las acciones de reducción:

$$(1)\ A \rightarrow (\ A\ )$$
$$(2)\ A \rightarrow \mathtt{a}$$

De este modo, la entrada r2 en el estado 3 con búsqueda hacia delante \$ indica una reducción por medio de la regla $A \rightarrow \mathtt{a}$.

Como los pasos en el análisis de una cadena particular en el análisis sintáctico LR(1) general son precisamente como serían en el análisis SLR(1) o LR(0), omitimos un ejemplo de un análisis de esta clase. En los ejemplos subsiguientes en esta sección también omitiremos

Tabla 5.10
Tabla de análisis sintáctico LR(1) general para el ejemplo 5.14

Estado	Entrada				Ir a
	(	a	)	$	A
0	s2	s3			1
1				aceptar	
2	s5	s6			4
3				r2	
4			s7		
5	s5	s6			8
6			r2		
7				r1	
8			s9		
9			r1		

§

la construcción explícita de la tabla de análisis sintáctico, porque se puede obtener muy fácilmente a partir del DFA.

En la práctica, casi todas las gramáticas razonables de lenguajes de programación son LR(1), a menos que sean ambiguas. Naturalmente, se pueden construir ejemplos de gramáticas no ambiguas que no pasan la prueba para ser LR(1), pero aquí no lo haremos (véanse los ejercicios). Tales ejemplos tienden a ser artificiales y por lo regular se pueden evitar en situaciones prácticas. De hecho, un lenguaje de programación rara vez necesita el poder que proporciona el análisis sintáctico LR(1) general. En realidad, el ejemplo que utilizamos para presentar el análisis sintáctico LR(1) (ejemplo 5.14) ya es una gramática LR(0) (y, por lo tanto, también SLR(1)).

El ejemplo siguiente muestra que el análisis sintáctico LR(1) general resuelve el problema de búsqueda hacia delante para la gramática del ejemplo 5.13 que no pasó la prueba para ser SLR(1).

Ejemplo 5.16

La gramática del ejemplo 5.13 en forma simplificada es como se presenta a continuación:

$$S \rightarrow \texttt{id} \mid V \texttt{:=} E$$
$$V \rightarrow \texttt{id}$$
$$E \rightarrow V \mid \texttt{n}$$

Construimos el DFA de conjuntos de elementos LR(1) para esta gramática. El estado inicial es la cerradura del elemento $[S' \rightarrow .S, \$]$. Contiene los elementos $[S \rightarrow .\texttt{id}, \$]$ y $[S \rightarrow .V \texttt{:=} E, \$]$. Este último elemento también da origen al elemento de cerradura $[V \rightarrow .\texttt{id}, \texttt{:=}]$, puesto que $S \rightarrow .V \texttt{:=} E$ indica que se puede reconocer una V, pero sólo si está seguida por un token de asignación. De este modo, el token `:=` aparece como la búsqueda hacia delante del elemento inicial $V \rightarrow .\texttt{id}$. En resumen, el estado inicial se compone de los siguientes elementos LR(1):

Estado 0: $[S' \rightarrow .S, \$]$
$[S \rightarrow .\texttt{id}, \$$
$[S \rightarrow .V \texttt{:=} E, \$]$
$[V \rightarrow .\texttt{id}, \texttt{:=}]$

A partir de este estado existe una transición con S al estado de un elemento

Estado 1: $[S' \rightarrow S\,.,\ \$]$

y una transición con `id` al estado de dos elementos

Estado 2: $[S \rightarrow .\,\mathtt{id},\ \$]$
$[V \rightarrow .\,\mathtt{id},\ :=]$

También existe una transición con V del estado 0 al estado de un elemento

Estado 3: $[S \rightarrow V\ .\ :=\ E\ ,\ \$]$

Desde los estados 1 y 2 no hay transiciones, pero existe una transición del estado 3 con `:=` a la cerradura del elemento $[S \rightarrow V\ :=.\ E,\ \$]$. Como E no tiene símbolos siguiéndolo en este elemento, esta cerradura incluye los elementos $[E \rightarrow .V,\ \$]$ y $[E \rightarrow .\,\mathtt{n},\ \$]$. Finalmente, el elemento $[E \rightarrow .V,\ \$]$ conduce al elemento de cerradura $[V \rightarrow .\,\mathtt{id},\ \$]$, ya que V tampoco tiene símbolos siguiéndolo en este caso. (Compare esto con la situación en el estado 0, donde una V estaba seguida por una asignación. Aquí V *no puede* estar seguida por una asignación, debido a que ya se vio un token de asignación.) El estado completo es entonces

Estado 4: $[S \rightarrow V\ :=.\ E\ ,\ \$]$
$[E \rightarrow .V\ ,\ \$]$
$[E \rightarrow .\,\mathtt{n},\ \$]$
$[V \rightarrow .\,\mathtt{id},\ \$]$

Los estados y las transiciones restantes son fáciles de construir, y las dejamos para que el lector las efectúe. El DFA completo de los conjuntos de elementos LR(1) se muestra en la figura 5.8.

Ahora consideremos el estado 2. Éste era el estado que daba origen al conflicto del análisis sintáctico SLR(1). Los elementos LR(1) distinguen claramente las dos reducciones mediante sus búsquedas hacia delante: reducción mediante $S \rightarrow \mathtt{id}$ con \$ y mediante $V \rightarrow \mathtt{id}$ con `:=`. Por consiguiente, esta gramática es LR(1).

Figura 5.8
El DFA de conjuntos de elementos LR(1) para el ejemplo 5.16

§

5.4.3 Análisis sintáctico LALR(1)

El análisis sintáctico LALR(1) está basado en la observación de que, en muchos casos, el tamaño del DFA de conjuntos de elementos LR(1) se debe en parte a la existencia de muchos estados diferentes que tienen el mismo conjunto de primeros componentes en sus elementos (los elementos LR(0)), mientras que difieren sólo en sus segundos componentes (los símbolos de búsqueda hacia delante). Por ejemplo, el DFA de los elementos LR(1) de la figura 5.7 tiene 10 estados, mientras que el DFA correspondiente de los elementos LR(0) (figura 5.5) tiene sólo 6. Efectivamente, en la figura 5.7 cada estado de los pares de estados 2 y 5, 4 y 8, 7 y 9, 3 y 6, difieren del otro sólo en los componentes de búsqueda hacia delante de sus elementos. Considere, por ejemplo, a los estados 2 y 5. Estos dos estados se distinguen sólo en su primer elemento, y únicamente en la búsqueda hacia delante de tal elemento: el estado 2 tiene el primer elemento [$A \rightarrow$ (. A), \$], con \$ como búsqueda hacia delante, mientras que el estado 5 tiene el primer elemento [$A \rightarrow$ (. A),)], con) como búsqueda hacia delante.

El algoritmo de análisis sintáctico LALR(1) expresa el hecho de que tiene sentido identificar todos esos estados y combinar sus búsquedas hacia delante. Al hacerlo así, siempre debemos finalizar con un DFA que sea idéntico al DFA de los elementos LR(0), excepto si cada estado se compone de elementos con conjuntos de búsqueda hacia delante. En el caso de elementos completos estos conjuntos de búsqueda hacia delante son frecuentemente más pequeños que los correspondientes conjuntos Siguiente. De este modo, el análisis sintáctico LALR(1) retiene algunos de los beneficios del análisis LR(1), mientras que mantiene el tamaño más pequeño del DFA de los elementos LR(0).

Formalmente, el **núcleo** de un estado del DFA de elementos LR(1) es el conjunto de elementos LR(0) que se compone de los primeros componentes de todos los elementos LR(1) en el estado. Puesto que la construcción del DFA de los elementos LR(1) utiliza transiciones que son las mismas que en la construcción del DFA de los elementos LR(0), excepto por sus efectos sobre las partes de búsqueda hacia delante de los elementos, obtenemos los siguientes dos hechos, que forman la base para la construcción del análisis sintáctico LALR(1).

PRIMER PRINCIPIO DEL ANÁLISIS SINTÁCTICO LALR(1)

El núcleo de un estado del DFA de elementos LR(1) es un estado del DFA de elementos LR(0).

SEGUNDO PRINCIPIO DEL ANÁLISIS SINTÁCTICO LALR(1)

Dados dos estados s_1 y s_2 del DFA de elementos LR(1) que tengan el mismo núcleo, suponga que hay una transición con el símbolo X desde s_1 hasta un estado t_1. Entonces existe también una transición con X del estado s_2 al estado t_2, y los estados t_1 y t_2 tienen el mismo núcleo.

Tomados juntos, estos dos principios nos permiten construir el **DFA de los elementos LALR(1)**, el cual se construye a partir del DFA de elementos LR(1) al identificar todos los estados que tienen el mismo núcleo y formar la unión de los símbolos de búsqueda hacia delante para cada elemento LR(0). De este modo, cada elemento LALR(1) en este DFA tendrá un elemento LR(0) como su primer componente y un conjunto de tokens de búsqueda hacia delante como su segundo componente.[5] En ejemplos posteriores denotaremos las búsquedas hacia delante múltiples escribiendo una / entre ellas. Así, el elemento LALR(1) [$A \rightarrow \alpha.\beta$, a / b / c] tiene un conjunto de búsqueda hacia delante compuesto de los símbolos a, b y c.

5. Los DFA de elementos LR(1), de hecho, también podrían utilizar conjuntos de símbolos de búsqueda hacia delante para representar elementos múltiples en el mismo estado en que comparten sus primeros componentes, pero consideramos conveniente emplear esta representación para la construcción LALR(1), donde es más apropiado.

Proporcionaremos un ejemplo para demostrar esta construcción.

Ejemplo 5.17

Considere la gramática del ejemplo 5.14, cuyo DFA de elementos LR(1) se dan en la figura 5.7. La identificación de los estados 2 y 5, 4 y 8, 7 y 9, 3 y 6 proporciona el DFA de los elementos LALR(1) en la figura 5.9. Conservamos en esta figura la numeración de los estados 2, 3, 4 y 7, y agregamos las búsquedas hacia delante de los estados 5, 6, 8 y 9. Como esperábamos, este DFA es idéntico al DFA de los elementos LR(0) (figura 5.5), excepto por las búsquedas hacia delante.

Figura 5.9
El DFA de conjuntos de elementos LALR(1) para el ejemplo 5.17

§

El algoritmo para el análisis sintáctico LALR(1) utilizando el DFA condensado de los elementos LALR(1) es idéntico al algoritmo de análisis sintáctico LR(1) general descrito en la sección anterior. Como antes, denominamos **gramática LALR(1)** a una gramática si no surgen conflictos de análisis en el algoritmo de análisis sintáctico LALR(1). Para la construcción LALR(1) es posible crear conflictos de análisis que no existen en el análisis sintáctico LR(1) general, pero esto rara vez ocurre en la práctica. En realidad, si una gramática es LR(1), entonces la tabla de análisis sintáctico LALR(1) no puede tener ningún conflicto de reducción por desplazamiento; no obstante, puede haber conflictos de reducción-reducción (véase los ejercicios). Sin embargo, si una gramática es SLR(1), entonces por supuesto es LALR(1), y los analizadores LALR(1) a menudo funcionan tan bien como los analizadores LR(1) generales en la eliminación de conflictos típicos que se presentan en el análisis sintáctico SLR(1). Por ejemplo, la gramática no SLR(1) del ejemplo 5.16 es LALR(1): el DFA de los elementos LR(1) de la figura 5.8 es también el DFA de los elementos LALR(1). Si, como en este ejemplo, la gramática ya es LALR(1), la única consecuencia de utilizar el análisis sintáctico LALR(1) en vez del análisis sintáctico LR general es que, en presencia de errores, pueden hacerse algunas reducciones espurias antes que se declare un error. Por ejemplo, en la figura 5.9 observamos que, dada la cadena de entrada errónea `a)`, un analizador sintáctico LALR(1) efectuará la reducción $A \rightarrow$ `a` antes de declarar el error, mientras que un analizador sintáctico LR(1) general declarará el error inmediatamente después de un desplazamiento del token `a`.

La combinación de estados LR(1) para formar el DFA de los elementos LALR(1) resuelve el problema de las tablas de análisis sintáctico grandes, pero todavía requiere del DFA completo de elementos LR(1) para calcularse. De hecho, es posible calcular el DFA de los elementos LALR(1) directamente del DFA de elementos LR(0) a través de un proceso de **propagación de búsquedas hacia delante**. Aunque no describiremos este proceso

de manera formal, es formativo ver cómo se puede hacer esto de manera relativamente fácil.

Considere el DFA LALR(1) de la figura 5.9. Comenzamos construyendo búsquedas hacia delante al agregar el marcador de finalización $ a la búsqueda hacia delante del elemento de aumento $A' \rightarrow .A$ en el estado 0. (Se dice que la búsqueda hacia delante $ se **genera espontáneamente**.) Entonces, por las reglas de la cerradura ε, el $ se propaga a los dos elementos de cerradura (la A en el lado derecho del elemento de núcleo $A' \rightarrow .A$ es seguido por la cadena vacía). Al seguir las tres transiciones desde el estado 0, el $ se propaga a los elementos de núcleo de los estados 1, 3 y 2. Continuando con el estado 2, los elementos de cerradura obtienen la búsqueda hacia delante), nuevamente por generación espontánea (debido a que A en el lado derecho del elemento de núcleo $A \rightarrow (.A)$ viene antes de un paréntesis derecho). Ahora la transición en **a** al estado 3 causa que el) se propague a la búsqueda hacia delante del elemento en ese estado. También, la transición con (del estado 2 hacia sí mismo causa que el) se propague a la búsqueda hacia delante del elemento de núcleo (aquí se explica por qué el elemento de núcleo tiene tanto $ como) en su conjunto de búsqueda hacia delante). Ahora el conjunto de búsqueda hacia delante $/) se propaga al estado 4 y posteriormente al estado 7. De este modo, a través de este proceso, obtuvimos el DFA de elementos LALR(1) de la figura 5.9 directamente del DFA de elementos LR(0).

5.5 Yacc: UN GENERADOR DE ANALIZADORES SINTÁCTICOS LALR(1)

Un **generador de analizadores sintácticos** es un programa que toma como su entrada una especificación de la sintaxis de un lenguaje en alguna forma, y produce como su salida un procedimiento de análisis sintáctico para ese lenguaje. Históricamente, los generadores de analizadores sintácticos fueron llamados **compiladores de compilador**, debido a que todos los pasos de compilación eran realizados de manera tradicional como acciones incluidas dentro del analizador sintáctico. La visión moderna es considerar al analizador como sólo una parte del proceso de compilación, de modo que este término se volvió obsoleto. Un generador de analizadores sintácticos ampliamente utilizado que incorpora el algoritmo de análisis sintáctico LALR(1) se conoce como **Yacc** ("Yet Another Compiler-Compiler", es decir, "otro compilador de compilador más" en inglés). En esta sección daremos una perspectiva general de Yacc, y en la sección siguiente lo utilizaremos para desarrollar un analizador sintáctico para el lenguaje TINY. Debido a que existen varias implementaciones diferentes de Yacc, además de varias versiones del dominio público comúnmente denominadas **Bison**,[6] existen numerosas variaciones en los detalles de su operación, las cuales pueden diferir en alguna forma de la versión que presentamos aquí.[7]

5.5.1 Fundamentos de Yacc

Yacc toma un archivo de especificación (por lo regular con un sufijo `.y`) y produce un archivo de salida compuesto del código fuente en C para el analizador sintáctico (por lo general en un archivo denominado `y.tab.c` o `ytab.c` o, más recientemente <nombre de archivo>`.tab.c`, donde <nombre de archivo>`.y` es el archivo de entrada). Un archivo de especificación de Yacc tiene el formato básico

6. Una versión popular, Gnu Bison, forma parte del software Gnu que distribuye la Free Software Foundation; véase la sección de notas y referencias.

7. Efectivamente, utilizamos varias versiones diferentes para generar los ejemplos posteriores.

```
{definiciones}
%%
{reglas}
%%
{rutinas auxiliares}
```

De este modo, existen tres secciones (la sección de definiciones, la sección de reglas y la sección de rutinas auxiliares) separadas mediante líneas que contienen doble signo de porcentaje.

La sección de definiciones contiene información acerca de los tokens, tipos de datos y reglas gramaticales que Yacc necesita para construir el analizador sintáctico. También incluye cualquier código en C que debería ir directamente en el archivo de salida a su inicio (sobre todo directivas **#include** de otros archivos de código fuente). Esta sección del archivo de especificación puede estar vacía.

La sección de reglas contiene reglas gramaticales en una forma BNF modificada, junto con acciones en código C que se ejecutarán siempre que se reconozca la regla gramatical asociada (es decir, se emplee en una reducción, de acuerdo con el algoritmo de análisis sintáctico LALR(1)). Las convenciones de metasímbolos utilizadas en las reglas gramaticales son de la manera siguiente. Como es habitual, la barra vertical se utiliza para las alternativas (las alternativas también se pueden escribir por separado). El símbolo de flecha → que hemos empleado para separar los lados izquierdo y derecho de una regla gramatical se reemplaza en Yacc por un signo de dos puntos. También un signo de punto y coma debe finalizar cada regla gramatical.

La tercera sección, de rutinas auxiliares, contiene declaraciones de procedimientos y funciones que de otra manera pueden no estar disponibles a través de archivos **#include** y que son necesarias para completar el analizador sintáctico y/o el compilador. Esta sección puede estar vacía, y si éste es el caso se puede omitir el segundo metasímbolo de porcentaje doble del archivo de especificación. De esta manera, un archivo de especificación mínimo de Yacc consistiría sólo de **%%** seguidos por reglas gramaticales y acciones (las acciones también se pueden omitir si sólo tratamos de analizar la gramática, un tema que se trata posteriormente en esta sección).

Yacc también permite insertar comentarios al estilo de C en el archivo de especificación en cualquier punto donde no interfieran con el formato básico.

Explicaremos el contenido del archivo de especificación de Yacc con más detalle utilizando un ejemplo simple. El ejemplo es una calculadora de expresiones aritméticas enteras simples con la gramática

$$
\begin{aligned}
&\textit{exp} \rightarrow \textit{exp opsuma term} \mid \textit{term}\\
&\textit{opsuma} \rightarrow + \mid -\\
&\textit{term} \rightarrow \textit{term opmult factor} \mid \textit{factor}\\
&\textit{opmult} \rightarrow *\\
&\text{factor} \rightarrow (\ \textit{exp}\ ) \mid \textbf{\textit{número}}
\end{aligned}
$$

Esta gramática se utilizó ampliamente como ejemplo en los capítulos anteriores. En la sección 4.1.2 desarrollamos un programa calculador descendente recursivo para esta gramática. Una especificación de Yacc completamente equivalente se proporciona en la figura 5.10. Analizaremos el contenido de cada una de las tres secciones de esta especificación en su momento.

En la sección de definiciones de la figura 5.10 existen dos elementos. El primero consta del código que se insertará al principio de la salida de Yacc. Este código se compone de dos directivas típicas **#include** y se destaca de otras declaraciones de Yacc en esta sección mediante los delimitadores **%{** y **%}**. (Advierta que los signos de porcentaje vienen antes que las llaves.) El segundo elemento en la sección de definiciones es una declaración del token **NÚMERO**, el cual representa una secuencia de dígitos.

Figura 5.10
Definición de Yacc para un programa calculador simple

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token NUMBER

%%

command : exp       { printf("%d\n",$1);}
        ; /* permite la impresión del resultado */

exp     : exp '+' term  {$$ = $1 + $3;}
        | exp '-' term  {$$ = $1 - $3;}
        | term    {$$ = $1;}
        ;

term    : term '*' factor {$$ = $1 * $3;}
        | factor  {$$ = $1;}
        ;

factor        : NUMBER          {$$ = $1;}
              | '(' exp ')'    {$$ = $2;}
              ;
%%

main()
{ return yyparse();
}

int yylex(void)
{ int c;
  while((c = getchar()) == ' ');
  /* elimina blancos */
  if ( isdigit(c) ) {
    ungetc(c,stdin);
    scanf("%d",&yylval);
    return(NUMBER);
  }
  if (c == '\n') return 0;
  /* hace que se detenga el análisis sintáctico */
  return(c);
}

void yyerror(char * s)
{ fprintf(stderr,"%s\n",s);
} /* permite la impresión de un mensaje de error */
```

Yacc tiene dos maneras de reconocer tokens. En primer lugar, cualquier carácter encerrado entre comillas simples en una regla gramatical será reconocido como él mismo. Así, los tokens de un solo carácter se pueden incluir directamente en reglas gramaticales de esta manera, como los tokens de operador `+`, `-` y `*` de la figura 5.10 (así como los tokens de paréntesis). En segundo lugar, los tokens simbólicos se pueden declarar en una declaración `%token` de Yacc, como el token **NUMBER** ("`número`") de la figura 5.10. A tales tokens se les asigna un valor numérico mediante Yacc que no entre en conflicto con ningún valor de carácter. Típicamente, Yacc comienza asignando valores de token con el número 258. Yacc inserta estas definiciones de token como sentencias `#define` en el código de salida. De este modo, en el archivo de salida probablemente encontraríamos la línea

```
#define NUMBER 258
```

como respuesta de Yacc a la declaración `%token` **NUMBER** en el archivo de especificación. Yacc insiste en definir todos los tokens simbólicos por sí mismo, en vez de importar una definición de alguna otra parte. Sin embargo, es posible especificar el valor numérico que se asignará al token escribiendo un valor después del nombre del token en la declaración de token. Por ejemplo, al escribir

```
%token NUMBER 18
```

se asigna a **NUMBER** el valor numérico 18 (en lugar de 258).

En la sección de reglas de la figura 5.10 observamos las reglas para los no terminales *exp*, *term* y *factor*. Como también queremos imprimir el valor de una expresión, tenemos una regla adicional, que denominaremos *command*, y a la cual asociamos la acción de impresión. Debido a que la regla para *command* se enumera primero, *command* se toma como el símbolo inicial de la gramática. De manera alternativa, podríamos haber incluido la línea

```
%start command
```

en la sección de definiciones, en cuyo caso no tendríamos que poner la regla para *command* primero.

Las acciones se especifican en Yacc al escribirlas como código en C real (dentro de llaves) con cada regla gramatical. De manera característica, el código de acción se ubica al final de cada selección de regla gramatical (pero antes de la barra vertical o del signo de punto y coma), aunque también es posible escribir **acciones incrustadas** en una selección (comentaremos esto en breve). Al escribir las acciones podemos aprovecharnos de las **pseudovariables** de Yacc. Cuando se reconoce una regla gramatical, cada símbolo en la regla posee un valor, que se supone es un entero a menos que sea cambiado por el programador (posteriormente veremos cómo hacer esto). Esos valores se conservan en una **pila de valor** mediante Yacc, la cual se mantiene paralela a la pila de análisis sintáctico. Se puede hacer referencia a cada valor de símbolo en la pila utilizando una pseudovariable que comience con un signo de $. `$$` representa el valor del no terminal que se acaba de reconocer, es decir, el símbolo en el lado izquierdo de la regla gramatical. Las pseudovariables `$1`, `$2`, `$3`, y así sucesivamente, representan los valores de cada símbolo en sucesión en el lado derecho de la regla gramatical. De esta manera, en la figura 5.10 la regla gramatical y acción

```
exp    : exp '+' term {$$ = $1 + $3;}
```

significan que cuando reconocemos la regla *exp* → *exp* + *term* asignamos el valor de la *exp* a la izquierda como la suma de los valores de la *exp* y el *term* de la derecha.

Todos los no terminales obtienen valores mediante tales acciones suministradas por el usuario. Los tokens también pueden tener valores asignados, pero esto se hace durante el proceso de análisis léxico. Yacc supone que el valor de un token se asigna a la variable **yylval**, la cual se define de manera interna por Yacc, y debe ser asignada cuando se reconoce el token. De este modo, en la regla gramatical y acción

```
factor    : NUMBER    {$$ = $1;}
```

el valor **$1** se refiere al valor del token **NUMBER** que fue asignado previamente a **yylval** cuando se reconoció el token.

La tercera sección de la figura 5.10 (la sección de rutinas auxiliares) contiene la definición de tres procedimientos. La primera es una definición de **main**, que está incluida de manera que la salida resultante de Yacc se pueda compilar directamente para un programa ejecutable. El procedimiento **main** llama a **yyparse**, que es el nombre que Yacc proporciona al procedimiento de análisis sintáctico que produce. Este procedimiento se declara para devolver un valor entero, que es siempre 0 si el análisis tiene éxito, y 1 si no es así (es decir, si se ha presentado un error y no se ha realizado recuperación de errores). El procedimiento **yyparse** generado por Yacc llama a su vez a un procedimiento de analizador léxico, que se supone tiene el nombre **yylex** para propósitos de compatibilidad con el generador de analizadores léxicos Lex (véase el capítulo 2). De esta manera, la especificación Yacc de la figura 5.10 también incluye una definición para **yylex**. En esta situación particular el procedimiento **yylex** es muy simple. Todo lo que necesita hacer es devolver el siguiente carácter que no sea blanco, a menos que este carácter sea un dígito, en cuyo caso debe reconocer el token simple de carácter múltiple **NUMBER** y devolver su valor en la variable **yylval**. La única excepción a esto es cuando el analizador léxico ha llegado al final de la entrada. En este caso, el final de la entrada se indica mediante un carácter de "retorno de línea" o "nueva línea" ('**\n**' en C), ya que estamos suponiendo que una expresión se introduce en una sola línea. Yacc espera que el final de la entrada sea señalado mediante una devolución del valor nulo 0 por **yylex** (nuevamente, esto es una convención compartida con Lex). Finalmente, se define un procedimiento **yyerror**. Yacc utiliza este procedimiento para imprimir un mensaje de error cuando se encuentra un error durante el análisis sintáctico (Yacc imprime específicamente la cadena "syntax error", pero este comportamiento puede ser modificado por el usuario).

5.5.2 Opciones de Yacc

Por lo regular habrá muchos procedimientos auxiliares a los que Yacc necesitará tener acceso además de **yylex** y **yyerror**, y éstos a menudo se colocan en archivos externos en lugar de directamente en el archivo de especificación Yacc. Es fácil proporcionar acceso a Yacc a estos procedimientos escribiendo archivos de encabezado apropiados y colocando directivas **#include** en la sección de definiciones de la especificación Yacc. Es más difícil dejar disponibles las definiciones específicas de Yacc para otros archivos. Esto es particularmente cierto para las definiciones de token, las cuales, como hemos dicho, Yacc insiste en generar por sí mismo (en lugar de importarlas), pero que deben estar disponibles para muchas otras partes de un compilador (en particular el analizador léxico). Por esta razón, Yacc tiene una opción disponible que automáticamente producirá un archivo de cabecera que contiene esta información, la cual entonces se puede incluir en cualquier otro archivo que necesite las definiciones. Este archivo de cabecera por lo regular se denomina **y.tab.h** o bien **ytab.h**, y es producido con la opción **-d** (por el término archivo de cabecera en inglés: "heaDer file").

Por ejemplo, si la especificación Yacc de la figura 5.10 está contenida en el archivo **calc.y**, entonces el comando

```
yacc -d calc.y
```

producirá (además del archivo `y.tab.c`) el archivo `y.tab.h` (un nombre similar), cuyo contenido varía, pero por lo regular incluye cosas como la siguiente:

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define      NUMBER      258

extern YYSTYPE yylval;
```

(Describiremos el significado de `YYSTYPE` con más detalle en breve.) Con este archivo se puede colocar el código para `yylex` en un archivo diferente insertando la línea

```
#include y.tab.h
```

en ese archivo.[8]

Una segunda opción muy útil de Yacc es la **opción verbose (opción descriptiva)** activada mediante la señal `-v` en la línea de comando. Esta opción produce todavía otro archivo, con el nombre `y.output` (o un nombre semejante). Este archivo contiene una descripción textual de la tabla de análisis sintáctico LALR(1) que es utilizada por el analizador. La lectura de este archivo permite al usuario determinar exactamente qué acción tomará el analizador sintáctico generado por Yacc en cualquier situación, y ésta puede ser una manera muy efectiva para rastrear ambigüedades e imprecisiones en una gramática. En realidad, es una buena idea ejecutar Yacc con esta opción sobre la gramática por sí misma, antes de agregar las acciones o procedimientos auxiliares a la especificación, para asegurarse de que el analizador sintáctico generado por Yacc funcionará en realidad como se espera.

Como ejemplo considere la especificación Yacc de la figura 5.11. Ésta es sólo una versión escueta de la especificación Yacc de la figura 5.10. Ambas especificaciones generarán el mismo archivo de salida cuando Yacc se utilice con la opción verbose:

```
yacc -v calc.y
```

Figura 5.11
Una especificación en esqueleto Yacc para su uso con la opción -v

```
%token NUMBER
%%
command      : exp
             ;
exp          : exp '+' term
             | exp '-' term
             | term
             ;
term         : term '*' factor
             | factor
             ;
factor       : NUMBER
             | '(' exp ')'
             ;
```

8. Las versiones más antiguas de Yacc sólo pueden colocar las definiciones de token (y no las de `yylval`) en `y.tab.h`. Esto puede requerir cambios de dirección manual o reacomodo del código.

Un archivo típico **y.output** se lista en su totalidad para esta gramática en la figura 5.12.[9] Analizaremos la interpretación de este archivo en los párrafos siguientes.

El archivo de salida de Yacc se compone de un listado de todos los estados en el DFA, seguido por un resumen de estadísticas internas. Los estados se enumeran comenzando por el 0. Bajo cada estado el archivo de salida enumera primero los elementos de núcleo (los elementos de cerradura no se enumeran), luego las acciones correspondientes a diversas búsquedas hacia delante, y finalmente las acciones Ir a en varios no terminales. Específicamente, Yacc utiliza un carácter de subraya o guión bajo _ para marcar la posición distinguida en un elemento, en lugar del punto que hemos estado utilizando en este capítulo. Yacc utiliza el punto en cambio para indicar una opción predeterminada, o un token de búsqueda hacia delante "sin importancia" en la sección de acción de cada listado de estado.

Yacc comienza en el estado 0 exhibiendo el elemento inicial de la producción de aumento, que es siempre el único elemento de núcleo en el estado inicial del DFA. En el archivo de salida de nuestro ejemplo, este elemento se escribe como

```
$accept : _command $end
```

Esto corresponde al elemento *command'* → . *command* en nuestra propia terminología. Yacc proporciona al no terminal de aumento el nombre **$accept**. También lista el pseudotoken de final de entrada de manera explícita como **$end**.

Examinemos brevemente la sección de acción del estado 0, que sigue a la lista de elementos de núcleo:

```
NUMBER shift 5
( shift 6
. error
command goto 1
exp goto 2
term goto 3
factor goto 4
```

La lista anterior especifica que el DFA se desplaza al estado 5 con el token de búsqueda hacia delante **NUMBER**, se desplaza al estado 6 en el token de búsqueda hacia delante (, y declara error en todos los otros token de búsqueda hacia delante. También se enumeran cuatro transiciones Ir a, para utilizar durante las reducciones de los no terminales dados. Estas acciones están exactamente como aparecerían en una tabla de análisis sintáctico construida a mano empleando los métodos de este capítulo.

Consideremos ahora el estado 2, con el listado de salida

```
state 2
        command : exp_   (1)
        exp : exp_+ term
        exp : exp_- term

        + shift 7
        - shift 8
        . reduce 1
```

9. Las versiones más recientes de Bison producen un formato muy diferente en los archivos de salida, pero los contenidos son esencialmente los mismos.

Figura 5.12 (inicio)
Un archivo típico y.output generado por la especificación Yacc de la figura 5.10 utilizando la opción descriptiva "verbose".

```
state 0
        $accept : _command $end

        NUMBER  shift 5
        (  shift 6
        .  error

        command  goto 1
        exp  goto 2
        term  goto 3
        factor  goto 4

state 1
        $accept :  command_$end

        $end  accept
        .  error

state 2
        command :  exp_    (1)
        exp :  exp_+ term
        exp :  exp_- term

        +  shift 7
        -  shift 8
        .  reduce 1

state 3
        exp :  term_    (4)
        term :  term_* factor

        *  shift 9
        .  reduce 4

state 4
        term :  factor_    (6)

        .  reduce 6

state 5
        factor :  NUMBER_    (7)

        .  reduce 7

state 6
        factor :  (_exp )

        NUMBER  shift 5
        (  shift 6
        .  error

        exp  goto 10
        term  goto 3
        factor  goto 4

state 7
        exp :  exp +_term

        NUMBER  shift 5
        (  shift 6
        .  error

        term  goto 11
        factor  goto 4

state 8
        exp :  exp -_term

        NUMBER  shift 5
        (  shift 6
        .  error

        term  goto 12
        factor  goto 4

state 9
        term :  term *_factor

        NUMBER  shift 5
        (  shift 6
        .  error

        factor  goto 13
```

Figura 5.12 (conclusión) Un archivo típico `y.output` generado por la especificación Yacc de la figura 5.10 utilizando la opción "verbose"

```
state 10
     exp :  exp_+ term
     exp :  exp_- term
     factor :  ( exp_)

     +  shift 7
     -  shift 8
     )  shift 14
     .  error

state 11
     exp :  exp + term_    (2)
     term :  term_* factor

     *  shift 9
     .  reduce 2

state 12
     exp :  exp - term_    (3)
     term :  term_* factor

     *  shift 9
     .  reduce 3

state 13
     term :  term * factor_   (5)

     .  reduce 5

state 14
     factor :  ( exp )_    (8)

       .  reduce 8

8/127 terminals, 4/600 nonterminals
9/300 grammar rules, 15/1000 states
0 shift/reduce, 0 reduce/reduce conflicts reported
9/601 working sets used
memory: states, etc. 36/2000, parser 11/4000
9/601 distinct lookahead sets
6 extra closures
18 shift entries, 1 exceptions
8 goto entries
4 entries saved by goto default
Optimizer space used: input 50/2000, output 218/4000
218 table entries, 202 zero
maximum spread: 257, maximum offset: 43
```

Aquí el elemento de núcleo es un elemento completo, de modo que habrá una reducción mediante la selección de producción asociada en la sección de acción. Para recordarnos el número de la producción que se está utilizando para la reducción, Yacc muestra el número después del elemento completo. En este caso el número de producción es 1, y hay una acción **reduce 1**, la cual indica una reducción mediante la producción *command* → *exp*. Yacc siempre enumera las producciones en el orden en el que se exhiben en el archivo de especificación. En nuestro ejemplo existen ocho producciones (una para *command*, tres para *exp*, dos para *term* y dos para *factor*).

Advierta que la acción de reducción en este estado es una acción *predeterminada*: se realizará una reducción en cualquier búsqueda hacia delante diferente de + y -. Aquí Yacc difiere de un analizador sintáctico LALR(1) puro (e incluso de un analizador SLR(1)) en que no se hace ningún intento por verificar la legalidad de las búsquedas hacia delante en las reducciones (más que para decidir entre varias reducciones). Un analizador sintáctico Yacc generalmente hace varias reducciones sobre errores antes de declarar finalmente un error (lo cual debe hacer en algún momento antes de que ocurra otro desplazamiento). Esto significa que los mensajes de error pueden no ser tan informativos como deberían, pero la tabla de análisis sintáctico se vuelve considerablemente más sencilla, puesto que se presentan menos casos (este punto se volverá a comentar en la sección 5.7).

Concluiremos este ejemplo mediante la construcción de una tabla de análisis sintáctico del archivo de salida de Yacc, exactamente como lo hubiéramos escrito a mano al principio de este capítulo. La tabla de análisis sintáctico aparece en la tabla 5.11.

Tabla 5.11
Tabla de análisis sintáctico correspondiente a la salida Yacc de la figura 5.12

Estado	Entrada							Ir a			
	NÚMERO	(	+	-	*	)	$	*command*	*exp*	*term*	*factor*
0	s5	s6						1	2	3	4
1							aceptar				
2	r1	r1	s7	s8	r1	r1	r1				
3	r4	r4	r4	r4	s9	r4	r4				
4	r6	r6	r6	r6	r6	r6	r6				
5	r7	r7	r7	r7	r7	r7	r7				
6	s5	s6							10	3	4
7	s5	s6								11	4
8	s5	s6								12	4
9	s5	s6									13
10			s7	s8		s14					
11	r2	r2	r2	r2	s9	r2	r2				
12	r3	r3	r3	r3	s9	r3	r3				
13	r5	r5	r5	r5	r5	r5	r5				
14	r8	r8	r8	r8	r8	r8	r8				

5.5.3 Conflictos de análisis sintáctico y reglas de eliminación de ambigüedad

Uno de los usos importantes de la opción verbose es investigar los conflictos de análisis sintáctico, de los que Yacc informará en el archivo `y.output`. Yacc tiene reglas de eliminación de ambigüedad integradas en él mismo que le permitirán producir un analizador sintáctico incluso en la presencia de conflictos de análisis (por consiguiente, hasta para gramáticas ambiguas). Con frecuencia estas reglas de eliminación de ambigüedad hacen bien su trabajo, pero en ocasiones no es así. Al examinar el archivo `y.output` se permite al usuario determinar cuáles son los conflictos de análisis sintáctico, y si el analizador sintáctico producido por Yacc los resolverá de manera correcta.

En el ejemplo de la figura 5.10 no había conflictos de análisis sintáctico, y Yacc informó de este hecho en la información resumida al final del archivo de salida como

```
0 shift/reduce, 0 reduce/reduce conflicts reported
```

Un ejemplo más interesante es la gramática ambigua a causa del else del ejemplo 5.12 (página 213). En la tabla 5.9 (página 215) proporcionamos la tabla de análisis sintáctico SLR(1) para esta gramática, con el conflicto de reducción por desplazamiento en el estado 5 eliminado al preferir el desplazamiento a la reducción (esto corresponde a la regla de eliminación de ambigüedad de anidación más próxima). Yacc informa de la ambigüedad

exactamente en los mismos términos, y resuelve la ambigüedad mediante la misma regla de eliminación de ambigüedad. En realidad, la tabla de análisis sintáctico mostrada por Yacc es idéntica a la tabla 5.9, excepto por las reducciones predeterminadas que Yacc inserta en las entradas de error. Por ejemplo, Yacc informa de las acciones del estado 5 de la manera siguiente en el archivo **y.output** (los tokens se definieron en letras mayúsculas en el archivo de especificación para evitar conflictos con las palabras reservadas del lenguaje C):

```
5: shift/reduce conflict (shift 6, red'n 3) on ELSE
state 5
        I : IF S_   (3)
        I : IF S_ELSE S

        ELSE shift 6
        . reduce 3
```

En la información resumida Yacc también informa del conflicto simple de reducción por desplazamiento:

```
1 shift/reduce, 0 reduce/reduce conflicts reported
```

En el caso de un conflicto de reducción-reducción, Yacc elimina la ambigüedad al preferir la reducción por la regla gramatical que se mostró primero en el archivo de especificación. Esto es más probable que sea un error en la gramática, aunque también puede resultar en un analizador sintáctico correcto. Ofrecemos el siguiente ejemplo simple.

Ejemplo 5.18

Consideremos la gramática siguiente:

$$S \rightarrow A \mid B$$
$$A \rightarrow a$$
$$B \rightarrow a$$

Ésta es una gramática ambigua, ya que la cadena legal simple a tiene las dos derivaciones $S \Rightarrow A \Rightarrow a$ y $S \Rightarrow B \Rightarrow a$. El archivo **y.output** completo para esta gramática se proporciona en la figura 5. 13. Observe el conflicto de reducción-reducción en el estado 4, el cual es resuelto al preferir la regla $A \rightarrow a$ en lugar de la regla $B \rightarrow a$. Esto da como resultado que nunca se utilice esta última regla en una reducción (lo que indica claramente un problema con la gramática) y Yacc informa de este hecho al final con la línea

```
Rule not reduced:  B : a
```

igura 5.13
rchivo de salida de Yacc
ara la gramática del
jemplo 5.18

```
state 0
        $accept : _S $end

        a   shift 4
        .   error

        S   goto 1
        A   goto 2
        B   goto 3
```

Figura 5.13 continuación
Archivo de salida de Yacc para la gramática del ejemplo 5.18

```
state 1
        $accept :   S_$end

        $end   accept
        .   error

state 2
        S :   A_     (1)

        .   reduce 1

state 3
        S :   B_     (2)

        .   reduce 2

4: reduce/reduce conflict (red'ns 3 and 4 ) on $end
state 4
        A :   a_     (3)
        B :   a_     (4)

        .   reduce 3

Rule not reduced:     B :   a

3/127 terminals, 3/600 nonterminals
5/300 grammar rules, 5/1000 states
0 shift/reduce, 1 reduce/reduce conflicts reported
...
```

§

Yacc tiene, además de la regla de eliminación de ambigüedad ya mencionada, varios mecanismos ex profeso para la especificación de la asociatividad y precedencia de operadores separadamente de una gramática que de otro modo es ambigua. Esto tiene diversas ventajas. En primer lugar, la gramática no necesita contener construcciones explícitas que especifiquen asociatividad y precedencia, y esto significa que la gramática puede ser más concisa y más simple. En segundo lugar, la tabla de análisis sintáctico asociada también puede ser más pequeña y el analizador sintáctico resultante, más eficiente.

Considere, por ejemplo, la especificación Yacc de la figura 5.14. En esa figura la gramática está escrita en forma ambigua sin precedencia o asociatividad de los operadores. En cambio, la precedencia y asociatividad de los operadores se especifica en la sección de definiciones al escribir las líneas

```
%left '+' '-'
%left '*'
```

Estas líneas le indican a Yacc que los operadores + y - tienen la misma precedencia y son asociativos por la izquierda, y que el operador * es asociativo por la izquierda y tiene una

precedencia más alta que + y - (ya que se muestra después de estos operadores en las declaraciones). Las otras posibles especificaciones de operador de Yacc son **%right** y **%nonassoc** ("nonassoc" significa que no se permiten operadores repetidos en el mismo nivel).

Figura 5.14
Especificación de Yacc para una calculadora simple con gramática ambigua y reglas de precedencia y asociatividad para operadores

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token NUMBER

%left '+' '-'
%left '*'

%%

command : exp        { printf("%d\n",$1);}
        ;

exp     : NUMBER          {$$ = $1;}
        | exp '+' exp     {$$ = $1 + $3;}
        | exp '-' exp     {$$ = $1 - $3;}
        | exp '*' exp     {$$ = $1 * $3;}
        | '(' exp ')'     {$$ = $2;}
        ;

%%
/* auxiliary procedure declarations as in Figure 5.10 */
```

5.5.4 Rastreo de la ejecución de un analizador sintáctico Yacc

Además de la opción verbose que exhibe la tabla de análisis sintáctico en el archivo **y.output**, también es posible obtener un analizador sintáctico generado por Yacc para imprimir un rastreo de su ejecución, incluyendo una descripción de la pila de análisis sintáctico y acciones del analizador semejantes a las descripciones que ya dimos antes en este capítulo. Esto se hace compilando **y.tab.c** con el símbolo **YYDEBUG** definido (utilizando la opción de compilador **-DYYDEBUG**, por ejemplo) y estableciendo la variable entera **yydebug** de Yacc a 1 en el punto donde se desea la información de rastreo. Por ejemplo, al agregarse las líneas siguientes al principio del procedimiento **main** de la figura 5.10,

```
extern int yydebug;
yydebug = 1;
```

se provocará que el analizador sintáctico produzca una salida parecida a la de la figura 5.15, y dará la expresión 2+3 como entrada. Invitamos al lector a que construya un rastreo a mano de las acciones del analizador sintáctico utilizando la tabla de análisis sintáctico de la tabla 5.11 y lo compare con esta salida.

Figura 5.15
Rastreo de salida utilizando yydebug para el analizador sintáctico de Yacc generado por la figura 5.10, dada la entrada 2+3

```
Starting parse
Entering state 0
Input: 2+3
Next token is NUMBER
Shifting token NUMBER, Entering state 5
Reducing via rule 7, NUMBER -> factor
state stack now 0
Entering state 4
Reducing via rule 6, factor -> term
state stack now 0
Entering state 3
Next token is '+'
Reducing via rule 4, term -> exp
state stack now 0
Entering state 2
Next token is '+'
Shifting token '+', Entering state 7
Next token is NUMBER
Shifting token NUMBER, Entering state 5
Reducing via rule 7, NUMBER -> factor
state stack now 0 2 7
Entering state 4
Reducing via rule 6, factor -> term
state stack now 0 2 7
Entering state 11
Now at end of input.
Reducing via rule 2, exp '+' term -> exp
state stack now 0
Entering state 2
Now at end of input.
Reducing via rule 1, exp -> command
5
state stack now 0
Entering state 1
Now at end of input.
```

5.5.5 Tipos de valor arbitrario en Yacc

En la figura 5.10 especificamos las acciones de una calculadora empleando las pseudovariables de Yacc asociadas con cada símbolo gramatical en una regla gramatical. Por ejemplo, escribimos `$$ = $1 + $3` para establecer el valor de una expresión como la suma de los

valores de sus dos subexpresiones (en las posiciones 1 y 3 en el lado derecho de la regla gramatical *exp* → *exp* + *term*). Esto está bien mientras que los valores con los que estamos tratando sean enteros, puesto que el tipo predeterminado de Yacc de estos valores es siempre entero. Sin embargo, es inadecuado si, por ejemplo, queremos una calculadora para hacer cálculos con valores de punto flotante. En este caso debemos incluir una redefinición del tipo de valor de las pseudovariables de Yacc en el archivo de especificación. Este tipo de datos siempre se define en Yacc mediante el símbolo de preprocesador en C **YYSTYPE**. La redefinición de este símbolo modificará apropiadamente el tipo de la pila de valores de Yacc. De este modo, si queremos una calculadora que haga cálculos de valores en punto flotante, debemos agregar la línea

```
#define YYSTYPE double
```

dentro de los delimitadores **%{...%}** en la sección de definiciones del archivo de especificación de Yacc.

En situaciones más complicadas podemos necesitar valores diferentes para reglas gramaticales diferentes. Por ejemplo, supongamos que queremos separar el reconocimiento de una selección de operadores de la regla que calcula con estos operadores, como en las reglas

$$exp \rightarrow exp\ opsuma\ term \mid term$$
$$opsuma \rightarrow + \mid -$$

(Éstas eran de hecho las reglas originales de la gramática de expresión, las cuales cambiamos para reconocer los operadores directamente en las reglas para *exp* de la figura 5.10.) Ahora *opsuma* debe devolver el operador (un carácter), mientras que *exp* debe devolver el valor calculado (digamos, un **double**), y estos dos tipos de datos son diferentes. Lo que necesitamos hacer es definir **YYSTYPE** como una unión de **double** y **char**. Podemos hacer esto de dos maneras. Una es declarar la unión directamente en la especificación Yacc utilizando la declaración Yacc **%union**:

```
%union { double val;
         char op; }
```

Ahora Yacc necesita estar informado del tipo de retorno de cada no terminal, y esto se consigue mediante el uso de la directiva **%type** en la sección de definiciones:

```
%type <val> exp term factor
%type <op> addop mulop
```

Advierta cómo los nombres de los campos de unión están encerrados entre picoparéntesis en la declaración **%type** de Yacc. Entonces, la especificación Yacc de la figura 5.10 se modificaría para comenzar de la siguiente manera (y dejaremos los detalles para los ejercicios):

```
...
%token NUMBER

%union { double val;
         char op; }

%type <val> exp term factor NUMBER
```

```
%type <op> addop mulop

%%
command : exp        { printf("%d\n",$1);}
        ;
exp     : exp op term { switch ($2) {
                          case '+': $$ = $1 + $3; break;
                          case '-': $$ = $1 - $3; break;
                          }
                        }
        | term   {$$ = $1;}
        ;
op : '+' { $$ = '+'; }
   | '-' { $$ = '-'; }
   ;
```

La segunda alternativa es definir un nuevo tipo de datos en un archivo incluido por separado (por ejemplo, un archivo de cabecera) y posteriormente definir **YYSTYPE** como de este tipo. Entonces los valores apropiados se deben construir a mano en el código de acción asociado. Un ejemplo de esto es el analizador sintáctico TINY de la sección siguiente.

5.5.6 Acciones incrustadas en Yacc

En ocasiones es necesario ejecutar algún código antes de completar el reconocimiento de una regla gramatical durante el análisis sintáctico. Considere como ejemplo el caso de las declaraciones simples:

$$\begin{aligned} &\textit{decl} \rightarrow \textit{type var-list} \\ &\textit{type} \rightarrow \texttt{int} \mid \texttt{float} \\ &\textit{var-list} \rightarrow \textit{var-list}\ \texttt{,}\ \textit{id} \mid \textit{id} \end{aligned}$$

Nos gustaría, cuando se reconociera una *var-list*, etiquetar cada identificador de variable con el tipo actual (entero o de punto flotante). En Yacc podemos hacer esto del modo que se muestra a continuación:

```
decl    : type { current_type = $1; }
          var_list
        ;
type    : INT { $$ = INT_TYPE; }
        | FLOAT {   $$ = FLOAT_TYPE; }
        ;
var_list : var_list ',' ID
             { setType(tokenString,current_type);}
         | ID
             { setType(tokenString,current_type);}
         ;
```

Observe cómo se establece **current_type** (el cual para ser declarado necesita una variable estática en la sección de declaraciones) mediante una acción de incrustación antes del reconocimiento de las variables en la regla **dec1**. En las secciones siguientes se verán más ejemplos de acciones incrustadas.

Yacc interpreta una acción incrustada

```
A : B {/* acción incrustada */} C ;
```

como equivalente a la creación de un nuevo no terminal retenedor y una producción ε para ese no terminal que, cuando se reduce, realiza la acción incrustada:

```
A : B E C ;
E : {/* acción incrustada */} ;
```

Para cerrar esta sección resumimos en la tabla 5.12 los nombres internos y mecanismos de definición de Yacc que hemos analizado.

Tabla 5.12
Nombres internos y mecanismos de definición de Yacc

Nombre interno de Yacc	Significado/Uso
y.tab.c	Nombre interno de Yacc
y.tab.h	Archivo de cabecera generado por Yacc que contiene definiciones de token
yyparse	Rutina de análisis sintáctico de Yacc
yylval	Valor del token actual en la pila
yyerror	Impresión de mensaje de error definido por el usuario utilizado mediante Yacc
error	Pseudotoken de error de Yacc
yyerrok	Procedimiento que restablece el analizador sintáctico después de un error
yychar	Contiene el token de búsqueda hacia delante que causó un error
YYSTYPE	Símbolo de preprocesador que define el tipo de valor de la pila de análisis sintáctico
yydebug	Variable que, si se establece a 1 por el usuario, provoca la generación de información en tiempo de ejecución acerca de las acciones del análisis sintáctico

Mecanismo de definición en Yacc	Significado/Uso
%token	Define símbolos de preprocesador de token
%start	Define el símbolo no terminal inicial
%union	Define un **YYSTYPE** de unión, permitiendo valores de tipos diferentes en la pila del analizador sintáctico
%type	Define el tipo de unión variante devuelto por un símbolo
%left %right %nonassoc	Define la asociatividad y precedencia (por posición) de operadores

5.6 GENERACIÓN DE UN ANALIZADOR SINTÁCTICO TINY UTILIZANDO Yacc

En la sección 3.7 se expuso la sintaxis de TINY, y en la sección 4.4 se describió un analizador sintáctico escrito a mano, remitimos al lector a esas descripciones. Aquí describiremos el archivo de especificación Yacc **tiny.y**, junto con los cambios necesarios a las definiciones globales **globals.h** (tomamos pasos para minimizar cambios a otros archivos, lo que también se comentará). El archivo completo **tiny.y** está listado en el apéndice B, líneas 4000-4162.

Comenzamos con un análisis de la sección de definiciones acerca de la especificación Yacc de TINY. El uso de la marca **YYPARSER** (línea 4007) se describirá posteriormente. Tenemos cuatro archivos **#include** representando la información que necesita Yacc de otra parte en el programa (líneas 4009-4012). La sección de definiciones tiene otras cuatro declaraciones. La primera (línea 4014) es una definición de **YYSTYPE** que define los valores devueltos por los procedimientos de análisis sintáctico de Yacc como apuntadores a estructuras de nodos (**TreeNode** mismo está definido en **globals.h**). Esto permite al analizador sintáctico de Yacc construir un árbol sintáctico. La segunda declaración es de una variable estática global **savedName** que se emplea para almacenar temporalmente cadenas de identificador que necesitan ser insertadas en nodos de árbol que todavía no se han construido cuando las cadenas se ven en la entrada (en TINY esto es necesario sólo en asignaciones). La variable **savedLineNo** se utiliza para el mismo propósito, de manera que los números de línea apropiados del código fuente estarán asociados con los identificadores. Finalmente, **savedTree** se utiliza para almacenar de manera temporal el árbol sintáctico producido por el procedimiento **yyparse** (**yyparse** mismo sólo puede devolver una marca de valor entero).

Procederemos a un análisis de las acciones asociadas con cada una de las reglas gramaticales de TINY (estas reglas son ligeras variaciones de la gramática BNF expuesta en el capítulo 3, figura 3.6). En la mayoría de los casos estas acciones representan la construcción del árbol sintáctico correspondiente al árbol de análisis gramatical en ese punto. Específicamente, se necesitan nuevos nodos para ser asignados mediante llamadas a **newStmtNode** y **newExpNode** desde el paquete **util** (éstos se describieron en la página 182), y se necesitan nodos hijo apropiados del nuevo nodo de árbol para ser asignados. Por ejemplo, las acciones correspondientes a **write_stmt** de TINY (líneas 4082 y siguientes) son como se ve a continuación:

```
write_stmt : WRITE exp
                 { $$ = newStmtNode(WriteK);
                   $$->child[0] = $2;
                 }
           ;
```

La primera instrucción llama a **newStmtNode** y asigna su valor devuelto como el valor de **write_stmt**. Entonces asigna el valor previamente construido de **exp** (la pseudovariable **$2** de Yacc, que es un apuntador al nodo de árbol de la expresión por imprimirse) para que sea el primer hijo del nodo de árbol de la sentencia de escritura. El código de acción para otras sentencias y expresiones es muy similar.

Las acciones para **program**, **stmt_seq** y **assign_stmt** tratan con pequeños problemas asociados con cada una de estas construcciones. En el caso de la regla gramatical para **program** la acción asociada (línea 4029) es

```
{ savedTree = $1;}
```

Esto asigna el árbol construido para la **stmt_seq** a la variable estática **savedTree**. Esto es necesario para que el árbol sintáctico pueda ser devuelto posteriormente mediante el procedimiento **parse**.

En el caso de **assign_stmt**, ya mencionamos que necesitamos almacenar la cadena del identificador de la variable que es el objetivo de la asignación de manera que esté disponible cuando se construya el nodo (así como su número de línea para una exploración posterior). Esto se consigue utilizando las variables estáticas **savedName** y **saveLineNo** (líneas 4067 y siguientes):

```
assign_stmt : ID { savedName = copyString(tokenString);
                   savedLineNo = lineno; }
              ASSIGN exp
                  { $$ = newStmtNode(AssignK);
                    $$->child[0] = $4;
                    $$->attr.name = savedName;
                    $$->lineno = saveLineNo;
                  }
            ;
```

La cadena de identificador y el número de línea se deben guardar como una acción incrustada antes del reconocimiento del token **ASSIGN**, puesto que, a medida que se igualen nuevos tokens, los valores de **tokenString** y **lineno** son modificados por el analizador léxico. Incluso el nuevo nodo para la asignación no puede ser completamente construido hasta que la **exp** sea reconocida. De aquí surge la necesidad de **savedName** y **saveLineNo**. (El uso del procedimiento utilitario **copyString** asegura que no se comparta la memoria entre estas cadenas. Advierta que también el valor **exp** es referido como **$4**. Esto se debe a que Yacc cuenta las acciones incrustadas como ubicaciones adicionales en los lados derechos de las reglas gramaticales: véase el análisis de la sección anterior.)

En el caso de **stmt_seq** (líneas 4031-4039), el problema es que las sentencias se unen en un árbol sintáctico de TINY utilizando apuntadores hermanos en lugar de apuntadores hijos. Como la regla para secuencias de sentencias se escribe de manera recursiva por la izquierda, esto requiere que el código rastree la lista de hermanos ya construida en busca de la sublista izquierda a fin de unir la sentencia actual al final. Esto no es eficiente, y puede evitarse volviendo a escribir la regla de manera recursiva por la derecha, pero esta solución tiene su propio problema, que consiste en que la pila de análisis sintáctico se acrecentará a medida que se procesen secuencias de sentencias más largas.

Finalmente, la sección de procedimientos auxiliares de la especificación Yacc (líneas 4144-4162) contiene la definición de los tres procedimientos, **yyerror**, **yylex** y **parse**. El procedimiento **parse**, el cual se invoca desde el programa principal, llama al procedimiento de análisis sintáctico definido en Yacc **yyparse** y luego devuelve el árbol sintáctico grabado. El procedimiento **yylex** es necesario debido a que Yacc supone que éste es el nombre del procedimiento del analizador léxico, y éste fue definido de manera externa como **getToken**. Escribimos esta definición de manera que el analizador sintáctico generado por Yacc funcionara con el compilador TINY con un mínimo de cambios para otros archivos del código. Se puede desear hacer los cambios apropiados al analizador léxico y eliminar esta definición, particularmente si se está utilizando la versión de Lex del analizador léxico. El procedimiento **yyerror** es llamado por Yacc cuando se presenta un error; imprime cierta información útil, tal como el número de línea, al archivo de listado. Utiliza una variable **yychar** interna de Yacc que contiene el número de token correspondiente al token que ocasionó el error.

Resta describir los cambios a los otros archivos en el analizador sintáctico de TINY que se hacen necesarios por el uso de Yacc para producir el analizador sintáctico. Como observamos, pretendimos mantener estos cambios en un mínimo, y confinamos todos los cambios al archivo **globals.h**. El archivo revisado se muestra en el apéndice B, líneas 4200-4320.

El problema básico es que el archivo de cabecera generado por Yacc, que contiene las definiciones de token, debe ser incluido en la mayoría de los otros archivos de código, pero no puede ser incluido directamente en el analizador sintáctico generado por Yacc, puesto que repetiría las definiciones internas. La solución es comenzar la sección de declaraciones de Yacc con la definición de una marca **YYPARSER** (línea 4007), la cual se incluirá en el analizador sintáctico de Yacc e indicará cuando el compilador de C esté dentro del analizador sintáctico. Utilizamos esa marca (líneas 4226-4236) para incluir de manera selectiva el archivo de cabecera generado por Yacc **y.tab.h** en **globals.h**.

Un segundo problema se presenta con el token **ENDFILE**, que genera el analizador léxico para indicar el fin del archivo de entrada. Yacc supone que este token siempre tiene el valor 0. Suministramos una definición directa de este token (líneas 4234) y la incluimos en la sección selectivamente compilada controlada por **YYPARSER**, puesto que Yacc no la necesita internamente.

El cambio final al archivo **globals.h** es volver a definir **TokenType** como un sinónimo para **int** (línea 4252), ya que todos los tokens de Yacc tienen valores enteros. Esto evita reemplazos innecesarios del tipo enumerado **TokenType** anterior en otros archivos.

5.7 RECUPERACIÓN DE ERRORES EN ANALIZADORES SINTÁCTICOS ASCENDENTES

5.7.1 Detección de errores en el análisis sintáctico ascendente

Un analizador sintáctico ascendente encontrará un error cuando se detecte una entrada en blanco (o de error) en la tabla de análisis sintáctico. Obviamente, tiene sentido que los errores deberían detectarse tan pronto como fuera posible, ya que los mensajes de error pueden ser más específicos y significativos. De este modo, una tabla de análisis sintáctico debería tener tantas entradas en blanco como fuera posible.

Por desgracia esta meta está en conflicto con otra de la misma importancia: la reducción del tamaño de la tabla de análisis sintáctico. Ya vimos (tabla 5.11) que Yacc rellena tantas entradas de la tabla como puede con reducciones predeterminadas, de manera que se pueden producir un gran número de reducciones en la pila de análisis sintáctico antes que se declare un error. Esto oculta la fuente precisa del error y puede conducir a mensajes de error poco informativos.

Una característica adicional del análisis sintáctico ascendente es que la potencia del algoritmo particular utilizado puede afectar la capacidad del analizador sintáctico para detectar los errores de inmediato. Por ejemplo, un analizador sintáctico LR(1) puede detectar errores en una etapa más temprana que un analizador LALR(1) o SLR(1), y estos últimos pueden detectar errores más pronto que un analizador LR(0). Como un ejemplo simple de esto compare la tabla de análisis sintáctico LR(0) de la tabla 5.4 (página 209) con la tabla de análisis sintáctico LR(1) de la tabla 5.10 (página 222) para la misma gramática. Dada la cadena de entrada incorrecta **(a**$, la tabla de análisis sintáctico LR(1) de la tabla 5. 10 desplazará **(** y **a** hacia la pila y se moverá al estado 6. En el estado 6 no hay entrada bajo $, de modo que se informará de un error. En contraste, el algoritmo LR(0) (así como el algoritmo SLR(1)) reducirán mediante $A \rightarrow$ **a** antes de descubrir la ausencia de un paréntesis derecho para completar la expresión. De manera similar, dada la cadena incorrecta **a)**$, el analizador sintáctico LR(1) general desplazará **a** y después declarará un error del estado 3 en el paréntesis derecho, mientras que el analizador LR(0) reducirá otra vez mediante $A \rightarrow$ **a** antes de declarar un error. Naturalmente, cualquier analizador sintáctico ascendente siempre terminará por informar del error, quizás después de un número de reducciones "erróneas". Ninguno de estos analizadores desplazará jamás un token en estado de error.

5.7.2 Recuperación de errores en modo de alarma

Como en el análisis sintáctico descendente, se puede conseguir una recuperación de errores razonablemente buena en analizadores ascendentes eliminando símbolos de manera prudente,

ya sea de la pila de análisis sintáctico o de la entrada, o bien de ambas. Como en los analizadores sintácticos LL(1), existen tres posibles acciones alternativas que se pueden contemplar:

1. Extraer un estado de la pila.
2. Extraer sucesivamente tokens desde la entrada hasta que se vea uno de ellos para el cual se pueda reiniciar el análisis sintáctico.
3. Insertar un nuevo estado en la pila.

Un método particularmente efectivo para elegir cuál de estas acciones realizar cuando se presenta un error es el siguiente:

1. Extraer estados de la pila de análisis sintáctico hasta encontrar un estado con entradas Ir a no vacías.
2. Si existe una acción legal sobre el token de entrada actual desde uno de los estados Ir a, insertar ese estado en la pila y reiniciar el análisis sintáctico. Si existen varios de tales estados, preferir un desplazamiento a una reducción. Entre las acciones de reducción preferir una cuyo no terminal asociado sea menos general.
3. Si no existe acción legal en el token de entrada actual desde uno de los estados Ir a, avanzar la entrada hasta que haya una acción legal o se alcance el final de la entrada.

Estas reglas tienen el efecto de obligar al reconocimiento de una construcción que estuviera en el proceso de ser reconocida cuando ocurre el error, y reiniciar el análisis sintáctico inmediatamente a partir de entonces. La recuperación de errores que utiliza éstas o unas reglas semejantes se podría denominar recuperación de errores en **modo de alarma**, puesto que es similar al modo de alarma descendente descrito en la sección 4.5.

Desgraciadamente, como el paso 2 inserta nuevos estados en la pila, estas reglas pueden dar como resultado un ciclo infinito. Si éste es el caso, existen varias posibles soluciones. Una es insistir en una acción de desplazamiento desde un estado Ir a en el paso 2. Sin embargo, esto puede ser demasiado restrictivo. Otra solución es, si el siguiente movimiento legal es una reducción, establecer una marca que provoque que el analizador sintáctico siga la pista de la secuencia de estados durante las siguientes reducciones, y si el mismo estado se repite, extraiga estados de la pila hasta que se elimine el estado original en el que ocurrió el error, y comience de nuevo con el paso 1. Si en cualquier momento se presenta una acción de desplazamiento, el analizador vuelve a establecer la marca y continúa con un análisis sintáctico normal.

Ejemplo 5.19

Considere la gramática de expresión aritmética simple cuya tabla de análisis sintáctico en Yacc se da en la tabla 5.11 (página 235). Considere ahora la entrada errónea `(2+*)`. El análisis sintáctico continúa normalmente hasta que se detecta el asterisco `*`. En ese punto el modo de alarma podría ocasionar que se produjeran las siguientes acciones en la pila de análisis sintáctico:

Pila de análisis sintáctico	Entrada	Acción
...	...	...
$ 0 (6 *E* 10 + 7	*) $	error: insertar *T*, ir a 11
$ 0 (6 *E* 10 + 7 *T* 11	*) $	desplazar 9
$ 0 (6 *E* 10 + 7 *T* 11 * 9	) $	error: insertar *F*, ir a 13
$ 0 (6 *E* 10 + 7 *T* 11 * 9 *F* 13	) $	reducir $T \rightarrow T * F$
...	...	...

En el primer error el analizador está en el estado 7, que tiene estados Ir a legales 11 y 4. Como el estado 11 tiene un desplazamiento con el token de entrada siguiente *, se prefiere ese Goto, y se desplaza el token. En ese punto el analizador sintáctico está en el estado 9, con un paréntesis derecho en la entrada. Esto de nueva cuenta es un error. En el estado 9 hay una entrada de Ir a simple (al estado 11), y el estado 11 tiene una acción legal con) (aunque sea una reducción). El análisis sintáctico procede entonces normalmente hasta concluir. §

5.7.3 Recuperación de errores en Yacc

Una alternativa al modo de alarma es emplear las denominadas **producciones de error**. Una producción de error es una producción que contiene el pseudotoken `error` como el único símbolo en su lado derecho. Una producción de error marca un contexto en el que los tokens erróneos se pueden eliminar hasta que se detectan tokens de sincronización apropiados, de donde el análisis sintáctico se puede reiniciar. En efecto, las producciones de error permiten al programador marcar manualmente aquellos no terminales cuyas entradas Ir a vayan a utilizarse para recuperación de errores.

Las producciones de error son el principal método disponible en Yacc para recuperación de errores. El comportamiento de un analizador sintáctico Yacc en la presencia de errores, así como su manejo de las producciones de error, es de la manera siguiente:

1. Cuando el analizador sintáctico detecta un error durante un análisis (es decir, encuentra una entrada Ir a vacía en la tabla de análisis sintáctico), extrae los estados de la pila de análisis sintáctico hasta que alcanza un estado en el cual el pseudotoken `error` es una búsqueda hacia delante legal. El efecto es descartar la entrada a la izquierda del error y visualizar la entrada como que contiene el pseudotoken `error`. Si no hay producciones de error, entonces `error` nunca es una búsqueda hacia delante legal para un desplazamiento, y la pila de análisis sintáctico se vaciará, cancelando el análisis sintáctico para el primer error. (Éste es el comportamiento del analizador sintáctico generado mediante la entrada de Yacc de la figura 5.10, página 228.)
2. Una vez que el analizador sintáctico encuentra un estado en la pila en el cual `error` es una búsqueda hacia delante legal, continúa en modo normal con desplazamientos y reducciones. El efecto es como si `error` fuera detectado en la entrada, seguido por la búsqueda hacia delante original (es decir, la búsqueda que causó el error). Si se desea, puede utilizarse la macro `yyclearin` de Yacc para descartar el token que provocó el error, y emplear el token que le sigue como la siguiente búsqueda hacia delante (después de `error`).
3. Si, después de que se presenta un error, el analizador sintáctico descubre más errores, entonces los tokens de entrada causantes de los errores se descartan silenciosamente, hasta que tres tokens sucesivos se desplazan de manera legal en la pila de análisis sintáctico. Durante este tiempo se dice que el analizador sintáctico está en un "estado de error". Este comportamiento está diseñado para evitar cascadas de mensajes de error ocasionadas por el mismo error. Sin embargo, esto puede dar como resultado que se desaproveche gran cantidad de entrada antes que el analizador sintáctico salga del estado de error (justamente como con el modo de alarma). El escritor de compiladores puede cancelar este comportamiento utilizando la macro `yyerrok` para eliminar el analizador del estado de error, de manera que ninguna entrada adicional será descartada sin una nueva recuperación de errores.

Describimos unos cuantos ejemplos simples de este comportamiento basados en la entrada de Yacc de la figura 5.10.

Ejemplo 5.20 Considere el siguiente reemplazo a la regla para *command* de la figura 5.10:

```
command : exp        { printf("%d\n",$1);}
        | error      { yyerror("expresión incorrecta"); }
        ;
```

Considere también la entrada errónea `2++3`. El análisis de esta cadena continúa normalmente hasta que se alcanza el segundo + erróneo. En este punto el análisis da la siguiente pila de análisis sintáctico y entrada (utilizamos la tabla 5.11 para describir esto, aunque la adición de la producción de error resultará en realidad en una tabla de análisis ligeramente diferente):

PILA DE ANÁLISIS SINTÁCTICO	ENTRADA
$0 *exp* 2 + 7	+3$

Ahora el analizador sintáctico introduce el "estado" de error (generando un mensaje de error tal como "syntax error" o "error de sintaxis") y comienza a extraer estados desde la pila, hasta que se descubre el estado 0. En este punto la producción de errores para *command* estipula que `error` es una búsqueda hacia delante legal, y se desplazará a la pila de análisis sintáctico, e inmediatamente reducirá a *command*, causando que se ejecute la acción asociada (lo que imprime el mensaje "incorrect expression" o "expresión incorrecta"). La situación resultante es ahora como se muestra a continuación:

PILA DE ANÁLISIS SINTÁCTICO	ENTRADA
$0 *command* 1	+3$

En este punto la única búsqueda hacia delante legal está al final de la entrada (indicado aquí por $, correspondiente a la devolución de 0 por `yylex`), y el analizador sintáctico eliminará los tokens de entrada restantes `+3` antes de salir (aunque todavía en el "estado de error"). De este modo, la adición de la producción de error tiene esencialmente el mismo efecto que la versión de la figura 5.10, sólo que ahora podemos proporcionar nuestro propio mensaje de error.

Un mecanismo de error aún mejor que éste permitiría al usuario reintroducir la línea después de la entrada errónea. En este caso se necesita un token de sincronización, y el fin del marcador de línea es el único sensible. Así, el analizador léxico se debe modificar para devolver el carácter de retorno de línea (en vez de 0), y con esta modificación podemos escribir (pero véase el ejercicio 5.32):

```
command : exp '\n'    { printf("%d\n",$1); exit(0);}
        | error '\n'
            { yyerrok;
              printf("reintroducir expresión: "); }
          command
        ;
```

Este código tiene el efecto de que, cuando ocurre un error, el analizador sintáctico saltará todos los tokens hasta uno de retorno de línea, cuando ejecutará la acción representada por `yyerrok` y la sentencia `printf`. Entonces intentará reconocer otro *command*. Aquí es necesaria la llamada a `yyerrok` para cancelar el "estado de error" después de que se detecta el retorno de línea, ya que de otro modo, se presenta un nuevo error justo después

del retorno de línea, Yacc eliminará silenciosamente los tokens hasta que encuentre una secuencia de tres tokens correctos. §

Ejemplo 5.21

Imaginemos lo que pasaría si se agregara una producción de error a la definición Yacc de la figura 5.10 de la manera siguiente:

```
factor    : NUMBER          {$$ = $1;}
          | '(' exp ')'     {$$ = $2;}
          | error {$$ = 0;}
          ;
```

Consideremos en primer lugar la entrada errónea `2++3` como en el ejemplo anterior. (Continuamos usando la tabla 5.11, aunque la producción de error adicional da como resultado una tabla ligeramente diferente.) Como antes, el analizador sintáctico alcanzará el punto siguiente:

PILA DE ANÁLISIS SINTÁCTICO	ENTRADA
$0 *exp* 2 + 7	+3$

Ahora la producción de error para *factor* estipulará que `error` es una búsqueda hacia delante legal en el estado 7, y `error` será inmediatamente desplazado a la pila y reducido a *factor*, provocando que el valor 0 sea devuelto. Ahora el analizador ha llegado al punto siguiente:

PILA DE ANÁLISIS SINTÁCTICO	ENTRADA
$0 *exp* 2 + 7 *factor* 4	+3$

Ésta es una situación normal, y el analizador continuará su ejecución normalmente hasta el final. El efecto es interpretar la entrada como `2+0+3` (el 0 entre los dos símbolos de + está allí porque allí fue donde se insertó el pseudotoken `error`, y por la acción para la producción de error, `error` se visualiza como equivalente a un factor con valor 0).

Ahora consideremos la entrada errónea `2 3` (es decir, dos números con un operador perdido). Aquí el analizador sintáctico alcanza la posición

PILA DE ANÁLISIS SINTÁCTICO	ENTRADA
$0 *exp* 2	3$

En este punto (si la regla para *command* no se ha cambiado), el analizador reducirá (erróneamente) mediante la regla *command* → *exp* (e imprimirá el valor 2), aun cuando un número no es un símbolo siguiente legal para *command*. El analizador alcanza entonces la posición

PILA DE ANÁLISIS SINTÁCTICO	ENTRADA
$0 *command* 1	3$

Ahora se detecta un error, y el estado 1 se extrae de la pila de análisis sintáctico, descubriendo el estado 0. En este punto la producción de error para *factor* permite que `error` sea una búsqueda hacia delante legal desde el estado 0, y `error` se desplaza a la pila de análisis sintáctico produciendo otra cascada de reducciones y la impresión del valor 0 (el valor devuelto por la producción de error). ¡Ahora el analizador está de regreso en la misma posición en el análisis sintáctico, con el número `3` todavía en la entrada! Afortunadamente, el

analizador sintáctico está ya en el "estado de error" y no desplazará de nueva cuenta a `error`, pero desechará el número `3`, descubriendo la búsqueda hacia delante correcta para el estado 1, con lo cual el analizador sintáctico sale de escena.[10] El resultado es que el analizador imprime lo siguiente (repetimos la entrada del usuario en la primera línea):

```
> 2 3
2
error de sintaxis
expresión incorrecta
0
```

Este comportamiento nos da una visión fugaz de las dificultades de una buena recuperación de errores en Yacc. (Véanse los ejercicios para más ejemplos.) §

5.7.4 Recuperación de errores en TINY

El archivo de especificación de Yacc `tiny.y` en el apéndice B contiene dos producciones de error, una para `stmt` (línea 4047) y otra para `factor` (línea 4139), cuyas acciones asociadas son para devolver el árbol sintáctico nulo. Estas producciones de error proporcionan un nivel de manejo de errores semejante al del analizador sintáctico descendente recursivo de TINY expuesto en el capítulo anterior, sólo que no se hace ningún intento de construir un árbol sintáctico significativo en la presencia de errores. Estas producciones de error tampoco proporcionan sincronización especial para el reinicio del análisis sintáctico, de modo que en la presencia de errores múltiples se pueden omitir muchos tokens.

EJERCICIOS

5.1 Considere la gramática siguiente:

$$E \rightarrow (\ L\ ) \mid \mathtt{a}$$
$$L \rightarrow L\ ,\ E \mid E$$

a. Construya el DFA de elementos LR(0) para esta gramática.
b. Construya la tabla de análisis sintáctico SLR(1).
c. Muestre la pila de análisis sintáctico y las acciones de un analizador SLR(1) para la cadena de entrada `((a),a,(a,a))`.
d. ¿Es esta gramática una gramática LR(0)? Si no es así, describa el conflicto LR(0). Si lo es, construya la tabla de análisis sintáctico LR(0) y describa cómo puede diferir un análisis sintáctico de un análisis SLR(1).

5.2 Considere la gramática del ejercicio anterior.
a. Construya el DFA de elementos LR(1) para esta gramática.
b. Construya la tabla de análisis sintáctico LR(1) general.
c. Construya el DFA de elementos LALR(1) para esta gramática.
d. Construya la tabla de análisis sintáctico LALR(1).

---

10. Algunas versiones de Yacc vuelven a extraer la pila de análisis sintáctico antes de eliminar cualquier entrada. Esto puede dar como resultado un comportamiento aún más complicado. Véanse los ejercicios.

e. Describa cualquier diferencia que se pueda presentar entre las acciones de un analizador sintáctico LR(1) general y un analizador sintáctico LALR(1).

5.3 Considere la gramática siguiente:

$$A \rightarrow A\ (\ A\ )\ |\ \varepsilon$$

a. Construya el DFA de elementos LR(0) para esta gramática.
b. Construya la tabla de análisis sintáctico SLR(1).
c. Muestre la pila de análisis sintáctico y las acciones de un analizador SLR(1) para la cadena de entrada
`(()())`.
d. ¿Es esta gramática una gramática LR(0)? Si no es así, describa el conflicto LR(0). Si lo es, construya la tabla de análisis sintáctico LR(0) y describa cómo puede diferir un análisis sintáctico de un análisis SLR(1).

5.4 Considere la gramática del ejercicio anterior.
a. Construya el DFA de elementos LR(1) para esta gramática.
b. Construya la tabla de análisis sintáctico LR(1) general.
c. Construya el DFA de elementos LALR(1) para esta gramática.
d. Construya la tabla de análisis sintáctico LALR(1).
e. Describa cualquier diferencia que se pueda presentar entre las acciones de un analizador sintáctico LR(1) general y un analizador sintáctico LALR(1).

5.5 Considere la gramática siguiente de secuencias simplificadas de sentencias:

$$\textit{secuencia-sent} \rightarrow \textit{secuencia-sent}\ \texttt{;}\ \textit{sent}\ |\ \textit{sent}$$
$$\textit{sent} \rightarrow \texttt{s}$$

a. Construya el DFA de elementos LR(0) para esta gramática.
b. Construya la tabla de análisis sintáctico SLR(1).
c. Muestre la pila de análisis sintáctico y las acciones de un analizador SLR(1) para la cadena de entrada `s;s;s`.
d. ¿Es esta gramática una gramática LR(0)? Si no es así, describa el conflicto LR(0). Si lo es, construya la tabla de análisis sintáctico LR(0) y describa cómo puede diferir un análisis sintáctico de un análisis SLR(1).

5.6 Considere la gramática del ejercicio anterior.
a. Construya el DFA de elementos LR(1) para esta gramática.
b. Construya la tabla de análisis sintáctico LR(1) general.
c. Construya el DFA de elementos LALR(1) para esta gramática.
d. Construya la tabla de análisis sintáctico LALR(1).
e. Describa cualquier diferencia que se pueda presentar entre las acciones de un analizador sintáctico LR(1) general y un analizador sintáctico LALR(1).

5.7 Considere la gramática siguiente:

$$E \rightarrow (\ L\ )\ |\ \texttt{a}$$
$$L \rightarrow E\ L\ |\ E$$

a. Construya el DFA de elementos LR(0) para esta gramática.
b. Construya la tabla de análisis sintáctico SLR(1).

c. Muestre la pila de análisis sintáctico y las acciones de un analizador SLR(1) para la cadena de entrada `((a)a(a a))`.

d. Construya el DFA de elementos LALR(1) mediante la propagación de búsquedas hacia delante a través del DFA de elementos LR(0).

e. Construya la tabla de análisis sintáctico LALR(1).

5.8 Considere la gramática siguiente:

declaración → *tipo var-list*
tipo → **`int`** | **`float`**
var-list → **`identificador,`** *var-list* | **`identificador`**

a. Vuelva a escribirla en una forma más adecuada para el análisis sintáctico ascendente.

b. Construya el DFA de elementos LR(0) para la gramática reescrita.

c. Construya la tabla de análisis sintáctico SLR(1) para la gramática reescrita.

d. Muestre la pila de análisis sintáctico y las acciones de un analizador SLR(1) para la cadena de entrada `int x,y, z` utilizando la tabla del inciso c.

e. Construya el DFA de elementos LALR(1) propagando las búsquedas hacia delante a través del DFA de elementos LR(0) del inciso b.

f. Construya la tabla de análisis sintáctico LALR(1) para la gramática reescrita.

5.9 Describa una descripción formal del algoritmo para construir el DFA de elementos LALR(1) propagando búsquedas hacia delante a través del DFA de elementos LR(0). (Este algoritmo se describió de manera informal en la sección 5.4.3.)

5.10 Todas las pilas de análisis sintáctico mostradas en este capítulo incluyeron tanto números de estado como símbolos gramaticales (por claridad). Sin embargo, la pila de análisis sintáctico necesita sólo almacenar los números de estado: los tokens y no terminales no necesitan ser almacenados en la pila. Describa el algoritmo de análisis sintáctico SLR(1) si sólo los números de estado se mantienen en la pila.

5.11 **a.** Muestre que la gramática siguiente no es LR(1):

$$A \rightarrow a\,A\,a \mid \varepsilon$$

b. ¿Esta gramática es ambigua? Justifique su respuesta.

5.12 Muestre que la siguiente gramática es LR(1) pero no LALR(1):

$$S \rightarrow a\,A\,d \mid b\,B\,d \mid a\,B\,e \mid b\,A\,e$$
$$A \rightarrow c$$
$$B \rightarrow c$$

5.13 Muestre que una gramática LR(1) que no es LALR(1) debe tener sólo conflictos de reducción-reducción.

5.14 Muestre que un prefijo de una forma sentencial derecha es un prefijo viable si y sólo si no se extiende más allá del controlador.

5.15 ¿Puede haber una gramática SLR(1) que no sea LALR(1)? Justifique su respuesta.

5.16 Muestre que un analizador sintáctico LR(1) general no hará reducción antes que se declare un error, si el siguiente token de entrada no puede ser desplazado con el tiempo.

5.17 ¿Puede un analizador sintáctico SLR(1) hacer más o menos reducciones que un analizador LALR(1) antes que declarar un error? Justifique su respuesta.

5.18 La siguiente gramática ambigua genera las mismas cadenas que la gramática del ejercicio 5.3 (a saber, todas las cadenas de paréntesis anidados):

$$A \rightarrow AA \mid (\ A\ ) \mid \varepsilon$$

¿Reconocerá un analizador sintáctico generado por Yacc todas las cadenas legales utilizando esta gramática? Explique su respuesta.

5.19 Dado un estado en el cual haya dos posibles reducciones (en diferentes búsquedas hacia delante), Yacc elegirá una de las reducciones como su acción predeterminada. Describa la regla que utiliza Yacc para efectuar esta elección. (*Sugerencia*: utilice la gramática del ejemplo 5.16, página 222, como un caso de prueba.)

5.20 Suponga que eliminamos las especificaciones de asociatividad y precedencia de los operadores de la especificación Yacc de la figura 5.14, página 238 (dejando, por consiguiente, una gramática ambigua). Describa qué asociatividad y precedencia resultan de las reglas de eliminación de ambigüedad predeterminadas de Yacc.

5.21 Como se describió en la nota a pie de página de la página 250, algunos analizadores de Yacc extraen la pila de análisis sintáctico otra vez antes de descartar cualquier entrada mientras está en "estado de error". Describa el comportamiento de un analizador de esta clase para la especificación de Yacc del ejemplo 5.21, dada la entrada errónea `2 3`.

5.22 **a.** Rastree el comportamiento de un mecanismo de recuperación de error en el modo de alarma como se describió en la sección 5.7.1 utilizando la tabla de análisis sintáctico 5.11, página 235, y la cadena `(*2`.

b. Sugiera una mejora para el comportamiento del inciso a.

5.23 Suponga que la regla para `command` en la especificación de la calculadora de Yacc de la figura 5.10, página 228, es reemplazada por la siguiente regla para un no terminal inicial `list`:

```
list   :     list '\n' {exit(0);}
       |     list exp '\n' {printf("%d\n",$2);}
       |     list error '\n' {yyerrok;}
       ;
```

y el procedimiento `yylex` en esa figura tiene la línea

```
if (c == '\n') return 0;'
```

eliminada.

a. Explique las diferencias en el comportamiento de esta versión de la calculadora simple respecto al de la versión de la figura 5.10.

b. Explique la razón para el `yyerrok` en la última línea de esta regla. Proporcione un ejemplo para mostrar lo que podría pasar si no estuviera allí.

5.24 **a.** Suponga que la regla para `command` en la especificación de la calculadora de Yacc de la figura 5.10, página 228, es reemplazada por la siguiente regla:

```
command : exp error {printf("%d\n",$1);}
```

Explique con precisión el comportamiento del analizador Yacc, dada la entrada `2 3`.

b. Suponga que la regla para `command` en la especificación de la calculadora de Yacc de la figura 5.10, página 228, es reemplazada en cambio por la siguiente regla:

```
command : error exp{printf("%d\n",$2);}
```

Explique con precisión el comportamiento del analizador Yacc, dada la entrada `2 3`.

5.25 Suponga que la producción de error Yacc del ejemplo 5.21, página 249, se reemplaza por la regla siguiente:

```
factor  : NUMBER          {$$ = $1;}
        | '(' exp ')'     {$$ = $2;}
        | error {yyerrok; $$ = 0;}
        ;
```

a. Explique el comportamiento del analizador Yacc dada la entrada errónea `2++3`.

b. Explique el comportamiento del analizador Yacc dada la entrada errónea `2 3`.

5.26 Compare la recuperación de errores del analizador sintáctico TINY generado por Yacc con la del analizador sintáctico descendente recursivo del capítulo anterior utilizando los programas de prueba de la sección 4.5.3. Explique las diferencias en comportamiento.

EJERCICIOS DE PROGRAMACIÓN

5.27 Vuelva a escribir la especificación Yacc de la figura 5.10, página 235, para utilizar las siguientes reglas gramaticales (en lugar de **scanf**) en el cálculo del valor de un número (y, por lo tanto, suprimir el token **NUMBER**):

número → *número dígito* | *dígito*
dígito → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

5.28 Agregue lo siguiente a la especificación de la calculadora de enteros Yacc de la figura 5.10 (asegúrese que tengan la asociatividad y precedencia correctas):

a. División entera con el símbolo `/`.

b. Módulo entero con el símbolo `%`.

c. Exponenciación entera con el símbolo `^`. (*Advertencia*: este operador tiene una precedencia mayor que la multiplicación y es asociativo por la derecha.)

d. Menos unitario con el símbolo `-`. (Véase el ejercicio 3.12.)

5.29 Vuelva a hacer la especificación de la calculadora Yacc de la figura 5.10 de manera que acepte números de punto flotante (y realice cálculos en modo de punto flotante).

5.30 Vuelva a escribir la especificación de la calculadora Yacc de la figura 5.10 de manera que *distinga* entre valores de punto flotante y valores enteros en vez de simplemente calcular todo como enteros o números de punto flotante. (*Sugerencia*: un "valor" es ahora un registro con una marca que indica si es entero o de punto flotante.)

5.31 **a.** Vuelva a escribir la especificación de la calculadora Yacc de la figura 5.10 de manera que devuelva un árbol sintáctico de acuerdo con las declaraciones de la sección 3.3.2.

b. Escriba una función que tome como parámetro el árbol sintáctico producido por su código del inciso a y devuelva el valor calculado al recorrer el árbol.

5.32 La técnica de recuperación de errores simple sugerida para el programa de la calculadora en la página 248 era errónea porque podía causar desbordamiento de la pila después de muchos errores. Vuelva a escribirla para eliminar este problema.

5.33 Vuelva a escribir la especificación del calculador Yacc de la figura 5.10 para agregar los siguientes mensajes de error útiles:

"paréntesis derecho olvidado", generado por la cadena `(2+3`

"paréntesis izquierdo olvidado", generado por la cadena `2+3)`

"operador olvidado", generado por la cadena `2 3`

"operando olvidado", generado por la cadena `(2+)`

5.34 La gramática siguiente representa expresiones aritméticas simples en una notación de prefijo tipo LISP:

$$
\begin{aligned}
&lexp \rightarrow \texttt{número} \mid \texttt{(}\ op\ sec\text{-}lexp\ \texttt{)}\\
&op \rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*}\\
&sec\text{-}lexp \rightarrow sec\text{-}lexp\ lexp \mid lexp
\end{aligned}
$$

Por ejemplo, la expresión `(* (-2) 3 4)` tiene el valor −24. Escriba una especificación Yacc para un programa que calculará e imprimirá el valor de las expresiones en esta sintaxis. (*Sugerencia*: esto requerirá volver a escribir la gramática y usar un mecanismo para pasar el operador a una *sec-lexp*.)

5.35 La siguiente gramática (ambigua) representa las expresiones regulares simples analizadas en el capítulo 2:

$$
\begin{aligned}
rexp \rightarrow\ & rexp\ \text{“|”}\ rexp\\
\mid\ & rexp\ rexp\\
\mid\ & rexp\ \text{“*”}\\
\mid\ & \text{“(”}\ rexp\ \text{“)”}\\
\mid\ & \texttt{letra}
\end{aligned}
$$

a. Escriba una especificación esqueleto en Yacc (es decir, sin acciones) que exprese la asociatividad y precedencia correctas de las operaciones.

b. Extienda su especificación del inciso a para incluir todas las acciones y procedimientos auxiliares necesarios para producir un "compilador de expresiones regulares", es decir, un programa que tomará una expresión regular como entrada y como salida un programa en C que, cuando se compile, buscará una cadena de entrada para la primera ocurrencia de una subcadena que iguale la expresión regular. (*Sugerencia*: puede utilizarse una tabla, o arreglo bidimensional, para representar los estados y transiciones del NFA asociado. El NFA se puede entonces simular utilizando una cola para almacenar estados. Sólo se necesita generar la tabla mediante las acciones Yacc; el resto del código siempre será el mismo. Véase el capítulo 2.)

5.36 Vuelva a escribir la especificación Yacc para TINY (apéndice B, líneas 4000-4162) en una forma más compacta en cualquiera de las siguientes dos maneras:

a. haciendo uso de una gramática ambigua para expresiones (con reglas de eliminación de ambigüedades Yacc para la precedencia y la asociatividad), y

b. colapsando el reconocimiento de operadores en una sola regla, como en

$$
\begin{aligned}
&exp \rightarrow exp\ op\ term \mid \ldots\\
&op \rightarrow \texttt{+} \mid \texttt{-} \mid \ldots
\end{aligned}
$$

y utilizando la declaración Yacc `%union` (dejando que *op* devuelva el operador, mientras que *exp* y otros no terminales devuelven apuntadores hacia los nodos de árbol). Asegúrese de que su analizador sintáctico produzca los mismos árboles sintácticos que antes.

5.37 Agregue los operadores de comparación `<=` (menor o igual que), `>` (mayor que), `>=` (mayor o igual que) y `<>` (distinto de) a la especificación Yacc para el analizador sintáctico TINY. (Esto requerirá agregar estos tokens y modificar el analizador léxico también, pero no debería requerir un cambio al árbol sintáctico.)

5.38 Agregue los operadores booleanos `and`, `or` y `not` a la especificación Yacc para el analizador sintáctico de TINY. Asígneles las propiedades descritas en el ejercicio 3.5, además de una

precedencia menor a la de todos los operadores aritméticos. Asegúrese que cualquier expresión pueda ser booleana o entera.

5.39 Vuelva a escribir la especificación Yacc para el analizador sintáctico de TINY con el fin de mejorar su recuperación de errores.

NOTAS Y REFERENCIAS

El análisis sintáctico LR general fue inventado por Knuth [1965], pero se pensaba que era poco práctico hasta que DeRemer [1969, 1971] desarrolló las técnicas SLR y LALR. Repetimos el conocimiento convencional de que los analizadores sintácticos LR(1) son demasiado complicados para el uso práctico. De hecho es posible construir analizadores LR(1) prácticos utilizando técnicas de mezcla de estados más delicadas que las del análisis sintáctico LALR(1) [Pager, 1977]. No obstante, la potencia agregada rara vez es necesaria. Un estudio detallado de la teoría de las técnicas del análisis sintáctico LR se puede encontrar en Aho y Ullman [1972].

Yacc fue desarrollado por Steve Johnson en los laboratorios AT&T Bell a mediados de la década de los setenta y está incluido en la mayoría de las implementaciones Unix [Johnson, 1975]. Fue empleado para desarrollar el compilador portátil de C [Johnson, 1978] y muchos otros compiladores. Bison fue desarrollado por Richard Stallman y otros, Gnu Bison es una parte de la distribución del software Gnu de la Free Software Foundation y está disponible en muchos sitios de Internet. Un ejemplo del uso de Yacc para desarrollar un programa de una calculadora poderosa y todavía compacta se proporciona en Kernighan y Pike [1984]. Un estudio profundo del uso de Yacc se puede encontrar en Schreiner y Friedman [1985].

Las técnicas de recuperación de errores LR se estudian en Graham, Haley y Joy [1979]; Penello y DeRemer [1978] y Burke y Fisher [1987]. Una técnica de reparación de errores LR se describe en Fischer y LeBlanc [1991]. Las ideas básicas de la técnica de modo de alarma descrita en la sección 5.7.2 son atribuidas por Fischer y LeBlanc a James [1972].

Capítulo 6

Análisis semántico

6.1 Atributos y gramáticas con atributos

6.2 Algoritmos para cálculo de atributos

6.3 La tabla de símbolos

6.4 Tipos de datos y verificación de tipos

6.5 Un analizador semántico para el lenguaje TINY

En este capítulo analizamos la fase del compilador que calcula la información adicional necesaria para la compilación una vez que se conoce la estructura sintáctica de un programa. Esta fase se conoce como análisis semántico debido a que involucra el cálculo de información que rebasa las capacidades de las gramáticas libres de contexto y los algoritmos de análisis sintáctico estándar, por lo que no se considera como sintaxis.[1] La información calculada también está estrechamente relacionada con el significado final, o semántica, del programa que se traduce. Como el análisis que realiza un compilador es estático por definición (tiene lugar antes de la ejecución), dicho análisis semántico también se conoce como **análisis semántico estático**. En un lenguaje típico estáticamente tipificado como C, el análisis semántico involucra la construcción de una tabla de símbolos para mantenerse al tanto de los significados de nombres establecidos en declaraciones e inferir tipos y verificarlos en expresiones y sentencias con el fin de determinar su exactitud dentro de las reglas de tipos del lenguaje.

El análisis semántico se puede dividir en dos categorías. La primera es el análisis de un programa que requiere las reglas del lenguaje de programación para establecer su exactitud y garantizar una ejecución adecuada. La complejidad de un análisis de esta clase requerido por una definición del lenguaje varía enormemente de lenguaje a lenguaje. En lenguajes orientados en forma dinámica, tales como LISP y Smalltalk, puede no haber análisis semántico estático en absoluto, mientras que en un lenguaje como Ada existen fuertes requerimientos que debe cumplir un programa para ser ejecutable. Otros lenguajes se encuentran entre estos extremos (Pascal, por ejemplo, no es tan estricto en sus requerimientos estáticos como Ada y C, pero no es tan condescendiente como LISP).

1. Este punto fue comentado con algún detalle en la sección 3.6.3 del capítulo 3.

La segunda categoría de análisis semántico es el análisis realizado por un compilador para mejorar la eficiencia de ejecución del programa traducido. Esta clase de análisis por lo regular se incluye en análisis de "optimización", o técnicas de mejoramiento de código. Investigaremos algunos de estos métodos en el capítulo sobre generación de código, mientras que en este capítulo nos enfocaremos en los análisis comunes que por exactitud son requeridos para una definición del lenguaje. Conviene advertir que las técnicas estudiadas aquí se aplican a ambas situaciones. También que las dos categorías no son mutuamente excluyentes, ya que los requerimientos de exactitud, tales como la verificación de tipos estáticos, también permiten que un compilador genere código más eficiente que para un lenguaje sin estos requerimientos. Además, vale la pena hacer notar que los requerimientos de exactitud que aquí se comentan nunca pueden establecer la exactitud completa de un programa, sino sólo una clase de exactitud parcial. Tales requerimientos todavía son útiles debido a que proporcionan al programador información para mejorar la seguridad y fortaleza del programa.

El análisis semántico estático involucra tanto la **descripción** de los análisis a realizar como la **implementación** de los análisis utilizando algoritmos apropiados. En este sentido, es semejante al análisis léxico y sintáctico. En el análisis sintáctico, por ejemplo, utilizamos gramáticas libres de contexto en la Forma Backus-Naus (BNF, por sus siglas en inglés) para describir la sintaxis y diversos algoritmos de análisis sintáctico descendente y ascendente para implementar la sintaxis. En el análisis semántico la situación no es tan clara, en parte porque no hay un método estándar (como el BNF) que permita especificar la semántica estática de un lenguaje, y en parte porque la cantidad y categoría del análisis semántico estático varía demasiado de un lenguaje a otro. Un método para describir el análisis semántico que los escritores de compiladores usan muy a menudo con buenos efectos es la identificación de **atributos**, o propiedades, de entidades del lenguaje que deben calcularse y escribir **ecuaciones de atributos** o **reglas semánticas**, que expresan cómo el cálculo de tales atributos está relacionado con las reglas gramaticales del lenguaje. Un conjunto así de atributos y ecuaciones se denomina **gramática con atributos**. Las gramáticas con atributos son más útiles para los lenguajes que obedecen el principio de la **semántica dirigida por sintaxis**, la cual asegura que el contenido semántico de un programa se encuentra estrechamente relacionado con su sintaxis. Todos los lenguajes modernos tienen esta propiedad. Por desgracia, el escritor de compiladores casi siempre debe construir una gramática con atributos a mano a partir del manual del lenguaje, ya que rara vez la da el diseñador del lenguaje. Aún peor, la construcción de una gramática con atributos puede complicarse innecesariamente debido a su estrecha adhesión con la estructura sintáctica explícita del lenguaje. Un fundamento mucho mejor para la expresión de los cálculos semánticos es la sintaxis abstracta, como se representa mediante un árbol sintáctico abstracto. Incluso, el diseñador del lenguaje, también suele dejar al escritor del compilador la especificación de la sintaxis abstracta.

Los algoritmos para la implementación del análisis semántico tampoco son tan claramente expresables como los algoritmos de análisis sintáctico. De nuevo, esto se debe en parte a los mismos problemas que se acaban de mencionar respecto a la especificación del análisis semántico. No obstante, existe un problema adicional causado por la temporización del análisis durante el proceso de compilación. Si el análisis semántico se puede suspender hasta que todo el análisis sintáctico (y la construcción de un árbol sintáctico abstracto) esté completo, entonces la tarea de implementar el análisis semántico se vuelve considerablemente más fácil y consiste en esencia en la especificación de orden para un recorrido del árbol sintáctico, junto con los cálculos a realizar cada vez que se encuentra un nodo en el recorrido. Sin embargo, esto implica que el compilador debe ser de paso

múltiple. Si, por otra parte, el compilador necesita realizar todas sus operaciones en un solo paso (incluyendo la generación del código), entonces la implementación del análisis semántico se convierte en mucho más que un proceso a propósito para encontrar un orden correcto y un método para calcular la información semántica (suponiendo que un orden correcto así exista en realidad). Afortunadamente, la práctica moderna cada vez más permite al escritor de compiladores utilizar pasos múltiples para simplificar los procesos de análisis semántico y generación de código.

A pesar de este estado algo desordenado del análisis semántico, es muy útil para estudiar gramáticas con atributos y cuestiones de especificación, ya que esto redundará en la capacidad de escribir un código más claro, más conciso y menos proclive a errores para análisis semántico, además de permitir una comprensión más fácil de ese código.

Por lo tanto, el capítulo comienza con un estudio de atributos y gramáticas con atributos. Continúa con técnicas para implementar los cálculos especificados mediante una gramática con atributos, incluyendo la inferencia de un orden para los cálculos y los recorridos del árbol que los acompañan. Dos secciones posteriores se concentran en las áreas principales del análisis semántico: tablas de símbolos y verificación de tipos. La última sección describe un analizador semántico para el lenguaje de programación TINY presentado en capítulos anteriores.

A diferencia de capítulos previos, este capítulo no contiene ninguna descripción de un **generador de analizador semántico** ni alguna herramienta general para construir analizadores semánticos. Aunque se han construido varias de tales herramientas, ninguna ha logrado el amplio uso y disponibilidad de Lex o Yacc. En la sección de notas y referencias al final del capítulo mencionamos algunas de estas herramientas y proporcionamos referencias a la literatura para el lector interesado.

6.1 ATRIBUTOS Y GRAMÁTICAS CON ATRIBUTOS

Un **atributo** es cualquier propiedad de una construcción del lenguaje de programación. Los atributos pueden variar ampliamente en cuanto a la información que contienen, su complejidad y en particular el tiempo que les toma realizar el proceso de traducción/ejecución cuando pueden ser determinados. Ejemplos típicos de atributos son

- El tipo de datos de una variable
- El valor de una expresión
- La ubicación de una variable en la memoria
- El código objeto de un procedimiento
- El número de dígitos significativos en un número

Los atributos se pueden establecer antes del proceso de compilación (o incluso la construcción de un compilador). Por ejemplo, el número de dígitos significativos en un número se puede establecer (o por lo menos dar un valor mínimo) mediante la definición de un lenguaje. Además, los atributos sólo se pueden determinar durante la ejecución del programa, tal como el valor de una expresión (no constante), o la ubicación de una estructura de datos dinámicamente asignada. El proceso de calcular un atributo y asociar su valor calculado con la construcción del lenguaje en cuestión se define como **fijación** del atributo. El tiempo que toma el proceso de compilación/ejecución cuando se presenta la fijación de un atributo se denomina **tiempo de fijación**. Los tiempos de fijación de atributos diferentes varían, e incluso el mismo atributo puede tener tiempos de fijación bastante diferentes de un lenguaje

a otro. Los atributos que pueden fijarse antes de la ejecución se denominan **estáticos**, mientras que los atributos que sólo se pueden fijar durante la ejecución son **dinámicos**. Naturalmente, un escritor de compiladores está interesado en aquellos atributos estáticos que se fijan durante el tiempo de traducción.

Considere la lista de muestra de atributos dada con anterioridad. Analizamos el tiempo de fijación y la significancia durante la compilación de cada uno de los atributos en la lista.

- En un lenguaje estáticamente tipificado como C o Pascal, el tipo de datos de una variable o expresión es un importante atributo en tiempo de compilación. Un **verificador de tipo** es un analizador semántico que calcula el atributo de tipo de datos de todas las entidades del lenguaje para las cuales están definidos los tipos de datos y verifica que estos tipos cumplan con las reglas de tipo del lenguaje. En un lenguaje como C o Pascal, un verificador de tipo es una parte importante del análisis semántico. Sin embargo, en un lenguaje como LISP, los tipos de datos son dinámicos, y un compilador LISP debe generar código para calcular tipos y realizar la verificación de tipos durante la ejecución del programa.
- Los valores de las expresiones por lo regular son dinámicos, y un compilador generará código para calcular sus valores durante la ejecución. Sin embargo, algunas expresiones pueden, de hecho, ser constantes (por ejemplo 3 + 4 * 5), y un analizador semántico puede elegir evaluarlas durante la compilación (este proceso se conoce como **incorporación de constantes**).
- La asignación de una variable puede ser estática o dinámica, dependiendo del lenguaje y las propiedades de la variable misma. Por ejemplo, en FORTRAN77 todas las variables se asignan de manera estática, mientras que en LISP todas las variables se asignan dinámicamente. C y Pascal tienen una mezcla de asignación de variables estática y dinámica. Un compilador por lo regular pospondrá los cálculos asociados con la asignación de variables hasta la generación de código, porque tales cálculos dependen del ambiente de ejecución y, ocasionalmente, de los detalles de la máquina objetivo. (El capítulo siguiente trata esto con más detalle.)
- El código objeto de un procedimiento es evidentemente un atributo estático. El generador de código de un compilador está comprometido por completo con el cálculo de este atributo.
- El número de dígitos significativos en un número es un atributo que con frecuencia no se trata explícitamente durante la compilación. Está implícito en el sentido de que el escritor de compiladores implementa las representaciones para los valores, lo que en general se considera como parte del ambiente de ejecución que se analiza en el capítulo siguiente. Sin embargo, incluso el analizador léxico puede necesitar conocer el número de dígitos significativos permitidos si va a producir constantes de manera correcta.

Como vimos a partir de estos ejemplos, los cálculos de atributos son muy variados. Cuando aparecen de manera explícita en un compilador pueden presentarse en cualquier momento durante la compilación: aun cuando asociamos el cálculo de atributos más estrechamente con la fase de análisis semántico, tanto el analizador léxico como el analizador sintáctico pueden necesitar disponer de información de atributo, y puede ser necesario realizar algún análisis semántico al mismo tiempo que el análisis sintáctico. En este capítulo nos enfocamos en cálculos típicos que se presentan antes de la generación de código y después del análisis sintáctico (pero véase la sección 6.2.5 para información sobre el análisis

semántico durante el análisis sintáctico). El análisis de atributos que se puede aplicar directamente a la generación de código se analizará en el capítulo 8.

6.1.1 Gramáticas con atributos

En la **semántica dirigida por sintaxis** los atributos están directamente asociados con los símbolos gramaticales del lenguaje (los terminales y no terminales).[2] Si X es un símbolo gramatical, y a es un atributo asociado con X, entonces escribimos $X.a$ para el valor de a asociado con X. Esta notación nos recuerda el indicador de campo de registro de Pascal o (de manera equivalente) una operación de miembro de estructura en C. En realidad, una manera típica de implementar cálculos de atributo es colocar valores de atributo en los nodos de un árbol sintáctico utilizando campos de registro (o miembros de estructura). Veremos esto con más detalle en la siguiente sección.

Dada una colección de atributos $a_1, \ldots, a_k$, el principio de la semántica dirigida por sintaxis implica que para cada regla gramatical $X_0 \rightarrow X_1 X_2 \ldots X_n$ (donde X_0 es un no terminal, mientras que las otras X_i son símbolos arbitrarios), los valores de los atributos $X_i.a_j$ de cada símbolo gramatical X_i están relacionados con los valores de los atributos de los otros símbolos en la regla. Si el mismo símbolo X_i apareciera más de una vez en la regla gramatical, entonces cada ocurrencia tendría que distinguirse de las otras mediante una subindización adecuada, de manera que se puedan diferenciar los valores de atributo de diferentes ocurrencias. Cada relación está especificada por una **ecuación de atributo** o **regla semántica**[3] de la forma

$$X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k, X_1.a_1, \ldots, X_1.a_k, \ldots, X_n.a_1, \ldots, X_n.a_k)$$

donde f_{ij} es una función matemática de sus argumentos. Una **gramática con atributos** para los atributos $a_1, \ldots, a_k$ es la colección de todas esas ecuaciones para todas las reglas gramaticales del lenguaje.

En esta generalidad las gramáticas con atributos pueden parecer muy complejas. En la práctica las funciones f_{ij} por lo regular son bastante simples. Por otro lado, es raro que los atributos dependan de un gran número de otros atributos, así que los atributos pueden separarse en pequeños conjuntos independientes de atributos interdependientes, y las gramáticas con atributos pueden escribirse de manera separada para cada conjunto.

Por lo general, las gramáticas con atributos se escriben en forma tabular, con cada regla gramatical enumerada con el conjunto de ecuaciones de atributo, o reglas semánticas asociadas con esa regla de la manera siguiente:[4]

---

2. La semántica dirigida por sintaxis podría fácilmente ser llamada **sintaxis dirigida por semántica**, puesto que en la mayoría de los lenguajes la sintaxis se diseña con la semántica (incidental) de la construcción ya en mente.

3. En un trabajo posterior veremos las reglas semánticas como algo más general que ecuaciones de atributo. Por ahora, el lector puede considerarlas idénticas.

4. Siempre usaremos el encabezado "reglas semánticas" en estas tablas en vez de "ecuaciones de atributo" para tener en cuenta una interpretación más general de las reglas semánticas más adelante.

Regla gramatical	Reglas semánticas
Regla 1	Ecuaciones de atributo asociadas
.	.
.	.
.	.
Regla *n*	Ecuaciones de atributo asociadas

Continuamos directamente con varios ejemplos.

Ejemplo 6.1

Consideremos la siguiente gramática simple para números sin signo:

$$\textit{número} \rightarrow \textit{número dígito} \mid \textit{dígito}$$
$$\textit{dígito} \rightarrow \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \mathtt{3} \mid \mathtt{4} \mid \mathtt{5} \mid \mathtt{6} \mid \mathtt{7} \mid \mathtt{8} \mid \mathtt{9}$$

El atributo más importante de un número es su valor, al cual le asignamos el nombre *val*. Cada dígito tiene un valor que es directamente calculable del dígito real que representa. Así, por ejemplo, la regla gramatical *dígito* → 0 implica que *dígito* tiene valor 0 en este caso. Esto se puede expresar mediante la ecuación de atributo *dígito.val* = 0, y asociamos esta ecuación con la regla *dígito* → 0. Además, cada número tiene un valor basado en el dígito que contiene. Si un número se deriva utilizando la regla,

$$\textit{número} \rightarrow \textit{dígito}$$

entonces el número contiene precisamente un dígito, y su valor es el valor de este dígito. La ecuación de atributo que expresa este hecho es

$$\textit{número.val} = \textit{dígito.val}$$

Si un número contiene más de un dígito, entonces se deriva utilizando la regla gramatical

$$\textit{número} \rightarrow \textit{número dígito}$$

y debemos expresar la relación entre el valor del símbolo en el lado izquierdo de la regla gramatical y los valores de los símbolos en el lado derecho. Advierta que las dos ocurrencias de *número* en la regla gramatical deben distinguirse, puesto que el *número* a la derecha tendrá un valor diferente del correspondiente al *número* en la izquierda. Los distinguiremos utilizando subíndices, y la regla gramatical la escribiremos como se muestra a continuación:

$$\textit{número}_1 \rightarrow \textit{número}_2\ \textit{dígito}$$

Ahora considere un número como **34**. Una derivación (por la izquierda) de este número es de la manera siguiente: *número* ⇒ *número dígito* ⇒ *dígito dígito* ⇒ **3** *dígito* ⇒ **34**. Considere el uso de la regla gramatical $número_1 \rightarrow número_2\ dígito$ en el primer paso de esa derivación. El no terminal $número_2$ corresponde al dígito **3**, mientras que *dígito* corresponde al dígito **4**. Los valores de cada uno son 3 y 4, respectivamente. Para obtener el valor de $número_1$ (es decir, 34), debemos multiplicar el valor de $número_2$ por 10 y agregar el valor de *dígito*: 34 = 3 * 10 + 4. En otras palabras, desplazamos el 3 un lugar decimal hacia la izquierda y sumamos 4. Esto corresponde a la ecuación de atributos

$$número_1.val = número_2.val * 10 + dígito.val$$

Una gramática con atributos completa para el atributo *val* se da en la tabla 6.1.

El significado de las ecuaciones de atributo para una cadena particular pueden visualizarse utilizando el árbol de análisis gramatical para la cadena. Por ejemplo, el árbol de análisis gramatical para el número **345** se da en la figura 6.1. En esta figura el cálculo correspondiente a la ecuación de atributo apropiada se muestra debajo de cada nodo interior. Visualizar las ecuaciones de atributo como cálculos en el árbol de análisis gramatical es importante para los algoritmos que calculan valores de atributo, como veremos en la sección siguiente.[5]

Tanto en la tabla 6.1 como en la figura 6.1 pusimos énfasis en la diferencia entre la representación sintáctica de un dígito y el valor, o contenido semántico, del dígito utilizando diferentes fuentes tipográficas. Por ejemplo, en la regla gramatical *dígito* → **0**, el dígito **0** es un token o carácter, mientras que *dígito.val* = 0 significa que el dígito tiene el valor numérico 0.

Tabla 6.1
Gramática con atributos para el ejemplo 6.1

Regla gramatical	Reglas semánticas
$número_1 \rightarrow$ $número_2\ dígito$	$número_1.val =$ $número_2.val * 10 + dígito.val$
número → *dígito*	*número.val* = *dígito.val*
dígito → **0**	*dígito.val* = 0
dígito → **1**	*dígito.val* = 1
dígito → **2**	*dígito.val* = 2
dígito → **3**	*dígito.val* = 3
dígito → **4**	*dígito.val* = 4
dígito → **5**	*dígito.val* = 5
dígito → **6**	*dígito.val* = 6
dígito → **7**	*dígito.val* = 7
dígito → **8**	*dígito.val* = 8
dígito → **9**	*dígito.val* = 9

5. De hecho, los números por lo regular se reconocen como token por el analizador léxico, y sus valores numéricos se calculan fácilmente al mismo tiempo. Sin embargo, al hacerlo así es probable que el analizador léxico utilice de manera implícita las ecuaciones de atributo especificadas aquí.

Figura 6.1
Árbol sintáctico que muestra cálculos de atributo para el ejemplo 6.1

§

Ejemplo 6.2 Considere la gramática siguiente para expresiones aritméticas enteras simples:

$$exp \rightarrow exp + term \mid exp - term \mid term$$
$$term \rightarrow term * factor \mid factor$$
$$factor \rightarrow (\ exp \ ) \mid \textbf{número}$$

Esta gramática es una versión un poco modificada de la gramática de expresión simple estudiada extensamente en capítulos anteriores. El atributo principal de una *exp* (o *term* o *factor*) es su valor numérico, el cual escribimos como *val*. Las ecuaciones de atributo para el atributo *val* se proporcionan en la tabla 6.2.

Tabla 6.2
Gramática con atributos para el ejemplo 6.2

Regla gramatical	Reglas semánticas
$exp_1 \rightarrow exp_2 + term$	$exp_1.val = exp_2.val + term.val$
$exp_1 \rightarrow exp_2 - term$	$exp_1.val = exp_2.val - term.val$
$exp \rightarrow term$	$exp.val = term.val$
$term_1 \rightarrow term_2 * factor$	$term_1.val = term_2.val * factor.val$
$term \rightarrow factor$	$term.val = factor.val$
$factor \rightarrow (\ exp \ )$	$factor.val = exp.val$
$factor \rightarrow$ **número**	$factor.val =$ **número**$.val$

Estas ecuaciones expresan la relación entre la sintaxis de las expresiones y la semántica de los cálculos aritméticos que se realizarán. Observe, por ejemplo, la diferencia entre el símbolo sintáctico + (un token) en la regla gramatical

$$exp_1 \rightarrow exp_2 + term$$

y la operación de adición o suma aritmética + que se realiza en la ecuación

$$exp_1.val = exp_2.val + term.val$$

Advierta también que no hay ecuación con **número**.*val* en el lado izquierdo. Como veremos en la sección siguiente, esto implica que **número**.*val* debe calcularse antes de cualquier análisis semántico que utilice esta gramática con atributos (por ejemplo, mediante el análisis léxico). De manera alternativa, si queremos que este valor esté explícito en la gramática con atributos, debemos agregar reglas gramaticales y ecuaciones de atributo a la gramática con atributos (por ejemplo, las ecuaciones del ejemplo 6.1).

También podemos expresar los cálculos implicados mediante esta gramática con atributos agregando ecuaciones a los nodos en un árbol de análisis gramatical, como en el ejemplo 6.1. Por ejemplo, dada la expresión `(34-3)*42`, podemos expresar la semántica de su valor en su árbol de análisis gramatical como en la figura 6.2. §

Figura 6.2
Árbol de análisis gramatical para `(34-3)*42` que muestra cálculos del atributo *val* para la gramática con atributos del ejemplo 6.2

Ejemplo 6.3

Considere la siguiente gramática simple para declaraciones de variable en una sintaxis tipo C:

decl → *type var-list*
type → `int` | `float`
var-list → `id`, *var-list* | `id`

Queremos definir un atributo de tipo de datos para las variables dadas por los identificadores en una declaración y escribir ecuaciones que expresen cómo está relacionado el atributo de tipo de datos con el tipo de la declaración. Hacemos esto construyendo una gramática con atributos para un atributo *dtype* (utilizamos el nombre *dtype* para distinguir el atributo del no terminal *type*). La gramática con atributos para *dtype* se da en la tabla 6.3. Haremos las observaciones siguientes respecto a las ecuaciones de atributo de esa figura.

En primer lugar, los valores de *dtype* son del conjunto {*integer, real*} que corresponde a los tokens `int` y `float`. El no terminal *type* tiene un *dtype* dado por el token que representa. Este *dtype* corresponde al *dtype* de la *var-list* entera, por la ecuación asociada con la regla gramatical para *decl*. Cada `id` en la lista tiene este mismo *dtype*, por las ecuaciones asociadas con *var-list*. Advierta que no hay ecuación que involucre el *dtype* del no terminal *decl*. En realidad, una *decl* no necesita tener un *dtype*: no es necesario especificar el valor de un atributo para todos los símbolos gramaticales.

Como antes, las ecuaciones de atributo se pueden exhibir en un árbol de análisis gramatical. Se proporciona un ejemplo en la figura 6.3. §

Tabla 6.3
Gramática con atributos para el ejemplo 6.3

Regla gramatical	Reglas semánticas
decl → *type var-list*	*var-list.dtype* = *type.dtype*
type → `int`	*type.dtype* = *integer*
type → `float`	*type.dtype* = *real*
$var\text{-}list_1$ → `id,` $var\text{-}list_2$	`id`.*dtype* = $var\text{-}list_1$.*dtype*
	$var\text{-}list_2$.*dtype* = $var\text{-}list_1$.*dtype*
var-list → `id`	`id`.*dtype* = *var-list.dtype*

Figura 6.3
Árbol de análisis gramatical para la cadena `float x,y` que muestra el atributo *dtype* como se especifica mediante la gramática con atributos de la tabla 6.3

En los ejemplos que hemos visto hasta ahora, sólo había un atributo. Las gramáticas con atributos pueden involucrar varios atributos interdependientes. El ejemplo siguiente es una situación simple donde existen atributos interdependientes.

Ejemplo 6.4

Considere una modificación de la gramática numérica del ejemplo 6.1, donde los números pueden ser octales o decimales. Imaginemos que esto se indica mediante un sufijo de un carácter `o` (para octal) o `d` (para decimal). Entonces tenemos la gramática siguiente:

num-base → *num carbase*
carbase → `o` | `d`
num → *num dígito* | *dígito*
dígito → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

En este caso *num* y *dígito* requieren un nuevo atributo *base*, que se utiliza para calcular el atributo *val*. La gramática con atributos para *base* y *val* se proporciona en la tabla 6.4.

Tabla 6.4
Gramática con atributos para el ejemplo 6.4

Regla gramatical	Reglas semánticas
num-base → *num carbase*	*num-base.val* = *num.val* *num.base* = *carbase.base*
carbase → o	*carbase.base* = 8
carbase → d	*carbase.base* = 10
$num_1 \rightarrow num_2$ *dígito*	num_1 *.val* = **if** *dígito.val* = *error* **or** num_2 *.val* = *error* **then** *error* **else** num_2 *.val* * num_1 *.base* + *dígito.val* num_2 *.base* = num_1 *.base* *dígito.base* = num_1 *.base*
num → *dígito*	*num.val* = *dígito.val* *dígito.base* = *num.base*
dígito → 0	*dígito.val* = 0
dígito → 1	*dígito.val* = 1
...	...
dígito → 7	*dígito.val* = 7
dígito → 8	*dígito.val* = **if** *dígito.base* = 8 **then** *error* **else** 8
dígito → 9	*dígito.val* = **if** *dígito.base* = 8 **then** *error* **else** 9

Deberían observarse dos nuevas características en esta gramática con atributos. En primer lugar, la gramática BNF no elimina por sí misma la combinación errónea de los dígitos (no octales) 8 y 9 con el sufijo o. Por ejemplo, la cadena 189o es sintácticamente correcta de acuerdo con la BNF anterior, pero no se le puede asignar ningún valor. De este modo, es necesario un nuevo valor *error* para tales casos. Además, la gramática con atributos debe expresar el hecho de que la introducción de 8 o 9 en un número con un sufijo o produce un valor de *error*. La manera más fácil de hacer esto es emplear una expresión **if-then-else** en las funciones de las ecuaciones y atributo apropiadas. Por ejemplo, la ecuación

$$
\begin{aligned}
& num_1 .val = \\
& \qquad \textbf{if } dígito.val = error \textbf{ or } num_2 .val = error \\
& \qquad \textbf{then } error \\
& \qquad \textbf{else } num_2 .val * num_1 .base + dígito.val
\end{aligned}
$$

correspondiente a la regla gramatical $num_1 \rightarrow num_2$ *dígito* expresa el hecho de que si cualquiera de $num_2.val$ o *dígito.val* es *error* entonces $num_1.val$ también debe ser *error*, y sólo si no es ése el caso $num_1.val$ está dado por la fórmula $num_2.val * num_1.base + dígito.val$.

Para concluir con este ejemplo mostramos de nueva cuenta los cálculos de atributo en un árbol de análisis gramatical. La figura 6.4 ofrece un árbol de análisis gramatical para el número 345o, junto con los valores de atributo calculados de acuerdo con la gramática con atributos de la tabla 6.4.

Figura 6.4
Árbol de análisis gramatical que muestra cálculos de atributo para el ejemplo 6.4

6.1.2 Simplificaciones y extensiones a gramática con atributos

El uso de una expresión **if-then-else** extiende las clases de expresiones que pueden aparecer en una ecuación de atributo de una manera útil. La colección de expresiones permisibles en una ecuación de atributo se conoce como **metalenguaje** para la gramática con atributos. Por lo regular queremos un metalenguaje cuyo significado sea suficientemente claro para que no surja confusión sobre su propia semántica. También deseamos un metalenguaje que sea cercano a un lenguaje de programación real, ya que, como veremos en breve, queremos convertir las ecuaciones de atributo en código de trabajo en un analizador semántico. En este libro utilizamos un metalenguaje que está limitado a expresiones aritméticas, lógicas y algunas otras clases de expresiones, junto con una expresión **if-then-else**, y ocasionalmente una expresión **case** o **switch**.

Una característica adicional útil al especificar ecuaciones de atributo es agregar al metalenguaje el uso de funciones cuyas definiciones se puedan dar en otra parte. Por ejemplo, en las gramáticas para números hemos estado escribiendo ecuaciones de atributo para cada una de las opciones de *dígito*. En cambio, podríamos adoptar una convención más concisa escribiendo la regla gramatical para *dígito* como *dígito* → `D` (donde se entiende que `D` es uno de los dígitos) y entonces escribir la ecuación de atributo correspondiente como

$$dígito.val = numval(\texttt{D})$$

Aquí *numval* es una función cuya definición debe ser especificada en otra parte como un suplemento para la gramática con atributos. Por ejemplo, podemos dar la siguiente definición de *numval* como código en C:

```
int numval(char D)
{ return (int)D - (int)'0';}
```

Una simplificación adicional que puede ser útil al especificar gramática con atributos es utilizar una forma ambigua, pero más simple, de la gramática original. De hecho, puesto que se supone que el analizador sintáctico ya fue construido, toda ambigüedad se habrá abordado en esta etapa, y la gramática con atributos puede basarse libremente en construcciones ambiguas, sin implicar ninguna ambigüedad en los atributos resultantes. Por ejemplo, la gramática de expresión aritmética del ejemplo 6.2 tiene la siguiente forma más sencilla, pero ambigua:

$$exp \rightarrow exp + exp \mid exp - exp \mid exp * exp \mid (exp) \mid \textbf{número}$$

Cuando se utiliza esta gramática el atributo *val* se puede definir mediante la tabla que se muestra en la tabla 6.5 (compare ésta con la tabla 6.2).

Tabla 6.5
Definición del atributo *val* para una expresión utilizando una gramática ambigua

Regla gramatical	Reglas semánticas
$exp_1 \rightarrow exp_2 + exp_3$	$exp_1.val = exp_2.val + exp_3.val$
$exp_1 \rightarrow exp_2 - exp_3$	$exp_1.val = exp_2.val - exp_3.val$
$exp_1 \rightarrow exp_2 * exp_3$	$exp_1.val = exp_2.val * exp_3.val$
$exp_1 \rightarrow (exp_2)$	$exp_1.val = exp_2.val$
$exp \rightarrow$ **número**	$exp.val =$ **número**$.val$

También se puede hacer una simplificación exhibiendo valores de atributo mediante el uso de un árbol sintáctico abstracto en lugar de un árbol de análisis gramatical. Un árbol sintáctico abstracto debe tener siempre la estructura suficiente, de manera que se pueda expresar la semántica definida mediante una gramática con atributos. Por ejemplo, la expresión `(34-3)*42`, cuyo árbol de análisis gramatical y atributos *val* se mostraron en la figura 6.2, pueden tener su semántica completamente expresada mediante el árbol sintáctico abstracto de la figura 6.5.

No es de sorprender que el árbol sintáctico mismo se pueda especificar mediante una gramática con atributos como se ve en el ejemplo siguiente.

Figura 6.5
Árbol sintáctico abstracto para `(34-3)*42` que muestra los cálculos de atributo de *val* para la gramática con atributos de la tabla 6.2 o la tabla 6.5

Ejemplo 6.5

Dada la gramática para expresiones aritméticas enteras simples del ejemplo 6.2 (página 264), podemos definir un árbol sintáctico abstracto para expresiones mediante la gramática con atributos dada en la tabla 6.6. En esta gramática con atributos utilizamos dos funciones auxiliares *mkOpNode* y *mkNumNode*. La función *mkOpNode* toma tres parámetros (un token de operador y dos árboles sintácticos) y construye un nuevo nodo de árbol cuya etiqueta de operador es el primer parámetro, y cuyos hijos son el segundo y tercer parámetros. La función *mkNumNode* toma un parámetro (un valor numérico) y construye un nodo hoja

representando un número con ese valor. En la tabla 6.6 escribimos el valor numérico como **número**.*lexval* para indicar que es construido por el analizador léxico. De hecho, éste podría ser el valor numérico real del número, o su representación de cadena, dependiendo de la implementación. (Compare las ecuaciones de la tabla 6.6 con la construcción descendente recursiva del árbol sintáctico de TINY en el apéndice B.)

Tabla 6.6
Gramática con atributos para árboles sintácticos abstractos o expresiones aritméticas enteras simples

Regla gramatical	Reglas semánticas
$exp_1 \rightarrow exp_2$ + *term*	exp_1 .*tree* = *mkOpNode* (+, exp_2 .*tree*, *term.tree*)
$exp_1 \rightarrow exp_2$ - *term*	exp_1 .*tree* = *mkOpNode*(-, exp_2 .*tree*, *term.tree*)
exp → *term*	*exp.tree* = *term.tree*
$term_1 \rightarrow term_2$ * *factor*	$term_1$.*tree* = *mkOpNode*(*, $term_2$.*tree*, *factor.tree*)
term → *factor*	*term.tree* = *factor.tree*
factor → (*exp*)	*factor.tree* = *exp.tree*
factor → **número**	*factor.tree* = *mkNumNode*(**número**.*lexval*)

§

Una cuestión que es fundamental para la especificación de atributos utilizando gramática con atributos es: ¿cómo podemos estar seguros de que una gramática con atributos en particular es consistente y completa, es decir, que define de manera única los atributos dados? La respuesta simple es que hasta ahora no podemos. Ésta es una cuestión similar a la de determinar si una gramática es ambigua. En la práctica, son los algoritmos de análisis sintáctico que utilizamos lo que determina la aceptabilidad de una gramática, y ocurre una situación semejante en el caso de las gramáticas con atributos. De este modo, los métodos algorítmicos para calcular atributos que estudiaremos en la siguiente sección determinarán si una gramática con atributos es adecuada para definir los valores de atributos.

6.2 ALGORITMOS PARA CÁLCULO DE ATRIBUTOS

En esta sección estudiaremos las maneras en que se puede utilizar una gramática con atributos como base para un compilador a fin de calcular y utilizar los atributos definidos por las ecuaciones de la gramática con atributos. Fundamentalmente, esto equivale a convertir las ecuaciones de atributo en reglas de cálculo. De este modo, la ecuación de atributo

$$X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k, X_1.a_1, \ldots, X_1.a_k, \ldots, X_n.a_1, \ldots, X_n.a_k)$$

se visualiza como una asignación del valor de la expresión funcional en el lado derecho para el atributo $X_i.a_j$. Para que esto tenga éxito, los valores de todos los atributos que aparecen en el lado derecho ya deben existir. Este requerimiento se puede pasar por alto en la especificación de las gramáticas con atributos, donde las ecuaciones se pueden escribir en orden arbitrario sin afectar su validez. El problema de implementar un algoritmo correspondiente

a una gramática con atributos consiste ante todo en hallar un orden para la evaluación y asignación de atributos que aseguran que todos los valores de atributo utilizados en cada cálculo estén disponibles cuando éste se realice. Las mismas ecuaciones de atributo indican las limitantes de orden en el cálculo de los atributos, y la primera tarea es hacer explícitas las limitantes de orden utilizando gráficas dirigidas para representarlas. Estas gráficas dirigidas se conocen como gráficas de dependencia.

6.2.1 Gráficas de dependencia y orden de evaluación

Dada una gramática con atributos, cada regla gramatical tiene una **gráfica de dependencia** asociada. Esta gráfica tiene un nodo etiquetado por cada atributo $X_i.a_j$ de cada símbolo en la regla gramatical, y para cada ecuación de atributo

$$X_i.a_j = f_{ij}(\ \ldots, X_m.a_k, \ldots\ )$$

asociada con la regla gramatical hay un borde desde cada nodo $X_m.a_k$ en el lado derecho para el nodo $X_i.a_j$ (que expresa la dependencia de $X_i.a_j$ respecto a $X_m.a_k$). Dada una cadena legal en el lenguaje generado por la gramática libre de contexto, la **gráfica de dependencia** de la cadena es la unión de las gráficas de dependencia de las reglas gramaticales que representan cada nodo (no hoja) del árbol de análisis gramatical de la cadena.

Al dibujar la gráfica de dependencia para cada regla gramatical o cadena, los nodos asociados con cada símbolo X son trazados en un grupo, de manera que las dependencias se puedan visualizar como estructuras alrededor de un árbol de análisis gramatical.

Ejemplo 6.6

Considere la gramática del ejemplo 6.1, con la gramática con atributos como se proporciona en la tabla 6.1. Hay sólo un atributo, *val*, de modo que para cada símbolo existe sólo un nodo en cada gráfica de dependencia, correspondiente a su atributo *val*. La regla gramatical $número_1 \rightarrow número_2\ dígito$ tiene la ecuación de atributo asociada simple

$$número_1.val = número_2.val * 10 + dígito.val$$

La gráfica de dependencia para esta regla gramatical es

(En futuras gráficas de dependencia omitiremos los subíndices para símbolos repetidos, ya que la representación gráfica distingue claramente las diferentes ocurrencias como nodos distintos.)

De manera semejante, la gráfica de dependencia para la regla gramatical $número \rightarrow dígito$ con la ecuación de atributo $número.val = dígito.val$ es

Para las reglas gramaticales restantes de la forma *dígito* → `D` las gráficas de dependencia son triviales (no hay bordes), ya que *dígito.val* se calcula directamente del lado derecho de la regla.

Finalmente, la cadena `345` tiene la siguiente gráfica de dependencia correspondiente a su árbol de análisis gramatical (véase la figura 6.1):

§

Ejemplo 6.7

Considere la gramática del ejemplo 6.3 con la gramática con atributos para el atributo *dtype* que se da en la tabla 6.3. En este ejemplo la regla gramatical $var\text{-}list_1 \rightarrow$ `id,` $var\text{-}list_2$ tiene las dos ecuaciones de atributo asociadas

$$\texttt{id}\ dtype = var\text{-}list_1.dtype$$
$$var\text{-}list_2.dtype = var\text{-}list_1.dtype$$

y la gráfica de dependencia

De manera similar, la regla gramatical *var-list* → `id` tiene la gráfica de dependencia

Las dos reglas *type* → `int` y *type* → `float` tienen gráficas de dependencia triviales.

Finalmente, la regla *decl* → *type var-list* con la ecuación asociada *var-list.dtype* = *type.dtype* tiene la gráfica de dependencia

En este caso, como *decl* no está involucrada de manera directa en la gráfica de dependencia, no está completamente claro cuál regla gramatical tiene su gráfica asociada a ella. Por esta razón (y algunas otras razones que comentaremos posteriormente), a menudo dibujamos la gráfica de dependencia sobrepuesta sobre un segmento de árbol de análisis gramatical correspondiente a la regla gramatical. De este modo, la gráfica de dependencia anterior se puede dibujar como

y esto hace más clara la regla gramatical a la que está asociada la dependencia. Advierta también que cuando dibujamos los nodos del árbol de análisis gramatical suprimimos la notación punto para los atributos, y representamos los atributos de cada nodo escribiéndolos a continuación de su nodo asociado. Así, la primera gráfica de dependencia en este ejemplo también se puede escribir como

Finalmente, la gráfica de dependencia para la cadena `float x, y` es

§

Ejemplo 6.8 Considere la gramática de números de base del ejemplo 6.4 con la gramática con atributos para los atributos *base* y *val* como se dan en la tabla 6.4. Dibujaremos las gráficas de dependencia para las cuatro reglas gramaticales

num-base → *num carbase*
num → *num dígito*
num → *dígito*
dígito → 9

y para la cadena `345o`, cuyo árbol de análisis gramatical se muestra en la figura 6.4.

Comenzamos con la gráfica de dependencia para la regla gramatical *num-base* → *num carbase*:

Esta gráfica expresa las dependencias de las dos ecuaciones asociadas *num-base.val* = *num.val* y *num.base* = *carbase.base*.

A continuación dibujamos la gráfica de dependencia correspondiente a la regla gramatical *num* → *num dígito*:

Esta gráfica expresa las dependencias de las tres ecuaciones de atributo

$num_1.val =$
  if *dígito.val* = *error* **or** $num_2.val$ = *error*
  then *error*
  else $num_2.val * num_1.base$ + *dígito.val*
$num_2.base = num_1.base$
dígito.base = $num_1.base$

La gráfica de dependencia para la regla gramatical *num* → *dígito* es similar:

Finalmente, dibujamos la gráfica de dependencia para la regla gramatical *dígito* → **9**:

Esta gráfica expresa la dependencia creada por la ecuación *dígito.val* = **if** *dígito.base* = 8 **then** *error* **else** 9, a saber, que *dígito.val* depende de *dígito.base* (es parte de la prueba en la expresión **if**). Sólo resta dibujar la gráfica de dependencia para la cadena `345o`. Esto se hace en la figura 6.6.

Figura 6.6
Gráfica de dependencia para la cadena `345o` (ejemplo 6.8)

§

Supongamos ahora que deseamos determinar un algoritmo que calcula los atributos de una gramática con atributos utilizando las ecuaciones de atributo como reglas para el cálculo. Dada una cadena particular de tokens para traducirse, la gráfica de dependencia del árbol de análisis gramatical de la cadena da un conjunto de limitantes de orden bajo el cual el algoritmo debe operar para calcular los atributos de esa cadena. En realidad, cualquier algoritmo debe calcular el atributo en cada nodo de la gráfica de dependencia *antes* de intentar calcular cualquier atributo sucesor. Un orden del recorrido de la gráfica de dependencia que obedece esta restricción se denomina **clasificación topológica**, y una muy conocida condición necesaria y suficiente para que exista una clasificación topológica es que la gráfica debe ser **acíclica**. Tales gráficas se conocen como **gráficas acíclicas dirigidas**, o **DAG**, por sus siglas en inglés.

Ejemplo 6.9

La gráfica de dependencia de la figura 6.6 es una DAG. En la figura 6.7 numeramos los nodos de la gráfica (y eliminamos el árbol de análisis gramatical subyacente para facilitar la visualización). Una clasificación topológica está dada por el orden de nodo en el que los nodos están numerados. Otra clasificación topológica está dada por el orden

12 6 9 1 2 11 3 8 4 5 7 10 13 14

Figura 6.7
Gráfica de dependencia para la cadena 3450 (ejemplo 6.9)

§

Una cuestión que surge en el uso de una clasificación topológica de la gráfica de dependencia para calcular valores de atributo es cómo se encuentran los valores de atributo en las raíces de la gráfica (una **raíz** de una gráfica es un nodo que no tiene predecesores). En la figura 6.7, los nodos 1, 6, 9 y 12 son todos raíces de la gráfica.[6] Los valores de atributo en estos nodos no dependen de ningún otro atributo, entonces se deben calcular utilizando la información que está directamente disponible. Esta información se encuentra a menudo en forma de tokens que son hijos de los correspondientes nodos de árbol de análisis gramatical. En la figura 6.7, por ejemplo, el *val* del nodo 6 depende del token **3** que es el hijo del nodo *dígito* al que *val* corresponde (véase la figura 6.6). De este modo, el valor de atributo en el nodo 6 es 3. Todos esos valores raíz necesitan ser calculables antes del cálculo de cualquier otro valor de atributo. Tales cálculos se realizan con frecuencia mediante los analizadores léxico o sintáctico.

6. No se debe confundir una raíz de la gráfica de dependencia con la raíz del árbol de análisis gramatical.

Es posible basar un algoritmo para análisis de atributos en una construcción de la gráfica de dependencia durante la compilación y una clasificación topológica subsiguiente de la gráfica de dependencia con el fin de determinar un orden para la evaluación de los atributos. Como la construcción de la gráfica de dependencia está basada en el árbol de análisis gramatical específico en tiempo de compilación, este método se conoce en ocasiones como **método de árbol de análisis gramatical**. Dicho método es capaz de evaluar los atributos en cualquier gramática con atributos que sea **no circular**, es decir, una gramática con atributos para la cual *toda* gráfica de dependencia posible es acíclica.

Existen varios problemas con este método. En primer lugar, tenemos la complejidad agregada requerida por la construcción en tiempo de compilación de la gráfica de dependencia. En segundo lugar, aunque este método puede determinar si una gráfica de dependencia es acíclica en tiempo de compilación, por lo general es inadecuado esperar hasta el momento de compilación para descubrir un círculo vicioso, ya que es casi seguro que éste representa un error en la gramática con atributos original. En realidad, para evitarlo debería probarse por adelantado una gramática con atributos. Existe un algoritmo para hacer esto (véase la sección de notas y referencias), pero es un algoritmo de tiempo exponencial. Naturalmente, este algoritmo sólo necesita ejecutarse una vez, en el tiempo de construcción del compilador, de modo que éste no es un argumento aplastante en su contra (al menos para los propósitos de la construcción del compilador). La complejidad de este método es un argumento más convincente.

La alternativa al método anterior para evaluación de atributos, por la que opta prácticamente todo compilador, es que el escritor de compiladores analice la gramática con atributos y fije un orden para la evaluación de atributos en tiempo de construcción de compilador. Aunque este método todavía utiliza el árbol de análisis gramatical como guía para evaluación de atributos, el método se conoce como **método basado en reglas**, ya que depende de un análisis de las ecuaciones de atributo, o reglas semánticas. Las gramáticas con atributos para las cuales un orden de evaluación de atributos se puede fijar en tiempo de construcción del compilador forman una clase que es menos general que la clase de todas las gramáticas con atributos no circulares, pero en la práctica todas las gramáticas con atributos razonables tienen esta propiedad. A menudo se les denomina gramáticas con atributos **completamente no circulares**. Continuaremos con una exposición de los algoritmos basados en reglas para esta clase de gramáticas con atributos después del ejemplo siguiente.

Ejemplo 6.10

Considere nuevamente las gráficas de dependencia del ejemplo 6.8 y las clasificaeiones topológicas de la gráfica de dependencia analizada en el ejemplo 6.9 (véase la figura 6.7). Aun cuando los nodos 6, 9 y 12 de la figura 6.7 son raíces de la DAG, y, por consiguiente, todo podría ocurrir al principio de una clasificación topológica, en un método basado en reglas esto no es posible. La razón es que cualquier *val* puede depender de la *base* de su nodo *dígito* asociado, si el token correspondiente es **8** o **9**. Por ejemplo, la gráfica de dependencia para *dígito* → **9** es

Así, en la figura 6.7 el nodo 6 puede haber dependido del nodo 5, el nodo 9 puede haber dependido del nodo 8, y el nodo 12 puede haber dependido del nodo 11. En un método basado en reglas estos nodos serían forzados a ser evaluados después que cualquier nodo del que ellos pudieran depender. Por lo tanto, un orden de evaluación que, digamos, evaluara prime-

ro el nodo 12 (véase el ejemplo 6.9), aunque es correcto para el árbol particular de la figura 6.7, no es un orden válido para un algoritmo basado en reglas, puesto que violaría el orden para otros árboles de análisis gramatical. §

6.2.2 Atributos sintetizados y heredados

La evaluación de atributos basados en reglas depende de un recorrido explícito o implícito del árbol de análisis gramatical o del árbol sintáctico. Diferentes clases de recorridos varían en potencia en términos de las clases de dependencias de atributo que se pueden manejar. Para estudiar las diferencias primero debemos clasificar los atributos mediante las clases de dependencias que exhiben. La clase más simple de dependencia a manejar es la de los atributos sintetizados, que se define a continuación.

Definición

Un atributo es **sintetizado** si todas sus dependencias apuntan de hijo a padre en el árbol de análisis gramatical. De manera equivalente, un atributo a es sintetizado si, dada una regla gramatical $A \rightarrow X_1X_2 \ldots X_n$, la única ecuación de atributo asociada con una a en el lado izquierdo es de la forma

$$A.a = f(X_1.a_1, \ldots, X_1.a_k, \ldots, X_n.a_1, \ldots, X_n.a_k)$$

Una gramática con atributos en la que todos los atributos están sintetizados se conoce como **gramática con atributos S**.

Ya vimos varios ejemplos de atributos sintetizados y gramáticas con atributos S. El atributo *val* de números en el ejemplo 6.1 está sintetizado (véanse las gráficas de dependencia en el ejemplo 6.6, página 271), como el atributo *val* de las expresiones aritméticas enteras simples en el ejemplo 6.2, página 264.

Dado que un árbol de análisis gramatical o árbol sintáctico se ha construido mediante un analizador sintáctico, los valores de atributo de una gramática con atributos S se pueden calcular mediante un recorrido ascendente, o postorden, del árbol. Esto se puede expresar mediante el siguiente pseudocódigo para un evaluador de atributo postorden recursivo:

```
procedure PostEval ( T: treenode );
begin
  for cada hijo C de T do
    PostEval ( C );
  calcule todos los atributos sintetizados de T;
end;
```

Ejemplo 6.11 Consideremos la gramática con atributos del ejemplo 6.2 para expresiones aritméticas simples, con el atributo sintetizado *val*. Dada la estructura siguiente para un árbol sintáctico (tal como el de la figura 6.5, página 269).

```
typedef enum {Plus,Minus,Times} OpKind;
typedef enum {OpKind,ConstKind} ExpKind;
typedef struct streenode
        { ExpKind kind;
          OpKind op;
          struct streenode *lchild,*rchild;
           int val;
        } STreeNode;
typedef STreeNode *SyntaxTree;
```

el pseudocódigo PostEval se traduciría en el código C de la figura 6.8 para un recorrido de izquierda a derecha.

Figura 6.8
Código C para el evaluador de atributo postorden para el ejemplo 6.11

```
void postEval(SyntaxTree t)
{ int temp;
  if (t->kind = OpKind)
  { postEval(t->lchild);
    postEval(t->rchild);
    switch (t->op)
    { case Plus:
        t->val = t->lchild->val + t->rchild->val;
        break;
      case Minus:
        t->val = t->lchild->val - t->rchild->val;
        break;
      case Minus:
        t->val = t->lchild->val * t->rchild->val;
        break;
    } /* end switch */
  } /* end if */
} /* end postEval */
```

§

Por supuesto, no todos los atributos son sintetizados.

Definición

Un atributo no sintetizado se denomina **heredado**.

Ejemplos de atributos heredados que ya hemos visto incluyen al atributo *dtype* del ejemplo 6.3 (página 265) y el atributo *base* del ejemplo 6.4 (página 266). Los atributos heredados tienen dependencias que fluyen ya sea de padre a hijos en el árbol de análisis gramatical (a lo que deben su nombre) o de hermano a hermano. Las figuras 6.9(a) y (b) ilustran las dos categorías básicas de dependencia de atributos heredados. Cada una de estas clases de dependencia se presentan en el ejemplo 6.7 para el atributo *dtype*. La razón de que ambas se clasifiquen como heredadas es que en algoritmos para calcular atributos heredados, la herencia entre hermanos a menudo se implementa de tal manera que los valores de atributo se pasen de hermano a hermano *a través* del padre. En efecto, esto es necesario si los bordes del árbol sintáctico sólo apuntan de padre a hijo (de este modo un hijo no puede

tener acceso a su padre o hermanos directamente). Por otra parte, si algunas estructuras en un árbol sintáctico se implementan a través de apuntadores hermanos, entonces la herencia entre hermanos puede continuar directamente a lo largo de una cadena de hermano, como se representa en la figura 6.9(c).

Figura 6.9
Diferentes clases de dependencias heredadas

(a) Herencia de padres a hermanos

(b) Herencia de hermano a hermano

(c) Herencia hermana vía apuntadores hermanos

Volvemos ahora a los métodos algorítmicos para la evaluación de atributos heredados. Los atributos heredados se pueden calcular mediante un recorrido preorden, o combinado preorden/enorden del árbol de análisis gramatical o del árbol sintáctico. De manera esquemática, esto se puede representar mediante el siguiente pseudocódigo:

```
procedure PreEval ( T: treenode );
begin
  for cada hijo C de T do
      calcule todos los atributos heredados de C;
      PreEval ( C );
end;
```

A diferencia de los atributos sintetizados, el orden en el que se calculan los atributos heredados de los hijos es importante, porque los atributos heredados pueden tener dependencias entre los atributos de los hijos. El orden en el que se visita cada hijo *C* de *T* en el pseudocódigo precedente debe, por lo tanto, adherirse a cualquier requerimiento de estas dependencias. En los dos ejemplos siguientes demostraremos esto utilizando los atributos heredados *dtype* y *base* de los ejemplos anteriores.

Ejemplo 6.12

Considere la gramática del ejemplo 6.3 que tiene el atributo heredado *dtype* y cuyas gráficas de dependencia se proporcionan en el ejemplo 6.7, página 272 (véase la gramática con atributos en la tabla 6.3, página 266). Supongamos primero que un árbol de análisis gramatical se ha construido explícitamente a partir de la gramática, la que repetimos aquí para poder hacer referencia a ella con facilidad:

decl → *type var-list*
type → `int` | `float`
var-list → `id,` *var-list* | `id`

Entonces el pseudocódigo para un procedimiento recursivo que calcula el atributo *dtype* para todos los nodos requeridos se muestra a continuación:

```
procedure EvalType ( T: treenode );
begin
  case clasenodo de T of
  decl:
      EvalType ( tipo hijo de T );
      Asignar dtype de tipo hijo de T a var-list hija de T;
      EvalType ( var-list hija de T );
  type:
      if hijo de T = int then T.dtype := integer
      else T.dtype := real;
  var-list:
      asignar T.dtype al primer hijo de T;
      if tercer hijo de T no es nil then
         asignar T.dtype al tercer hijo;
         EvalType ( tercer hijo de T );
  end case;
end EvalType;
```

Advierta cómo operaciones preorden y en orden están mezcladas, dependiendo de la clase diferente de nodo que se está procesando. Por ejemplo, un nodo *decl* requiere que el *dtype* de su primer hijo se calcule primero y después se asigne a su segundo hijo antes de una llamada recursiva de *EvalType* a ese hijo; éste es un proceso en orden. Por otra parte, un nodo *var-list* asigna *dtype* a sus hijos antes de hacer cualquier llamada recursiva; éste es un proceso preorden.

En la figura 6.10 mostramos el árbol de análisis gramatical para la cadena `float x, y` junto con las gráficas de dependencia para el atributo *dtype*, y numeramos los nodos en el orden en el cual se calcula *dtype* de acuerdo con el pseudocódigo anterior.

Figura 6.10
Árbol de análisis gramatical que muestra orden de recorrido para el ejemplo 6.12

Para proporcionar una forma completamente concreta a este ejemplo, convertimos el pseudocódigo anterior a código C real. También, en vez de utilizar un árbol de análisis gramatical explícito, suponemos que se ha construido un árbol sintáctico, en el que *var-list* se representa mediante una lista de hermanos de nodos `id`. Entonces una cadena de declaración tal como `float x, y` tendría el árbol sintáctico (compare esto con la figura 6.10)

y el orden de evaluación de los hijos del nodo *decl* sería de izquierda a derecha (primero el nodo *type*, luego el nodo **x** y finalmente el nodo **y**). Observe que en este árbol ya incluimos el *dtype* del nodo *type*; suponemos que esto se ha calculado previamente durante el análisis sintáctico.

La estructura del árbol sintáctico la proporcionan las siguientes declaraciones en C:

```
typedef enum {decl,type,id} nodekind;
typedef enum {integer,real} typekind;
typedef struct treeNode
  { nodekind kind;
    struct treeNode
        * lchild, * rchild, * sibling;
    typekind dtype;
    /* para nodos id y type */
    char * name;
    /* sólo para nodos id */
  } * SyntaxTree;
```

El procedimiento *EvalType* tiene ahora el código correspondiente en C:

```
void evalType (SyntaxTree t)
{ switch (t->kind)
  { case decl:
      t->rchild->dtype = t->lchild->dtype;
      evalType(t->rchild);
      break;
    case id:
      if (t->sibling != NULL)
      { t->sibling->dtype = t->dtype;
        evalType(t->sibling);
      }
      break;
  } /* end switch */
} /* end evalType */
```

Este código se puede simplificar al siguiente procedimiento no recursivo, que funciona enteramente al nivel del nodo raíz (`decl`):

```
void evalType (SyntaxTree t)
{ if (t->kind == decl)
  { SyntaxTree p = t->rchild;
    p->dtype = t->lchild->dtype;
    while (p->sibling != NULL)
    { p->sibling->dtype = p->dtype;
      p = p->sibling;
    }
  } /* end if */
} /* end evalType */
```

§

Ejemplo 6.13 Considere la gramática del ejemplo 6.4 (página 266), que tiene el atributo heredado *base* (las gráficas de dependencia están dadas en el ejemplo 6.8, página 273). Repetiremos la gramática de ese ejemplo aquí:

num-base → *num carbase*
carbase → o | d
num → *num dígito* | *dígito*
dígito → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Esta gramática tiene dos nuevas características. En primer lugar, existen dos atributos, el atributo sintetizado *val* y el atributo heredado *base*, del cual depende *val*. En segundo lugar, el atributo *base* es heredado del hijo derecho al hijo izquierdo de un *num-base* (es decir, desde *carbase* hasta *num*). Así, en este caso debemos evaluar los hijos de un *num-base* de derecha a izquierda en lugar de izquierda a derecha. Procederemos a dar el pseudocódigo para un procedimiento *EvalWithBase* que calcula tanto *base* como *val*. En este caso *base* se calcula en preorden y *val* en postorden durante un paso simple (en breve comentaremos la cuestión de pasos y atributos múltiples). El pseudocódigo se presenta a continuación (véase la gramática con atributos en la tabla 6.4, página 267).

```
procedure EvalWithBase ( T: treenode );
begin
  case clasenodo de T of
  num-base:
      EvalWithBase ( hijo derecho de T );
      asignar base del hijo derecho de T a la base del hijo izquierdo;
      EvalWithBase ( hijo izquierdo de T );
      asignar valor de hijo izquierdo de T a T.val;
  num:
      asignar T.base a la base del hijo izquierdo de T;
      EvalWithBase ( hijo derecho de T );
      if hijo derecho de T no es nil then
         asignar T.base a la base del hijo derecho de T;
         EvalWithBase ( hijo derecho de T );
         if valores de hijos izquierdo y derecho ≠ error then
              T.val := T.base*(valor de hijo izquierdo) + valor de hijo derecho
         else T.val := error;
      else T.val := valor de hijo izquierdo;
```

```
    carbase:
        if hijo de T = o then T.base := 8
        else T.base := 10;
    dígito:
        if T.base = 8 and hijo de T = 8 or 9 then T.val := error
        else T.val := numval ( hijo de T );
    end case;
end EvalWithBase;
```

Dejamos para los ejercicios la construcción de las declaraciones en C en el caso de un árbol sintáctico apropiado y una traducción del pseudocódigo para *EvalWithBase* a código en C. §

En las gramáticas con atributos con combinaciones de atributos sintetizados y heredados, si los atributos sintetizados dependen de los atributos heredados (además de otros atributos sintetizados), pero los atributos heredados no dependen de ningún atributo sintetizado, entonces es posible calcular todos los atributos en un solo paso sobre el árbol de análisis gramatical o el árbol sintáctico. El ejemplo anterior es un buen ejemplo de cómo se hace esto, y el orden de evaluación puede resumirse combinando los procedimientos en pseudocódigo *PostEval* y *PreEval*:

```
procedure CombinedEval ( T: treenode );
begin
  for cada hijo C de T do
     calcula todos los atributos heredados de C;
     CombinedEval ( C );
  calcula todos los atributos sintetizados de T;
end;
```

Las situaciones en las cuales los atributos heredados dependen de atributos sintetizados son más complejas y requieren de más de un paso sobre los árboles de análisis gramatical o sintáctico, como lo muestra el siguiente ejemplo.

Ejemplo 6.14

Considere la siguiente versión simple de una gramática de expresión:

$$exp \rightarrow exp \ \texttt{/} \ exp \mid \texttt{num} \mid \texttt{num.num}$$

Esta gramática tiene una sola operación, de división, indicada por el token `/`. También tiene dos versiones de números: números enteros compuestos de secuencias de dígitos, las cuales se indican mediante el token `num` y números de punto flotante, que se indican por medio del token `num.num`. La idea de esta gramática es que las operaciones se pueden interpretar de manera diferente, dependiendo de si son operaciones de punto flotante o estrictamente enteras. La división, en particular, es bastante diferente, dependiendo de si se permiten las fracciones. Si no, la división con frecuencia se llama operación **div**, y el valor de `5/4` es 5 **div** 4 = 1. Si se refiere a la división de punto flotante, entonces `5/4` tiene un valor de 1.2.

Supongamos ahora que un lenguaje de programación requiere expresiones mezcladas para ser promovido completamente a expresiones de punto flotante, y las operaciones apropiadas para utilizarse en su semántica. De este modo, el significado de la expresión

`5/2/2.0` (suponiendo la asociatividad por la izquierda de la división) es 1.25, mientras que el significado de `5/2/2` es 1.[7] Para describir estas semánticas se requieren tres atributos: un atributo booleano sintetizado *isFloat*, que indica si alguna parte de una expresión tiene un valor de punto flotante; un atributo heredado *etype*, con dos valores *int* y *float*, que da el tipo de cada subexpresión y que depende de *isFloat*; y finalmente, el *val* calculado de cada subexpresión, que depende del *etype* heredado. Esta situación también requiere que se identifique la expresión de nivel superior (de manera que sepamos que no hay más subexpresiones por considerar). Hacemos esto aumentando la gramática con un símbolo inicial:

$$S \rightarrow exp$$

Las ecuaciones de atributo se dan en la tabla 6.7. En las ecuaciones para la regla gramatical $exp \rightarrow$ **num** utilizamos *Float* (**num**.*val*) para indicar una función que convierte el valor entero **num**.*val* en un valor de punto flotante. También empleamos la / para la división de punto flotante y **div** para la división entera.

Los atributos *isFloat*, *etype* y *val* en este ejemplo se pueden calcular mediante dos pasos sobre el árbol de análisis gramatical o el árbol sintáctico. El primer paso calcula el atributo sintetizado *isFloat* mediante un recorrido postorden. El segundo paso calcula tanto el atributo heredado *etype* como el atributo sintetizado *val* en un recorrido combinado preorden/postorden. Dejamos la descripción de estos pasos y los correspondientes cálculos de atributo para la expresión 5/2/2.0 para los ejercicios, además de la construcción del pseudocódigo o código en C para efectuar los pasos sucesivos en el árbol sintáctico.

Tabla 6.7
Gramática con atributos para el ejemplo 6.14

Regla gramatical	Reglas semánticas
$S \rightarrow exp$	$exp.etype =$ **if** $exp.isFloat$ **then** *float* **else** *int* $S.val = exp.val$
$exp_1 \rightarrow exp_2 \ / \ exp_3$	$exp_1.isFloat =$ $exp_2.isFloat$ **or** $exp_3.isFloat$ $exp_2.etype = exp_1.etype$ $exp_3.etype = exp_1.etype$ $exp_1.val =$ **if** $exp_1.etype = int$ **then** $exp_2.val$ **div** $exp_3.val$ **else** $exp_2.val \ / \ exp_3.val$
$exp \rightarrow$ **num**	$exp.isFloat =$ **false** $exp.val =$ **if** $exp.etype = int$ **then** **num**.*val* **else** *Float*(**num**.*val*)
$exp \rightarrow$ **num.num**	$exp.isFloat =$ **true** $exp.val =$ **num.num**.*val*

§

7. Esta regla no es la misma regla que la empleada en C. Por ejemplo, el valor de `5/2/2.0` es 1.0 en C, no 1.25.

6.2.3 Atributos como parámetros y valores devueltos

Con frecuencia al calcular atributos tiene sentido utilizar parámetros y valores de función devueltos para comunicar los valores de atributo en vez de almacenarlos como campos en una estructura de registro de árbol sintáctico. Esto es particularmente cierto si muchos de los valores de atributo son los mismos o se utilizan sólo de manera temporal para calcular otros valores de atributo. En este caso no tiene mucho sentido emplear el espacio en el árbol sintáctico para almacenar valores de atributo en cada nodo. Un procedimiento de recorrido recursivo simple que calcule atributos heredados en preorden y atributos sintetizados en postorden puede, de hecho, pasar los valores de atributos heredados como parámetros a llamadas recursivas de hijos y recibir valores de atributos sintetizados como valores devueltos de esas mismas llamadas. En capítulos anteriores ya se presentaron varios ejemplos de esto. En particular, el cálculo del valor sintetizado de una expresión aritmética se puede hacer mediante un procedimiento de análisis sintáctico recursivo que devuelve el valor de la expresión actual. De manera similar, el árbol sintáctico como un atributo sintetizado debe él mismo ser calculado mediante un valor devuelto durante el análisis sintáctico, puesto que hasta que se haya construido, no existe todavía estructura de datos en la cual pueda registrarse a sí mismo como un atributo.

En situaciones más complejas, como cuando debe ser devuelto más de un atributo sintetizado, puede ser necesario emplear una estructura de registro o unión como un valor devuelto, o puede dividirse el procedimiento recursivo en varios procedimientos para cubrir los diferentes casos. Ilustramos esto con un ejemplo.

Ejemplo 6.15

Considere el procedimiento recursivo *EvalWithBase* del ejemplo 6.13. En este procedimiento el atributo *base* de un número se calcula sólo una vez y entonces se utiliza para todos los cálculos subsiguientes del atributo *val*. De manera semejante, el atributo *val* de una parte del número se emplea sólo como algo temporal en el cálculo del valor del número completo. Tiene sentido convertir a *base* en un parámetro (como un atributo heredado) y a *val* en un valor devuelto. El procedimiento modificado *EvalWithBase* queda como se presenta a continuación:

```
function EvalWithBase ( T: treenode; base: integer ): integer;
var temp, temp2: integer;
begin
  case clasenodo de T of
  num-base:
      temp := EvalWithBase ( hijo derecho de T );
      return EvalWithBase ( hijo izquierdo de T, temp );
  num:
      temp := EvalWithBase ( hijo izquierdo de T, base );
      if hijo derecho de T no es nil then
        temp2 := EvalWithBase ( hijo derecho de T, base );
        if temp ≠ error and temp2 ≠ error then
            return base*temp + temp2
        else return error;
      else return temp;
  carbase:
      if hijo de T = o then return 8
      else return 10;
  dígito:
      if base = 8 and hijo de T = 8 or 9 then return error
      else return numval ( hijo de T );
  end case;
end EvalWithBase;
```

Naturalmente, esto funciona sólo porque el atributo *base* y el atributo *val* tienen el mismo tipo de datos *integer* o "entero", ya que en un caso *EvalWithBase* devuelve el atributo *base* (cuando el nodo del árbol de análisis gramatical es un nodo *carbase*) y en los otros casos *EvalWithBase* devuelve el atributo *val*. También es algo irregular que la primera llamada a *EvalWithBase* (en el nodo del árbol de análisis gramatical *num-base* raíz) deba tener un valor *base* facilitado, aun cuando todavía no exista, y lo cual es ignorado posteriormente. Por ejemplo, para iniciar el cálculo se tendría que hacer una llamada como

EvalWithBase(*nodorraíz*,0);

con un valor de base de prueba 0. Por lo tanto, sería más racional distinguir tres casos: el caso *num-base*, el caso *carbase* y el caso *num* y *dígito*, así como escribir tres procedimientos separados para estos tres casos. El pseudocódigo aparecería entonces como se muestra enseguida:

```
function EvalBasedNum ( T: treenode ): integer;
(* sólo llamado en nodo raíz *)
begin
   return EvalNum ( hijo izquierdo de T, EvalBase(hijo derecho de T) );
end EvalBasedNum;

function EvalBase ( T: treenode ): integer;
(* sólo llamado en nodo carbase *)
begin
   if hijo de T = o then return 8
   else return 10;
end EvalBase;

function EvalNum ( T: treenode; base: integer ): integer;
var temp, temp2: integer;
begin
   case clasenodo de T of
   num:
         temp := EvalWithBase ( hijo izquierdo de T, base );
         if hijo derecho de T no es nil (nulo) then
            temp2 := EvalWithBase ( hijo derecho de T, base );
            if temp ≠ error and temp2 ≠ error then
                  return base*temp + temp2
            else return error;
         else return temp;
   dígito:
         if base = 8 and hijo de T = 8 or 9 then return error
         else return numval ( hijo de T );
   end case;
end EvalNum;
```

§

6.2.4 El uso de estructuras de datos externas para almacenar valores de atributo

En aquellos casos en que los valores de atributo no se prestan fácilmente para el método de los parámetros y valores devueltos (lo que es cierto en particular cuando los valores de atributo tienen una estructura significativa y puede ser necesaria en puntos arbitrarios durante

la traducción), todavía puede no ser razonable almacenar valores de atributo en los nodos del árbol sintáctico. En tales casos las estructuras de datos tales como las tablas de búsqueda, gráficas y otras estructuras pueden ser útiles para obtener el comportamiento correcto y accesibilidad de los valores de atributo. La misma gramática con atributos se puede modificar para reflejar esta necesidad reemplazando ecuaciones de atributo (que representan asignaciones de valores de atributo) mediante llamadas a procedimientos que representan operaciones en las estructuras de datos apropiadas que se emplean para mantener los valores de atributo. Las reglas semánticas resultantes ya no representan entonces una gramática con atributos, pero todavía son útiles en la descripción de la semántica de los atributos, mientras que la operación de los procedimientos se ve con claridad.

Ejemplo 6.16

Considere el ejemplo anterior con el procedimiento *EvalWithBase* empleando parámetros y valores devueltos. Como el atributo *base*, una vez que se establece, se mantiene fijo el tiempo que dure el cálculo del valor, podemos utilizar una variable no local para almacenar su valor, en vez de pasarlo como un parámetro en cada ocasión. (Si *base* no estuviera fijo, en un proceso recursivo así esto sería riesgoso o incluso incorrecto.) De este modo, podemos alterar el pseudocódigo para *EvalWithBase* de la manera siguiente:

```
function EvalWithBase ( T: treenode ): integer;
var temp, temp2: integer;
begin
  case clasenodo de T of
  num-base:
      SetBase ( hijo derecho de T );
      return EvalWithBase ( hijo izquierdo de T );
  num:
      temp := EvalWithBase ( hijo izquierdo de T );
      if hijo derecho de T no es nil (nulo) then
        temp2 := EvalWithBase ( hijo derecho de T );
        if temp ≠ error and temp2 ≠ error then
            return base*temp + temp2
        else return error;
      else return temp;
  dígito:
      if base = 8 and hijo de T = 8 or 9 then return error
      else return numval ( hijo de T );
  end case;
end EvalWithBase;

procedure SetBase ( T: treenode );
begin
  if hijo de T = o then base := 8
  else base := 10;
end SetBase ;
```

Aquí separamos el proceso de asignación a la variable *base* no local en el procedimiento *SetBase*, el cual sólo es llamado en un nodo *carbase*. El resto del código de *EvalWithBase* entonces se refiere simplemente a *base* de manera directa, sin pasarlo como un parámetro.

También podemos cambiar las reglas semánticas para reflejar el uso de la variable no local *base*. En este caso las reglas tendrían un aspecto algo parecido a lo siguiente, donde utilizamos la asignación para indicar de manera explícita el establecimiento de la variable no local *base*:

Regla gramatical	Reglas semánticas
num-base → *num carbase*	*num-base.val* = *num.val*
carbase → `o`	*base* := 8
carbase → `d`	*base* := 10
num_1 → num_2 *dígito*	num_1 *.val* = **if** *dígito.val* = *error* **or** num_2 *.val* = error **then** *error* **else** num_2 *.val* * *base* + *dígito.val*
etc.	etc.

Ahora *base* ya no es un atributo en el sentido en que lo hemos utilizado hasta ahora, y las reglas semánticas ya no forman una gramática con atributos. No obstante, si se sabe que *base* es una variable con las propiedades adecuadas, entonces estas reglas todavía definen de manera adecuada la semántica de un *num-base* para el escritor del compilador. §

Uno de los ejemplos principales de una estructura de datos externas al árbol sintáctico es la **tabla de símbolos**, que almacena atributos asociados con procedimientos, variables y constantes declaradas en un programa. Una tabla de símbolos es una estructura de datos de diccionario con operaciones tales como *inserción, búsqueda* y *eliminación.* En la siguiente sección analizaremos cuestiones en torno a la tabla de símbolos en lenguajes de programación típicos. Nos contentaremos en esta sección con el siguiente ejemplo simple.

Ejemplo 6.17

Considere la gramática con atributos de las declaraciones simples de la tabla 6.3 (página 266) y el procedimiento de evaluación de atributo para esta gramática con atributos dado en el ejemplo 6.12 (página 279). Por lo general, la información en las declaraciones se inserta en una tabla de símbolos utilizando los identificadores declarados como claves y almacenados allí para una búsqueda posterior durante la traducción de otras partes del programa. Por consiguiente, suponemos para esta gramática con atributos que existe una tabla de símbolos que almacenará el nombre del identificador junto con su tipo de datos declarado, y que se insertarán pares de nombre-tipo de datos en la tabla de símbolos mediante un procedimiento de inserción denominado *insert*, que se declara como se muestra a continuación:

procedure *insert* (*name*: *string*; *dtype*: *typekind*);

De este modo, en vez de almacenar el tipo de datos de cada variable en el árbol sintáctico, lo insertamos en la tabla de símbolos utilizando este procedimiento. También, puesto que cada declaración tiene sólo un tipo asociado, podemos utilizar una variable no local para almacenar el *dtype* constante de cada declaración durante el procesamiento. Las reglas semánticas resultantes son las siguientes:

Regla gramatical	Reglas semánticas
decl → *type var-list*	
type → `int`	*dtype* = *integer*
type → `float`	*dtype* = *real*
$var\text{-}list_1$ → `id` , $var\text{-}list_2$	*insert*(`id`*.name, dtype*)
var-list → `id`	*insert*(`id`*.name, dtype*)

En las llamadas a *insert* hemos utilizado `id`.*name* para hacer referencia a la cadena de identificador, que suponemos es calculada por el analizador léxico o analizador sintáctico. Estas reglas semánticas son muy diferentes de la gramática con atributos correspondiente; la regla gramatical para una *decl* no tiene, de hecho, ninguna regla semántica. Las dependencias no están expresadas tan claramente, aunque salta a la vista que las reglas de *type* deben procesarse antes que las reglas asociadas de *var-list*, debido a que las llamadas a *insert* dependen de *dtype*, lo que está establecido en las reglas de *type*.

El pseudocódigo correspondiente a un procedimiento de evaluación de atributo *EvalType* es como se presenta a continuación (compare esto con el código de la página 280):

```
procedure EvalType ( T: treenode );
begin
  case clasenodo de T of
  decl:
      EvalType ( tipo hijo de T );
      EvalType ( var-list hijo de T );
  type:
      if hijo de T = int then dtype := integer
      else dtype := real;
  var-list:
      insert(nombre del primer hijo de T, dtype)
      if tercer hijo de T no es nil then
         EvalType ( tercer hijo de T );
  end case;
end EvalType;
```

§

6.2.5 El cálculo de atributos durante el análisis sintáctico

Una cuestión que se presenta de manera natural es hasta qué punto los atributos se pueden calcular al mismo tiempo que la etapa de análisis sintáctico, sin esperar a realizar pasos adicionales sobre el código fuente por medio de recorridos recursivos del árbol sintáctico. Esto es particularmente importante respecto al árbol sintáctico mismo, el cual es un atributo sintetizado que debe ser construido durante el análisis sintáctico, si se va a emplear para análisis semántico adicional. Históricamente, la posibilidad de calcular todos los atributos durante el análisis sintáctico era incluso de mayor interés, ya que se había puesto mucho énfasis en la capacidad de un compilador para realizar la traducción en un paso. En la actualidad esto es menos importante, así que no proporcionaremos un análisis exhaustivo de todas las técnicas especiales que se han desarrollado. Sin embargo, vale la pena ofrecer una perspectiva básica de sus ideas y requerimientos.

Saber cuáles atributos se pueden calcular con éxito durante un análisis sintáctico depende en gran medida de la potencia y propiedades del método de análisis sintáctico empleado. Una restricción importante es determinada por el hecho de que todos los métodos de análisis sintáctico principales procesan el programa de entrada de izquierda a derecha (esto es lo que significa la primera L en las técnicas de análisis sintáctico LL y LR que se estudiaron en los dos capítulos anteriores). Esto es equivalente al requerimiento de que los atributos puedan evaluarse mediante un recorrido del árbol de análisis gramatical de izquierda a derecha. Para los atributos sintetizados esto no es una restricción, porque los hijos de un nodo pueden procesarse en orden arbitrario, en particular de izquierda a derecha. Pero, para los atributos heredados, esto significa que puede no haber dependencias "hacia atrás" en la gráfica de dependencia (las dependencias apuntan de derecha a izquierda en el árbol de análisis gramatical). Por ejemplo, la gramática con atributos del ejemplo 6.4 (página 266) viola

esta propiedad, porque la base de un *num-base* tiene su base dada por el sufijo o o d, y el atributo *val* no se puede calcular hasta que el sufijo al final de la cadena se ve y se procesa. Las gramáticas con atributos que satisfacen esta propiedad se llaman de atributo L (por la **L** del término izquierda a derecha en inglés) y se define de la manera siguiente.

Definición

Una gramática con atributos para los atributos $a_1, \ldots, a_k$ es de **atributo L** si, para cada atributo heredado a_j, y para cada regla gramatical

$$X_0 \rightarrow X_1 X_2 \ldots X_n$$

las ecuaciones asociadas para a_j son todas de la forma

$$X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k, X_1.a_1, \ldots, X_1.a_k, \ldots, X_{i-1}.a_1, \ldots, X_{i-1}.a_k)$$

Es decir, el valor de a_j para X_i sólo puede depender de los atributos de los símbolos $X_0, \ldots, X_{i-1}$ que se presentan a la izquierda de X_i en la regla gramatical.

Como un caso especial, ya advertimos que una gramática con atributos S es de atributo L.

Dada una gramática con atributos L en la que los atributos heredados no dependan de los sintetizados, un analizador sintáctico descendente recursivo puede evaluar todos los atributos convirtiendo los heredados en parámetros y los sintetizados en valores devueltos de la manera antes descrita. Por desgracia, los analizadores LR, como un analizador sintáctico LALR(1) generado por Yacc, son adecuados para controlar principalmente atributos sintetizados. La razón reside, irónicamente, en la mayor potencia de los analizadores LR respecto a los analizadores LL. Los atributos sólo se pueden calcular cuando la regla gramatical a utilizarse en una derivación se vuelve conocida, porque sólo entonces las ecuaciones son determinadas por el cálculo de atributo. Sin embargo, los analizadores LR posponen la decisión de qué regla gramatical utilizar en una derivación hasta que el lado derecho de una regla gramatical está completamente formado. Esto hace difícil que los atributos heredados estén disponibles, a menos que sus propiedades permanezcan fijas para todas las posibles selecciones del lado derecho. Expondremos brevemente el uso de la pila de análisis sintáctico para calcular los atributos en los casos más comunes, con aplicaciones para Yacc. Las fuentes para técnicas más complejas se mencionan en la sección de notas y referencias.

Cálculo de atributos sintetizados durante el análisis sintáctico LR Éste es el caso más sencillo para un analizador LR. Un analizador LR generalmente tendrá una **pila de valores** en la que se almacenan los atributos sintetizados (quizá como uniones o estructuras, si hay más de un atributo para cada símbolo gramatical). La pila de valores será manipulada en paralelo con la pila de análisis sintáctico, con los nuevos valores que se están calculando de acuerdo con las ecuaciones de atributo cada vez que se presenta un desplazamiento o una reducción en la pila de análisis sintáctico. Ilustramos esto en la tabla 6.8 para la gramática con atributos de la tabla 6.5 (página 269), que es la versión ambigua de la gramática de expresión aritmética simple. Por simplicidad utilizamos notación abreviada para la gramática e ignoramos algunos de los detalles de un algoritmo de análisis sintáctico LR en la tabla. En particular, no

se indican los números de estado, no se muestra el símbolo inicial aumentado y no se expresan las reglas de eliminación de ambigüedad implícitas. La tabla contiene dos nuevas columnas además de las acciones habituales de análisis sintáctico: la pila de valores y acciones semánticas. Las acciones semánticas indican cómo ocurren los cálculos en la pila de valores a medida que se presentan las reducciones en la pila de análisis sintáctico. (Los desplazamientos se ven como la inserción de valores de token tanto en la pila de análisis sintáctico como en la pila de valores, aunque esto puede diferir en analizadores sintácticos individuales.)

Como ejemplo de una acción semántica considere el paso 10 de la tabla 6.8. La pila de valores contiene los valores enteros 12 y 5, separados por el token +, con el 5 en la parte superior de la pila. La acción del análisis sintáctico es reducir mediante $E \rightarrow E + E$, y la acción semántica correspondiente de la tabla 6.5 es calcular de acuerdo con la ecuación $E_1.val = E_2.val + E_3.val$. Las acciones correspondientes en la pila de valores que realiza el analizador sintáctico son las que se muestran a continuación (en pseudocódigo):

pop t3	{ obtiene $E_3.val$ de la pila de valores }
pop	{ descarta el token + }
pop t2	{ obtiene $E_2.val$ de la pila de valores }
t1 = t2 + t3	{ suma }
push t1	{ inserta el resultado de nuevo en la pila de valores }

En Yacc la situación representada por la reducción en el paso 10 se escribiría como la regla

```
E : E + E { $$ = $1 + $3; }
```

Aquí las pseudovariables `$i` representan posiciones en el lado derecho de la regla que se está reduciendo y se convierten a posiciones en la pila de valores contando hacia atrás a partir de la derecha. De este modo, `$3`, que corresponde al `E` más a la derecha, se encontrará en la parte superior o tope de la pila, mientras que `$1` se hallará dos posiciones por debajo del tope.

Tabla 6.8
Acciones semánticas y de análisis sintáctico para la expresión `3*4+5` durante un análisis sintáctico LR

	Pila de análisis sintáctico	Entrada	Acción de análisis sintáctico	Pila de valores	Acción semántica
1	$	3*4+5 $	desplazamiento	$	
2	$ **n**	*4+5 $	reducción de $E \rightarrow$ **n**	$ **n**	$E.val =$ **n**$.val$
3	$ E	*4+5 $	desplazamiento	$ 3	
4	$ E *	4+5 $	desplazamiento	$ 3 *	
5	$ E * **n**	+5 $	reducción de $E \rightarrow$ **n**	$ 3 * **n**	$E.val =$ **n**$.val$
6	$ E * E	+5 $	reducción de $E \rightarrow E * E$	$ 3 * 4	$E_1.val = E_2.val * E_3.val$
7	$ E	+5 $	desplazamiento	$ 12	
8	$ E +	5 $	desplazamiento	$ 12 +	
9	$ E + **n**	$	reducción de $E \rightarrow$ **n**	$ 12 + **n**	$E.val =$ **n**$.val$
10	$ E + E	$	reducción de $E \rightarrow E + E$	$ 12 + 5	$E_1.val = E_2.val + E_3.val$
11	$ E	$		$ 17	

Herencia de un atributo sintetizado previamente calculado durante el análisis sintáctico LR Debido a la estrategia de evaluación de izquierda a derecha del análisis sintáctico LR, una acción asociada a un no terminal en el lado derecho de una regla puede utilizar atributos sintetizados de los símbolos a la izquierda del mismo en la regla, puesto que estos valores ya se insertaron en la pila de valores. Para ilustrar esto brevemente considere la opción de producción $A \rightarrow B\ C$, y suponga que C tiene un atributo heredado i que depende de alguna manera del atributo sintetizado s de B: $C.i = f(B.s)$. El valor de $C.i$ puede almacenarse en una variable antes del reconocimiento de C introduciendo una producción ε entre B y C que programe el almacenamiento de la parte superior de la pila de valores:

Regla gramatical	Reglas semánticas
$A \rightarrow B\ D\ C$	
$B \rightarrow \ldots$	{ calcule $B.s$ }
$D \rightarrow \varepsilon$	$saved_i = f(valstack[top])$
$C \rightarrow \ldots$	{ ahora está disponible $saved_i$ }

En Yacc este proceso se vuelve aún más fácil, ya que la producción ε no necesita ser introducida explícitamente. En su lugar, la acción de almacenamiento del atributo calculado se escribe simplemente en el lugar de la regla donde se va a registrar:

```
A : B { saved_i = f($1); } C ;
```

(Aquí la pseudovariable `$1` se refiere al valor de `B`, que está en la parte superior de la pila cuando se ejecuta la acción.) Tales acciones incrustadas en Yacc se estudiaron en la sección 5.5.6 del capítulo anterior.

Una alternativa para esta estrategia está disponible cuando siempre puede predecirse la posición de un atributo sintetizado previamente calculado en la pila de valores. En este caso, el valor no necesita ser copiado en una variable, sino que se puede acceder a él directamente en la pila de valores. Considere, por ejemplo, la siguiente gramática con atributos L con un atributo *dtype* heredado:

Regla gramatical	Reglas semánticas
decl → *type var-list*	*var-list.dtype* = *type.dtype*
type → `int`	*type.dtype* = *integer*
type → `float`	*type.dtype* = *real*
$var\text{-}list_1 \rightarrow var\text{-}list_2$ `, id`	*insert*(`id`.*name*, $var\text{-}list_1.dtype$)
	$var\text{-}list_2.dtype = var\text{-}list_1.dtype$
var-list → `id`	*insert*(`id`.*name*, *var-list.dtype*)

En este caso el atributo *dtype*, que es un atributo sintetizado para el no terminal *type*, se puede calcular en la pila de valores justo antes de que se reconozca la primera *var-list*. Entonces, cuando se reduce cada regla para *var-list*, *dtype* se puede encontrar en una posición fija en la pila de valores contando hacia atrás desde la parte superior o tope: cuando se reduce *var-list* → `id`, *dtype* está justo debajo de la parte superior de la pila, mientras que cuando se reduce $var\text{-}list_1 \rightarrow var\text{-}list_2$ `, id`, *dtype* está tres posiciones debajo del tope de la pila. Podemos implementar esto en un analizador sintáctico LR mediante la eliminación de las dos

reglas de copia para *dtype* en la gramática con atributos anterior y entrando a la pila de valores de manera directa:

Regla gramatical	Reglas semánticas
decl → *type var-list*	
type → `int`	*type.dtype* = *integer*
type → `float`	*type.dtype* = *real*
$var\text{-}list_1 \rightarrow var\text{-}list_2$ `, id`	*insert*(`id`.*name*, *valstack*[*top*−3])
var-list → `id`	*insert*(`id`.*name*, *valstack*[*top*−1])

(Advierta que, como *var-list* no tiene atributo sintetizado *dtype*, el analizador sintáctico debe insertar un valor de prueba en la pila para mantener la ubicación apropiada en ésta.)

Existen varios problemas con este método. En primer lugar, requiere que el programador tenga acceso de manera directa a la pila de valores durante un análisis sintáctico, y esto puede ser peligroso en analizadores sintácticos generados de manera automática. Por ejemplo, Yacc no tiene una convención de pseudovariable para tener acceso a la pila de valores bajo la regla actual que se está reconociendo, como se requeriría con el método anterior. De este modo, para implementar un esquema de esta clase en Yacc, se tendría que escribir código especial para hacerlo. El segundo problema es que esta técnica sólo funciona si la posición del atributo previamente calculado es predecible a partir de la gramática. Por ejemplo, si hubiéramos escrito la gramática de la declaración anterior de manera que *var-list* fuera recursiva por la derecha (como hicimos en el ejemplo 6.17), entonces un número arbitrario de `id` podría estar en la pila, y no se conocería la posición de *dtype* en la pila.

Con mucho, la mejor técnica para tratar con atributos heredados en el análisis sintáctico LR es utilizar estructuras de datos externas, tal como una tabla de símbolos o variables no locales, con el fin de mantener los valores de atributos heredados, y de agregar producciones ε (o acciones incrustadas como en Yacc) para permitir que se presenten cambios a estas estructuras de datos en los momentos apropiados. Por ejemplo, una solución para el problema de *dtype* que se acaba de comentar puede encontrarse en el análisis de las acciones incrustadas en Yacc (sección 5.5.6).

Sin embargo, deberíamos estar conscientes de que incluso este último método no carece de riesgos: la adición de producciones ε a una gramática puede agregar conflictos de análisis sintáctico, de manera que una gramática LALR(1) se puede convertir en una gramática no LR(k) para cualquier k (véase el ejercicio 6.15 y la sección de notas y referencias). En situaciones prácticas, no obstante, esto rara vez sucede.

6.2.6 La dependencia del cálculo de atributos respecto a la sintaxis

Como el tema final en esta sección, vale la pena advertir que las propiedades de los atributos dependen en gran parte de la estructura de la gramática. Puede pasar que las modificaciones a la gramática que no cambian las cadenas legales del lenguaje provoquen que el cálculo de los atributos se vuelva más simple o más complejo. En realidad, tenemos el siguiente:

Teorema

(De Knuth [1968]). Dada una gramática con atributos, todos los atributos heredados se pueden cambiar en atributos sintetizados mediante la modificación adecuada de la gramática, sin cambiar el lenguaje de la misma.

Daremos un ejemplo de cómo un atributo heredado se puede convertir en un atributo sintetizado mediante modificación de la gramática.

Ejemplo 6.18

Vuelva a considerar la gramática de declaraciones simples de los ejemplos anteriores:

decl → *type var-list*
type → **int** | **float**
var-list → **id** , *var-list* | **id**

El atributo *dtype* de la gramática con atributos de la tabla 6.3 es heredado. Sin embargo, si volvemos a escribir la gramática de la manera siguiente,

decl → *var-list* **id**
var-list → *var-list* **id** , | *type*
type → **int** | **float**

entonces se aceptan las mismas cadenas, pero el atributo *dtype* ahora se vuelve sintetizado, de acuerdo con la siguiente gramática con atributos:

Regla gramatical	Reglas semánticas
decl → *var-list* **id**	**id**.*dtype* = *var-list.dtype*
$var\text{-}list_1 \rightarrow var\text{-}list_2$ **id** ,	**id**.*dtype* = $var\text{-}list_2$.*dtype*
	$var\text{-}list_1$.*dtype* = $var\text{-}list_2$.*dtype*
var-list → *type*	*var-list.dtype* = *type.dtype*
type → **int**	*type.dtype* = *integer*
type → **float**	*type.dtype* = *real*

Ilustraremos cómo este cambio en la gramática afecta al árbol de análisis gramatical y el cálculo del atributo *dtype* en la figura 6.11, que exhibe el árbol de análisis gramatical para la cadena **float x,y**, junto con las dependencias y valores de atributo. Las dependencias de los dos valores **id**.*dtype* respecto a los valores del padre o hermano se trazan como líneas discontinuas en la figura. Aunque estas dependencias parecen ser violaciones a la afirmación de que no hay atributos heredados en esta gramática con atributos, de hecho estas dependencias son siempre hacia hojas en el árbol de análisis sintáctico (es decir, no recursivo) y pueden conseguirse mediante operaciones en los nodos padre apropiados. De este modo, estas dependencias no se ven como herencias. §

Figura 6.11
Árbol de análisis gramatical para la cadena **float x,y** que muestra el atributo *dtype* como se especificó mediante la gramática con atributos del ejemplo 6.18

De hecho, el teorema establecido es menos útil de lo que puede parecer. Modificar la gramática con el fin de cambiar los atributos heredados a atributos sintetizados con frecuencia hace a la gramática y las reglas semánticas mucho más complejas y difíciles de comprender. Por lo tanto, no es una manera recomendable para tratar con los problemas de cálculo de atributos heredados. Por otra parte, si un cálculo de atributo parece anormalmente difícil, puede ser porque la gramática está definida de manera inadecuada para su cálculo, y un cambio en la gramática puede valer la pena.

6.3 LA TABLA DE SÍMBOLOS

La tabla de símbolos es el principal atributo heredado en un compilador y, después del árbol sintáctico, también forma la principal estructura de datos. Aunque hemos demorado, con unas cuantas excepciones, el análisis de la tabla de símbolos hasta este punto, donde corresponde mejor al marco conceptual de la fase del análisis sintáctico, el lector debería estar consciente de que en los compiladores prácticos la tabla de símbolos con frecuencia está íntimamente involucrada con el analizador sintáctico e incluso el analizador léxico, cualquiera de los cuales puede necesitar introducir información de manera directa en la tabla de símbolos o consultarla para resolver ambigüedades (para un ejemplo de esta clase en C véase el ejercicio 6.22). No obstante, en un lenguaje diseñado muy cuidadosamente, como Ada o Pascal, es posible e incluso razonable posponer las operaciones de la tabla de símbolos hasta después de realizar un análisis sintáctico completo, cuando se sepa que el programa que se está traduciendo es sintácticamente correcto. Hicimos esto, por ejemplo, en el compilador TINY, cuya tabla de símbolos se estudiará más adelante en este capítulo.

Las principales operaciones de la tabla de símbolos son la *inserción, búsqueda* y *eliminación*; pero también pueden ser necesarias otras operaciones. La operación de *inserción* se utiliza para almacenar la información proporcionada por las declaraciones de nombre cuando se procesan estas declaraciones. La operación de *búsqueda* es necesaria para recuperar la información asociada con un nombre cuando éste se utiliza en el código asociado. La operación de *eliminación* es necesaria para eliminar la información proporcionada por una declaración cuando ésta ya no se aplica.[8] Las propiedades de estas operaciones son dictadas por las reglas del lenguaje de programación que se está traduciendo. En particular, la información que se necesita almacenar en la tabla de símbolos está en función de la estructura y propósito de las declaraciones. Esto por lo regular incluye información de tipo de datos, información sobre la región de aplicabilidad (ámbito, que se analiza más adelante) e información acerca de la ubicación posible en la memoria.

En esta sección comentaremos, en primer lugar, la organización de la estructura de datos de la tabla de símbolos para obtener velocidad y facilidad de acceso. Posteriormente describiremos algunos requerimientos típicos del lenguaje y el efecto que tendrán sobre la operación de la tabla de símbolos. Por último proporcionaremos un ejemplo extendido del uso de una tabla de símbolos con una gramática con atributos.

6.3.1 La estructura de la tabla de símbolos

La tabla de símbolos en un compilador es una típica estructura de datos de diccionario. La eficiencia de las tres operaciones básicas de *inserción, búsqueda* y *eliminación* varía de acuerdo con la organización de la estructura de datos. El análisis de esta eficiencia para organizaciones diferentes, así como la investigación de buenas estrategias de organización, es uno de los temas principales de un curso de estructura de datos. Por lo tanto, en este texto

8. Antes que destruir esta información, es más probable que una operación de *eliminación* (*delete*) la retire de la vista, ya sea almacenándola en otra parte, o marcándola como inactiva.

no trataremos este tema de manera detallada, sino que remitiremos al lector que desee más información a las fuentes que se mencionan en la sección de notas y referencias al final de este capítulo. Sin embargo, proporcionaremos aquí una perspectiva general de las estructuras de datos más útiles para estas tablas en la construcción de compiladores.

Las implementaciones típicas de estructuras de diccionario incluyen listas lineales, diversas estructuras de árbol de búsqueda (árboles de búsqueda binarios, árboles AVL y árboles B), así como tablas de dispersión (búsqueda de dirección). Las listas lineales son una buena estructura de datos básica que puede proporcionar implementaciones directas y fáciles de las tres operaciones básicas, con una operación de *inserción* en tiempo constante (insertando siempre al frente o la parte posterior) y operaciones de *búsqueda* y *eliminación* que son de tiempo lineal en las dimensiones de la lista. Esto puede ser suficientemente bueno para una implementación de compilador en la que el interés principal no sea la velocidad de compilación, tal como un compilador prototipo o experimental, o un intérprete para programas muy pequeños. Las estructuras de árbol de búsqueda son un poco menos útiles para la tabla de símbolos, en parte porque no proporcionan la mejor eficiencia del caso, pero también debido a la complejidad de la operación de *eliminación*. La tabla de dispersión proporciona a menudo la mejor selección para implementar la tabla de símbolos, ya que las tres operaciones se pueden realizar en un tiempo casi constante, y se utiliza muy frecuentemente en la práctica. Por lo tanto, comentaremos el caso de la tabla de dispersión con algo más de detalle.

Una tabla de dispersión es un arreglo de entradas, denominadas **cubetas**, indizadas mediante un intervalo entero, generalmente desde el 0 hasta el tamaño de la tabla menos uno. Una **función de dispersión** convierte la clave de búsqueda (en este caso, el nombre del identificador, compuesto de una cadena de caracteres) en un valor entero de dispersión en el intervalo del índice, y el elemento correspondiente a la clave de búsqueda se almacena en la cubeta en este índice. Se debe tener cuidado para garantizar que la función de dispersión distribuya los índices clave tan uniformemente como sea posible sobre todo el intervalo de índices, porque las **colisiones** de dispersión (donde se mapean dos claves hacia el mismo índice mediante la función de dispersión) provocan una degradación del rendimiento en las operaciones de *búsqueda* y *eliminación*. La función de dispersión también necesita funcionar en un tiempo constante, o por lo menos en un tiempo que sea lineal en las dimensiones de la clave (esto equivale a tiempo constante si existe una cota superior en las dimensiones de la clave). Examinaremos funciones de dispersión en breve.

Una cuestión importante es cómo una tabla de dispersión se ocupa de las colisiones (esto se conoce como **resolución de colisión**). Un método asigna sólo el espacio suficiente para un solo elemento en cada cubeta y resuelve las colisiones mediante la inserción de nuevos elementos en cubetas sucesivas (esto en ocasiones se denomina **direccionamiento abierto**). En este caso el contenido de la tabla de dispersión está limitado por el tamaño del arreglo utilizado para la tabla y, a medida que el arreglo se va llenando, las colisiones se vuelven más y más frecuentes; esto causa una degradación importante del rendimiento. Un problema adicional con este método, al menos para la construcción del compilador, es que es difícil implementar una operación de *eliminación*, y estas eliminaciones no mejoran el rendimiento posterior de la tabla.

Quizás el mejor esquema para la construcción de compiladores es la alternativa al direccionamiento abierto, denominada **encadenamiento por separado**. En este método cada cubeta es en realidad una lista lineal, y las colisiones se resuelven insertando el nuevo elemento en la lista de la cubeta. La figura 6.12 muestra un ejemplo simple de este esquema, con una tabla de dispersión de tamaño 5 (irrealmente pequeño, para propósitos de demostración). En esa tabla partimos del supuesto de que se han insertado cuatro identificadores (`i`, `j`, `size` y `temp`) y que `size` ("`tamaño`") y `j` tienen el mismo valor de dispersión (a saber, 1). En la ilustración mostramos `size` antes que `j` en el número 1 de la lista de la cubeta; el orden de los elementos en la lista depende del orden de inserción y de cómo se mantenga la lista. Un método común es insertar siempre al principio de la lista, de manera que usando este método `size` se habría insertado después de `j`.

Figura 6.12
Tabla de dispersión encadenada por separado que muestra la resolución de colisión

La figura 6.12 también muestra las listas en cada cubeta implementadas como listas ligadas (se utilizan puntos sólidos para indicar apuntadores nulos). Éstos se pueden asignar utilizando la asignación dinámica de apuntadores del lenguaje de implementación del compilador, o se pueden asignar a mano a partir de un arreglo de espacio dentro del compilador mismo.

Una cuestión que debe contestar el escritor de compiladores es qué tan grande hacer el arreglo inicial de cubetas. Por lo regular, este tamaño se fijará durante el tiempo de construcción del compilador.[9] Los tamaños típicos abarcan desde algunos cientos hasta más de un millar. Si se utiliza la asignación dinámica para las entradas reales, incluso arreglos pequeños permitirán la compilación de programas muy grandes, con el costo de un aumento en el tiempo de compilación. En cualquier caso el tamaño real del arreglo de cubetas debería elegirse como un número primo, ya que esto hace que las funciones de dispersión típicas se comporten mejor. Por ejemplo, si se desea un arreglo de cubetas de tamaño 200, se debería seleccionar 211 como el tamaño desde el 200 mismo (211 es el número primo más pequeño mayor que 200).

Volvamos ahora a una descripción de las funciones de dispersión comunes. Una función de dispersión, para su uso en una implementación de tabla de símbolos, convierte una cadena de caracteres (el nombre del identificador) en un entero en el intervalo 0..*size* – 1. Por lo común, esto se hace mediante un proceso de tres pasos. En primer lugar, cada carácter en la cadena de caracteres se convierte a un entero no negativo. Luego, estos enteros se combinan de alguna manera para formar un entero simple. Al final, el entero resultante se escala al intervalo 0..*size* – 1.

La conversión de cada carácter a un entero no negativo generalmente se hace utilizando el mecanismo de conversión integrado del lenguaje de implementación del compilador. Por ejemplo, la función `ord` de Pascal convierte un carácter en un entero, por lo general su valor ASCII. De manera similar, el lenguaje C convertirá automáticamente un carácter a un entero si se utiliza en una expresión aritmética o se asigna a una variable entera.

Escalar un entero no negativo para que caiga en el intervalo 0..*size* – 1 también se hace fácilmente utilizando la función **módulo** de matemáticas, la cual devuelve el residuo dividiendo *size* entre el número. Esa función se llama `mod` en Pascal y `%` en C. Cuando se utiliza este método, es importante que *size* sea un número primo. De otro modo, un conjunto de enteros distribuido al azar puede no tener sus valores escalados distribuidos aleatoriamente en el intervalo 0..*size* – 1.

Resta al implementador de la tabla de dispersión elegir un método para combinar los diferentes valores enteros de los caracteres en un solo entero no negativo. Un método simple es ignorar muchos de los caracteres y sumar sólo los valores de unos cuantos caracteres primeros, o el primero, el medio y el último. Esto es inadecuado para un compilador, porque los programadores tienden a asignar nombres de variable en grupos, tal como `temp1`, `temp2` o `m1tmp`, `m2tmp`, y así sucesivamente, así que este método ocasionará frecuentes

9. Existen métodos para incrementar el tamaño "size" del arreglo (y modificar la función de dispersión) al vuelo si la tabla de dispersión crece demasiado, pero son complejos y rara vez se utilizan.

colisiones entre tales nombres. Por consiguiente, el método elegido debería involucrar todos los caracteres en cada nombre. Otro método popular pero inadecuado es sumar simplemente los valores de todos los caracteres. Si se usa este método, todas las permutaciones de los mismos caracteres, tal como **tempx** y **xtemp**, provocarán colisiones.

Una buena solución a estos problemas es emplear repetidamente un número constante α como factor multiplicativo cuando se sume al valor del siguiente carácter. De modo que, si c_i es el valor numérico del i-ésimo carácter, y h_i es el valor de dispersión parcial calculado en el i-ésimo paso, entonces las h_i se calculan de acuerdo con las fórmulas recursivas $h_0 = 0$ y $h_{i+1} = \alpha h_i + c_i$, con el valor de dispersión final h calculado como $h = h_n$ **mod** *size*, donde n es el número de caracteres en el nombre donde se está aplicando la dispersión. Esto equivale a la fórmula

$$h = (\alpha^{n-1}c_1 + \alpha^{n-2}c_2 + \cdots + \alpha c_{n-1} + c_n) \textbf{ mod } size = \left(\sum_{i=1}^{n} \alpha^{n-i} c_i \right) \textbf{ mod } size$$

La selección de α en esa fórmula tiene, por supuesto, un efecto importante sobre la salida. Una selección razonable para α es una potencia de dos, tal como 16 o 128, de manera que la multiplicación se pueda realizar como un desplazamiento. Efectivamente, seleccionando $\alpha = 128$ se tiene el efecto de visualizar la cadena de caracteres como un número en base 128, suponiendo que todos los valores de carácter c_i sean menores que 128 (verdadero para los caracteres ASCII). Otras posibilidades que se mencionan en la literatura son varios números primos (véase la sección de notas y referencias).

En ocasiones el desbordamiento puede ser un problema en la fórmula para h, especialmente para valores grandes de α en máquinas con enteros de dos bytes. Si se utilizan valores enteros para los cálculos, el desbordamiento puede dar como resultado valores negativos (en representación de complemento a dos), lo que causará errores de ejecución. En ese caso es posible obtener el mismo efecto realizando la operación **mod** en el bucle de sumatoria. En la figura 6.13 se proporciona una muestra de código C para una función de dispersión h que efectúa esto utilizando la fórmula anterior y un valor de 16 para α (un desplazamiento de bit de 4, puesto que $16 = 2^4$).

Figura 6.13
Función de dispersión para una tabla de símbolos

```
#define SIZE ...
#define SHIFT 4

int hash ( char * key )
{ int temp = 0;
  int i = 0;
  while (key[i] != '\0')
  { temp = ((temp << SHIFT) + key[i]) % SIZE;
    ++i;
  }
  return temp;
}
```

6.3.2 Declaraciones

El comportamiento de una tabla de símbolos depende mucho de las propiedades de las declaraciones del lenguaje que se está traduciendo. Por ejemplo, la manera en que las operaciones de *inserción* y *eliminación* actúan sobre la tabla de símbolos, cuando se necesita

llamar a esas operaciones, y cuáles atributos se insertan en la tabla, varía ampliamente de lenguaje a lenguaje. Incluso el tiempo que toma el proceso de traducción/ejecución, cuando se puede construir la tabla de símbolos, y la extensión que necesita tener la tabla símbolos para existir, pueden diferir bastante de un lenguaje a otro. En esta sección indicaremos algunas de las cuestiones de lenguaje involucradas en declaraciones que afectan el comportamiento y la implementación de la tabla de símbolos.

Existen cuatro clases básicas de declaraciones que se presentan con frecuencia en los lenguajes de programación: declaraciones de constantes, declaraciones de tipo, declaraciones de variable y declaraciones de procedimiento/función.

Las **declaraciones de constantes** incluyen las declaraciones `const` de C, tales como

```
const int SIZE = 199;
```

(C también tiene un mecanismo `#define` para la creación de constantes, pero éste, más que por el propio compilador, es manejado mediante un preprocesador.)

Las **declaraciones de tipo** incluyen las declaraciones de tipo de Pascal, tales como

```
type Table = array [1..SIZE] of Entry;
```

y declaraciones de C `struct` y `union` tales como

```
struct Entry
{ char * name;
  int count;
  struct Entry * next;
};
```

las cuales definen un tipo de estructura con nombre `Entry`. C también tiene un mecanismo `typedef` para declaración de nombres que son alias para tipos

```
typedef struct Entry * EntryPtr;
```

Las **declaraciones de variables** son la forma más común de declaración e incluyen declaraciones en FORTRAN tales como

```
integer a,b(100)
```

y declaraciones de C como

```
int a,b[100];
```

Finalmente, existen **declaraciones de procedimiento/función**, tales como la función de C definida en la figura 6.13. Éstas en realidad son sólo una declaración constante de tipo procedimiento/función, pero por lo regular se destacan en definiciones del lenguaje debido a su naturaleza especial. Estas declaraciones son **explícitas**, ya que se utiliza una construcción del lenguaje especial para declaraciones. También es posible tener declaraciones **implícitas**, en las cuales las declaraciones se unen a instrucciones ejecutables sin mención explícita. FORTRAN y BASIC, por ejemplo, permiten que las variables se utilicen sin declaración explícita. En tales declaraciones implícitas se utilizan convenciones para proporcionar la información que de otro modo sería dada por una declaración explícita. FORTRAN, por ejemplo, tiene la convención de tipo de que las variables que comiencen con el intervalo de letras que va de la I a la N sean automáticamente enteros si se utilizan sin una declaración explícita, mientras que todas las otras son automáticamente reales. Las

declaraciones implícitas también se pueden denominar **declaraciones por el uso**, porque el primer uso de una variable que no esté declarada de manera explícita se puede considerar como si contuviera implícitamente su declaración.

Con frecuencia es más fácil utilizar una tabla de símbolos para mantener los nombres de todas las diferentes clases de declaraciones, particularmente cuando el lenguaje prohíbe usar el mismo nombre en distintos tipos de declaraciones. De cuando en cuando es más fácil emplear una tabla de símbolos diferente para cada clase de declaración, de manera que, por ejemplo, todas las declaraciones de tipo estén contenidas en una tabla de símbolos, mientras que todas las declaraciones de variable estén contenidas en otra. Con ciertos lenguajes, en particular los derivados de Algol, tales como C, Pascal y Ada, podemos desear asociar tablas de símbolos separadas con diferentes regiones de un programa (tales como los procedimientos) y vincularlas de acuerdo con las reglas semánticas del lenguaje (comentaremos esto con más detalle en breve).

Los atributos ligados a un nombre por medio de una declaración varían con la clase de declaración. Las declaraciones de constante asocian valores a nombres; por esta razón en ocasiones las declaraciones de constante se conocen como **fijaciones** o **especificaciones de valor**. Los valores que pueden ser fijados determinan cómo los trata el compilador. Por ejemplo, Pascal y Modula-2 requieren que los valores en una declaración constante sean estáticos y, por lo tanto, calculables por el compilador. El compilador puede entonces utilizar la tabla de símbolos para reemplazar los nombres de constante con sus valores durante la compilación. Otros lenguajes, tales como C y Ada, permiten que las constantes sean dinámicas; es decir, sólo calculables durante la ejecución. Tales constantes se deben tratar más bien como variables, en el sentido que se debe generar el código para calcular sus valores durante la ejecución. Sin embargo, tales constantes son de **asignación única**: una vez que sus valores se han determinado no pueden cambiar. Las declaraciones de constante también pueden ligar de manera implícita o explícita tipos de datos con nombres. En Pascal, por ejemplo, los tipos de datos de constantes se determinan implícitamente de sus valores (estáticos), mientras que en C los tipos de datos se proporcionan de manera explícita, como en las declaraciones de variable.

Las declaraciones de tipo vinculan nombres a tipos recién construidos y también pueden crear alias para tipos nombrados que ya existen. Los nombres de tipo por lo regular se utilizan en conjunto con un algoritmo de equivalencia de tipo para realizar la verificación de tipos de un programa de acuerdo con las reglas del lenguaje. Dedicamos una sección posterior en este capítulo a la verificación de tipos, así que por el momento no hablaremos más de las declaraciones de tipo.

Las declaraciones de variable vinculan más frecuentemente nombres a tipos de datos, como en el caso de C

```
Table symtab;
```

que vincula el tipo de datos representado por el nombre `Table` a la variable con el nombre `symtab`. Las declaraciones de variable también pueden fijar otros atributos implícitamente. Uno de tales atributos que tiene un efecto importante sobre la tabla de símbolos es el **ámbito** de una declaración o la región del programa donde la declaración se aplica (es decir, donde las variables definidas por la declaración son accesibles). El ámbito suele indicarse mediante la posición de la declaración dentro del programa, pero también puede ser afectado por notaciones sintácticas explícitas e interacciones con otras declaraciones. Puede, además, ser una propiedad de las declaraciones de constante, tipo y procedimiento. Abordaremos las reglas del ámbito con más detalle dentro de poco.

Un atributo de las variables relacionado con el ámbito que también está ligado de manera implícita o explícita por una declaración es la asignación de memoria para la variable declarada, así como el tiempo que dura la ejecución de la asignación (que a veces se denomina **tiempo de vida** o **extensión** de la declaración). Por ejemplo, en C, todas las variables cuyas declaraciones son externas a funciones se asignan **estáticamente** (es decir, antes del principio

de la ejecución), así que tienen una extensión igual a todo el tiempo que tome la ejecución del programa, mientras las variables que se declaran dentro de las funciones se asignan sólo mediante la duración de cada llamada de función (lo que se denomina asignación **automática**). C también tiene en cuenta el cambio de la extensión de una declaración dentro de una función, a ser cambiada de automática a estática mediante el uso de la palabra reservada `static` en la declaración, como en

```
int count(void)
{ static int counter = 0;
  return ++counter;
}
```

La función `count` tiene una variable local estática `counter` que retiene su valor de llamada en llamada, de manera que `count` devuelve como su valor el número actual de veces que ha sido llamada.

C también distingue entre declaraciones que se utilizan para controlar la asignación de memoria y aquellas que se emplean para verificar tipos. En C, no se emplea cualquier declaración de variable que comience con la palabra reservada `extern` para efectuar asignación. De este modo, si la función anterior se escribiera como

```
int count(void)
{ extern int counter;
  return ++counter;
}
```

la variable `counter` se tendría que asignar (e inicializar) en otro sitio del programa. El lenguaje C se refiere a las declaraciones que asignan memoria como **definiciones**, mientras que reserva la palabra "declaración" para las declaraciones que no asignan memoria. De manera que, una declaración que comience con la palabra reservada `extern` no es una definición, pero una declaración estándar de una variable, tal como

```
int x;
```

es una definición. En C, puede haber muchas declaraciones de la misma variable, pero sólo una definición.

Las estrategias de asignación de memoria, como la verificación de tipo, forman una importante y compleja área en el diseño de compiladores que es parte de la estructura del **ambiente de ejecución**. Como dedicaremos todo el capítulo siguiente al estudio de los ambientes, aquí no hablaremos más de la asignación. En vez de eso regresaremos a un análisis de ámbito y estrategias para el mantenimiento del ámbito en una tabla de símbolos.

6.3.3 Reglas de ámbito y estructura de bloques

Las reglas de ámbito en los lenguajes de programación varían mucho, pero existen varias reglas que son comunes a muchos lenguajes. En esta sección analizaremos todas estas reglas, la declaración antes del uso y la regla de anidación más próxima para estructura de bloques.

La **declaración antes del uso** es una regla común, utilizada en C y Pascal, que requiere que se declare un nombre en el texto de un programa antes de cualquier referencia al nombre. La declaración antes del uso permite construir la tabla de símbolos a medida que el análisis sintáctico continúa y que las búsquedas se realicen tan pronto como se encuentra una referencia de nombre en el código; si la búsqueda falla, es que ha ocurrido una violación de la declaración antes del uso, y el compilador emitirá un mensaje de error apropiado. De este modo, la declaración antes del uso fomenta la compilación de un paso. Algunos lenguajes

no requieren de la declaración antes del uso (Modula-2 es un ejemplo), y en ellos se requiere de un paso por separado para la construcción de la tabla de símbolos; la compilación en un paso no es posible.

La **estructura de bloques** es una propiedad común de los lenguajes modernos. Un **bloque** en un lenguaje de programación es cualquier construcción que pueda contener declaraciones. Por ejemplo, en Pascal, los bloques son el programa principal y las declaraciones de procedimiento/función. En C, los bloques son las unidades de compilación (es decir, los archivos de código), las declaraciones de procedimiento/función y las sentencias compuestas (secuencias de sentencias encerradas entre llaves `{...}`). Las estructuras y uniones en C (registros en Pascal) también se pueden visualizar como bloques, ya que contienen declaraciones de campo. De manera similar, las declaraciones de clase de los lenguajes de programación orientados a objetos son bloques. Un lenguaje está **estructurado en bloques** si permite la anidación de bloques dentro de otros bloques, y si el ámbito de declaraciones en un bloque está limitado a ése y otros bloques contenidos en el mismo, sujeto a la **regla de anidación más próxima**: dadas varias declaraciones diferentes para el mismo nombre, la declaración que se aplica a una referencia es la única en el bloque anidado más próximo a la referencia.

Para ver cómo la estructura de bloques y la regla de anidación más próxima afectan a la tabla de símbolos, consideremos el fragmento de código en C de la figura 6.14. En este código existen cinco bloques. En primer lugar, existe el bloque del código completo, que contiene las declaraciones de las variables enteras `i` y `j` y la función `f`. En segundo lugar, existe la declaración de `f` misma, que contiene la declaración del parámetro `size`. En tercer lugar, existe la sentencia completa del cuerpo de `f`, que contiene las declaraciones de las variables carácter `i` y `temp`. (La declaración de función y su cuerpo asociado se pueden visualizar alternativamente como la representación de un bloque simple.) En cuarto lugar existe la sentencia compuesta que contiene la declaración `double j`. Finalmente, existe la sentencia compuesta que contiene la declaración `char * j`. Dentro de la función `f` tenemos declaraciones simples de variables `size` y `temp` en la tabla de símbolos, y todos los usos de estos nombres se refieren a estas declaraciones. En el caso del nombre `i`, hay una declaración local de `i` como un `char` dentro de la sentencia compuesta de `f`, y por la regla de anidación más próxima esta declaración reemplaza la declaración no local de `i` como un `int` en el bloque de archivo de código circundante. (Se dice que la `int i` no local tiene un **hueco de ámbito** dentro de `f`.) De manera similar, las declaraciones de `j` en las dos sentencias compuestas posteriores dentro de `f` reemplazan la declaración no local de `j` como un `int` dentro de sus bloques respectivos. En cada caso las declaraciones originales de `i` y `j` se recuperan cuando se sale de los bloques de las declaraciones locales.

Figura 6.14
Fragmento de código en C que ilustra los ámbitos anidados

```
int i,j;

int f(int size)
{ char i, temp;
  ...
  { double j;
    ...
  }
  ...
  { char * j;
    ...
  }
}
```

En muchos lenguajes, tales como Pascal y Ada (pero no C), los procedimientos y funciones también pueden estar anidados. Esto presenta un factor de complicación en el ambiente de tiempo de ejecución para tales lenguajes (el cual se estudiará en el capítulo siguiente), pero no presenta complicaciones particulares para ámbitos anidados. Por ejemplo, el código en Pascal de la figura 6.15 contiene los procedimientos anidados **g** y **h**, pero tiene esencialmente la misma estructura de tabla de símbolos que la del código en C de la figura 6.14 (excepto, por supuesto, para los nombres adicionales **g** y **h**).

Figura 6.15
Fragmento de código en Pascal que ilustra los ámbitos anidados

```
program Ex;
var i,j: integer;

function f(size: integer): integer;
var i,temp: char;

  procedure g;
  var j: real;
  begin
    ...
  end;

  procedure h;
  var j: ^char;
  begin
    ...
  end;

begin (* f *)
  ...
end;

begin (* programa principal *)
  ...
end.
```

Para implementar ámbitos anidados y la regla de anidación más próxima, la operación de *inserción* de la tabla de símbolos no debe sobrescribir declaraciones anteriores, sino que las debe ocultar temporalmente, de manera que la operación de *búsqueda* sólo encuentre la declaración para un nombre que se haya insertado más recientemente. De manera similar, la operación de *eliminación* no debe eliminar todas las declaraciones correspondientes a un nombre, sino sólo la más reciente, revelando cualquier declaración previa. Entonces la construcción de la tabla de símbolos puede continuar realizando operaciones de *inserción* para todos los nombres declarados en la entrada dentro de cada bloque y efectuando operaciones de *eliminación* correspondientes de los mismos nombres a la salida del bloque. En otras palabras, la tabla de símbolos se comporta de una manera parecida a una pila durante el procesamiento de los ámbitos anidados.

Para ver cómo puede mantenerse esa estructura de una manera práctica, considere la implementación de la tabla de dispersión de una tabla de símbolos que se escribió con anterioridad. Si hacemos suposiciones de simplificación semejantes a las de la figura 6.12 (página 297), entonces después de que son procesadas las declaraciones del cuerpo del procedimiento

f en la figura 6.14, la tabla de símbolos podría parecerse a la de la figura 6.16(a). Durante el procesamiento de la segunda sentencia compuesta dentro del cuerpo de f (la que contiene la declaración char * j), la tabla de símbolos se vería como la de la figura 6.16(b). Finalmente, después de salir del bloque de la función f, la tabla de símbolos podría verse como en la figura 6.16(c). Advierta como, para cada nombre, las listas ligadas en cada cubeta se comportan como una pila para las diferentes declaraciones de ese nombre.

Figura 6.16
Contenido de tabla de símbolos en varios lugares en el código de la figura 6.14

(a) Después del procesamiento de las declaraciones del cuerpo de f

(b) Después del procesamiento de la declaración de la segunda sentencia compuesta anidada dentro del cuerpo de f

(c) Después de salir del cuerpo de f (y de eliminar sus declaraciones)

Existen varias alternativas posibles para esta implementación de ámbitos anidados. Una solución es construir una nueva tabla de símbolos para cada ámbito y vincular las tablas desde ámbitos internos a ámbitos externos, de manera que la operación de *búsqueda* continúe buscando automáticamente con una tabla encerrada, si falla al encontrar un nombre en la tabla actual. Abandonar un ámbito requiere entonces menos esfuerzo, ya que no es necesario volver a procesar las declaraciones utilizando operaciones de *eliminación*. En vez de eso, toda la tabla de símbolos correspondiente al ámbito se puede liberar en un paso. Un ejemplo de esta estructura correspondiente a la figura 6.16(b) se proporciona en la figura 6.17. En esa figura se tienen tres tablas, una para cada ámbito, ligadas desde la más interna hasta la más externa. Para abandonar un ámbito sólo se requiere volver a establecer el apuntador de acceso (indicado a la izquierda) al siguiente ámbito más externo.

igura 6.17
structura de tabla de mbolos correspondiente a la gura 6.16(b) en la que se ilizaron tablas separadas ra cada ámbito

Un procesamiento adicional y cálculo de atributos también pueden ser necesarios durante la construcción de la tabla de símbolos, dependiendo del lenguaje y los detalles de la operación del compilador. Un ejemplo es el requerimiento en Ada de que todavía sea visible un nombre no local dentro de un hueco de ámbito, donde puede ser referenciado utilizando una notación semejante a la selección de campo de registro, por medio del nombre asociado con el ámbito en el cual se declara el nombre no local. Por ejemplo, en el código Pascal de la figura 6.15, dentro de la función `f` la variable entera global `i`, en Ada, todavía será visible como la variable `Ex.i` (utilizando el nombre del programa para identificar el ámbito global). De este modo, puede tener sentido, mientras se construye la tabla de símbolos, identificar cada ámbito mediante un nombre y asignar un prefijo a cada nombre declarado dentro de un ámbito mediante sus nombres de ámbito anidados acumulados. Así, todas las ocurrencias del nombre `j` en la figura 6.15 se distinguirían como `Ex.j`, `Ex.f.g.j` y `Ex.f.h.j`. De manera adicional o alternativa, puede necesitarse asignar un **nivel de anidación** o **profundidad de anidación** a cada ámbito y registrar en cada entrada de la tabla de símbolos el nivel de anidación de cada nombre. De esta forma, en la figura 6.15 el ámbito global del programa tiene nivel de anidación 0, las declaraciones de `f` (sus parámetros y variables locales) tienen nivel de anidación 1 y las declaraciones locales tanto de `g` como de `h` tienen nivel de anidación 2.

Un mecanismo que es semejante a las características de selección de ámbito de Ada que se acaba de describir es el **operador de resolución de ámbito `::`** de C++. Este operador permite que se pueda tener acceso al ámbito de una declaración de clase desde el exterior de la declaración misma. Esto puede ser empleado para completar la definición de funciones miembro desde el exterior de la declaración de clase:

```
class A
{ ... int f();...}; // f es una función miembro

A::f() // ésta es la definición de f en A
{ ... }
```

Todas las clases, procedimientos (en Ada) y estructuras de registro pueden visualizarse como representación de un ámbito nombrado, con el conjunto de declaraciones locales como un atributo. En situaciones donde estos ámbitos se pueden referenciar de manera externa, conviene construir una tabla de símbolos por separado para cada ámbito (como en la figura 6.17).

Hasta ahora hemos estado comentando la regla de ámbito estándar que sigue la estructura textual del programa. En ocasiones esto se llama **ámbito léxico** o **estático** (porque la tabla de símbolos se construye de manera estática). Una regla de ámbito alternativa que se utiliza en algunos lenguajes orientados dinámicamente (versiones antiguas de LISP, SNOBOL y algunos lenguajes de consulta de base de datos) se conoce como **ámbito dinámico**. Esta regla requiere que la aplicación de los ámbitos anidados siga la trayectoria de ejecución, en vez del diseño textual del programa. Un ejemplo simple que muestra diferencias entre las

dos reglas es el código en C de la figura 6.18. Este código imprime 1 utilizando la regla de ámbito estándar de C debido a que el ámbito de la variable `i` al nivel de archivo se extiende al procedimiento `f`. Si se utilizara ámbito dinámico, entonces el programa imprimiría 2, porque `f` es llamado desde `main`, y `main` contiene una declaración de `i` (con valor 2) que se extiende a `f` si se utiliza la secuencia de llamadas para resolver referencias no locales. El ámbito dinámico requiere que la tabla de símbolos se construya durante la ejecución realizando operaciones de *inserción* y *eliminación* a medida que los ámbitos se introducen y extraen en tiempo de ejecución. De este modo, el uso del ámbito dinámico requiere que la tabla de símbolos se vuelva parte del ambiente y que el código se genere mediante un compilador para mantenerlo, en vez de tener la tabla de símbolos construida directamente (y estáticamente) por el compilador. El ámbito dinámico compromete, además, la legibilidad de los programas, porque las referencias no locales no pueden resolverse sin simular la ejecución del programa. Finalmente, el ámbito dinámico es incompatible con la verificación de tipo estático, ya que los tipos de datos de las variables deben mantenerse por medio de la tabla de símbolos. (Advierta el problema que se presenta en el código de la figura 6.18 si la `i` dentro de `main` se declara como `double`.) De este modo, el ámbito dinámico rara vez se utiliza en los lenguajes modernos, por ello no comentaremos más al respecto aquí.

6.18
C que ilustra la
ia entre ámbito
y dinámico

```
#include <stdio.h>

int i = 1;

void f(void)
{ printf("%d\n",i);}

void main(void)
{ int i = 2;
  f();
  return 0;
}
```

Finalmente, deberíamos advertir que la operación de *eliminación* se ha mostrado en la figura 6.16 para eliminar del todo declaraciones de la tabla de símbolos. De hecho puede ser necesario mantener las declaraciones en la tabla (o por lo menos no dejar de asignar sus espacios en memoria), porque otras partes del compilador pueden volver a necesitar hacer referencia a ellas posteriormente. Si deben mantenerse en la tabla de símbolos, entonces la operación de *eliminación* necesitaría simplemente marcarlas como ya no activas, y entonces la *búsqueda omitirá* tales declaraciones marcadas mientras busca la tabla.

6.3.4 Interacción de declaraciones del mismo nivel

Otra cuestión que se relaciona con el ámbito es la interacción entre declaraciones del mismo nivel de anidación (es decir, asociadas a un solo bloque). Ésta pueden variar con la clase de declaración, y con el lenguaje que se está traduciendo. Un requerimiento típico en muchos lenguajes (C, Pascal, Ada) es que no se puede volver a utilizar el mismo nombre en declaraciones al mismo nivel. De este modo, en C, las siguientes declaraciones consecutivas provocarán un error de compilación:

```
typedef int i;
int i;
```

Para verificar este requerimiento un compilador debe realizar una *búsqueda* antes de cada *inserción* y determinar mediante algún mecanismo (el nivel de anidación, por ejemplo) si cualquier declaración preexistente con el mismo nombre está o no en el mismo nivel.

Algo más difícil es la cuestión de cuánta información tienen disponible entre sí las declaraciones en una secuencia al mismo nivel. Por ejemplo, considere el fragmento de código en C

```
int i = 1;

void f(void)
{ int i = 2, j = i+1;
  ...
}
...
```

La cuestión es si `j` dentro de `f` es inicializada al valor 2 o 3, es decir, si se utiliza la declaración local para `i` o la declaración no local para `i`. Puede parecer natural que se utilice la declaración más reciente (la local), mediante la regla de anidación más próxima. Ésta es en realidad la manera en que se comporta C. Pero esto presupone que cada declaración se agregue a la tabla de símbolos a medida que se procesa, lo que se conoce como **declaración secuencial**. Es posible que en vez de esto todas las declaraciones se procesen "simultáneamente" y se agreguen a la vez a la tabla de símbolos al final de la sección de declaraciones. Entonces cualquier nombre en las expresiones dentro de las declaraciones haría referencia a las declaraciones anteriores, no a las nuevas declaraciones que se están procesando. Una estructura de declaración así se denomina **declaración colateral**, y algunos lenguajes funcionales, como ML y Scheme, tienen una estructura de declaración como ésta. Una regla para declaraciones de esta naturaleza requiere que las declaraciones no se agreguen inmediatamente a la tabla de símbolos existente, sino que se acumulen en una nueva tabla (o estructura temporal) y después se agreguen a la tabla existente cuando todas las declaraciones se hayan procesado.

Finalmente, existe el caso de la estructura de **declaración recursiva**, en la cual las declaraciones pueden hacer referencia a sí mismas o entre sí. Esto es necesario en particular para declaraciones de procedimiento/función, donde los grupos de funciones mutuamente recursivas son comunes (por ejemplo, en un analizador sintáctico descendente recursivo). En su forma más simple, una función recursiva se llama a sí misma, como en el siguiente código en C, para una función que calcula el máximo común divisor de dos enteros:

```
int gcd(int n, int m)
{ if (m == 0) return n;
  else return gcd(m,n % m);
}
```

Para que esto sea compilado correctamente, el compilador debe agregar el nombre de la función `gcd` a la tabla de símbolos *antes* de procesar el cuerpo de la función. De otro modo, el nombre `gcd` no se hallará (o no tendrá el significado correcto) cuando se encuentre la llamada recursiva. En casos más complejos de un grupo de funciones mutuamente recursivas, como en el siguiente fragmento de código en C

```
void f(void)
{... g() ...}

void g(void)
{... f() ...}
```

no es suficiente con agregar cada función a la tabla de símbolos antes que su cuerpo sea procesado. En realidad, el fragmento anterior de código C producirá un error durante la compilación para la llamada a `g` dentro de `f`. En C, este problema se elimina agregando la denominada declaración de **función prototipo** para `g` antes de la declaración para `f`:

```
void g(void); /* declaración de prototipo
                                de función */

void f(void)
{... g() ...}

void g(void)
{... f() ...}
```

Tal modificación se puede ver como un **modificador de ámbito**, extendiendo el ámbito del nombre `g` para incluir `f`. De este modo, el compilador agrega `g` a la tabla de símbolos cuando se alcanza la (primera) declaración de prototipo para `g` (junto con su propio atributo de ámbito posicional). Naturalmente, el cuerpo de `g` no existe hasta que se alcanza la declaración (o definición) principal de `g`. Además, todos los prototipos de `g` deben tener verificación de tipo para asegurar que son idénticos en estructura.

El problema de la recursividad mutua puede ser resuelto de varias maneras. Por ejemplo, en Pascal se proporciona una declaración **forward** como extensor de ámbito de procedimiento/función. En Modula-2 la regla de ámbito para procedimientos y funciones (además de variables) extiende sus ámbitos para incluir el bloque completo de sus declaraciones, de manera que son mutuamente recursivos de manera natural y no es necesario un mecanismo de lenguaje adicional. Esto requiere un paso de procesamiento en el cual todos los procedimientos y funciones se agreguen a la tabla de símbolos antes de procesar cualquiera de sus cuerpos. Declaraciones mutuamente recursivas similares también se encuentran disponibles en varios otros lenguajes.

6.3.5 Un ejemplo extendido de una gramática con atributos mediante una tabla de símbolos

Ahora queremos considerar un ejemplo que demuestre varias de las propiedades de las declaraciones que hemos descrito y desarrollar una gramática con atributos que haga explícitas estas propiedades en el comportamiento de la tabla de símbolos. La gramática que utilizamos para este ejemplo es la siguiente condensación de la gramática de expresión aritmética simple, junto con una extensión que involucra declaraciones:

$$
\begin{aligned}
&S \rightarrow exp \\
&exp \rightarrow \texttt{(}\ exp\ \texttt{)} \mid exp + exp \mid \texttt{id} \mid \texttt{num} \mid \texttt{let}\ dec\text{-}list\ \texttt{in}\ exp \\
&dec\text{-}list \rightarrow dec\text{-}list\ \texttt{,}\ decl \mid decl \\
&decl \rightarrow \texttt{id} = exp
\end{aligned}
$$

Esta gramática incluye un símbolo inicial S en nivel superior, debido a que la gramática con atributos contiene atributos heredados que necesitarán inicialización en las raíces del árbol sintáctico. La gramática contiene sólo una operación (la adición, con token +) para hacerla más simple. También es ambigua, y se parte del supuesto de que un analizador sintáctico ya construyó un árbol sintáctico o manejó de alguna manera las ambigüedades (dejamos para los ejercicios la construcción de una gramática no ambigua equivalente). Sin embargo,

incluimos paréntesis, de manera que podamos escribir expresiones de manera no ambigua en esta gramática si así lo deseamos. Como en ejemplos anteriores, utilizando gramáticas similares suponemos que **num** e **id** son tokens cuya estructura está determinada por un analizador léxico (puede suponerse que **num** es una secuencia de dígitos y que **id** es una secuencia de caracteres).

La adición a la gramática que involucre las declaraciones es la **expresión *let***:

exp → `let` *dec-list* `in` *exp*

En una expresión *let* las declaraciones se componen de una secuencia de declaraciones separadas por comas de declaraciones de la forma **id** = *exp*. Un ejemplo de esto es

```
let x = 2+1, y = 3+4 in x + y
```

De manera informal, la semántica de una expresión *let* es como sigue. Las declaraciones después del token `let` establecen nombres para expresiones que, cuando aparecen en la *exp* después del token `in`, simbolizan los valores de las expresiones que representan. (La *exp* es el **cuerpo** de la expresión *let*.) El valor de la expresión *let* es el valor de su cuerpo, que se calcula reemplazando cada nombre en las declaraciones por el valor de su *exp* correspondiente y posteriormente calculando el valor del cuerpo de acuerdo con las reglas de la aritmética. Por ejemplo, en el caso anterior, `x` representa el valor 3 (el valor de `2+1`) y, por otro lado, `y` representa el valor 7 (el valor de `3+4`). De esta manera, la expresión *let* misma tiene el valor 10 (= el valor de `x+y` cuando `x` tiene valor 3 e `y` tiene el valor 7).

A partir de la semántica que se acaba de dar, vemos que las declaraciones dentro de una expresión *let* representan una clase de declaración constante (o fija), y que las expresiones *let* representan los bloques de este lenguaje. Para completar la exposición informal de la semántica de estas expresiones, necesitamos describir las reglas de ámbito e interacciones de las declaraciones en las expresiones *let*. Advierta que la gramática permite anidaciones arbitrarias de expresiones *let* entre sí mismas, por ejemplo, en la expresión

```
let x = 2, y = 3 in
  (let x = x+1, y=(let z=3 in x+y+z)
   in (x+y)
  )
```

Estableceremos las siguientes reglas de ámbito para las declaraciones de expresiones *let*. En primer lugar, no habrá declaración repetida del mismo nombre dentro de la misma expresión *let*. De este modo, una expresión de la forma

```
let x=2,x=3 in x+1
```

es ilegal, y da como resultado un error. En segundo lugar, si no es declarado cualquier nombre en alguna expresión *let* circundante, también se obtiene un error. De esta forma, la expresión

```
let x=2 in x+y
```

es errónea. En tercer lugar, el ámbito de cada declaración en una expresión *let* se extiende sobre el cuerpo de *let* de acuerdo con la regla de anidación más próxima para estructura de bloques. Así, el valor de la expresión

```
let x=2 in (let x=3 in x)
```

es 3, no 2 (puesto que la **x** en el *let* interno se refiere a la declaración **x=3**, no a la declaración **x=2**).

Finalmente, definimos la interacción de las declaraciones en una lista de declaraciones en el mismo *let* para que sea secuencial. Es decir, cada declaración utiliza las declaraciones anteriores para resolver nombres dentro de su propia expresión. De este modo, en la expresión

```
let x=2,y=x+1 in (let x=x+y,y=x+y in y)
```

la primera **y** tiene valor 3 (utilizando la declaración anterior de **x**), la segunda **x** tiene valor 5 (utilizando las declaraciones de *let* entre paréntesis) y la segunda **y** tiene valor 8 (empleando la declaración anidada de **y** además del valor de la **x** apenas declarada). Así, la expresión completa tiene valor 8. Invitamos al lector a calcular de manera parecida el valor de la expresión *let* triplemente anidada de la página anterior.

Ahora deseamos desarrollar ecuaciones de atributo que utilicen una tabla de símbolos para estar al tanto de las declaraciones en expresiones *let* y que expresen las reglas de ámbito e interacciones que se acaban de describir. Por simplicidad, sólo utilizaremos la tabla de símbolos para determinar si una expresión es o no errónea. No escribiremos ecuaciones para calcular el valor de las expresiones, pero dejaremos que el lector lo haga como ejercicio. En vez de eso calculamos el atributo booleano sintetizado *err* que tiene el valor **true** ("verdadero") si la expresión es errónea y **false** ("falso") si la expresión es correcta, de acuerdo con las reglas previamente establecidas. Para hacer esto necesitamos tanto un atributo heredado *symtab*, que represente a la tabla de símbolos ("bol le"), como un atributo heredado *nestlevel* ("nivel anidado") para determinar si dos declaraciones están dentro del mismo bloque *let*. El valor de *nestlevel* es un entero no negativo que expresa el nivel de anidación actual de los bloques *let*. Se inicializa con el valor 0 en el nivel más externo.

El atributo *symtab* necesitará operaciones de tabla de símbolos típica. Como queremos escribir ecuaciones de atributo, expresamos las operaciones de tabla de símbolos en una manera libre de efectos colaterales escribiendo la operación de *inserción* para tomar una tabla de símbolos como un parámetro y devolver una nueva tabla de símbolos con la nueva información agregada, pero con la tabla de símbolos original sin sufrir cambios. De esta forma, la inserción *insert*(s, n, l) devolverá una nueva tabla de símbolos que contiene toda la información de la tabla de símbolos s y además asocia el nombre n con el nivel de anidación l, sin cambiar s. (Como estamos determinando sólo la exactitud, no es necesario asociar un valor a n, únicamente un nivel de anidación.) Puesto que esta notación garantiza que podamos recuperar la tabla de símbolos s original, no se necesita una operación *delete* explícita. Finalmente, para probar los dos criterios que deben satisfacerse para la exactitud (que todos los nombres que aparezcan en expresiones se hayan declarado previamente y que no ocurran declaraciones repetidas en el mismo nivel), debemos poder verificar la presencia de un nombre en una tabla de símbolos, y también recuperar el nivel de anidación asociado con un nombre que esté presente. Haremos esto con las dos operaciones *isin*(s, n), que devuelven un valor booleano dependiendo de si n está o no en la tabla de símbolos s, y *lookup*(s, n), que devuelve un valor entero dando el nivel de anidación de la declaración más reciente de n, si existe, o -1, si n no está en la tabla de símbolos s (esto permite expresar ecuaciones utilizando *lookup* sin tener que realizar primero una operación *isin*). Finalmente, debemos tener una notación para una tabla de símbolos inicial que no tenga entradas; escribiremos esto como *emptytable* ("tabla vacía").

Con estas operaciones y convenciones para tabla de símbolos, volvemos ahora a la escritura de las ecuaciones de atributo para los tres atributos *symtab*, *nestlevel* y *err* de las expresiones. La gramática con atributos completa se resume en la tabla 6.9.

Tabla 6.9
Gramática con atributos para expresiones con bloques *let*

Regla gramatical	Reglas semánticas
$S \rightarrow exp$	$exp.symtab = emptytable$ $exp.nestlevel = 0$ $S.err = exp.err$
$exp_1 \rightarrow exp_2 + exp_3$	$exp_2.symtab = exp_1.symtab$ $exp_3.symtab = exp_1.symtab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_3.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$ **or** $exp_3.err$
$exp_1 \rightarrow (\ exp_2\ )$	$exp_2.symtab = exp_1.symtab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$
$exp \rightarrow$ `id`	$exp.err =$ **not** $isin(exp.symtab, \texttt{id}.name)$
$exp \rightarrow$ `num`	$exp.err =$ **false**
$exp_1 \rightarrow$ `let` $dec\text{-}list$ `in` exp_2	$dec\text{-}list.intab = exp_1.symtab$ $dec\text{-}list.nestlevel = exp_1.nestlevel + 1$ $exp_2.symtab = dec\text{-}list.outtab$ $exp_2.nestlevel = dec\text{-}list.nestlevel$ $exp_1.err = (decl\text{-}list.outtab = errtab)$ **or** $exp_2.err$
$dec\text{-}list_1 \rightarrow dec\text{-}list_2$ `,` $decl$	$dec\text{-}list_2.intab = dec\text{-}list_1.intab$ $dec\text{-}list_2.nestlevel = dec\text{-}list_1.nestlevel$ $decl.intab = dec\text{-}list_2.outtab$ $decl.nestlevel = dec\text{-}list_2.nestlevel$ $dec\text{-}list_1.outtab = decl.outtab$
$dec\text{-}list \rightarrow decl$	$decl.intab = dec\text{-}list.intab$ $decl.nestlevel = dec\text{-}list.nestlevel$ $dec\text{-}list.outtab = decl.outtab$
$decl \rightarrow$ `id` = exp	$exp.symtab = decl.intab$ $exp.nestlevel = decl.nestlevel$ $decl.outtab =$ **if** $(decl.intab = errtab)$ **or** $exp.err$ **then** $errtab$ **else if** $(lookup(decl.intab, \texttt{id}.name) =$ $decl.nestlevel)$ **then** $errtab$ **else** $insert(decl.intab, \texttt{id}.name, decl.nestlevel)$

En el nivel más alto, asignamos los valores de los dos atributos heredados y elevamos el valor del atributo sintetizado. De este modo, la regla gramatical $S \rightarrow exp$ tiene las tres ecuaciones de atributo asociadas

$$exp.symtab = emptytable$$
$$exp.nestlevel = 0$$
$$S.err = exp.err$$

Unas reglas similares se mantienen para la regla gramatical $exp \rightarrow (\ exp\ )$.

Para la regla $exp_1 \rightarrow exp_2 + exp_3$ (como es habitual cuando numeramos los no terminales escribiendo ecuaciones de atributo), las reglas siguientes expresan el hecho de que las expresiones del lado derecho heredan los atributos *symtab* y *nestlevel* de la expresión del lado izquierdo, la cual contiene un error si por lo menos una de las expresiones del lado derecho contiene un error:

$$exp_2.symtab = exp_1.symtab$$
$$exp_3.symtab = exp_1.symtab$$
$$exp_2.nestlevel = exp_1.nestlevel$$
$$exp_3.nestlevel = exp_1.nestlevel$$
$$exp_1.err = exp_2.err \textbf{ or } exp_3.err$$

La regla $exp \rightarrow$ **id** produce un error sólo si el nombre del **id** no puede hallarse en la tabla de símbolos actual (escribiremos **id**.*name* para el nombre del identificador y supondremos que se calcula mediante el analizador léxico o el analizador sintáctico), de modo que la ecuación de atributo asociada es

$$exp.err = \textbf{not } isin(exp.symtab, \texttt{id}.name)$$

Por otra parte, la regla $exp \rightarrow$ **num** nunca puede producir un error, de modo que la ecuación de atributo es

$$exp.err = \textbf{false}$$

Ahora volveremos a un análisis de las expresiones *let* con la regla gramatical

$$exp_1 \rightarrow \texttt{let}\ dec\text{-}list\ \texttt{in}\ exp_2$$

y las reglas asociadas para declaraciones. En la regla para las expresiones *let*, la *decl-list* se compone de cierto número de declaraciones que se deben agregar a la tabla de símbolos actual. Expresamos esto asociando dos tablas de símbolos por separado con *decl-list*: la tabla entrante *intab*, que está heredada de exp_1, y la tabla saliente *outtab*, que contiene las nuevas declaraciones (además de las antiguas) y que debe pasarse a exp_2. Debido a que las nuevas declaraciones pueden contener errores (tales como la declaración repetida de los mismos nombres), debemos expresar esto teniendo en cuenta una tabla de símbolos especial *errtab* que se pase desde una *dec-list*. Finalmente, la expresión *let* tiene un error sólo si *dec-list* contiene un error (en cuyo caso su *outtab* sería *errtab*) o si el cuerpo exp_2 de la expresión *let* contiene un error. Las ecuaciones de atributo son

$$dec\text{-}list.intab = exp_1.symtab$$
$$dec\text{-}list.nestlevel = exp_1.nestlevel + 1$$
$$exp_2.symtab = dec\text{-}list.outtab$$
$$exp_2.nestlevel = dec\text{-}list.nestlevel$$
$$exp_1.err = (decl\text{-}list.outtab = errtab) \textbf{ or } exp_2.err$$

Advierta que el nivel de anidación también se incrementa en uno tan pronto como se introduce el bloque *let*.

Resta desarrollar ecuaciones para listas de declaraciones y para declaraciones individuales. Una lista de declaraciones debe acumular, mediante la regla secuencial para declaraciones, declaraciones individuales a medida que el procesamiento de la lista continúa. De este modo, dada la regla

$$dec\text{-}list_1 \rightarrow dec\text{-}list_2 \ , \ decl$$

la *outtab* de $dec\text{-}list_2$ se pasa como la *intab* a *decl*, con lo que se da acceso a *decl* a las declaraciones que la preceden en la lista. En el caso de una declaración simple (*dec-list* → *decl*), se realiza copia y herencia estándar. Las ecuaciones completas se muestran en la tabla 6.9.

Finalmente analizamos el caso de una declaración individual

$$decl \rightarrow \texttt{id} = exp$$

En este caso la *decl.intab* heredada se pasa de inmediato a *exp* (porque las declaraciones en este lenguaje no son recursivas, de modo que *exp* debe encontrar sólo declaraciones anteriores del nombre de este `id`, no del actual). Entonces, si no hay errores, `id`*.name* se inserta en la tabla con el nivel de anidación actual, y esto se pasa hacia atrás como la *outtab* sintetizada de la declaración. Pueden presentarse errores de tres maneras. La primera, un error puede haberse presentado ya en una declaración anterior, en cuyo caso *decl.intab* es *errtab*; esta *errtab* debe ser pasada como *outtab*. La segunda, puede presentarse un error dentro de la *exp*; esto se indica mediante *exp.err* y, si es verdadero, también debe provocar que *outtab* sea *errtab*. La tercera, la declaración puede ser una declaración repetida de un nombre en el mismo nivel de anidación. Esto se debe verificar realizando una *búsqueda* ("*lookup*"), y este error también se debe informar obligando a que *outtab* sea *errtab*. (Advierta que si *lookup* falla en su intento de encontrar `id`*.name*, entonces devuelve -1, el cual nunca coincide con el nivel de anidación actual, de modo que no se genera ningún error.) La ecuación completa para *decl.outtab* se proporciona en la tabla 6.9.

Con esto concluye nuestro análisis de las ecuaciones de atributo.

6.4 TIPOS DE DATOS Y VERIFICACIÓN DE TIPOS

Una de las tareas principales de un compilador es el cálculo y mantenimiento de la información en tipos de datos (**inferencia de tipos**), y el uso de tal información para asegurar que cada parte de un programa tenga sentido bajo las reglas de tipo del lenguaje (**verificación de tipo**). Estas dos tareas por lo regular están estrechamente relacionadas, se realizan juntas, y se hace referencia a ellas simplemente como verificación de tipos. La información del tipo de datos puede ser estática o dinámica, o una mezcla de las dos. En la mayoría de los dialectos de LISP la información de tipo es enteramente dinámica. En un lenguaje así, un compilador debe generar código para realizar inferencia de tipo y verificación de tipos durante la ejecución. En lenguajes más tradicionales, como Pascal, C y Ada, la información de tipo es principalmente estática, y se utiliza como el mecanismo principal para verificar la exactitud de un programa antes de la ejecución. La información de tipo estático también se emplea para determinar el tamaño de memoria necesario para la asignación de cada variable y la manera en que se puede tener acceso a la memoria, y esto puede utilizarse para simplificar el ambiente en tiempo de ejecución. (Comentaremos esto en el capítulo siguiente.) En esta sección sólo nos ocuparemos de la verificación de tipos estática.

La información de tipo de datos puede presentarse en un programa en varias formas diferentes. Teóricamente, un **tipo de datos** es un conjunto de valores, o de manera más precisa, un conjunto de valores con ciertas operaciones sobre ellos. Por ejemplo, el tipo de datos `integer` ("`entero`") en un lenguaje de programación hace referencia a un subconjunto de los enteros matemáticos, junto con las operaciones aritméticas, tales como + y *, que son proporcionadas por la definición del lenguaje. En el terreno práctico de la construcción de compiladores, estos conjuntos por lo regular se describen mediante una **expresión de tipo**, que es un nombre de tipo, tal como `integer`, o una expresión estructurada, tal como

`array [1..10] of real`, cuyas operaciones generalmente se suponen o implican. Las expresiones de tipo se pueden presentar en diversos lugares en un programa. Éstas incluyen declaraciones de variable tales como

```
var x: array [1..10] of real;
```

la cual asocia un tipo con un nombre de variable, y declaraciones de tipo, tales como

```
type RealArray = array [1..10] of real;
```

la cual define un nuevo nombre de tipo para utilizarlo en un tipo posterior o declaraciones de variable. Tal información de tipo es **explícita**. También es posible que la información de tipo sea **implícita**, por ejemplo en la declaración de constante en Pascal

```
const greeting = "Hello!";
```

donde el tipo de `greeting` ("`saludos`") es implícitamente `array [1..6] of char`, de acuerdo con las reglas de Pascal.

La información de tipo, ya sea explícita o implícita, que está contenida en declaraciones, se mantiene en la tabla de símbolos y se recupera mediante el verificador de tipo siempre que se hace referencia a los nombres asociados. Los tipos nuevos se infieren entonces de estos tipos y se asocian con los nodos apropiados en el árbol sintáctico. Por ejemplo, en la expresión

```
a[i]
```

los tipos de datos de los nombres `a` e `i` se obtienen de la tabla de símbolos. Si `a` es del tipo `array [1..10] of real`, e `i` tiene tipo `integer`, entonces se determina que la subexpresión `a[i]` es correcto en tipo y tiene tipo `real`. (La cuestión de si `i` tiene un valor que esté entre 1 y 10 es una cuestión de **verificación de intervalo**, que en general no es estadísticamente determinable.)

La manera en que se representan los tipos de datos mediante un compilador, la forma en que una tabla de símbolos mantiene la información del tipo y las reglas que utiliza un verificador de tipo para inferir los tipos, dependen todas de las clases de expresiones de tipo disponibles en un lenguaje y las reglas de tipo del lenguaje que gobiernen el uso de estas expresiones de tipo.

6.4.1 Expresiones de tipo y constructores de tipos

Un lenguaje de programación siempre contiene un número de tipos ya incluidos, con nombres tales como `int` y `double`. Estos tipos **predefinidos** corresponden, ya sea a tipos de datos numéricos que son provistos internamente mediante diversas arquitecturas de máquina y cuyas operaciones ya existen como instrucciones de máquina, o a tipos elementales como `boolean` y `char`, cuyo comportamiento es fácil de implementar. Tales tipos de datos son **tipos simples**, ya que sus valores no exhiben estructura interna explícita. Una representación típica para enteros es en la forma de complemento a dos de dos o cuatro bytes. Una representación típica para números reales, o de punto flotante, es de cuatro u ocho bytes, con un bit de signo, un campo de exponente y un campo de fracción (o mantisa). Una representación común para caracteres es la de los códigos ASCII de un byte, y para los valores booleanos es la de un byte, del cual sólo se utiliza el bit menos significativo (1 = "true" o verdadero, 0 = "false" o falso). En ocasiones un lenguaje impondrá restricciones sobre cómo se van a implementar estos tipos predefinidos. Por ejemplo, el lenguaje C estándar requiere que un tipo `double` de punto flotante tenga por lo menos 10 dígitos decimales de precisión.

Un tipo predefinido interesante en el lenguaje C es el tipo `void`. Este tipo no tiene valores, así que representa el conjunto vacío. Se le utiliza para representar una función que no devuelva algún valor (es decir, un procedimiento), así como para representar un apuntador que (de momento) apunte a un tipo desconocido.

En algunos lenguajes es posible definir nuevos tipos simples. Ejemplos típicos de éstos son los **tipos subrange** ("subintervalo") y los **tipos enumerated** ("enumerados"). Por ejemplo, en Pascal el subintervalo de los enteros compuesto de los valores del 0 hasta el 9 puede declararse como

```
type Digit = 0..9;
```

mientras que un tipo enumerado compuesto de los valores nombrados `red`, `green` y `blue` puede declararse en C como

```
typedef enum {red,green,blue} Color;
```

Las implementaciones tanto de subintervalos como enumeraciones podrían ser como enteros, o se podría utilizar una cantidad más pequeña de memoria que fuera suficiente para representar todos los valores.

Dado un conjunto de tipos predefinidos, se pueden crear nuevos tipos de datos utilizando **constructores de tipo**, tales como `array` y `record` o `struct`. Tales constructores se pueden ver como funciones que toman tipos existentes como sus parámetros y devuelven nuevos tipos con una estructura que depende del constructor. Tales tipos con frecuencia son denominados **tipos estructurados**. En la clasificación de estos tipos es importante comprender la naturaleza del conjunto de valores que un tipo así representa. A menudo, un constructor de tipo equivaldrá estrechamente a una operación de conjuntos en los conjuntos de valores subyacentes de sus parámetros. Ilustraremos esto mencionando varios constructores comunes y comparándolos con las operaciones de conjuntos.

Array (Arreglo) El constructor de tipo *array* ("arreglo") o ("matriz") toma dos parámetros de tipo, en donde uno es el **tipo-índice** y el otro el **tipo-componente** y produce un nuevo tipo array. En una sintaxis tipo Pascal escribiríamos

`array` [*tipo-índice*] `of` *tipo-componente*

Por ejemplo, la expresión tipo Pascal

```
array [Color] of char;
```

crea un tipo array cuyo tipo-índice es `Color` y cuyo tipo-componente es `char`. Con frecuencia existen restricciones en los tipos que se pueden presentar como tipos de índice. Por ejemplo, en Pascal un tipo de índice está limitado a los denominados **tipos ordinales**: tipos para los cuales todo valor tiene un predecesor inmediato y un sucesor inmediato. Tales tipos incluyen subintervalos enteros y de carácter, así como tipos enumerados. En contraste, en lenguaje C sólo se permiten intervalos enteros comenzando en 0, y sólo se especifica el tamaño en lugar del tipo-índice. En realidad, no hay palabra reservada que corresponda a `array` en C: los tipos arreglo se declaran escribiendo simplemente como sufijo del nombre el intervalo entre corchetes. De modo que no hay un equivalente directo en C para la expresión anterior tipo Pascal, pero la declaración de tipo en C

```
typedef char Ar[3];
```

define un tipo `Ar` que tiene una estructura de tipo equivalente al tipo precedente (suponiendo que `Color` tenga tres valores).

Un arreglo representa valores que son secuencias de valores del tipo componente, indizadas mediante valores del tipo índice. Es decir, si *tipo-índice* tiene conjunto de valores I y *tipo-componente* tiene conjunto de valores C, entonces el conjunto de valores correspondiente al tipo `array [`*tipo-índice*`] of` *tipo-componente* es el conjunto de secuencias finitas de elementos de C indizados mediante elementos de I, o, en términos matemáticos, el conjunto de las funciones $I \rightarrow C$. Las operaciones asociadas sobre los valores del tipo arreglo se componen de la operación única de subindización, la cual se puede utilizar para asignar valores a componentes o buscar valores de componentes: `x := a[red]` o `a[blue] := y`.

Por lo regular, a los arreglos se les asigna almacenamiento contiguo desde los índices más pequeños a los más grandes, para permitir el uso de cálculos de desplazamiento automáticos durante la ejecución. La cantidad de memoria necesaria es n * *tamaño*, donde n es el número de valores en el tipo de índice y *tamaño* es la cantidad de memoria necesaria para un valor del tipo componente. De este modo, una variable del tipo `array [0..9] of integer` necesita 40 bytes de almacenamiento si cada entero ocupa 4 bytes.

Una complicación en la declaración de arreglos es la de los **arreglos multidimensionales**. Éstos a menudo se pueden declarar utilizando aplicaciones repetidas del constructor de tipo arreglo, como en

```
array [0..9] of array [Color] of integer;
```

o una versión más breve en la cual los conjuntos de índice se enumeran juntos:

```
array [0..9,Color] of integer;
```

La subindización en el primer caso aparece como `a[1][red]`, mientras que en el segundo caso escribimos `a[1,red]`. Un problema con los subíndices múltiples es que la secuencia de valores se puede organizar de diferentes maneras en la memoria, dependiendo de si la indización se hace primero en el primer índice y después sobre el segundo índice, o al contrario. La indización sobre el primer índice produce una secuencia en la memoria parecida a `a[0,red]`, `a[1,red]`, `a[2,red]`, . . . , `a[9,red]`, `a[0,blue]`, `a[1,blue]`, . . . , y así sucesivamente (esto se denomina **forma de columna mayor**), mientras que la indización en el segundo índice produce el orden en la memoria de `a[0,red]`, `a[0,blue]`, `a[0,green]`, `a[1,red]`, `a[1,blue]`, `a[1,green]`, . . . , y así sucesivamente (esto se denomina **forma de renglón mayor**). Si la versión repetida de la declaración de arreglo multidimensional va a ser equivalente a la versión abreviada (es decir, `a[0,red] = a[0][red]`), entonces se debe utilizar la forma de renglón mayor, ya que los índices diferentes se pueden agregar por separado: `a[0]` debe tener tipo `array [Color] of integer` y debe hacer referencia a una porción contigua de memoria. El lenguaje FORTRAN, en el que no hay indización parcial de arreglos multidimensionales, se ha implementado tradicionalmente utilizando la forma de columna mayor.

En ocasiones un lenguaje permitirá el uso de un arreglo cuyo intervalo de índices no esté especificado. Un **arreglo de indización abierta** es especialmente útil en la declaración de parámetros de arreglo para funciones, de manera que la función pueda manejar arreglos de diferentes tamaños. Por ejemplo, se puede usar la declaración en C

```
void sort (int a[], int first, int last)
```

para definir un procedimiento de clasificación que funcionará en cualquier arreglo de tamaño `a`. (Por supuesto, se debe emplear algún método para determinar el tamaño real en el momento de la llamada. En este ejemplo otros parámetros lo hacen.)

Record (Registro) Un constructor de tipo **record** o **structure** ("**estructura**") toma una lista de nombres y tipos asociados y construye un nuevo tipo, como en el ejemplo en C

```
struct
{ double r;
  int i;
}
```

Los registros difieren de los arreglos en que se pueden combinar los componentes de diferentes tipos (en un arreglo todos los componentes tienen el mismo tipo) y en que se utilizan nombres (más que índices) para tener acceso a los diferentes componentes. Los valores de un tipo "registro" corresponden aproximadamente al producto cartesiano de los valores de sus tipos componentes, sólo que se utilizan nombres en vez de posiciones para tener acceso a los componentes. Por ejemplo, el registro previamente dado corresponde aproximadamente al producto cartesiano $R \times I$, donde R es el conjunto correspondiente al tipo `double` e I es el conjunto correspondiente a `int`. De manera más precisa, el registro dado corresponde al producto cartesiano $(\mathtt{r} \times R) \times (\mathtt{i} \times I)$, donde los nombres `r` e `i` identifican los componentes. Estos nombres generalmente se utilizan con una **notación punto** para seleccionar sus componentes correspondientes en expresiones. De este modo, si `x` es una variable del tipo de registro dado, entonces `x.r` se refiere a su primer componente y `x.i` al segundo.

Algunos lenguajes tienen constructores de tipo de producto cartesiano puros. Un lenguaje de esta clase es ML, donde `int*real` es la notación para el producto cartesiano de los números enteros y los reales. Los valores del tipo `int*real` se escriben como tuplas, tales como `(2,3.14)`, y se tiene acceso a los componentes mediante funciones de proyección `fst` y `snd` (de las palabras en inglés *first* y *second*): `fst(2, 3.14) = 2` y `snd(2, 3.14) = 3.14`.

El método de implementación estándar para un registro o producto cartesiano consiste en asignar memoria de manera secuencial, con un bloque de memoria para cada tipo de componente. Así, si un número real requiere cuatro bytes y un entero dos bytes, entonces la estructura del registro dado anteriormente necesita seis bytes de almacenamiento, que se asignan como se muestra a continuación:

Union (Unión) Un tipo unión corresponde a la operación de unión de conjuntos. Está disponible directamente en C por medio de la declaración `union`, por ejemplo la declaración

```
union
{ double r;
  int i;
}
```

la cual define un tipo que es la unión de los números reales y los números enteros. Estrictamente hablando, ésta es una **unión disjunta**, porque cada valor se visualiza como un real o como un entero, pero nunca como ambos. La interpretación que se desea se aclara mediante el nombre del componente utilizado para tener acceso al valor: si `x` es una variable del tipo unión dado, entonces `x.r` significa el valor de `x` como un número real, mientras que `x.i` significa el valor de `x` como un entero. Matemáticamente esta unión se define escribiendo $(\mathtt{r} \times R) \cup (\mathtt{i} \times I)$.

El método de implementación estándar para una unión es asignar la memoria en paralelo para cada componente, de manera que la memoria para cada tipo del componente se superponga con todas las demás. Así, si un número real requiere de cuatro bytes, y un entero de dos bytes, entonces la estructura de unión previamente dada necesita sólo cuatro bytes de almacenamiento (el máximo de los requerimientos de memoria de sus componentes), y esta memoria se asigna de la manera siguiente:

Una implementación así requiere, de hecho, que la unión se interprete como una unión disjunta, puesto que la representación de un entero no concordará con su correspondiente representación como uno real. En realidad, sin algún medio para que un programador distinga los valores, una unión así conduce a errores en la interpretación de los datos, y también proporciona una forma para burlar a un verificador de tipo. Por ejemplo en C, si **x** es una variable que tiene el tipo unión dado, entonces al escribir

```
x.r = 2.0;
printf("%d",x.i);
```

no se provocará un error de compilación, pero en lugar de que se imprima el valor 2, se imprimirá un valor erróneo.

Esta inseguridad en los tipos unión ha sido abordada por muchos diseños de lenguajes diferentes. En Pascal, por ejemplo, un tipo unión se declara utilizando el denominado **registro variante**, en el cual los valores de un tipo ordinal se registran en un componente **discriminante**, el cual se utiliza para distinguir los valores deseados. De este modo, el anterior tipo unión se puede escribir en Pascal como

```
record case isReal: boolean of
  true:(r: real);
  false:(i: integer);
end;
```

Ahora una variable **x** de este tipo tiene tres componentes: `x.isReal` (un valor booleano), `x.r` y `x.i`, y el campo `isReal` está asignado en un espacio separado que no se superpone en la memoria. Cuando se asigna un valor real, como `x.r := 2.0`, también se debería, entonces, hacer la asignación `x.isReal := true`. En realidad, este mecanismo es relativamente inútil (al menos para el uso del compilador en verificación de tipo), puesto que el discriminante se puede asignar por separado de los valores que se están discriminando. Efectivamente, Pascal permite que el componente discriminante (pero no el uso de sus valores para distinguir los casos) se omita eliminando su nombre en la declaración, como en

```
record case boolean of
  true:(r: real);
  false:(i: integer);
end;
```

lo que ya no provoca que se asigne espacio para el discriminante en la memoria. De esta forma, los compiladores de Pascal casi nunca intentan usar el discriminante como una verificación de legalidad. Ada, por otra parte, tiene un mecanismo semejante, pero insiste

en que, en cualquier momento que se asigne un componente de unión, se debe asignar el discriminante de manera simultánea con un valor que pueda ser verificado para asegurar la legalidad.

Mediante los lenguajes funcionales, como el ML, se adopta un enfoque diferente. En este lenguaje, un tipo unión se define utilizando una barra vertical para indicar unión, y dando nombres a cada componente para discriminarlos, como en

```
IsReal of real | IsInteger of int
```

Ahora también se deben emplear los nombres `IsReal` e `IsInteger` siempre que se utilice un valor de este tipo, como en (`IsReal 2.0`) o en (`IsInteger 2`). Los nombres `IsReal` e `IsInteger` se conocen como **constructores de valor**, porque ellos "construyen" los valores de este tipo. Puesto que siempre se deben utilizar cuando se hace la referencia a valores de este tipo, no puede ocurrir ningún error de interpretación.

Pointer (Apuntador) Un tipo pointer o apuntador se compone de valores que son referencias a valores de otro tipo. De esta forma, un valor de un tipo apuntador es una dirección de memoria cuya ubicación mantiene un valor de su tipo base. Los tipos apuntador se imaginan con frecuencia como tipos numéricos, porque se pueden realizar cálculos aritméticos en ellos, tales como desplazamientos de suma y multiplicación por factores de escala. No son en realidad tipos simples, ya que se construyen a partir de tipos existentes mediante la aplicación de un constructor de tipo apuntador. Tampoco existe una operación de conjuntos estándar que corresponda directamente al constructor de tipo apuntador, del modo que el producto cartesiano corresponde al constructor de registro. De este modo, los tipos apuntador ocupan una posición algo especial en un sistema de tipos.

En Pascal el carácter `^` corresponde al constructor de tipo apuntador, de manera que la expresión de tipo `^integer` significa "apuntador a un entero". En C, una expresión de tipo equivalente es `int*`. La operación básica estándar en un valor de tipo apuntador es la operación de **desreferenciación**. Por ejemplo, en Pascal, el `^` representa el operador de desreferenciación (además del constructor de tipo apuntador). Y si `p` es una variable del tipo `^integer`, entonces `p^` es el valor desreferenciado de `p` y tiene tipo `integer`. Una regla similar se mantiene en C, donde `*` desreferencia a una variable apuntador y se escribe `*p`.

Los tipos apuntador son muy útiles para describir tipos recursivos, los cuales comentaremos en breve. Al hacerlo así, el constructor de tipo apuntador se aplica más frecuentemente a tipos de registro.

Los tipos apuntador reciben espacio de asignación sobre la base del tamaño de dirección de la máquina objetivo. Por lo regular, éste es de cuatro, o en ocasiones, ocho bytes. A veces, una arquitectura de máquina obligará a un esquema de asignación más complejo. En PC basadas en DOS, por ejemplo, se hace una distinción entre apuntadores **near** (para direcciones dentro de un segmento, con un tamaño de dos bytes) y apuntadores **far** (direcciones que pueden estar fuera de un segmento simple, con un tamaño de cuatro bytes).

Function (Función) Ya advertimos que un arreglo se puede ver como una función de su conjunto de índices hacia su conjunto de componentes. Muchos lenguajes (pero no Pascal o Ada) tienen una capacidad más general para describir tipos función. Por ejemplo, en Modula-2 la declaración

```
VAR f: PROCEDURE (INTEGER): INTEGER;
```

declara la variable `f` como de tipo función (o procedimiento), con un parámetro entero y produciendo un resultado entero. En notación matemática este conjunto se describiría como

el conjunto de funciones $\{f: I \rightarrow I\}$, donde I es el conjunto de los enteros. En el lenguaje ML este mismo tipo se escribiría como `int -> int`. El lenguaje C también tiene tipos función, pero se deben escribir como "apuntador a función", en una notación algo torpe. Por ejemplo, la declaración de Modula-2 que acabamos de dar en C se debe escribir como

```
int (*f) (int);
```

Se asigna espacio a los tipos función de acuerdo con el tamaño de dirección de la máquina objetivo. Dependiendo del lenguaje y la organización del ambiente en tiempo de ejecución, un tipo función puede necesitar tener espacio asignado a un apuntador de código sólo (apuntando al código que implementa la función) o a un apuntador de código y un apuntador de ambiente (apuntando a una ubicación en el ambiente en tiempo de ejecución). El papel del apuntador de ambiente se comentará en el capítulo siguiente.

Class (Clase) La mayoría de los lenguajes orientados a objetos tienen una declaración *class* semejante a una declaración de registro, excepto porque incluye la definición de operaciones, denominadas **métodos** o **funciones miembro**. Las declaraciones de clase pueden o no crear nuevos tipos en un lenguaje orientado a objetos (en C++ lo hacen). Incluso si éste es el caso, las declaraciones de clase no son *únicamente* tipos, porque permiten el uso de características que están más allá del sistema de tipos, tales como la herencia y el enlace dinámico.[10] Estas últimas propiedades deben ser mantenidas por estructuras de datos separadas, tales como la **jerarquía de clase** (una gráfica acíclica dirigida), la cual implementa la herencia, y la **tabla de método virtual**, la cual implementa el enlace dinámico. Volveremos a hablar de estas estructuras en el capítulo siguiente.

6.4.2 Nombres de tipos, declaraciones de tipo y tipos recursivos

Los lenguajes que tienen un rico conjunto de constructores de tipo por lo regular también tienen un mecanismo para que un programador asigne nombres a expresiones de tipo. Tales **declaraciones de tipo** (en ocasiones también conocidas como **definiciones de tipo**) incluyen el mecanismo `typedef` de C y las definiciones de tipo de Pascal. Por ejemplo, el código en C

```
typedef struct
        { double r;
          int i;
        } RealIntRec;
```

define el nombre `RealIntRec` como un nombre para el tipo de registro construido por la expresión de tipo `struct` que lo precede. En el lenguaje ML, una declaración similar (pero sin los nombres de campo) es la siguiente:

```
type RealIntRec = real*int;
```

10. En algunos lenguajes, como en C++, la herencia se refleja o se duplica en el sistema de tipos, puesto que las subclases se visualizan como **subtipos** (un tipo S es un subtipo del tipo T si todos sus valores se pueden contemplar como valores de T o, en terminología de conjuntos, si $S \subset T$).

El lenguaje C tiene un mecanismo de nombrado de tipos adicional en el que un nombre se puede asociar directamente con un constructor de **struct** o **union** sin utilizar una **typedef** directamente. Por ejemplo, el código en C

```
struct RealIntRec
      { double r;
        int i;
      };
```

también declara un nombre de tipo **RealIntRec**, pero se debe usar con el nombre de constructor **struct** en declaraciones de variables:

```
struct RealIntRec x; /* declara x como una variable
                                        de tipo RealIntRec */
```

Las declaraciones de tipo ocasionan que se introduzcan los nombres de tipo declarados en la tabla de símbolos de la misma manera que las declaraciones de variable determinan que se introduzcan los nombres de variables. Una cuestión que se presenta aquí es si se pueden o no volver a utilizar los nombres de tipo como nombres de variable. Por lo regular no se puede (excepto cuando se permite mediante las reglas de anidación de ámbito). El lenguaje C tiene una pequeña excepción a esta regla, ya que los nombres asociados con las declaraciones **struct** o **union** se pueden volver a utilizar como nombres **typedef**:

```
struct RealIntRec
      { double r;
        int i;
      };
typedef struct RealIntRec RealIntRec;
/* ¡es una declaración legal! */
```

Esto se puede lograr considerando que el nombre de tipo introducido mediante la declaración **struct** es la cadena completa "**struct RealIntRec**", la cual es diferente del nombre de tipo **RealIntRec** introducido mediante la **typedef**.

Los nombres de tipo están asociados con atributos en la tabla de símbolos en una forma similar a las declaraciones de variable. Esos atributos incluyen el ámbito (el cual puede ser inherente en la estructura de la tabla de símbolos) y la expresión de tipo correspondiente al nombre de tipo. Como los nombres de tipo pueden aparecer en expresiones de tipo, se presentan cuestiones acerca del uso recursivo de los nombres de tipo que son similares a las definiciones recursivas de las funciones analizadas en la sección anterior. Estos **tipos de datos recursivos** son muy importantes en los lenguajes de programación modernos e incluyen listas, árboles y muchas otras estructuras.

Los lenguajes se dividen en dos grupos generales en su manejo de los tipos recursivos. El primer grupo se compone de aquellos lenguajes que permiten el uso directo de la recursividad en declaraciones de tipo. Un lenguaje de esta clase es ML. En ML, por ejemplo, un árbol de búsqueda binaria que contiene enteros se puede declarar como

```
datatype intBST = Nil | Node of int*intBST*intBST
```

Esto se puede ver como una definición de **intBST**, o bien, como la unión del valor **Nil** con el producto cartesiano de los enteros con dos copias de **intBST** mismo (una para el

subárbol izquierdo y una para el subárbol derecho). Una declaración equivalente en C (en forma ligeramente alterada) es

```
struct intBST
{ int isNull;
  int val;
  struct intBST left,right;
};
```

Esta declaración, sin embargo, generará un mensaje de error en C, el cual es causado por el uso recursivo del nombre del tipo **intBST**. El problema es que estas declaraciones no determinan el tamaño de la memoria necesaria para asignar una variable de tipo **intBST**. La clase de lenguajes, como ML, que pueden aceptar tales declaraciones son aquellos que no necesitan tal información antes de la ejecución, y ésos son los lenguajes que proporcionan *un mecanismo de asignación y desasignación de memoria automático general*. Tales facilidades para la administración de memoria son parte del ambiente en tiempo de ejecución y se comentan en el capítulo siguiente. C no tiene un mecanismo de esta naturaleza, entonces debe hacer tales declaraciones de tipo recursivo ilegales. C es representativo del segundo grupo de lenguajes: aquellos que no permiten el uso directo de la recursividad en declaraciones de tipo.

La solución para tales lenguajes es permitir la recursividad sólo **indirectamente**, a través de apuntadores. Una declaración correcta para **intBST** en C es

```
struct intBST
{ int val;
  struct intBST *left,*right;
};
typedef struct intBST * intBST;
```

o

```
typedef struct intBST * intBST;
struct intBST
{ int val;
  intBST left,right;
};
```

(En C, las declaraciones recursivas requieren del uso de la forma **struct** o **union** para la declaración del nombre de tipo recursivo.) En estas declaraciones el tamaño de la memoria de cada tipo se calcula directamente por el compilador, pero el espacio para los valores se debe asignar manualmente por el programador a través del uso de procedimientos de asignación tales como **malloc**.

6.4.3 Equivalencia de tipos

Dadas las posibles expresiones de tipo de un lenguaje, un verificador de tipo con frecuencia debe responder la cuestión de cuándo dos expresiones de tipo representan al mismo tipo. Ésta es la cuestión de la **equivalencia de tipos**. Existen muchas maneras posibles para que la equivalencia de tipos sea definida por un lenguaje. En esta sección comentaremos brevemente sólo las formas más comunes de equivalencia. En este análisis representamos la equivalencia de tipos como sería en un analizador semántico de compilador, a saber, como una función

function *typeEqual* (*t1*, *t2* : *TypeExp*) : *Boolean*;

que toma dos expresiones de tipo y devuelve *verdadero* si representan el mismo tipo de acuerdo con las reglas de equivalencia de tipo del lenguaje y *falso* si no es así. Daremos varias descripciones en pseudocódigo diferentes de esta función para distintos algoritmos de equivalencia de tipos.

Una cuestión que se relaciona directamente con la descripción de los algoritmos de equivalencia de tipos es la manera en que se representan las expresiones de tipo dentro de un compilador. Un método directo es utilizar una representación de árbol sintáctico, ya que esto facilita traducir directamente desde la sintaxis en declaraciones hasta la representación interna de los tipos. Para tener un ejemplo concreto de tales representaciones considere la gramática para expresiones de tipo y declaraciones de variable dadas en la figura 6.19. Allí aparecen versiones simples de muchas de las estructuras de tipo que hemos analizado. Sin embargo, no hay declaraciones de tipo que permitan la asociación de nuevos nombres de tipo a expresiones de tipo (por lo tanto, no son posibles los tipos recursivos, a pesar de la presencia de tipos apuntador). En esa figura queremos describir una posible estructura de árbol sintáctico para expresiones de tipo correspondiente a las reglas gramaticales.

Considere, en primer lugar, la expresión de tipo

```
record
  x: pointer to real;
  y: array [10] of int
end
```

Esta expresión de tipo se puede representar mediante el árbol sintáctico

Aquí los nodos hijos del registro están representados como una lista de hermanos, puesto que el número de componentes de un registro es arbitrario. Advierta que los nodos que representan tipos simples forman las hojas del árbol.

Figura 6.19
Una gramática simple para expresiones de tipo

var-decls → *var-decls* **;** *var-decl* | *var-decl*
var-decl → **id** **:** *type-exp*
type-exp → *simple-type* | *structured-type*
simple-type → **int** | **bool** | **real** | **char** | **void**
structured-type → **array** **[num]** **of** *type-exp* |
record *var-decls* **end** |
union *var-decls* **end** |
pointer **to** *type-exp* |
proc **(** *type-exps* **)** *type-exp*
type-exps → *type-exps* **,** *type-exp* | *type-exp*

De manera similar, la expresión de tipo

```
proc(bool,union a:real; b:char end,int):void
```

se puede representar mediante el árbol sintáctico

Advierta que los tipos de parámetro también están dados como una lista de hermanos, mientras que el tipo de resultado (en este ejemplo `void`) se distingue haciéndolo directamente un hijo de `proc`.

La primera clase de equivalencia de tipo que describimos, y la única disponible en la ausencia de nombres para tipos, es la **equivalencia estructural**. En esta vista de equivalencia dos tipos son los mismos si y sólo si tienen la misma estructura. Si los árboles sintácticos se emplean para representar tipos, esta regla nos dice que dos tipos son equivalentes si y sólo si tienen árboles sintácticos que sean idénticos en estructura. Como un ejemplo de cómo la equivalencia estructural se verifica en la práctica, la figura 6.20 proporciona una descripción en pseudocódigo de la función *typeEqual* para la equivalencia estructural de dos expresiones de tipo dadas por la gramática de la figura 6.19, utilizando la estructura de árbol sintáctico que acabamos de describir.

Observamos del pseudocódigo de la figura 6.20 que esta versión de equivalencia estructural implica que dos arreglos no son equivalentes a menos que tengan el mismo tamaño y tipo de componente, y dos registros no son equivalentes a menos que tengan los mismos componentes con los mismos nombres y en el mismo orden. Es posible hacer diferentes selecciones en un algoritmo de equivalencia estructural. Por ejemplo, el tamaño del arreglo se podría ignorar en la determinación de la equivalencia, y también se podría permitir que los componentes de una estructura o unión se presenten en un orden diferente.

Una equivalencia de tipos mucho más restrictiva se puede definir cuando se pueden declarar nuevos nombres de tipo para expresiones de tipo en una declaración de tipo. En la figura 6.21 modificamos la gramática de la figura 6.19 para incluir declaraciones de tipo, y también restringimos declaraciones de variable y subexpresiones de tipo a tipos simples y nombres de tipo. Con estas declaraciones ya no se puede escribir

```
record
  x: pointer to real;
  y: array [10] of int
end
```

en vez de eso debemos escribir

```
t1 = pointer to real;
t2 = array [10] of int;
t3 = record
       x: t1;
       y: t2
end
```

Figura 6.20
Pseudocódigo para una función *typeEqual* que prueba la equivalencia estructural de expresiones de tipo de la gramática de la figura 6.19

```
function typeEqual ( t1, t2 : TypeExp ) : Boolean;
var temp : Boolean ;
    p1, p2 : TypeExp ;
begin
  if t1 y t2 son de tipo simple then return t1 = t2
  else if t1.clase = arreglo and t2.clase = arreglo then
     return t1.tamaño = t2.tamaño and typeEqual ( t1.hijo1, t2.hijo1 )
  else if t1.clase = registro and t2.clase = registro
       or t1.clase = unión and t2.clase = unión then
  begin
    p1 := t1.hijo1 ;
    p2 := t2.hijo1 ;
    temp := verdadero ;
    while temp and p1 ≠ nil and p2 ≠ nil do
        if p1.nombre ≠ p2.nombre then
          temp := falso
        else if not typeEqual ( p1.hijo1 , p2.hijo1 )
        then temp := falso
        else begin
          p1 := p1.hermano ;
          p2 := p2.hermano ;
        end;
    return temp and p1 = nil and p2 = nil ;
  end
  else if t1.clase = apuntador and t2.clase = apuntador then
     return typeEqual ( t1.hijo1 , t2.hijo1 )
  else if t1.clase = proc and t2.clase = proc then
  begin
    p1 := t1.hijo1 ;
    p2 := t2.hijo1 ;
    temp := verdadero ;
    while temp and p1 ≠ nil and p2 ≠ nil do
        if not typeEqual ( p1.hijo1 , p2.hijo1 )
        then temp := falso
        else begin
          p1 := p1.hermano ;
          p2 := p2.hermano ;
        end;
    return temp and p1 = nil and p2 = nil
            and typeEqual ( t1.hijo2 , t2.hijo2 )
  end
  else return falso ;
end ; (* typeEqual *)
```

Figura 6.21
Expresiones de tipo con declaraciones de tipo

var-decls → *var-decls* **;** *var-decl* | *var-decl*
var-decl → **id** **:** *simple-type-exp*
type-decls → *type-decls* **;** *type-decl* | *type-decl*
type-decl → **id** **=** *type-exp*
type-exp → *simple-type-exp* | *structured-type*
simple-type-exp → *simple-type* | **id**
simple-type → **int** | **bool** | **real** | **char** | **void**
structured-type → **array** **[num]** **of** *simple-type-exp* |
record *var-decls* **end** |
union *var-decls* **end** |
pointer to *simple-type-exp* |
proc(*type-exps* **)** *simple-type-exp*
type-exps → *type-exps* **,** *simple-type-exp* | *simple-type-exp*

Ahora podemos definir la equivalencia de tipos basados en nombres de tipo, una forma de equivalencia de tipos que se denomina **equivalencia de nombre**: dos expresiones de tipo son equivalentes si y sólo si son el mismo tipo simple o el mismo nombre de tipo. Ésta es una clase muy fuerte de equivalencia de tipo, puesto que ahora, dadas las declaraciones de tipo

```
t1 = int;
t2 = int
```

los tipos `t1` y `t2` no son equivalentes (porque tienen nombres diferentes) y tampoco son equivalentes respecto a `int`. La equivalencia de nombre pura es muy fácil de implementar, ya que una función *typeEqual* se puede escribir en unas cuantas líneas:

function *typeEqual* (*t1, t2* : *TypeExp*) : *Boolean*;
var *temp* : *Boolean* ;
p1, p2 : *TypeExp* ;
begin
if *t1 y t2 son de tipo simple* **then**
return *t1* = *t2*
else if *t1 y t2 son nombres de tipo* **then**
return *t1* = *t2*
else return *false* ;
end;

Naturalmente, las expresiones de tipo reales correspondientes a nombres de tipo se deben introducir en la tabla de símbolos para permitir el cálculo posterior del tamaño de memoria para asignación de almacenamiento y para verificar la validez de operaciones tales como la desreferenciación de apuntador y selección de componente.

Una complicación en la equivalencia de nombre es cuando otras expresiones de tipo además de los tipos simples o nombres de tipo, continúan permitiéndose en declaraciones de variable, o como subexpresiones de expresiones tipo. En tales casos una expresión de tipo puede no tener un nombre explícito asignado, y un compilador tendrá que generar un nombre interno para la expresión de tipo que es diferente a cualquier otro nombre. Por ejemplo, dadas las declaraciones de variable

```
x: array [10] of int;
y: array [10] of int;
```

las variables **x** y **y** tienen asignados nombres de tipo diferentes (y únicos) correspondientes a la expresión de tipo `array [10] of int`.

Es posible conservar la equivalencia estructural en presencia de nombres de tipo. En este caso, cuando se encuentra un nombre, su expresión de tipo correspondiente debe obtenerse de la tabla de símbolos. Esto se puede llevar a cabo agregando el caso siguiente al código de la figura 6.20,

else if *t1 y t2 son nombres de tipo* **then**
  return *typeEqual(getTypeExp(t1), getTypeExp(t2))*

donde *getTyptExp* es una operación de tabla de símbolos que devuelve la estructura de expresión de tipo asociada con su parámetro (que debe ser un nombre de tipo). Esto requiere que se inserte cada nombre de tipo en la tabla de símbolos con una expresión de tipo que represente su estructura, o cuando menos aquellas cadenas de nombres de tipo resultantes de declaraciones de tipo tales como

```
t2 = t1;
```

conduzcan finalmente de regreso a una estructura de tipo en la tabla de símbolos.

Debe tenerse cuidado al implementar la equivalencia estructural cuando sean posibles referencias de tipo recursivos, porque el algoritmo que se acaba de describir puede resultar en un bucle infinito. Esto se puede evitar modificando el comportamiento de la llamada *typeEqual(t1, t2)* en el caso en que *t1* y *t2* son nombres de tipo para incluir la suposición de que ya son potencialmente iguales. Entonces, si la función *typeEqual* siempre regresa a la misma llamada, se puede declarar un éxito en ese punto. Considere, por ejemplo, las declaraciones de tipo

```
t1 = record
        x: int;
        t: pointer to t2;
     end;

t2 = record
        x: int;
        t: pointer to t1;
     end;
```

Dada la llamada *typeEqual(t1, t2)*, la función *typeEqual* supondrá que *t1* y *t2* son potencialmente iguales. Entonces se obtendrán las estructuras tanto de *t1* como de *t2*, y el algoritmo continuará con éxito hasta que se haga la llamada *typeEqual(t2, t1)* para analizar los tipos hijos de las declaraciones de apuntador. Entonces esta llamada inmediatamente devolverá *true*, debido a la suposición de igualdad potencial que se hizo en la llamada original. En general, este algoritmo necesitará efectuar suposiciones sucesivas de que los pares de nombres de tipo pueden ser iguales y acumular las suposiciones en una lista. Finalmente, por supuesto, el algoritmo debe obtener éxito o fallar (es decir, no puede entrar en un bucle infinito), porque existe sólo un número finito de nombres de tipo en cualquier programa dado. Dejamos los detalles de las modificaciones al pseudocódigo de *typeEqual* para un ejercicio (véase también la sección de notas y referencias).

Una variación final acerca de la equivalencia de tipos es una versión menor de la equivalencia de nombres utilizada por Pascal y C, denominada **equivalencia de declaración**. En este tipo de método, las declaraciones del tipo siguiente

```
t2 = t1;
```

se interpretan como las que establecen **alias** de tipo, en vez de nuevos tipos (como en la equivalencia de nombre). De este modo, dadas las declaraciones

```
t1 = int;
t2 = int
```

tanto `t1` como `t2` son equivalentes a `int` (es decir, son sólo alias para el nombre del tipo `int`). En esta versión de equivalencia de tipo, todo nombre de tipo es equivalente a algún nombre de tipo base, el cual es, o bien un tipo predefinido, o está dado mediante una expresión de tipo resultante de la aplicación de un constructor de tipo. Por ejemplo, dadas las declaraciones

```
t1 = array [10] of int;
t2 = array [10] of int;
t3 = t1;
```

los nombres de tipo `t1` y `t3` son equivalentes bajo la equivalencia de declaración, pero ninguno es equivalente a `t2`.

Para implementar la equivalencia de declaración, debe proporcionarse una nueva operación *getBaseTypeName* mediante la tabla de símbolos, la cual obtiene el nombre de tipo base en vez de la expresión de tipo asociada. Un nombre tipo puede distinguirse en la tabla de símbolos como un nombre de tipo base si es un tipo predefinido o si está dado por una expresión de tipo que no sea simplemente otro nombre de tipo. Observe que la equivalencia de declaración, similar a la equivalencia de nombre, resuelve el problema de bucles infinitos en la verificación de tipos recursivos, puesto que dos nombres de tipo base sólo pueden ser equivalentes en declaración si son el mismo nombre.

Pascal utiliza de manera uniforme la equivalencia de declaración, mientras que C emplea equivalencia de declaración para estructuras y uniones, pero equivalencia estructural para apuntadores y arreglos.

Ocasionalmente, un lenguaje ofrecerá una selección de equivalencia estructural, de declaración o de nombre, donde se utilizan diferentes declaraciones de tipo para diferentes formas de equivalencia. Por ejemplo, el lenguaje ML permite que los nombres de tipo se declaren como alias utilizando la palabra reservada `type`, por ejemplo en la declaración

```
type RealIntRec = real*int;
```

Esto declara `RealIntRec` como un alias del tipo de producto cartesiano `real*int`. Por otra parte, la declaración `datatype` crea un tipo completamente nuevo, como la declaración

```
datatype intBST = Nil | Node of int*intBST*intBST
```

Advierta que una declaración `datatype` también debe contener nombres de constructor de valores (`Nil` y `Node` en la declaración dada), a diferencia de una declaración `type`. Esto

hace los valores del nuevo tipo distinguibles de los valores de tipos existentes. De este modo, dada la declaración

```
datatype NewRealInt = Prod of real*int;
```

el valor `(2.7,10)` es del tipo `RealIntRec` o `real*int`, mientras que el valor `Prod(2.7,10)` es del tipo `NewRealInt` (pero no `real*int`).

6.4.4 Inferencia de tipo y verificación de tipo

Procederemos ahora con la descripción de un verificador de tipo para un lenguaje simple en términos de acciones semánticas, basadas en una representación de tipos y una operación *typeEqual* como se comentó en sesiones anteriores. El lenguaje que usamos tiene la gramática dada en la figura 6.22, la cual incluye un pequeño subconjunto de las expresiones de tipo de la figura 6.19, con un pequeño número de expresiones y sentencias agregadas. También suponemos la disponibilidad de una tabla de símbolos que contenga nombres de variable y tipos asociados, con operaciones de inserción (*insert*), las cuales insertan un nombre y un tipo en la tabla, y de búsqueda (*lookup*), que devuelve el tipo asociado de un nombre. No especificaremos las propiedades de estas operaciones en sí mismas en la gramática con atributos. Analizamos las reglas de verificación de tipo y la inferencia de tipo para cada clase de construcción de lenguaje de manera individual. La lista completa de acciones semánticas se proporciona en la tabla 6.10 (página 330). Estas acciones no están dadas en forma de gramática con atributos pura, y utilizamos el símbolo := en vez del de igualdad en las reglas de la tabla 6.10 para indicar esto.

Figura 6.22
Una gramática simple para ilustrar la verificación de tipos

programa → *var-decls* **;** *sents*
var-decls → *var-decls* **;** *var-decl* | *var-decl*
var-decl → **id** **:** *type-exp*
type-exp → **int** | **bool** | **array** **[num]** **of** *type-exp*
sents → *sents* **;** *sent* | *sent*
sent → **if** *exp* **then** *sent* | **id** **:=** *exp*

Declaraciones Las declaraciones causan que el tipo de un identificador se introduzca en la tabla de símbolos. De este modo, la regla gramatical

var-decl → **id** **:** *type-exp*

tiene la acción semántica asociada

insert(**id**.*name*, *type-exp.type*)

que inserta un identificador en la tabla de símbolos y asocia un tipo al mismo. El tipo asociado en esta inserción se construye de acuerdo con las reglas gramaticales para *type-exp*.

Tabla 6.10
Gramática con atributos para verificación de tipos de la gramática simple de la figura 6.22

Regla gramatical	Reglas semánticas
var-decl → **id** : *type-exp*	*insert*(**id** .*name*, *type-exp.type*)
type-exp → **int**	*type-exp.type* := *integer*
type-exp → **bool**	*type-exp.type* := *boolean*
type-exp$_1$ → **array** [**num**] **of** *type-exp*$_2$	*type-exp*$_1$.*type* := *makeTypeNode*(*array*, **num** .*size*, *type-exp*$_2$.*type*)
sent → **if** *exp* **then** *sent*	**if not** *typeEqual*(*exp.type*, *boolean*) **then** *type-error*(*sent*)
sent → **id** := *exp*	**if not** *typeEqual*(*lookup*(**id** .*name*), *exp.type*) **then** *type-error*(*sent*)
exp$_1$ → *exp*$_2$ + *exp*$_3$	**if not** (*typeEqual*(*exp*$_2$.*type*, *integer*) **and** *typeEqual*(*exp*$_3$.*type*, *integer*)) **then** *type-error*(*exp*$_1$) ; *exp*$_1$.*type* := *integer*
exp$_1$ → *exp*$_2$ **or** *exp*$_3$	**if not** (*typeEqual*(*exp*$_2$.*type*, *boolean*) **and** *typeEqual*(*exp*$_3$.*type*, *boolean*)) **then** *type-error*(*exp*$_1$) ; *exp*$_1$.*type* := *boolean*
exp$_1$ → *exp*$_2$ [*exp*$_3$]	**if** *isArrayType*(*exp*$_2$.*type*) **and** *typeEqual*(*exp*$_3$.*type*, *integer*) **then** *exp*$_1$.*type* := *exp*$_2$.*type.child1* **else** *type-error*(*exp*$_1$)
exp → **num**	*exp.type* := *integer*
exp → **true**	*exp.type* := *boolean*
exp → **false**	*exp.type* := *boolean*
exp → **id**	*exp.type* := *lookup*(**id** .*name*)

Se supone que los tipos se mantienen como alguna clase de estructura de árbol, de manera que el tipo estructurado **array** en la gramática de la figura 6.22 corresponde a la acción semántica

makeTypeNode(*array*, *size*, *type*)

la cual construye un nodo de tipo

donde el hijo del nodo array (arreglo) es el árbol de tipo dado por el parámetro *type*. Se supone que los tipos simples *integer* y *boolean* se construyen como nodos de hoja estándar en la representación de árbol.

Sentencias Las sentencias no tienen tipos en sí mismas, sino subestructuras que es necesario verificar en cuanto a la exactitud de tipo. Situaciones típicas se muestran mediante las dos

reglas de sentencias en la gramática simple, la sentencia if y la sentencia de asignación. En el caso de la sentencia if, la expresión condicional debe tener tipo booleano. Esto se indica mediante la regla

if not *typeEqual(exp.type, boolean)* **then** *type-error (sent)*

donde *type-error* indica un mecanismo de informe de error cuyo comportamiento se describirá en breve.

En el caso de la sentencia de asignación, el requerimiento es que la variable que se está asignando debe tener el mismo tipo que la expresión cuyo valor está por recibir. Esto depende del algoritmo de equivalencia de tipos, como se expresó mediante la función *typeEqual*.

Expresiones Las expresiones constantes, tales como números, y los valores booleanos `true` y `false`, tienen tipos implícitamente definidos *integer* y *boolean*. Los nombres de variable tienen sus tipos determinados mediante una *búsqueda* en la tabla de símbolos. Otras expresiones se forman mediante operadores, tales como el operador aritmético +, el operador booleano `or` y el operador de subíndice `[]`. En cada caso las subexpresiones deben ser del tipo correcto para la operación indicada. En el caso de los subíndices, esto se indica mediante la regla

if *isArrayType*(exp_2*.type*)
  and *typeEqual*(exp_3*.type, integer*)
then exp_1*.type* := exp_2*.type.child1* **else** *type-error*(exp_1)

Aquí la función *isArrayType* prueba que su parámetro es un tipo arreglo ("array"), es decir, que la representación de árbol del tipo tenga un nodo raíz que represente el constructor de tipo arreglo. El tipo de la expresión de subíndice resultante es el tipo base del arreglo, el cual es el tipo representado por el (primer) hijo "*child*" del nodo raíz en la representación de árbol de un tipo arreglo, y esto se indica escribiendo exp_2*.type.child1*.

El resto describe el comportamiento de un verificador de tipo así en la presencia de errores, como se indica mediante el procedimiento *type-error* en las reglas semánticas de la tabla 6.10. Las cuestiones principales son cuándo generar un mensaje de error y cómo continuar la verificación de tipos en la presencia de errores. *No* se debería generar un mensaje de error cada vez que ocurre un error de tipo; de otro modo, un error simple podría causar que se imprimiera una cascada de muchos errores (algo que también puede pasar con errores de análisis sintáctico). En su lugar, si el procedimiento *type-error* puede determinar que un error de tipo ya debe haber ocurrido en un lugar relacionado, entonces debería suprimirse la generación de un mensaje de error. Esto se puede señalar teniendo un tipo de error interno especial (el cual se puede representar mediante un árbol de tipo nulo). Si *typeError* encuentra este tipo de error en una subestructura, entonces no se genera ningún mensaje de error. Al mismo tiempo, si un error de tipo también significa que no se puede determinar el tipo de una estructura, entonces el verificador de tipo puede usar el tipo de error como su (en realidad desconocido) tipo. Por ejemplo, en las reglas semánticas de la tabla 6.10, dada una expresión con subíndice $exp_1 \rightarrow exp_2$ `[`exp_3`]`, si exp_2 no es un tipo arreglo, entonces no se puede asignar a exp_1 un tipo válido, y no hay asignación de un tipo en las acciones semánticas. Esto supone que los campos de tipo se han inicializado para algún tipo de error. Por otra parte, en el caso de las operaciones + y `or`, incluso en la presencia de un error de tipo, puede hacerse la suposición de que el resultado que es significativo es de tipo entero o booleano, y las reglas de la tabla 6.10 utilizan ese hecho para asignar un tipo al resultado.

6.4.5 Temas adicionales en la verificación de tipos

En esta subsección comentaremos brevemente algunas de las extensiones comunes para los algoritmos de verificación de tipo que hemos analizado.

Sobrecarga Un operador está **sobrecargado** si se utiliza el mismo nombre de operador para dos operaciones diferentes. Un ejemplo común de sobrecarga es el caso de los operadores aritméticos, que con frecuencia representan operaciones sobre valores numéricos diferentes. Por ejemplo, `2+3` representa la suma entera, mientras que `2.1+3.0` representa la suma de punto flotante, que debe ser implementada de manera interna mediante una instrucción o conjunto de instrucciones diferente. Una sobrecarga así se puede extender a procedimientos y funciones definidos por el usuario, donde se puede emplear el mismo nombre para operaciones relacionadas, pero definido para parámetros de tipos diferentes. Por ejemplo, podemos desear definir un procedimiento de valor máximo para valores tanto enteros como reales:

```
procedure max (x,y: integer): integer;
procedure max (x,y: real): real;
```

En Pascal y C esto es ilegal, ya que representa una declaración repetida del mismo nombre dentro del mismo ámbito. Sin embargo, en Ada y C++, una declaración así es legal, porque el verificador de tipo puede decidir cuál procedimiento **max** es significativo con base en los tipos de los parámetros. Un uso así de verificación de tipo para eliminar la ambigüedad de significados múltiples de nombres puede implementarse de varias maneras, dependiendo de las reglas del lenguaje. Una manera sería aumentar el procedimiento de *búsqueda* de la tabla de símbolos con parámetros de tipo, permitiendo que la tabla de símbolos encuentre la coincidencia correcta. Una solución diferente para la tabla de símbolos es mantener conjuntos de tipos posibles para los nombres y devolver el conjunto al verificador de tipo para eliminar la ambigüedad. Esto es útil en situaciones más complejas, donde un tipo único no puede ser determinado de inmediato.

Coerción y conversión de tipos Una extensión común de las reglas de tipo de un lenguaje es permitir expresiones aritméticas de tipo mezclado, tal como `2.1 + 3`, donde se suman un número real y un número entero. En tales casos, debe hallarse un tipo común que sea compatible con todos los tipos de las subexpresiones, y deben aplicarse operaciones para convertir los valores en el tiempo de ejecución a las representaciones apropiadas antes de aplicar el operador. Por ejemplo, en la expresión `2.1 + 3`, el valor entero `3` se debe convertir a punto flotante antes de la suma, y la expresión resultante tendrá el tipo de punto flotante. Existen dos métodos que un lenguaje puede tomar para tales conversiones. Modula-2, por ejemplo, requiere que el programador suministre una función de conversión, de manera que el ejemplo que se acaba de dar debe escribirse como `2.1 + FLOAT(3)`, o se causará un error de tipo. La otra posibilidad (utilizada en el lenguaje C) es que el verificador de tipo suministre la operación de conversión de manera automática, basándose en los tipos de las subexpresiones. Una conversión automática de esta naturaleza se conoce como **coerción**. La coerción se puede expresar de manera implícita mediante el verificador de tipo, ya que el tipo inferido de una expresión cambia a partir de una subexpresión, como en

Esto requiere que un generador de código posterior examine los tipos de expresiones para determinar si se debe aplicar una conversión. De manera alternativa, la conversión se puede suministrar de manera explícita mediante el verificador de tipo insertando un nodo de conversión en el árbol sintáctico, como en

Las coerciones y conversiones de tipo también se aplican a las asignaciones, tal como

```
r = i;
```

en C, en donde, si `r` es de tipo `double` e `i` es `int`, se sujeta el valor de `i` a `double` antes de ser almacenado como el valor de `r`. Tales asignaciones pueden perder información durante la conversión, como podría ocurrir si la asignación se hiciera en la dirección opuesta (también legal en C):

```
i = r;
```

Una situación similar ocurre en los lenguajes orientados a objetos, donde generalmente es permitida la asignación de objetos de subclases a objetos de superclases (con una correspondiente pérdida de información). Por ejemplo, en C++, si `A` es una clase y `B` es una subclase, y si `x` está en `A` y además `y` está en `B`, entonces la asignación `x=y` está permitida, pero no lo contrario. (Esto se conoce como **principio del subtipo**.)

Tipificado polimórfico Un lenguaje es **polimórfico** si permite que las construcciones de lenguaje tengan más de un tipo. Hasta ahora hemos analizado la verificación de tipo para un lenguaje que es esencialmente **monomórfico** debido a que todos los nombres y expresiones que se requieren tienen tipos únicos. Una relajación de este requerimiento de monomorfismo es la sobrecarga. Pero la sobrecarga, aunque es una forma de polimorfismo, se aplica sólo a situaciones donde se realizan varias declaraciones por separado al mismo nombre. Una situación diferente ocurre cuando sólo una declaración se puede aplicar a cualquier tipo. Por ejemplo, un procedimiento que intercambia los valores de dos variables podría en principio aplicarse a variables de cualquier tipo (mientras que tengan el mismo tipo):

```
procedure swap (var x,y:anytype);
```

El tipo de este procedimiento `swap` ("`intercambio`") se dice que está **parametrizado** mediante el tipo *`anytype`* ("*`cualquiertipo`*") y *`anytype`* se considera una **variable de tipo**, la cual puede suponerse de cualquier tipo real. Podemos expresar el tipo de este procedimiento como

procedure(var anytype, var anytype): void

donde cada ocurrencia de *anytype* se refiere al mismo (pero no especificado) tipo. Un tipo así es en realidad un **patrón de tipo** o **esquema de tipo**, más que un tipo real, y un verificador de tipo debe determinar un tipo real en toda situación donde se utilice `swap` que coincida con este patrón de tipo o declarar un error de tipo. Por ejemplo, dado el código

```
var x,y: integer;
    a,b: char;
. . .
swap(x,y);
swap(a,b);
swap(a,x);
```

en la llamada **swap(x,y)**, el procedimiento **swap** está "especializado" desde su patrón de tipo polimórfico dado hasta el tipo (monomórfico)

procedure(var integer, var integer): void

mientras que en la llamada **swap(a,b)** está especializado para el tipo

procedure(var char, var char): void

Por otra parte, en la llamada **swap(a,x)**, el procedimiento **swap** tendría que ser del tipo

procedure(var char, var integer): void

y esto es un tipo que no se puede generar desde el patrón de tipo de **swap** mediante el reemplazo de la variable de tipo *anytype*. Existen algoritmos de verificación de tipo para tal verificación de tipo polimórfico general, de manera más notable en lenguajes funcionales modernos tales como ML, pero involucran sofisticadas técnicas de coincidencia de patrones que no estudiaremos aquí. (Véase la sección de notas y referencias.)

6.5 UN ANALIZADOR SEMÁNTICO PARA EL LENGUAJE TINY

En esta sección desarrollaremos el código para un analizador semántico en el caso del lenguaje TINY, basado en los analizadores sintácticos para TINY que se construyeron en capítulos anteriores. La sintaxis de TINY, así como la estructura del árbol sintáctico en el que basamos el analizador semántico, se describieron en la sección 3.7.

El lenguaje TINY es muy simple en sus requerimientos semánticos estáticos, y el analizador semántico reflejará esa simplicidad. No hay declaraciones explícitas en TINY, y no hay constantes nombradas, tipos de datos o procedimientos: los nombres sólo pueden referirse a variables. Las variables están declaradas implícitamente por el uso, y todas las variables tienen tipo de datos entero. Tampoco hay ámbitos anidados, de manera que un nombre de variable tiene el mismo significado a lo largo de un programa, y la tabla de símbolos no necesita mantener ninguna información referente al ámbito.

La verificación de tipo también es muy simple en TINY. Existen sólo dos tipos simples: entero y booleano. Los únicos valores booleanos son aquellos que resultan de la comparación de dos valores enteros. Como no hay operadores o variables booleanas, un valor booleano sólo puede aparecer en la expresión de prueba de una sentencia if o repeat, y no como el operando de un operador o el valor en una asignación. Finalmente, un valor booleano no puede ser una salida utilizando una sentencia de escritura.

Separaremos el análisis del código para el analizador semántico TINY en dos partes. Primero comentaremos la estructura de la tabla de símbolos y sus operaciones asociadas. Luego describiremos la operación del analizador semántico mismo, incluyendo la construcción de la tabla de símbolos y la verificación de tipo.

6.5.1 Una tabla de símbolos para TINY

En el diseño de la tabla de símbolos para el analizador semántico de TINY, primero debemos determinar qué información necesita mantenerse en la tabla. En casos típicos esta información incluye el tipo de datos y la información de ámbito. Como TINY no tiene información de ámbito, y puesto que todas las variables tienen tipo de datos entero, una tabla de símbolos de TINY no necesita contener esta información. Sin embargo, durante la generación del código, las variables necesitarán ser asignadas en ubicaciones de memoria y, puesto que no hay declaraciones en el árbol sintáctico, la tabla de símbolos es el lugar lógico para almacenar estas ubicaciones. Por ahora, las ubicaciones se pueden visualizar simplemente como índices enteros que se incrementan cada vez que se encuentra una nueva variable. Con el fin de hacer más interesante y más útil la tabla de símbolos, también la utilizamos para generar un listado de referencia cruzada de números de línea donde se tiene acceso a las variables.

Como un ejemplo de la información que se generará mediante la tabla de símbolos, considere el programa muestra de TINY (con números de línea agregados):

```
 1: { Programa simple
 2:    en lenguaje TINY --
 3:    calcula el factorial
 4: }
 5: read x; { introduce un entero }
 6: if 0 < x then { no calcule si x <= 0 }
 7:    fact := 1;
 8:    repeat
 9:      fact := fact * x;
10:      x := x - 1
11:    until x = 0;
12:    write fact { salida del factorial de x }
13: end
```

Después de la generación de la tabla de símbolos para este programa, el analizador semántico ofrecerá como salida (con `TraceAnalyze = TRUE`) la siguiente información al archivo del listado:

```
Symbol table:

Variable Name    Location    Line Numbers
-------------------------------------------------------------
x                0           5    6    9    10    10    11
fact             1           7    9    9    12
```

Advierta que múltiples referencias en la misma línea generan múltiples entradas para esa línea en la tabla de símbolos.

El código para la tabla de símbolos está contenido en los archivos **symtab.h** y **symtab.c** listados en el apéndice B (líneas 1150-1179 y 1200-1321, respectivamente).

La estructura que usamos para la tabla de símbolos es la tabla de dispersión encadenada separadamente como se describió en la sección 6.3.1, y la función de dispersión es la que se dio en la figura 6.13 (página 298). Como no hay información de ámbito, no hay necesidad de una operación de eliminación *delete*, y la operación de inserción *insert* necesita sólo parámetros de ubicación y número de línea, además del identificador. Las otras dos operaciones necesarias son un procedimiento para imprimir la información del resumen como se acaba de mostrar para el archivo del estado, y una operación de búsqueda *lookup* para obtener los números de ubicación de la tabla (necesarios más adelante para el generador de código, y también para el constructor de la tabla de símbolos a fin de verificar si ya se ha visto una variable). Por lo tanto, el archivo de cabecera **symtab.h** contiene las declaraciones siguientes:

```
void st_insert( char * name, int lineno, int loc );
int st_lookup ( char * name );
void printSymTab(FILE * listing);
```

Puesto que sólo existe una tabla de símbolos, su estructura no necesita ser declarada en el archivo de cabecera, ni necesita aparecer como un parámetro para estos procedimientos.

El código de implementación asociado en **symtab.c** utiliza una lista ligada dinámicamente asignada con un nombre de tipo **LineList** (líneas 1236-1239) para almacenar los números de línea asociados con cada registro de identificador en la tabla de dispersión. Los registros de identificador en sí mismos se conservan en una cubeta con nombre de tipo **BucketList** (líneas 1247-1252). El procedimiento **st_insert** (líneas 1262-1295) agrega nuevos registros de identificador al frente de cada cubeta, pero los números de línea se agregan al final de cada lista de números de línea, a fin de mantener el orden numérico. (La eficiencia de **st_insert** se podría mejorar utilizando una lista circular de apuntadores superiores/inferiores para la lista de números de línea; véanse los ejercicios.)

6.5.2 Un analizador semántico para TINY

La semántica estática de TINY comparte con los lenguajes de programación estándar las propiedades de que la tabla de símbolos es un atributo heredado, mientras que el tipo de datos de una expresión es un atributo sintetizado. Así, la tabla de símbolos se puede construir mediante un recorrido preorden del árbol sintáctico, y la verificación de tipos se puede efectuar mediante un recorrido postorden. Mientras que estos dos recorridos se pueden combinar fácilmente en uno solo, para hacer clara la distinción de las operaciones de ambos pasos de procesamiento, los dividimos en dos pasos por separado sobre el árbol sintáctico. De este modo, la interfase del analizador semántico para el resto del compilador, que ubicamos en el archivo **analyze.h** (apéndice B, líneas 1350-1370), se compone de dos procedimientos, dados por las declaraciones

```
void buildSymtab(TreeNode *);
void typeCheck(TreeNode *);
```

El primer procedimiento realiza un recorrido preorden del árbol sintáctico, llamando al procedimiento **st_insert** de la tabla de símbolos a medida que encuentra identificadores de variable en el árbol. Después que se completa el transversal, llama entonces a **printSymTab** para imprimir la información almacenada al archivo del listado. El segundo procedimiento realiza un recorrido postorden del árbol sintáctico, insertando tipos de datos en los nodos del

árbol a medida que se calculan e informando de cualquier error de verificación de tipo al archivo del listado. El código para estos procedimientos y sus procedimientos auxiliares está contenido en el archivo **analyze.c** (apéndice B, líneas 1400-1558).

Para enfatizar las técnicas involucradas de recorrido de un árbol estándar, implementamos tanto **buildSymtab** como **typeCheck** utilizando la misma función de transversal genérica **traverse** (líneas 1420-1441), que toman dos procedimientos (además del árbol sintáctico) como parámetros, uno para realizar el procesamiento preorden de cada nodo y el otro para efectuar el procesamiento postorden:

```
static void traverse( TreeNode * t,
                      void (* preProc) (TreeNode *),
                      void (* postProc) (TreeNode *) )
{ if (t != NULL)
  { preProc(t);
    { int i;
      for (i=0; i < MAXCHILDREN; i++)
        traverse(t->child[i],preProc,postProc);
    }
    postProc(t);
    traverse(t->sibling,preProc,postProc);
  }
}
```

Dado este procedimiento, para obtener un recorrido preorden necesitamos definir un procedimiento que suministre procesamiento preorden y lo pase a **traverse** como **preproc**, mientras pasa un procedimiento "que no hace nada" como **postproc**. En el caso de la tabla de símbolos de TINY, el procesador de preorden se llama **insertNode**, porque realiza inserciones en la tabla de símbolos. El procedimiento "que no hace nada" se denomina **nullProc** y se define con un cuerpo vacío (líneas 1438-1441). Entonces el recorrido preorden que construye la tabla de símbolos se llevará a cabo mediante la llamada simple

```
traverse(syntaxTree,insertNode,nullProc);
```

dentro del procedimiento **buildSymtab** (líneas 1488-1494). De manera similar, el recorrido postorden requerido por **typeCheck** (líneas 1556-1558) se puede obtener mediante la llamada simple

```
traverse(syntaxTree,nullProc,checkNode);
```

donde **checkNode** es un procedimiento definido de manera apropiada que calcula y verifica el tipo en cada nodo. Resta describir el funcionamiento de los procedimientos **insertNode** y **checkNode**.

El procedimiento **insertNode** (líneas 1447-1483) debe determinar cuándo insertar un identificador (junto con su número de línea y ubicación) en la tabla de símbolos, basado en la clase de nodo del árbol sintáctico que recibe a través de su parámetro (un apuntador a un nodo de árbol sintáctico). En el caso de nodos de sentencia, los únicos nodos que contienen referencias de variable son nodos de asignación y nodos de lectura, donde el nombre de variable a asignar o a leer está contenido en el campo **attr.name** del nodo. En el caso de nodos de expresión, los únicos nodos de interés son los nodos de identificador, donde el nombre nuevamente se almacena en **attr.name**. De este modo, en esos tres lugares, el procedimiento **insertNode** contiene una llamada

```
st_insert(t->attr.name,t->lineno,location++);
```

si la variable todavía no se ha contemplado (lo que almacena e incrementa el contador de ubicación además del número de línea) y

```
st_insert(t->attr.name,t->lineno,0);
```

si la variable ya está en la tabla de símbolos (que almacena el número de línea pero no la ubicación).

Finalmente, después que se ha construido la tabla de símbolos, el **buildSymtab** efectúa una llamada a **printSymTab** para escribir la información del número de línea al archivo de listado, bajo el control del marcador **TraceAnalyze** (que está establecido en **main.c**).

El procedimiento **checkNode** *del paso de verificación de tipo tiene dos tareas.* En primer lugar, debe determinar si, basado en los tipos de los nodos hijo, se ha presentado un error de tipo. En segundo lugar, debe inferir un tipo para el nodo actual (si tiene un tipo) y asignar este tipo a un campo nuevo en el nodo del árbol. Éste campo se conoce como campo **type** en **TreeNode** (definido en **globals.h**; véase el apéndice B, línea 216). Como sólo los nodos de expresión tienen tipos, esta inferencia del tipo únicamente se presenta para nodos de expresión. En TINY existen sólo dos tipos, entero y booleano, y estos tipos están definidos en un tipo enumerado en las declaraciones globales (véase el apéndice B, línea 203):

```
typedef enum {Void,Integer,Boolean} ExpType;
```

Aquí el tipo **Void** es un tipo "no-tipo" que se utiliza sólo en la inicialización y para verificación de errores. Cuando se presenta un error, el procedimiento **checkNode** llama al procedimiento **typeError**, el cual imprime un mensaje de error al archivo del listado, basado en el nodo actual.

Resta catalogar las acciones de **checkNode**. En el caso de un nodo de expresión, el nodo es un nodo hoja (una constante o un identificador, de clase **ConstK** o **IdK**) o un nodo de operador (de clase **OpK**). En el caso de un nodo hoja (líneas 1517-1520), el tipo es siempre **Integer** (y no ocurre verificación de tipo). En el caso de un nodo de operador (líneas 1508-1516), las dos subexpresiones hijas deben ser de tipo **Integer** (y sus tipos ya se han calculado, puesto que se está realizando un recorrido postorden). El tipo del nodo **OpK** se determina entonces a partir del operador mismo (sin tener en cuenta si se ha presentado un error de tipo): si el operador es un operador de comparación (< o =), entonces el tipo es **booleano** (**Boolean**), si no lo es, es entero (**Integer**).

En el caso de un nodo de sentencia no se infiere ningún tipo, pero en todos los demás casos aparte de éste debe realizarse alguna verificación de tipo. Éste es el caso de una sentencia **ReadK**, donde la variable que se está leyendo debe ser automáticamente de tipo **Integer**, de modo que no es necesaria la verificación de tipo. Las otras cuatro clases de sentencia necesitan alguna forma de verificación de tipo: las sentencias **IfK** y **RepeatK** necesitan sus expresiones de prueba verificadas para asegurarse que tienen tipo **Boolean** (líneas 1527-1530 y 1539-1542), y las sentencias **WriteK** y **AssignK** necesitan una verificación (líneas 1531-1538) de que la expresión que se está escribiendo o asignando no es booleana (ya que las variables sólo pueden mantener valores enteros, y sólo los valores enteros se pueden escribir):

```
x := 1 < 2; { error - el valor booleano
                    no se puede asignar }
write 1 = 2; { un error también }
```

EJERCICIOS

6.1 Escriba una gramática con atributos para el valor entero de un número dado por la gramática siguiente:

$$\textit{número} \rightarrow \textit{dígito número} \mid \textit{dígito}$$
$$\textit{dígito} \rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}$$

6.2 Escriba una gramática con atributos para el valor de punto flotante de un número decimal dado por la gramática siguiente. (*Sugerencia*: utilice un atributo *conteo* para contar el número de dígitos a la derecha del punto decimal.)

$$\textit{dnum} \rightarrow \textit{num.num}$$
$$\textit{num} \rightarrow \textit{num dígito} \mid \textit{dígito}$$
$$\textit{dígito} \rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}$$

6.3 La gramática para números decimales del ejercicio anterior se puede volver a escribir de manera que sea innecesario un atributo *conteo* (y se eviten las ecuaciones que involucran exponentes). Vuelva a escribir la gramática para hacer esto y proporcione una nueva gramática con atributos para el valor de un *dnum*.

6.4 Considere una gramática de expresión como se escribiría para un analizador sintáctico predictivo con la recursividad por la izquierda eliminada:

$$\textit{exp} \rightarrow \textit{term exp}'$$
$$\textit{exp}' \rightarrow \texttt{+}\ \textit{term exp}' \mid \texttt{-}\ \textit{term exp}' \mid \varepsilon$$
$$\textit{term} \rightarrow \textit{factor term}'$$
$$\textit{term}' \rightarrow \texttt{*}\ \textit{factor term}' \mid \varepsilon$$
$$\textit{factor} \rightarrow \texttt{(}\ \textit{exp}\ \texttt{)} \mid \texttt{número}$$

Escriba una gramática con atributos para el valor de una expresión dada por esta gramática.

6.5 Vuelva a escribir la gramática con atributos de la tabla 6.2 para calcular un atributo de cadena *postfijo* en lugar de *val*, que contenga la forma postfija para la expresión entera simple. Por ejemplo, el atributo *postfijo* para `(34-3)*42` resultaría ser "34 3 – 42 *". Se puede suponer un operador de concatenación || y la existencia de un atributo `número`.*valcadena*.

6.6 Considere la gramática siguiente para árboles binarios enteros (en forma linealizada):

$$\textit{árbol} \rightarrow \texttt{(}\ \texttt{número}\ \textit{árbol árbol}\ \texttt{)} \mid \texttt{nil}$$

Escriba una gramática con atributos para verificar que está ordenado un árbol binario, es decir, que los valores de los números del primer subárbol sean ≤ que el valor del número actual y los valores de todos los números del segundo subárbol sean ≥ al valor del número actual. Por ejemplo `(2 (1 nil nil) (3 nil nil))` está ordenado, pero `(1 (2 nil nil) (3 nil nil))` no lo está.

6.7 Considere la gramática siguiente para declaraciones simples estilo Pascal:

$$\textit{decl} \rightarrow \textit{var-list}\ \texttt{:}\ \textit{type}$$
$$\textit{var-list} \rightarrow \textit{var-list}\ \texttt{,}\ \texttt{id} \mid \texttt{id}$$
$$\textit{type} \rightarrow \texttt{integer} \mid \texttt{real}$$

Escriba una gramática con atributos para el tipo de una variable.

6.8 Considere la gramática del ejercicio 6.7. Vuelva a escribir la gramática de manera que el tipo de una variable se pueda definir como un atributo puramente sintetizado, y proporcione una nueva gramática con atributos para el tipo que tiene esta propiedad.

6.9 Vuelva a escribir la gramática y gramática con atributos del ejemplo 6.4 (página 266) de manera que el valor de un *num-base* se calcule sólo por atributos sintetizados.

6.10 **a.** Dibuje las gráficas de dependencias correspondientes a cada regla gramatical del ejemplo 6.14 (página 283) y para la expresión `5/2/2.0`.

b. Describa los dos pasos que se requieren para calcular los atributos en el árbol sintáctico de `5/2/2.0`, incluyendo un posible orden en el cual se podrían visitar los nodos y los valores de atributo calculados en cada punto.

c. Escriba el pseudocódigo para procedimientos que realizarían los cálculos descritos en el inciso b.

6.11 Dibuje las gráficas de dependencia correspondientes a cada regla gramatical para la gramática con atributos del ejercicio 6.4, y para la cadena `3*(4+5)*6`.

6.12 Dibuje las gráficas de dependencia correspondientes a cada regla gramatical para la gramática con atributos del ejercicio 6.7, y dibuje una gráfica de dependencia para la declaración `x,y,z:real`.

6.13 Considere la siguiente gramática con atributos:

Regla gramatical	Reglas semánticas
$S \rightarrow A\ B\ C$	$B.u = S.u$
	$A.u = B.v + C.v$
	$S.v = A.v$
$A \rightarrow a$	$A.v = 2 * A.u$
$B \rightarrow b$	$B.v = B.u$
$C \rightarrow c$	$C.v = 1$

a. Dibuje el árbol de análisis gramatical para la cadena *abc* (la única cadena en el lenguaje), y dibuje la gráfica de dependencia para los atributos asociados. Describa un orden correcto para la evaluación de los atributos.

b. Suponga que se asigna a $S.u$ el valor 3 antes de comenzar la evaluación de atributo. ¿Cuál es el valor de $S.v$ cuando ha terminado la evaluación?

c. Suponga que las ecuaciones de atributo se modifican como sigue:

Regla gramatical	Reglas semánticas
$S \rightarrow A\ B\ C$	$B.u = S.u$
	$C.u = A.v$
	$A.u = B.v + C.v$
	$S.v = A.v$
$A \rightarrow a$	$A.v = 2 * A.u$
$B \rightarrow b$	$B.v = B.u$
$C \rightarrow c$	$C.v = C.u - 2$

¿Qué valor tiene $S.v$ después de la evaluación de atributo, si $S.u = 3$ antes de comenzar la evaluación?

6.14 Muestre que, dada la gramática con atributos,

Regla gramatical	Reglas semánticas
decl → *type var-list*	*var-list.dtype* = *type.dtype*
type → `int`	*type.dtype* = *integer*
type → `float`	*type.dtype* = *real*
$var\text{-}list_1$ → `id` , $var\text{-}list_2$	`id`.*dtype* = $var\text{-}list_1.dtype$
	$var\text{-}list_2.dtype = var\text{-}list_1.dtype$
var-list → `id`	`id`.*dtype* = *var-list.dtype*

si el atributo *type.dtype* se mantiene en la pila de valores durante un análisis sintáctico LR, entonces este valor no se puede encontrar en una posición fija en la pila cuando ocurren reducciones a una *var-list*.

6.15 **a.** Demuestre que la gramática $B \rightarrow B\ b \mid a$ es SLR(1), pero que la gramática

$$B \rightarrow A\ B\ b \mid a$$
$$A \rightarrow \varepsilon$$

(construida a partir de la gramática anterior al agregar una producción ε simple) no es LR(k) para ninguna k.

b. Dada la gramática del inciso a (con la producción ε), ¿qué cadenas aceptarían un analizador sintáctico generado por Yacc?

c. ¿Es probable que esta situación se presente durante el análisis semántico de un lenguaje de programación "real"? Justifique su respuesta.

6.16 Vuelva a escribir la gramática de expresión de la sección 6.3.5 en una gramática no ambigua, de manera tal que las expresiones escritas en esa sección sigan siendo legales, y vuelva a escribir la gramática con atributos de la tabla 6.9 (página 311) para esta nueva gramática.

6.17 Vuelva a escribir la gramática con atributos de la tabla 6.9 para utilizar declaraciones colaterales en lugar de declaraciones secuenciales. (Véase la página 307.)

6.18 Escriba una gramática con atributos que calcule el valor de cada expresión para la gramática de expresión de la sección 6.3.5.

6.19 Modifique el pseudocódigo para la función *typeEqual* de la figura 6.20 (página 325) a fin de incorporar nombres de tipo y el algoritmo sugerido para determinar la equivalencia estructural de tipos recursivos como se describió en la página 327.

6.20 Considere la siguiente gramática (ambigua) de expresiones:

$$exp \rightarrow exp + exp \mid exp - exp \mid exp * exp \mid exp\ /\ exp$$
$$\mid (\ exp\ ) \mid \texttt{num} \mid \texttt{num.num}$$

Suponga que se siguen las reglas de C al calcular el valor de cualquier expresión así: si dos subexpresiones son de tipo mixto, entonces la subexpresión en modo entero se convierte a modo de punto flotante, y se aplica el operador de punto flotante. Escriba una gramática con atributos que convertirá a tales expresiones en expresiones legales en Modula-2: las conversiones de números enteros a números de punto flotante se expresan aplicando la función `FLOAT`, y el operador de división `/` se considera como `div` si sus operandos son enteros.

6.21 Considere la siguiente extensión de la gramática de la figura 6.22 (página 329) para incluir llamadas y declaraciones de funciones:

programa → *var-decls* **;** *fun-decls* **;** *sents*
var-decls → *var-decls* **;** *var-decl* | *var-decl*
var-decl → **id** **:** *type-exp*
type-exp → **int** | **bool** | **array** **[num]** **of** *type-exp*
fun-decls → **fun** **id** **(** *var-decls* **)** **:** *type-exp* **;** *cuerpo*
cuerpo → *exp*
sents → *sents* **;** *sent* | *sent*
sent → **if** *exp* **then** *sent* | **id** **:=** *exp*
exp → *exp* **+** *exp* | *exp* **or** *exp* | *exp* **[** *exp* **]** | **id** **(** *exps* **)**
| **num** | **true** | **false** | **id**
exps → *exps* **,** *exp* | *exp*

a. Diseñe una estructura adecuada de árbol para las nuevas estructuras de tipo de función, y escriba una función *typeEqual* para dos tipos de función.

b. Escriba reglas semánticas para la verificación de tipo de declaraciones de función y llamadas de función (representada por la regla *exp* → **id** **(** *exps* **)**) similares a las reglas de la tabla 6.10 (página 330).

6.22 Considere la siguiente ambigüedad en expresiones en C. Dada la expresión

```
(A)-x
```

si **x** es una variable entera, y **A** está definida en una **typedef** como equivalente a **double**, entonces esta expresión asigna el valor de **-x** a **double**. Por otra parte, si **A** es una variable entera, entonces esto calcula la diferencia en valor entero de las dos variables.

a. Describa cómo puede utilizar el analizador sintáctico la tabla de símbolos para eliminar la ambigüedad de estas dos interpretaciones.

b. Describa cómo puede emplear el analizador léxico la tabla de símbolos para eliminar la ambigüedad de estas dos interpretaciones.

6.23 Escriba una gramática con atributos correspondiente a las restricciones de tipo impuestas por el verificador de tipos de TINY.

6.24 Escriba una gramática con atributos para la construcción de la tabla de símbolos del analizador semántico de TINY.

EJERCICIOS DE PROGRAMACIÓN

6.25 Escriba declaraciones en C para un árbol sintáctico en el caso de los números base del ejemplo 6.4 (página 266), y utilícelas para traducir el pseudocódigo *EvalWithBase* del ejemplo 6.13 (página 282) en código C.

6.26 **a.** Vuelva a escribir el evaluador descendente recursivo de la figura 4.1 (página 148) de manera que imprima la traducción postfija en lugar del valor de una expresión (véase el ejercicio 6.5).

b. Vuelva a escribir el evaluador descendente recursivo de manera que imprima *tanto* el valor como la traducción postfija.

6.27 **a.** Vuelva a escribir la especificación Yacc de la figura 5.10 (página 228) para una calculadora de enteros simple de manera que imprima una traducción postfija en lugar del valor (véase el ejercicio 6.5).

b. Vuelva a escribir la especificación Yacc de manera que imprima *tanto* el valor como la traducción postfija.

6.28 Escriba una especificación Yacc que imprimirá la traducción en Modula-2 de una expresión dada para la gramática del ejercicio 6.20.

6.29 Escriba una especificación Yacc para un programa que calculará el valor de una expresión con bloques *let* (tabla 6.9, página 311). (Se pueden abreviar los tokens `let` e `in` a caracteres simples y también restringir identificadores o usar Lex para generar un analizador léxico adecuado.)

6.30 Escriba una especificación Yacc y Lex de un programa para verificar tipos del lenguaje cuya gramática está dada en la figura 6.22 (página 329).

6.31 Vuelva a escribir la implementación de la tabla de símbolos para el analizador semántico de TINY con el fin de agregar un apuntador inferior a la estructura de datos `LineList` y mejorar la eficiencia de la operación `insert`.

6.32 Vuelva a escribir el analizador semántico de TINY de manera que sólo realice un paso sobre el árbol sintáctico.

6.33 El analizador semántico de TINY no intenta asegurar que se ha asignado una variable antes de utilizarse. De este modo, el siguiente código TINY se considera semánticamente correcto:

```
y := 2+x;
x := 3;
```

Vuelva a escribir el analizador TINY de manera que haga verificaciones "razonables" de que ha ocurrido una asignación a una variable antes de utilizar esa variable en una expresión. ¿Qué impide a tales verificaciones ser infalibles?

6.34 **a.** Vuelva a escribir el analizador semántico de TINY para permitir que se almacenen valores booleanos en variables. Esto requerirá que se dé a las variables tipos de datos booleano o entero en la tabla de símbolos y que la verificación de tipo incluya una búsqueda en la tabla de símbolos. La exactitud del tipo de un programa TINY ahora debe incluir el requerimiento de que todas las asignaciones a (y usos de) una variable sean consistentes con sus tipos de datos.

b. Escriba una gramática con atributos para el verificador de tipo TINY como fue modificado en el inciso a.

NOTAS Y REFERENCIAS

El trabajo inicial más importante acerca de gramática con atributos es el de Knuth [1968]. Importantes estudios del uso de las gramáticas con atributos en la construcción de compiladores aparecen en Lorho [1984]. El uso de gramáticas con atributos para especificar formalmente la semántica de los lenguajes de programación se estudia en Slonneger y Kurtz [1995], donde se proporciona una gramática con atributos completa para la semántica estática de un lenguaje similar a TINY. Más propiedades matemáticas de las gramáticas con atributos se pueden encontrar en Mayoh [1981]. Una prueba para la no circularidad se puede hallar en Jazayeri, Ogden y Rounds [1975].

La cuestión de la evaluación de atributos durante el análisis sintáctico se estudia con algo más de detalle en Fischer y LeBlanc [1991]. Las condiciones que aseguran que esto se puede hacer durante un análisis sintáctico LR aparecen en Jones [1980]. La cuestión de mantener análisis sintáctico LR determinístico mientras se agregan nuevas producciones ε para programar acciones (como en Yacc) se estudia en Purdom y Brown [1980].

Las estructuras de datos para la implementación de la tabla de símbolos, incluyendo tablas de dispersión y análisis de sus eficiencias, se pueden encontrar en muchos textos; véase, por ejemplo, Aho, Hopcroft y Ullman [1983] o Cormen, Leiserson y Rivest [1990]. Un cuidadoso estudio de la selección de una función de dispersión aparece en Knuth [1973].

Los sistemas de tipos, exactitud de tipos e inferencia de tipos forman uno de los campos principales del estudio de la ciencia computacional teórica, con aplicaciones a muchos

lenguajes. Para una perspectiva general tradicional, véase Louden [1993]. Para una perspectiva más profunda, véase Cardelli y Wegner [1985]. C y Ada utilizan formas mixtas de equivalencia de tipos similar a la equivalencia de declaración de Pascal, que son difíciles de describir de manera concisa. Lenguajes más antiguos, tales como FORTRAN77, Algol60 y Algol68 utilizan equivalencia estructural. Los lenguajes funcionales modernos como ML y Haskell utilizan equivalencia de nombre estricta, con sinónimos de tipo tomando el lugar de la equivalencia estructural. El algoritmo para equivalencia estructural descrito en la sección 6.4.3 se puede encontrar en Koster [1969]; una moderna aplicación de un algoritmo similar se encuentra en Amadio y Cardelli [1993]. Los sistemas de tipo polimórficos y algoritmos de inferencia de tipos se describen en Peyton Jones [1987] y Reade [1989]. La versión de la inferencia de tipos polimórfica utilizada en ML y Haskell se conoce como **inferencia de tipos Hindley-Milner** [Hindley, 1969; Milner, 1978].

En este capítulo no describimos todas las herramientas para la construcción automática de evaluadores de atributos, puesto que ninguna es de uso general (a diferencia de los generadores de analizadores léxicos y analizadores sintácticos Lex y Yacc). Algunas herramientas interesantes basadas en gramáticas con atributos son LINGUIST [Farrow, 1984] y GAG [Kastens, Hutt y Zimmermann, 1982]. El Synthesizer Generator ("Generador Sintetizador") [Reps y Teitelbaum, 1989] es una herramienta madura para la generación de editores sensibles al contexto basados en gramáticas con atributos. Una de las cosas interesantes que se pueden hacer con esta herramienta es construir un editor de lenguaje que suministra de manera automática declaraciones de variable basadas en el uso.

Capítulo 7

Ambientes de ejecución

7.1 Organización de memoria durante la ejecución del programa
7.2 Ambientes de ejecución completamente estáticos
7.3 Ambientes de ejecución basados en pila
7.4 Memoria dinámica
7.5 Mecanismos de paso de parámetros
7.6 Un ambiente de ejecución para el lenguaje TINY

En capítulos anteriores estudiamos las fases de un compilador que realiza análisis estático del lenguaje fuente. Esto incluyó análisis léxico, análisis sintáctico y análisis semántico estático. Este análisis sólo depende de las propiedades del lenguaje fuente: es independiente por completo del lenguaje objetivo (ensamblador o de máquina) y de las propiedades de la máquina objetivo y su sistema operativo.

En este capítulo y el siguiente volveremos a la tarea de estudiar cómo un compilador genera código ejecutable. Esto puede involucrar análisis adicional, como el que realiza un optimizador, y parte de éste puede ser independiente de la máquina. Pero gran parte de la tarea de la generación de código depende de los detalles de la máquina objetivo. No obstante, las características generales de la generación de código permanecen iguales a través de una amplia variedad de arquitecturas. Esto es particularmente cierto para el **ambiente en tiempo de ejecución** o **ambiente de ejecución**, el cual es la estructura de la memoria y los registros de la computadora objetivo con los que se administra la memoria y se mantiene la información necesaria para guiar el proceso de la ejecución. De hecho, casi todos los lenguajes de programación utilizan una de tres clases de ambiente en tiempo de ejecución, cuya estructura esencial no depende de los detalles específicos de la máquina objetivo. Estas tres clases de ambientes son: el **ambiente completamente estático** característico de FORTRAN77; el **ambiente basado en pila** de lenguajes como C, C++, Pascal y Ada; y el **ambiente completamente dinámico** de lenguajes funcionales como LISP. También pueden utilizar híbridos a partir de todos los anteriores.

En este capítulo comentaremos cada una de estas clases de ambientes por turno, junto con las características del lenguaje que determinan qué ambientes son factibles, y cuáles deben ser sus propiedades. Esto incluye cuestiones de ámbito y asignación, la naturaleza de llamadas a procedimientos, y las variedades de mecanismos para paso de parámetros. Aquí, el enfoque se centrará en la estructura general del ambiente y en el siguiente capítulo, en el código real que es necesario generar para mantener el ambiente. En este aspecto es importante tomar en cuenta que un compilador puede mantener un

ambiente sólo de manera indirecta, ya que debe generar código para realizar las operaciones de mantenimiento necesarias durante la ejecución del programa. En contraste, un intérprete tiene una tarea más fácil, puesto que puede mantener el ambiente directamente dentro de sus propias estructuras de datos.

La primera sección de este capítulo contiene un análisis de las características generales de todos los ambientes en tiempo de ejecución y sus relaciones con la arquitectura de la máquina objetivo. Las siguientes dos secciones analizan los ambientes estático y basado en pila, junto con ejemplos de su funcionamiento durante la ejecución. Como el ambiente basado en pila es el más común, se proporciona algo de detalle acerca de las diferentes variedades y la estructura de un sistema basado en pila. Una sección posterior analiza cuestiones de memoria dinámica, incluyendo ambientes completamente dinámicos y ambientes orientados a objetos. A continuación se tiene un análisis del efecto de diversas técnicas de paso de parámetros sobre el funcionamiento de un ambiente. El capítulo se cierra con una breve descripción del ambiente simple necesario para implementar el lenguaje TINY.

7.1 ORGANIZACIÓN DE MEMORIA DURANTE LA EJECUCIÓN DEL PROGRAMA

La memoria de una computadora típica está dividida en un área de registro y una memoria de acceso aleatorio (RAM), más lenta, que se puede abordar directamente. El área de RAM también puede dividirse en un área de código y un área de datos. En la mayoría de lenguajes compilados no es posible efectuar cambios al área de código durante la ejecución, y el área de código y de datos pueden visualizarse como separadas de manera conceptual. Además, como el área de código se fija antes de la ejecución, todas las direcciones de código se pueden calcular en tiempo de compilación, y el área de código se puede contemplar de la manera siguiente:

Código en memoria

En particular, el punto de entrada de cada procedimiento y función se conoce en tiempo de compilación.[1] No se puede decir lo mismo de la asignación de datos, ya que solamente una pequeña parte de la misma puede asignarse en ubicaciones fijas en memoria antes de que comience la ejecución. Gran parte del resto del capítulo se dedica a cómo manejar la asignación de datos que no es fija, o bien, que es dinámica.

1. Lo más probable es que el código se carga mediante un cargador en un área en memoria que está asignada al principio de la ejecución y, de este modo, no es absolutamente predecible. Sin embargo, todas las direcciones reales son entonces calculadas de manera automática mediante desplazamiento desde una dirección de carga base fija, de forma que el principio de direcciones fijas permanece igual. En ocasiones, el escritor del compilador debe tener cuidado de generar el denominado **código relocalizable**, en el cual los saltos, llamadas y referencias son todas realizadas en relación con alguna **base**, por lo regular un registro. Ejemplos de esto se proporcionarán en el capítulo siguiente.

Existe una clase de datos que pueden ser fijados en memoria antes de la ejecución y que comprenden los datos globales y/o estáticos de un programa. (En FORTRAN77, a diferencia de la mayoría de los lenguajes, todos los datos son de esta clase.) Tales datos por lo regular se asignan de manera separada en un área fija de manera similar al código. En Pascal, las variables globales se encuentran en esta clase, lo mismo que las variables estáticas y externas de C.

Una cuestión que surge en la organización del área global/estática involucra constantes que son conocidas en tiempo de compilación. Éstas incluyen las declaraciones `const` de C y Pascal, además de valores de literales utilizadas en el código mismo, tal como la cadena `"Hello%d\n"` y el valor entero 12345 en la sentencia C

```
printf("Hello %d\n",12345);
```

Las constantes pequeñas de tiempo de compilación, tales como 0 y 1, por lo regular se insertan directamente en el código por medio del compilador y no se les asigna ningún espacio de datos. Tampoco se necesita asignar espacio en el área de datos global para procedimientos o funciones globales, puesto que el compilador conoce sus puntos de entrada y también se pueden insertar directamente en el código. Sin embargo, a los valores enteros grandes, los valores de punto flotante y, en particular, las literales de cadena se les asigna por lo regular memoria en el área global/estática, se almacenan una vez en el inicio y posteriormente se buscan en esas ubicaciones mediante el código de ejecución. (En realidad, en las literales de cadena C son visualizados como apuntadores, de modo que deben ser almacenadas de esta manera.)

El área de memoria utilizada para la asignación de datos dinámicos puede ser organizada de muchas maneras diferentes. Una organización típica divide esta memoria en un área de **pila** y un área de **apilamiento** o **montículo (heap)**, donde el área de pila es utilizada para los datos cuya asignación se presenta en modo LIFO (por las siglas del término en inglés "último en entrar, primero en salir", es decir, "last-in, first-out") y el área de apilamiento es empleada para la asignación dinámica que no cumple con un protocolo LIFO (la asignación de apuntadores en C, por ejemplo).[2] A menudo la arquitectura de la máquina objetivo incluirá una pila de procesador, y al usar esta pila permitirá emplear soporte de procesador para llamadas de procedimiento y retornos (el mecanismo principal que utiliza asignación de memoria basada en pila). En ocasiones, un compilador tendrá que hacer preparativos para la asignación explícita de la pila del procesador en un lugar apropiado en la memoria.

Una organización general de almacenamiento en tiempo de ejecución que tiene todas las categorías de memoria descritas puede tener el aspecto que se presenta a continuación:

2. Debería observarse que el apilamiento es por lo regular una simple área de memoria lineal. Se denomina apilamiento o montículo (heap) por razones históricas, y no está relacionado con la estructura de datos de pilas utilizada en algoritmos tales como el de ordenamiento de pilas.

Las flechas en esta imagen indican la dirección de crecimiento de la pila y el apilamiento (heap). Tradicionalmente, la pila se representa incrementándose hacia abajo en la memoria, de manera que su tope o parte superior en realidad está en la parte inferior de su área ilustrada. El apilamiento también se representa de manera similar a la pila, pero no es una estructura LIFO, y su incremento y decremento son más complicados de lo que indica la flecha (véase la sección 7.4). En algunas organizaciones se asigna a la pila y al apilamiento secciones separadas de memoria, en vez de ocupar la misma área.

Una unidad importante de la asignación de memoria es el **registro de activación de procedimiento**, el cual contiene la memoria asignada a los datos locales de un procedimiento o función cuando son llamados o activados. Un registro de activación, como mínimo, debe contener las secciones siguientes:

espacio para argumentos (parámetros)
espacio para información de administración, incluyendo direcciones de retorno
espacio para datos locales
espacio para temporales locales

Aquí hacemos énfasis (y lo haremos de manera repetida en las secciones siguientes), en que esta imagen sólo ilustra la organización general de un registro de activación. Los detalles específicos, incluyendo el orden de los datos que contiene, dependerán de la arquitectura de la máquina objetivo, de las propiedades del lenguaje que se está compilando, e incluso de las preferencias del escritor del compilador.

Algunas partes de un registro de activación tienen el mismo tamaño para todos los procedimientos; el espacio para la información de administración, por ejemplo. Otras partes, como el espacio para los argumentos y datos locales, pueden permanecer fijas para cada procedimiento individual, pero variarán entre un procedimiento y otro. Algunas partes del registro de activación también pueden ser asignadas de manera automática mediante el procesador o llamadas de procedimiento (el almacenamiento de la dirección de retorno, por ejemplo). Otras partes (como el espacio temporal local) pueden necesitar ser asignadas de manera explícita mediante instrucciones generadas por el compilador. Dependiendo del lenguaje, los registros de activación pueden ser asignados en el área estática (FORTRAN77), el área de la pila (C, Pascal) o el área de apilamiento (LISP). Cuando los registros de activación se conservan en la pila, en ocasiones se hace referencia a ellos como **marcos de pila**.

Los registros de procesador también forman parte de la estructura del ambiente de ejecución. Los registros pueden ser utilizados para almacenar elementos temporales, variables locales o incluso variables globales. Cuando un procesador tiene muchos registros, como es el caso en los procesadores RISC más recientes, toda el área estática y los registros de activación completos pueden ser conservados por completo en registros. Los procesadores también tienen registros de propósito específico para mantenerse al tanto de la ejecución, tales como el **contador de programa (pc, program counter)** y el **apuntador de pila (sp, stack pointer)** en la mayoría de las arquitecturas. También puede haber registros específicamente diseñados para mantenerse al tanto de las activaciones de procedimiento. Algunos registros típicos de esta naturaleza son el **apuntador de marco (fp, frame pointer)**, que

apunta al registro de activación actual, y el **apuntador de argumento (ap, argument pointer)**, que apunta al área del registro de activación reservado para argumentos (valores de parámetro).[3]

Una parte particularmente importante del diseño de un ambiente de ejecución es la determinación de la secuencia de operaciones que deben ocurrir cuando se llama a un procedimiento o función. Tales operaciones pueden incluir la asignación de memoria para el registro de activación, el cálculo y almacenamiento de los argumentos, y el almacenamiento y establecimiento de los registros necesarios para efectuar la llamada. Estas operaciones por lo regular se conocen como **secuencia de llamada**. Las operaciones adicionales necesarias cuando un procedimiento, o una función, es devuelto, tal como la ubicación del valor de retorno al que se puede tener acceso mediante el elemento que llama, el reajuste de los registros y la posible liberación de la memoria del registro de activación, por lo general también se consideran parte de la secuencia de llamada. Si es necesario, haremos referencia a la parte de la secuencia de llamada que se realiza durante una llamada como la **secuencia de llamada** y la parte que es efectuada al regreso como la **secuencia de retorno**.

Los aspectos importantes del diseño de la secuencia de llamada son 1) cómo dividir las operaciones de la secuencia de llamada entre el elemento que llama y el elemento llamado (es decir, cuánto código de la secuencia de llamada colocar en el punto de llamada y cuánto colocar al principio del código de cada procedimiento) y 2) hasta qué punto depender del soporte del procesador de llamadas en vez de generar código explícito para cada paso de la secuencia de llamada. El punto 1) es un asunto particularmente espinoso, puesto que por lo regular es más fácil generar el código de la secuencia de llamada en el punto de llamada en vez de hacerlo desde el interior del elemento llamado, pero al hacerlo así se provoca que crezca el tamaño del código generado, puesto que el mismo código debe ser duplicado en cada sitio de llamada. Más adelante se tratarán estas cuestiones con más detalle.

Como mínimo, el elemento que llama es responsable de calcular los argumentos y colocarlos en ubicaciones donde puedan ser encontrados por el elemento llamado (quizás directamente en el registro de activación del elemento llamado). Además, el estado de la máquina en el punto de llamada, incluyendo la dirección de retorno y, posiblemente, los registros que se encuentran en uso, deben ser guardados, ya sea por el elemento que llama o por el elemento llamado, o de manera parcial por ambos. Finalmente, también debe ser establecida cualquier información adicional de administración, de nueva cuenta, quizá de alguna manera en la que intervengan tanto el elemento que llama como el elemento llamado.

7.2 AMBIENTES DE EJECUCIÓN COMPLETAMENTE ESTÁTICOS

La clase más simple de un ambiente de ejecución es aquella en la cual todos los datos son estáticos y permanecen fijos en la memoria mientras dure la ejecución del programa. Un ambiente de esta clase puede utilizarse para implementar un lenguaje en el cual no hay apuntadores o asignación dinámica, y en el cual los procedimientos no puedan ser llamados de manera recursiva. El ejemplo estándar de un lenguaje así es FORTRAN77.

En un ambiente completamente estático no sólo las variables globales, sino *todas* las variables, son asignadas de manera estática. Así, cada procedimiento tiene sólo un registro de activación único, que es asignado estáticamente antes de la ejecución. Se puede tener acceso directamente a todas las variables, ya sean locales o globales, mediante direcciones fijas, y la memoria completa del programa se puede visualizar como sigue:

3. Estos nombres son tomados de la arquitectura VAX, pero nombres similares se presentan en otras arquitecturas.

En un ambiente así hay relativamente poco gasto general en términos de administración de información a mantener en cada registro de activación, y no es necesario mantener información extra acerca del ambiente (aparte de, tal vez, la dirección de retorno) en un registro de activación. La secuencia de llamada para un ambiente de esta naturaleza también es particularmente simple. Cuando se llama a un procedimiento, cada argumento es calculado y almacenado en su ubicación de parámetro apropiada en la activación del procedimiento que se está llamando. Entonces la dirección de retorno en el código del elemento que llama se graba, y se efectúa un salto al inicio del código del procedimiento llamado. Al regreso, se hace un simple salto a la dirección de retorno.[4]

Ejemplo 7.1

Como un ejemplo concreto de esta clase de ambiente, considere el programa en FORTRAN77 de la figura 7.1. Este programa tiene un procedimiento principal y un solo procedimiento adicional **QUADMEAN**.[5] Existe una sola variable global dada por la declaración **COMMON MAXSIZE** tanto en el procedimiento principal como en **QUADMEAN**.[6]

---

4. En la mayoría de las arquitecturas, un salto de subrutina guarda automáticamente la dirección de retorno; esta dirección también se recarga de manera automática cuando se ejecuta una instrucción de retorno.

5. Ignoramos la función de biblioteca **SQRT**, la cual es llamada por **QUADMEAN** y está vinculada antes de la ejecución.

6. De hecho, FORTRAN77 permite que variables COMMON tengan nombres diferentes en procedimientos diferentes, mientras que todavía hacen referencia a la misma ubicación de memoria. Desde ahora en este ejemplo ignoraremos discretamente tales complejidades.

Figura 7.1
Un programa de muestra de FORTRAN77

```
      PROGRAM TEST
      COMMON MAXSIZE
      INTEGER MAXSIZE
      REAL TABLE(10),TEMP
      MAXSIZE = 10
      READ *, TABLE(1),TABLE(2),TABLE(3)
      CALL QUADMEAN(TABLE,3,TEMP)
      PRINT *, TEMP
      END

      SUBROUTINE QUADMEAN(A,SIZE,QMEAN)
      COMMON MAXSIZE
      INTEGER MAXSIZE,SIZE
      REAL A(SIZE),QMEAN, TEMP
      INTEGER K
      TEMP = 0.0
      IF ((SIZE.GT.MAXSIZE).OR.(SIZE.LT.1)) GOTO 99
      DO 10 K = 1,SIZE
         TEMP = TEMP + A(K)*A(K)
10    CONTINUE
99    QMEAN = SQRT(TEMP/SIZE)
      RETURN
      END
```

Ignorando la posible diferencia en tamaño entre los valores enteros y de punto flotante en memoria, ilustramos un ambiente de ejecución para este programa en la figura 7.2 (página 352).[7] En esta imagen indicamos con flechas los valores que los parámetros **A**, **SIZE** y **QMEAN** del procedimiento **QUADMEAN** tienen durante la llamada desde el procedimiento principal. En FORTRAN77, los valores de parámetro son implícitamente referencias a memoria, de modo que las ubicaciones de los argumentos de la llamada (**TABLE**, 3 y **TEMP**) se copian en las ubicaciones de parámetro de **QUADMEAN**. Lo anterior tiene varias consecuencias. La primera es que, se requiere otra derreferenciación para tener acceso a valores de parámetro. La segunda es que los parámetros de arreglo no necesitan ser reubicados y copiados (así, se asigna al parámetro de arreglo **A** en **QUADMEAN** sólo un espacio, que apunta a la ubicación base de **TABLE** durante la llamada). La tercera es que los argumentos constantes, como el valor 3 en la llamada, deben almacenarse en una localidad de memoria y ésta debe utilizarse durante la llamada. (Los mecanismos de paso de parámetro se comentan más ampliamente en la sección 7.5.)

Otra característica de la figura 7.2 que requiere ser explicada es la ubicación sin nombre que se asignó al final del registro de activación de **QUADMEAN**. Esta ubicación es una localidad "improvisada" que se utiliza para almacenar valores temporales durante el cálculo de expresiones aritméticas. En **QUADMEAN** existen dos cálculos donde esto puede

7. Nuevamente enfatizamos que los detalles de esta figura son sólo ilustrativos. Las implementaciones reales pueden diferir en forma sustancial de las proporcionadas aquí.

ser necesario. El primero es el cálculo de **TEMP + A(K) * A(K)** en el ciclo, y el segundo es el cálculo de **TEMP/SIZE** como el parámetro en la llamada a **SQRT**. Ya comentamos la necesidad de asignar espacio a valores de parámetro (aunque en una llamada a una función de biblioteca la convención puede ser, de hecho, diferente). La razón de que una localidad de memoria temporal también pueda ser necesaria para el cálculo del ciclo es que cada operación aritmética debe aplicarse en un solo paso, de manera que **A(K) * A(K)** se calcula y luego se suma al valor de **TEMP** en el siguiente paso. Si no hay suficientes registros disponibles para mantener este valor temporal, o si se hace una llamada requiriendo que este valor sea guardado, entonces el valor será almacenado en el registro de activación antes de la culminación del cálculo. Un compilador siempre puede predecir si esto será necesario durante la ejecución, y hacer preparativos para la asignación del número (y tamaño) apropiado de las ubicaciones temporales.

Figura 7.2
Un ambiente de ejecución para el programa de la figura 7.1

§

7.3 AMBIENTES DE EJECUCIÓN BASADOS EN PILA

En un lenguaje en el que se permitan las llamadas recursivas, y en el cual las variables locales sean nuevamente asignadas en cada llamada, los registros de activación no pueden ser asignados de manera estática. En vez de esto, los registros de activación deben ser asignados de manera basada en pila, en la cual cada nuevo registro de activación es asignado en la parte superior de la pila a medida que se hace una nueva llamada de procedimiento (una **inserción** del registro activación) y desasignado cuando la llamada termina (una **extracción** del registro de activación). La **pila de registros de activación** (también conocida como **pila de ejecución** o **pila de llamadas**) entonces se incrementa y decrementa con la cadena de llamadas que se han presentado en el programa en ejecución. Cada procedimiento puede tener varios registros de activación diferentes a la vez en la pila de llamadas, y cada una representará una llamada distinta. Un ambiente de esta clase requiere una estrategia más compleja para su administración y acceso a variables que un ambiente completamente estático. En particular, la información de administración adicional debe mantenerse en los registros de activación, y la secuencia de llamada también debe incluir los pasos

necesarios para establecerla y mantenerla. La exactitud de un ambiente basado en pila, y la cantidad de información de administración requerida, depende fuertemente de las propiedades de lenguaje que se está compilando. En esta sección consideraremos la organización de los ambientes basados en pila en un orden de complejidad creciente, clasificados por las propiedades de lenguaje involucrado.

7.3.1 Ambientes basados en pila sin procedimientos locales

En un lenguaje donde los procedimientos son globales (como el lenguaje C), un ambiente basado en pila requiere de dos cosas: el mantenimiento de un apuntador hacia el registro de activación actual para permitir acceso a variables locales y un registro de la posición o tamaño del registro de activación inmediatamente precedente (el registro de activación del elemento que llama) con el fin de permitir que el registro de activación sea recuperado (y la activación actual sea descartada) cuando la llamada actual finalice. El apuntador a la activación actual por lo regular se conoce como **apuntador de marco**, o **fp**, por las siglas del término en inglés, es decir, **frame pointer**, y generalmente se mantiene en un registro (a menudo también conocido como fp). La información acerca de la activación anterior por lo común se conserva en la activación actual como un apuntador hacia el registro de activación previo al que se conoce como **vínculo de control** o **vínculo dinámico** (*dinámico*, porque apunta al registro de activación del elemento que llama durante la ejecución). En ocasiones este apuntador es denominado **fp antiguo**, porque representa el valor anterior del fp. Por lo general este apuntador se conserva en algún lugar a la mitad de la pila, entre el área de parámetro y el área de variable local, y apunta a la ubicación del vínculo de control del registro de activación previo. Adicionalmente, puede haber un **apuntador de pila**, o **sp** (por las siglas del término en inglés, es decir, **stack pointer**), que siempre apunta a la última ubicación asignada en la pila de llamada (la cual a menudo se denomina **tope del apuntador de pila**, o **tos**, por las siglas del término en inglés, es decir, top of stack pointer).

Veamos algunos ejemplos.

Ejemplo 7.2

Considere la implementación recursiva simple del algoritmo de Euclides para calcular el máximo común divisor de dos enteros no negativos, cuyo código (en C) se proporciona en la figura 7.3.

Figura 7.3
Código en C para el ejemplo 7.2

```
#include <stdio.h>

int x,y;

int gcd( int u, int v)
{ if (v == 0) return u;
  else return gcd(v,u % v);
}

main()
{ scanf("%d%d",&x,&y);
  printf("%d\n",gcd(x,y));
  return 0;
}
```

Suponga que el usuario introduce los valores 15 y 10 a este programa, de manera que **main** inicialmente hace la llamada **gcd(15, 10)**. Esta llamada da como resultado una segunda llamada recursiva **gcd(10, 5)** (puesto que 15 % 10 = 5), y ésta produce una tercera llamada **gcd(5,0)** (puesto que 10 % 5 = 0), lo que devuelve entonces el valor 5. Durante la tercera llamada el ambiente de ejecución puede visualizarse como en la figura 7.4. Advierta cómo cada llamada a **gcd** agrega un nuevo registro de activación de exactamente el mismo tamaño a la parte superior o tope de la pila, y en cada nuevo registro de activación el vínculo de control apunta al vínculo de control del registro de activación anterior. Advierta también que el fp apunta al vínculo de control del registro de activación actual, de modo que en la siguiente llamada el fp actual se convierte en el vínculo de control del siguiente registro de activación.

Figura 7.4
Ambiente basado en pila para el ejemplo 7.2

Después de la llamada final a **gcd**, cada una de las activaciones es eliminada por turno de la pila, de manera que cuando la sentencia **printf** es ejecutada en **main**, sólo el registro de activación para **main** y el área global/estática permanecen en el ambiente. (Mostramos el registro de activación de **main** como vacío. En realidad, contendría información que sería utilizada para transferir el control de regreso al sistema operativo.)

Finalmente, observamos que no se necesita espacio en el elemento que llama para los valores de argumento en las llamadas a **gcd** (a diferencia de la constante 3 en el ambiente FORTRAN77 de la figura 7.2), ya que el lenguaje C emplea parámetros de valor. Este punto se analizará con más detalle en la sección 7.5. §

Ejemplo 7.3

Considere el código C de la figura 7.5. Este código contiene variables que serán utilizadas para ilustrar otros puntos más adelante en esta sección, pero su funcionamiento básico es de

Figura 7.5
Programa en C del ejemplo 7.3

```
int x = 2;

void g(int); /* prototipo */

void f(int n)
{ static int x = 1;
  g(n);
  x--;
}

void g(int m)
{ int y = m--1;
  if (y > 0)
  { f(y);
    x--;
    g(y);
  }
}

main()
{ g(x);
  return 0;
}
```

la manera siguiente. La primera llamada desde **main** es a **g(2)** (puesto que **x** tiene el valor 2 en ese punto). En esta llamada, **m** se convierte en 2 mientras que **y** toma el valor de 1. Entonces, **g** hace la llamada **f(1)**, y **f** a su vez hace la llamada **g(1)**. En esta llamada a **g**, **m** se convierte en 1 y **y** en 0, de modo que no se hacen más llamadas. El ambiente de ejecución en este punto (durante la segunda llamada a **g**) se muestra en la figura 7.6a) (página 356).

Ahora las llamadas a **g** y **f** salen de escena (con **f** disminuyendo su variable local estática **x** antes de regresar), sus registros de activación son extraídos de la pila y se devuelve el control al punto directamente a continuación de la llamada a **f** en la primera llamada a **g**. Ahora **g** disminuye la variable externa **x** y hace una llamada adicional **g(1)**, la cual establece **m** a 1 y **y** a 0, lo que da como resultado el ambiente de ejecución mostrado en la figura 7.6b). Una vez que ya no se hacen llamadas adicionales, los registros de activación restantes son extraídos de la pila, y el programa llega a su fin.

Advierta cómo, en la figura 7.6b), el registro de activación de la tercera llamada a **g** ocupa (y sobreescribe) el área de memoria previamente ocupada por el registro de activación de **f**. Observe también que la variable estática **x** en **f** no puede ser asignada en un registro de activación de **f**, puesto que debe continuar a través de todas las llamadas a **f**. De este modo, debe ser asignada en el área global/estática junto con la variable externa **x**, aun cuando no sea una variable global. En efecto, no puede haber confusión con la variable externa **x** porque la tabla de símbolos siempre las distinguirá y determinará la variable correcta para tener acceso a cada punto del programa.

Figura 7.6

(a) Ambiente de ejecución del programa de la figura 7.5 durante la segunda llamada a g

(b) Ambiente de ejecución del programa de la figura 7.5 durante la tercera llamada a g

§

Una herramienta útil para el análisis de estructuras de llamada complejas en un programa es el **árbol de activación**: cada registro (o llamada) de activación se convierte en un nodo en este árbol, y los descendientes de cada nodo representan todas las llamadas hechas durante la llamada correspondiente a ese nodo. Por ejemplo, el árbol de activación del programa de la figura 7.3 es lineal y se representa (para las entradas 15 y 10) en la figura 7.7a), mientras que el árbol de activación del programa de la figura 7.5 se ilustra en la figura 7.7b). Advierta que los ambientes mostrados en las figuras 7.4 y 7.6 representan los ambientes durante las llamadas representadas por cada una de las hojas de los árboles de activación. En general, la pila de los registros de activación al principio de una llamada particular tiene una estructura equivalente a la trayectoria desde el nodo correspondiente en el árbol de activación hasta la raíz.

Acceso a nombres En un ambiente basado en pila, ya no se puede tener acceso a los parámetros y las variables locales mediante direcciones fijas como en un ambiente completamente estático. En vez de eso, deben ser encontradas mediante desplazamiento desde el apuntador

Figura 7.7
Árboles de activación para los programas de las figuras 7.3 y 7.5

de marco actual. En la mayoría de los lenguajes el desplazamiento para cada declaración local se calcula todavía de manera estática mediante el compilador, puesto que las declaraciones de un procedimiento están fijadas en tiempo de compilación y el tamaño de memoria para asignarse a cada declaración está fijado por su tipo de datos.

Consideremos el procedimiento **g** en el programa C de la figura 7.5 (véanse también los ambientes de ejecución ilustrados en la figura 7.6). Cada registro de activación de **g** tiene exactamente la misma forma, y el parámetro **m**, así como la variable local **y**, se encuentran siempre en exactamente la misma ubicación relativa en el registro de activación. Llamemos a estas distancias **mOffset** y **yOffset**. Entonces, durante cualquier llamada a **g**, tenemos la siguiente ilustración del ambiente local

Se puede tener acceso tanto a **m** como a **y** mediante sus desplazamientos ("offsets") fijados desde el fp. Por ejemplo, supongamos concretamente que la pila de ejecución se incrementa desde las direcciones de memoria más altas hasta las más bajas, que las variables enteras requieren de 2 bytes de almacenamiento y que las direcciones requieren de 4 bytes. Con la organización de un registro de activación como el que se muestra, tenemos que **mOffset** = +4 y **yOffset** = −6, y las referencias a **m** y a **y** pueden escribirse en código de máquina (suponiendo convenciones de ensamblador estándar) como **4(fp)** y **-6(fp)**, respectivamente.

Las estructuras y arreglos locales sólo tienen dificultad para asignar y calcular direcciones en el caso de las variables simples, como lo demuestra el ejemplo siguiente.

Ejemplo 7.4

Considere el procedimiento C

```
void f(int x, char c)
{ int a[10];
  double y;
  ...
}
```

El registro de activación para una llamada a `f` aparecería como

y, suponiendo dos bytes para enteros, cuatro bytes para direcciones, un byte para caracteres y ocho bytes para punto flotante de precisión doble, tendríamos los siguientes valores de desplazamiento (suponiendo nuevamente una dirección negativa de crecimiento para la pila), todos los cuales se calculan en tiempo de compilación:

Nombre	Desplazamiento
`x`	+5
`c`	+4
`a`	−24
`y`	−32

Ahora un acceso a, digamos, `a[i]`, requeriría el cálculo de la dirección

```
(-24+2*i)(fp)
```

(aquí el factor de 2 en el producto `2*i` es el **`factor de escala`** resultante de la suposición de que los valores enteros ocupan dos bytes). Un acceso a memoria así, dependiendo de la ubicación de `i` y la arquitectura, puede necesitar una simple instrucción. §

No puede tenerse acceso a los nombres estáticos y no locales en este ambiente de la misma manera que a los nombres locales. En realidad, en el caso que estamos considerando aquí (lenguajes con ausencia de procedimientos locales) todos los nombres no locales son globales y, por lo tanto, estáticos. De este modo, en la figura 7.6, la variable `x` externa (global) de C tiene una ubicación estática fija, y así se puede tener acceso a ella directamente (o mediante

desplazamiento desde algún otro apuntador base aparte del fp). Se tiene acceso a la variable local estática **x** de **f** exactamente de la misma manera. Advierta que este mecanismo implementa ámbito estático (o léxico), como se describió en el capítulo anterior. Si se desea ámbito dinámico, entonces se requiere de un mecanismo de acceso más complejo (éste se describirá más adelante en esta sección).

La secuencia de llamada La secuencia de llamada ahora comprende aproximadamente los pasos siguientes.[8] Cuando un procedimiento es llamado,

1. Calcula los argumentos y los almacena en sus posiciones correctas en el nuevo registro de activación del procedimiento (esto se logra insertándolos en orden en la pila de ejecución).
2. Almacena (inserta) el fp como el vínculo de control en el nuevo registro de activación.
3. Cambia el fp de manera que apunte al principio del nuevo registro de activación (si existe un sp, esto se consigue copiando éste en el fp en este punto).
4. Almacena la dirección de retorno en el nuevo registro de activación (si es necesario).
5. Realiza un salto hacia el código del procedimiento a ser llamado.

Cuando un procedimiento sale de escena,

1. Copia el fp al sp.
2. Carga el vínculo de control en el fp.
3. Realiza un salto hacia la dirección de retorno.
4. Cambia el sp para extraer los argumentos.

Ejemplo 7.5 Considere la situación precisamente antes de la última llamada a **g** en la figura 7.6b):

Cuando se realiza la nueva llamada a **g**, se inserta primero el valor del parámetro **m** en la pila de ejecución:

8. Esta descripción ignora cualquier grabación de registros que deba tener lugar. También ignora la necesidad de situar un valor de retorno en una ubicación disponible.

Entonces el fp se inserta en la pila:

Ahora se copia el sp en el fp, la dirección de retorno se inserta en la pila y se hace el salto a la nueva llamada de g:

Finalmente, **g** asigna e inicializa la nueva **y** en la pila para completar la construcción del nuevo registro de activación:

§

Tratamiento de datos de longitud variable Hasta ahora hemos descrito una situación en la cual todos los datos, ya sean locales o globales, pueden encontrarse en un lugar fijo o en un desplazamiento fijo del fp que puede ser calculado por el compilador. En ocasiones un compilador debe tratar con la posibilidad de que los datos puedan variar, tanto en el número de objetos de datos como en el tamaño de cada objeto. Dos ejemplos que se presentan en lenguajes que soportan ambientes basados en pila son 1) el número de argumentos en una llamada puede tener variaciones de una llamada a otra, y 2) el tamaño de un parámetro de arreglo o una variable de arreglo local puede variar de llamada en llamada.

Un ejemplo típico de la situación número 1 es la función **printf** de C, donde el número de argumentos se determina a partir de la cadena de formato que se pasa como el primer argumento. Por consiguiente,

```
printf("%d%s%c",n,prompt,ch);
```

tiene cuatro argumentos (incluyendo la cadena de formato **"%d%s%c"**), mientras que

```
printf("Hello, world\n");
```

tiene sólo un argumento. Los compiladores de C por lo regular manejan esto insertando los argumentos para una llamada **en orden inverso** en la pila de ejecución. Entonces, el primer parámetro (el cual le dice al código para **printf** cuántos parámetros más hay) está siempre ubicado en un desplazamiento fijo desde el fp en la implementación descrita anteriormente (de hecho +4, empleando las suposiciones del ejemplo anterior). Otra opción es utilizar un mecanismo preprocesador como el ap (argument pointer, apuntador de argumentos) en las arquitecturas VAX. Ésta y otras posibilidades se tratan con más detalle en los ejercicios.

Un ejemplo de la situación número 2 es el **arreglo no restringido** de Ada:

```
type Int_Vector is
      array(INTEGER range <>) of INTEGER;

procedure Sum (low,high: INTEGER;
                    A: Int_Vector) return INTEGER
is
   temp: Int_Array (low..high);
begin
   ...
end Sum;
```

(Advierta que la variable local **temp** también tiene un tamaño impredecible.) Un método típico para manejar esta situación es emplear un nivel extra de indirección para los datos de longitud variable, en tanto que se almacena un apuntador a los datos reales en una ubicación que puede ser predecida en tiempo de compilación, mientras se realiza la asignación real en el tope de la pila de ejecución de una manera que pueda ser administrada mediante el sp durante la ejecución.

Ejemplo 7.6

Dado el procedimiento **Sum** de Ada, como se definió anteriormente, y suponiendo la misma organización para el ambiente que antes,[9] podríamos implementar un registro de activación para **Sum** de la manera siguiente (esta imagen muestra, en concreto, una llamada a **Sum** con un arreglo de tamaño 10):

Ahora, por ejemplo, el acceso a **A[i]** puede conseguirse calculando

```
@6 (fp) +2*i
```

9. Esto en realidad no es suficiente para Ada, el cual permite procedimientos anidados; véase el análisis más adelante en esta sección.

donde la @ significa indirección, y donde de nueva cuenta suponemos que hay dos bytes para enteros y cuatro bytes para direcciones. §

Advierta que en la implementación descrita en el ejemplo anterior, el elemento que llama debe conocer el tamaño de cualquier registro de activación de `Sum`. El tamaño de la parte del parámetro y la parte de administración es conocido por el compilador en el punto de llamada (puesto que se pueden contar los tamaños de argumentos y la parte de administración es igual para todos los procedimientos), pero el tamaño de la parte de la variable local, por lo común no se conoce en el punto de llamada. De este modo, esta implementación requiere que el compilador calcule previamente un atributo de tamaño de variable local para cada procedimiento y lo almacene en la tabla de símbolos para su uso posterior. Las variables locales de longitud variable pueden ser tratadas de manera similar.

Vale la pena reiterar que los arreglos en C no caen dentro de la clase de tales datos de longitud variable. En efecto, los arreglos en C son apuntadores, de modo que los parámetros de arreglo son pasados por referencia en C y no asignados localmente (y no conducen información de tamaño).

Declaraciones anidadas y temporales locales Éstas son dos complicaciones más para el ambiente de ejecución básico basado en pila que merecen mencionarse: las declaraciones anidadas y temporales locales.

Las temporales locales son resultados parciales de cálculos que deben ser grabados a través de las llamadas de procedimiento. Considere, por ejemplo, la expresión en C

```
x[i] = (i + j)*(i/k + f(j))
```

En una evaluación de izquierda a derecha de esta expresión, es necesario grabar tres resultados parciales por medio de la llamada a `f`: la dirección de `x[i]` (para la asignación pendiente), la suma `i+j` (en espera de la multiplicación), y el cociente `i/k` (pendiente de la suma con el resultado de la llamada `f(j)`). Estos resultados parciales podrían calcularse en registros y grabarse y recuperarse según algún mecanismo de administración de registros, o podrían almacenarse como temporales en la pila de ejecución antes de la llamada a `f`. En este último caso la pila de ejecución puede tener el aspecto que sigue en el punto justo antes de la llamada a `f`:

En esta situación, la secuencia de llamada descrita previamente utilizando el sp funciona sin cambio. De manera alternativa, el compilador también puede calcular fácilmente la posición del tope de la pila a partir del fp (cuando no hay datos de longitud variable), ya que el número de temporales requeridos es una cantidad en tiempo de compilación.

Las declaraciones anidadas presentan un problema semejante. Considere el código en C

```
void p( int x, double y)
{ char a;
  int i;
  ...
A:{ double x;
    int j;
    ...
  }
  ...
B:{ char * a;
    int k;
    ...
  }
  ...
}
```

En este código tenemos dos bloques (también llamados *sentencias compuestas*) etiquetados **A** y **B**, anidados dentro del cuerpo del procedimiento **p**, cada uno con dos declaraciones locales cuyo ámbito se extiende sólo sobre el bloque en el cual se encuentran localizados (es decir, hasta la siguiente llave de cierre). Las declaraciones locales de cada uno de estos bloques no necesitan asignarse hasta que se entre al bloque, y las declaraciones del bloque **A** y el bloque **B** no necesitan asignarse simultáneamente. Un compilador *trataría* un bloque de la misma manera que trataría un procedimiento y crearía un nuevo registro de activación cada vez que se entre a un bloque y lo descartaría a la salida. Sin embargo, esto sería ineficaz, puesto que tales bloques son mucho más simples que los procedimientos: un bloque así no tiene parámetros ni dirección de retorno y siempre se ejecuta de inmediato, en vez del que se llama de otra parte. Un método más simple es manejar las declaraciones en bloques anidados como se manejan las expresiones temporales, asignándolas en la pila a medida que se entra al bloque y desasignándolas a la salida.

Por ejemplo, justo después de introducir el bloque **A** en el código C de muestra que se acaba de proporcionar, la pila de ejecución tendría el aspecto que sigue:

y justo después de entrar al bloque B tendría el aspecto que sigue:

Al realizar una implementación de esta naturaleza se debe tener cuidado de asignar declaraciones anidadas de manera tal que los desplazamientos desde el fp del bloque de procedimiento circundante se puedan calcular en tiempo de compilación. En particular, tales datos deben ser asignados antes de cualquier dato de longitud variable. Por ejemplo, en el código que se acaba de dar, la variable local j para el bloque A tendría un desplazamiento de −17 desde el fp de p (suponiendo otra vez 2 bytes para enteros, 4 bytes para direcciones, 8 bytes para números reales de punto flotante y 1 byte para caracteres), mientras que k en el bloque B tendría un desplazamiento de −13.

7.3.2 Ambientes basados en pila con procedimientos locales

Si se permiten declaraciones de procedimiento local en el lenguaje que se está compilando, entonces el ambiente de ejecución que hemos descrito hasta ahora es insuficiente, puesto que no se ha previsto nada para referencias no locales y no globales.

Considere, por ejemplo, el código en Pascal de la figura 7.8, página 366 (en Ada podrían escribirse programas similares). Durante la llamada a q el ambiente de ejecución podría parecerse al de la figura 7.9. Si se utiliza la regla de ámbito estático estándar, cualquier mención de n dentro de q debe hacer referencia a la variable entera local n de p. Como podemos ver en la figura 7.9, esta n no se puede encontrar utilizando algo de la información de administración que se conserva en el ambiente de ejecución hasta ahora.

Se *podría* encontrar a n empleando los vínculos de control, si estamos dispuestos a aceptar el ámbito dinámico. Al observar la figura 7.9 vemos que la n en el registro de activación de r se podría encontrar siguiendo el vínculo de control, y si r no tuviera declaración de n, entonces la n de p se podría encontrar siguiendo un segundo vínculo de control (este proceso se conoce como **encadenamiento**, un método que veremos de nuevo en breve). Desgraciadamente, no sólo esto implementa el ámbito dinámico, sino que los desplazamientos para los cuales n puede encontrarse pueden variar con diferentes llamadas (observe que la n en r tiene un desplazamiento diferente de la n en p). De este modo, en una implementación

así, las tablas de símbolos locales de cada procedimiento deben conservarse durante la ejecución con el fin de permitir que un identificador sea buscado en cada registro de activación para ver si existe, y determinar su desplazamiento. Ésta es una complicación adicional importante para el ambiente de ejecución.

Figura 7.8
Programa en Pascal que muestra la referencia no local y no global

```
program nonLocalRef;

procedure p;
var n: integer;

      procedure q;
      begin
        (* una referencia a n es ahora
           no local no global *)
      end; (* q *)

      procedure r(n: integer);
      begin
        q;
      end; (* r *)

begin (* p *)
  n := 1;
  r(2);
end; (* p *)

begin (* main *)
  p;
end.
```

Figura 7.9
Pila de ejecución para el programa de la figura 7.8

La solución a este problema, que también implementa el ámbito estático, es agregar una pieza extra de información de administración que se conoce como **vínculo de acceso** a cada registro de activación. Este vínculo de acceso es como el vínculo de control, sólo que éste apunta al registro de activación que representa el *ambiente de definición* del procedimiento en vez de hacerlo al ambiente de llamada. Por esta razón, el vínculo de acceso en ocasiones también se conoce como **vínculo estático**, aun cuando no es una cantidad propia del tiempo de compilación.[10]

La figura 7.10 muestra la pila de ejecución de la figura 7.9 modificada para incluir vínculos de acceso. En este nuevo ambiente, los vínculos de acceso de los registros de activación, tanto de `r` como de `q`, apuntan al registro de activación de `p`, ya que tanto `r` como `q` están declarados dentro de `p`. Ahora una referencia no local a `n` dentro de `q` provocará que se siga el vínculo de acceso, en donde `n` se encontrará en un desplazamiento fijado, puesto que éste siempre será un registro de activación de `p`. Por lo común, esto puede conseguirse en código cargando el vínculo de acceso en un registro y posteriormente accediendo a `n` mediante desplazamiento desde este registro (que ahora funciona como el fp). Por ejemplo, utilizando las convenciones de tamaño descritas anteriormente, si el registro r se utiliza para el vínculo de acceso, entonces se puede tener acceso a `n` dentro de `p` como −6(r) después de que r se haya cargado con el valor 4(fp) (el vínculo de acceso tiene desplazamiento +4 desde el fp de la figura 7.10).

Figura 7.10
Pila de ejecución para el programa de la figura 7.8 con vínculos de acceso agregados

Advierta que el registro de activación del procedimiento `p` mismo no contiene vínculo de acceso, como se indica por el comentario entre picoparéntesis en la ubicación donde podría ir. Esto se debe a que `p` es un procedimiento global, de modo que ninguna referencia no local dentro de `p` debe ser una referencia global, y se tiene acceso a él mediante el mecanismo de referencia global. De este modo, no hay necesidad de un vínculo de acceso. (De hecho, se puede insertar un vínculo de acceso nulo o de otro modo arbitrario sólo para ser consistentes con los otros procedimientos.)

El caso que hemos estado describiendo es realmente la situación más simple, donde la referencia no local es a una declaración en el siguiente ámbito exterior. También es posible que las referencias no locales hagan referencia a declaraciones en ámbitos más distantes. Considere, por ejemplo, el código en la figura 7.11.

10. El procedimiento de definición es por supuesto conocido, pero no la ubicación exacta de su registro de activación.

Figura 7.11
Código en Pascal que muestra el encadenamiento de acceso

```
program chain;

procedure p;
var x: integer;

   procedure q;
      procedure r;
      begin
        x := 2;
        ...
        if ... then p;
      end; (* r *)
   begin
     r;
   end; (* q *)

begin
  q;
end; (* p *)

begin (* main *)
  p;
end.
```

En este código el procedimiento **r** es declarado en el procedimiento **q**, el cual a su vez es declarado en el procedimiento **p**. De este modo, la asignación para **x** dentro de **r**, que hace referencia a la **x** de **p**, debe atravesar dos niveles de ámbito para encontrar **x**. La figura 7.12 muestra la pila de ejecución después de la (primera) llamada a **r** (puede haber más de una llamada a **r**, puesto que **r** puede llamar de manera recursiva a **p**). En este ambiente, **x** debe ser alcanzada siguiendo *dos* vínculos de acceso, un proceso que se denomina **encadenamiento de acceso**. Este encadenamiento de acceso es implementado al ir a buscar de manera repetida el vínculo de acceso utilizando el vínculo previamente alcanzado como si fuera el fp. Como ejemplo concreto, se puede tener acceso a **x** de la figura 7.12 (utilizando las convenciones de tamaño anteriores) de la manera siguiente:

Cargar 4(fp) en el registro r.

Cargar 4(r) en el registro r.

Acceder ahora a **x** como −6(r).

Para que el método de encadenamiento de acceso funcione, el compilador debe ser capaz de determinar a través de cuántos niveles de anidación se hará el encadenamiento antes de tener acceso al nombre localmente. Esto requiere que el compilador calcule previamente un atributo de **nivel de anidación** para cada declaración. Por lo regular, el ámbito más remoto (el nivel de programa principal en Pascal o el ámbito externo de C) tiene asignado el nivel de anidación 0, y cada vez que se introduce una función o procedimiento (durante la compilación), el nivel de anidación se incrementa en 1, y se decrementa en la misma cantidad a la salida. Por ejemplo, en el código de la figura 7.11 el procedimiento **p** tiene asignado el nivel de anidación 0 porque es global; la variable **x** tiene asignado el nivel de anidación 1, porque el nivel de anidación se incrementa cuando se introduce **p**; el procedimiento **q** también tiene asignado un nivel de anidación 1, porque es local a **p**, y el procedimiento **r** tiene asignado el nivel de anidación 2, porque nuevamente el nivel de anidación se incrementa cuando se introduce **q**. Finalmente, dentro de **r** el nivel de anidación nuevamente se incrementa a 3.

Figura 7.12
Pila de ejecución después de la primera llamada a r en el código de la figura 7.11

Ahora la cantidad de encadenamiento necesaria para tener acceso a un nombre no local se puede determinar comparando el nivel de anidación en el punto de acceso con el nivel de anidación de la declaración del nombre; el número de vínculos de acceso a seguir es la diferencia entre estos dos niveles de anidación. Por ejemplo, en la situación anterior, la asignación a **x** se presenta en el nivel de anidación 3, y **x** tiene nivel de anidación 1, de modo que deben seguirse dos vínculos de acceso. En general, si la diferencia entre niveles de anidación es ***m***, entonces el código que debe ser generado para el encadenamiento de acceso debe tener ***m*** cargas para un registro r, la primera utilizando el fp, y el resto utilizando r.

Puede parecer que el encadenamiento de acceso es un método ineficiente para el acceso de variables, ya que es necesario ejecutar una larga secuencia de instrucciones para cada referencia no local con una gran diferencia de anidación. Sin embargo, en la práctica, los niveles de anidación rara vez tienen más de dos o tres niveles de profundidad, y la mayoría de las referencias no locales son para variables globales (nivel de anidación 0), a las cuales se puede continuar teniendo acceso mediante los métodos directos previamente analizados. Existe un método para implementar vínculos de acceso en una tabla de búsqueda indizada mediante nivel de anidación que no conduce al gasto de ejecución del encadenamiento. La estructura de datos utilizada para este método se denomina **despliegue** ("**display**"). Su estructura y uso se tratan en los ejercicios.

La secuencia de llamada Los cambios a la secuencia de llamada necesarios para implementar los vínculos de acceso son relativamente directos. En la implementación mostrada, durante una llamada, el vínculo de acceso debe ser insertado en la pila de ejecución justo antes del fp, y después de una salida el sp debe ser ajustado mediante una cantidad extra para eliminar tanto el vínculo de acceso como los argumentos.

El único problema es el de encontrar el vínculo de acceso de un procedimiento durante una llamada. Esto se puede conseguir empleando la información del nivel de anidación (en tiempo de compilación) adjunta a la declaración del procedimiento que se está llamando. En realidad, todo lo que necesitamos hacer es generar una cadena de acceso, precisamente como si fuéramos a tener acceso a una variable en el mismo nivel de anidación que el del procedimiento que se está llamando. La dirección así calculada será la del vínculo de acceso apropiado. Naturalmente, si el procedimiento es local (la diferencia en niveles de anidación es 0), el vínculo de acceso y el vínculo de control son iguales (y también son iguales al fp en el punto de la llamada).

Considere, por ejemplo, la llamada a `q` desde dentro de `r` en la figura 7.8. En el interior de `r` estamos a nivel de anidación 2, mientras que la declaración de `q` conlleva un nivel de anidación de 1 (porque `q` es local a `p` y dentro de `p` el nivel de anidación es 1). De esta forma, se requiere un paso de acceso para calcular el vínculo de acceso de `q`, y, efectivamente en la figura 7.10, el vínculo de acceso de `q` apunta al registro de activación de `p` (y es el mismo que el vínculo de acceso de `r`).

Advierta que incluso en la presencia de activaciones múltiples del ambiente de definición, este proceso calculará el vínculo de acceso correcto, ya que el cálculo es realizado en tiempo de ejecución (utilizando los niveles de anidación en tiempo de compilación), no en tiempo de compilación. Por ejemplo, dado el código de la figura 7.11, la pila de ejecución después de la *segunda* llamada a `r` (suponiendo una llamada recursiva a `p`) se parecería al de la figura 7.13. En esta ilustración `r` tiene dos registros de activación diferentes con dos vínculos de acceso diferentes, apuntando a los diferentes registros de activación de `q`, los cuales representan diferentes ambientes de definición para `r`.

Figura 7.13
Pila de ejecución después de la segunda llamada a `r` en el código de la figura 7.11

7.3.3 Ambientes basados en pila con parámetros de procedimiento

En algunos lenguajes no sólo se permiten procedimientos locales, sino que los procedimientos también se pueden pasar como parámetros. En un lenguaje así, cuando se llama un procedimiento que ha sido pasado como un parámetro, es imposible para un compilador generar código que permita calcular el vínculo de acceso al momento de la llamada, de la manera en que se describió en la sección anterior. En vez de esto, se debe calcular previamente el

vínculo de acceso para un procedimiento y pasarse junto con un apuntador al código para el procedimiento cuando éste se pasa como un parámetro. De esta forma, un valor de parámetro de procedimiento ya no puede ser visto como un simple apuntador de código, sino que también debe incluir un apuntador de acceso que defina el ambiente en el cual las referencias no locales son resueltas. Este par de apuntadores (un apuntador de código y un vínculo de acceso, o bien, un **apuntador de instrucción** y un **apuntador de ambiente**) juntos representan el valor de un parámetro de procedimiento o función y se les conoce comúnmente como **cerradura** (porque el vínculo de acceso "cierra" los "huecos" causados por las referencias no locales).[11] Escribiremos las cerraduras como <ip, ep>, ip para referirnos al apuntador de instrucción (apuntador de código o apuntador de entrada) del procedimiento, y ep para referirnos al apuntador de ambiente (vínculo de acceso) del procedimiento.

Ejemplo 7.7

Considere el programa en Pascal estándar de la figura 7.14, el cual tiene un procedimiento `p`, con un parámetro `a` que también es un procedimiento. Después de la llamada a `p` en `q`, en la cual el procedimiento local `r` de `q` es pasado a `p`, la llamada a `a` dentro de `p` en realidad llama a `r`, esta llamada todavía debe encontrar la variable no local `x` en la activación de `q`. Cuando `p` es llamado, `a` es construido como una cerradura <ip, ep>, donde ip es un apuntador al código de `r` y ep es una copia del fp en el punto de la llamada (es decir, apunta al ambiente de la llamada a `q` en la cual está definida `r`). El valor del ep de `a` se indica mediante la línea discontinua de la figura 7.15 (página 372), la cual representa el ambiente justo después de la llamada a `p` en `q`. Entonces, cuando `a` es llamada dentro de `p`, el ep de `a` es utilizado como el vínculo estático en su registro de activación, como se indica en la figura 7.16. §

Figura 7.14
Código en Pascal estándar con un procedimiento como parámetro

```
program closureEx(output);

procedure p(procedure a);
begin
  a;
end;

procedure q;
var x:integer;

      procedure r;
      begin
        writeln(x);
      end;

begin
  x := 2;
  p(r);
end; (* q *)

begin (* main *)
  q;
end.
```

§

11. Este término tiene su origen en el cálculo lambda y no debe confundirse con la operación de cerradura (Kleene) en expresiones regulares o la cerradura ε de un conjunto de estados NFA.

Figura 7.15
Pila de ejecución justo después de la llamada a p en el código de la figura 7.14

Figura 7.16
Pila de ejecución justo después de la llamada a a en el código de la figura 7.14

La secuencia de llamada en un ambiente como el que acabamos de describir debe ahora distinguir claramente entre procedimientos ordinarios y parámetros de procedimiento. Un procedimiento ordinario, como antes, es llamado buscando el vínculo de acceso por medio del nivel de anidación del procedimiento y saltando directamente al código del procedimiento (el cual es conocido en tiempo de compilación). Un parámetro de procedimiento, por otra parte, ya dispone de un vínculo de acceso almacenado en el registro de activación local, el cual debe ser obtenido e insertado en el nuevo registro de activación. La ubicación del código para el procedimiento, no obstante, no se conoce directamente por el compilador; en su lugar debe realizarse una llamada indirecta al ip almacenado en el registro de activación actual.

Un escritor de compiladores puede querer evitar, por razones de simplicidad o uniformidad, esta distinción entre procedimientos ordinarios y parámetros de procedimiento y conservar todos los procedimientos como cerraduras en el ambiente. En realidad, cuanto más general es un lenguaje en su tratamiento de procedimientos, más razonable se vuelve un método así. Por ejemplo, si se permiten variables de procedimiento, o si los valores de procedimiento se pueden calcular dinámicamente, entonces la representación <ip, ep> de los procedimientos se vuelve un requerimiento para todas esas situaciones. La figura 7.17 muestra que el ambiente de la figura 7.16 daría la impresión de que todos los valores de procedimiento están almacenados en el ambiente como cerraduras.

Figura 7.17
Pila de ejecución justo después de la llamada a a en el código de la figura 7.14 con todos los procedimientos mantenidos como cerraduras en el ambiente

Finalmente, observamos que tanto C como Modula-2 y Ada evitan las complicaciones descritas en esta subsección: C, porque no tiene procedimientos locales (aunque tiene variables y parámetros de procedimiento); Modula-2, porque tiene una regla especial que restringe los valores de parámetro de procedimiento y variable de procedimiento a procedimientos globales; y Ada, porque no tiene variables o parámetros de procedimiento.

7.4 MEMORIA DINÁMICA

7.4.1 Ambientes de ejecución completamente dinámicos

Los ambientes de ejecución basados en pila que se analizaron en la sección anterior son las formas más comunes de ambiente entre los lenguajes imperativos estándar tales como C, Pascal y Ada. Sin embargo, tales ambientes tienen limitaciones. En particular, en un lenguaje donde una referencia a una variable local en un procedimiento puede ser devuelta al elemento que llama, ya sea de manera implícita o explícita, un ambiente basado en pila producirá una **referencia colgante** cuando se salga del procedimiento, ya que el registro de activación del procedimiento será desasignado de la pila. El ejemplo más simple de esto es cuando la dirección de una variable local es devuelta, como en el código de C:

```
int * dangle(void)
{ int x;
  return &x;}
```

Una asignación `addr = dangle( )` causa ahora que `addr` apunte a una ubicación insegura en la pila de activación cuyo valor puede ser cambiado arbitrariamente por llamadas posteriores a cualquier procedimiento. C evita este problema declarando simplemente que un programa así es erróneo (aunque el compilador no dará un mensaje de error). En otras palabras, la semántica de C está construida alrededor del ambiente fundamental basado en pila.

Un ejemplo algo más complejo de una referencia colgante se presenta si una función local puede ser devuelta mediante una llamada. Por ejemplo, si C permitiera definiciones de función local, el código de la figura 7.18 produciría una referencia colgante indirecta al parámetro **x** de **g**, al cual se puede tener acceso llamando a **f** después de que **g** ha salido de escena. Naturalmente, C evita este problema prohibiendo procedimientos locales. Otros lenguajes, como Modula-2, que tienen procedimientos locales además de variables de procedimiento, parámetros y valores devueltos, deben establecer una regla especial que haga erróneos a tales programas. (En Modula-2 la regla es que sólo los procedimientos globales pueden ser argumentos o valores devueltos: una retirada importante incluso de parámetros de procedimiento estilo Pascal.)

Figura 7.18
Código pseudo-C que muestra una referencia colgante causada por el retorno de una función local

```
typedef int (* proc)(void);

proc g(int x)
{ int f(void) /* función local ilegal */
  { return x;}
  return f; }

main()
{ proc c;
  c = g(2);
  printf("%d\n",c()); /* debería imprimir 2 */
  return 0;
}
```

Sin embargo, existe una extensa clase de lenguajes donde tales reglas son inaceptables, es decir, los lenguajes de programación funcionales, como LISP y ML. Un principio esencial en el diseño de un lenguaje funcional es que las funciones sean tan generales como sea posible, y esto significa que las funciones deben poderse definir localmente, pasarse como parámetros y devolverse como resultados. De este modo, para esta amplia clase de lenguajes, un ambiente de ejecución basado en pila es inadecuado, y se requiere una forma de ambiente más general. Llamaremos a un ambiente así **completamente dinámico**, porque pueden desasignar registros de activación sólo cuando han desaparecido todas las referencias a ellos, y esto requiere que los registros de activación sean dinámicamente liberados en momentos arbitrarios durante la ejecución. Un ambiente de ejecución completamente dinámico es mucho más complicado que un ambiente basado en pila, ya que involucra el seguimiento de las referencias durante la ejecución, y la capacidad de encontrar y desasignar áreas inaccesibles de memoria en momentos arbitrarios durante la ejecución (este proceso se denomina **recolección de basura**).

A pesar de la complejidad adicional de esta clase de ambiente, la estructura básica de un registro de activación permanece igual: el espacio debe ser asignado para parámetros y variables locales, y todavía se necesitan vínculos de acceso y de control. Naturalmente, ahora cuando el control es devuelto al elemento que llama (y el vínculo de control es utilizado para restaurar el ambiente anterior), el registro de activación que sale permanece en la memoria, para ser desasignado en algún momento posterior. De esta forma, toda la complejidad adicional de este ambiente puede ser encapsulada en un administrador de memoria que reemplaza las operaciones de la pila de ejecución con rutinas más generales de asignación

y desasignación. Analizaremos algunas de estas cuestiones en el diseño de un administrador de memoria de esta naturaleza, más adelante en esta sección.

7.4.2 Memoria dinámica en lenguajes orientados a objetos

Los lenguajes orientados a objetos requieren de mecanismos especiales en el ambiente de ejecución para incrementar sus características adicionales: objetos, métodos, herencia y fijación dinámica. En esta subsección proporcionamos una breve perspectiva general de la variedad de técnicas de implementación para estas características. Suponemos que el lector está familiarizado con la terminología y conceptos básicos orientados a objetos.[12]

Los lenguajes orientados a objetos varían en gran medida en sus requerimientos para el ambiente de ejecución. Smalltalk y C++ son buenos representantes de los extremos: Smalltalk requiere un ambiente completamente dinámico similar al de LISP, mientras que gran parte del esfuerzo de diseño en C++ se ha ido en retener el ambiente basado en pila de C, sin la necesidad de administración de memoria dinámica automática. En ambos lenguajes un objeto en memoria puede ser visto como una combinación de una estructura de registro tradicional y un registro de activación, con las variables de instancia (miembros de datos) como los campos del registro. Esta estructura difiere de un registro tradicional en la manera en que se accede a las características de herencia y métodos.

Un mecanismo directo para implementar objetos podría ser copiar por código de inicialización todas las características (y métodos) comúnmente heredadas de manera directa en la estructura del registro (con los métodos como apuntadores de código). Sin embargo, esto implica un gasto excesivo de espacio. Una alternativa es mantener una descripción completa de la estructura de clase en la memoria en cada punto durante la ejecución, con la herencia mantenida mediante apuntadores de superclase, y todos los apuntadores de método conservados como campos en la estructura de clase (esto se conoce en ocasiones como **gráfica de herencia**). Cada objeto conserva entonces, junto con campos para sus variables de instancia, un apuntador a su clase de definición, a través del cual son encontrados todos los métodos (tanto locales como heredados). De esta manera, los apuntadores de método se registran sólo una vez (en la estructura de clase) y no se copian en la memoria para cada objeto. Este mecanismo también implementa herencia y fijación dinámica, porque los métodos son encontrados mediante una búsqueda de la jerarquía de clase. La desventaja de esto es que, mientras que las variables de instancia pueden tener desplazamientos predecibles (del mismo modo que las variables locales en un ambiente estándar), no ocurre lo mismo con los métodos, por lo que deben ser mantenidos por nombre en una estructura de tabla de símbolos con capacidad de búsqueda. Aun así, ésta es una estructura razonable para un lenguaje altamente dinámico como Smalltalk, donde pueden ocurrir cambios a la estructura de clase durante la ejecución.

Una alternativa a conservar toda la estructura de clase dentro del ambiente es calcular la lista de apuntadores de código para métodos disponibles de cada clase, y almacenarla en la memoria (estática) como una **tabla de función virtual** (en la terminología de C++). Ésta tiene la ventaja de que puede ser arreglada de manera que cada método tenga un desplazamiento predecible, y ya no es necesario un recorrido en la jerarquía de clase con una serie de búsquedas en tabla. Ahora cada objeto contiene un apuntador a la tabla de función virtual apropiada, en vez de a la estructura de clase. (Por supuesto, la ubicación de este apuntador también debe tener un desplazamiento predecible.) Esta simplificación sólo funciona si la misma estructura de clase está fijada antes de la ejecución. Es el método de elección en C++.

---

12. El análisis siguiente también supone que sólo está disponible la herencia simple. La herencia múltiple se trata en alguno de los trabajos citados en la sección de notas y referencias.

Ejemplo 7.8 Considere las siguientes declaraciones de clase de C++:

```
class A
{ public:
  double x,y;
  void f();
  virtual void g();
};

class B: public A
{ public:
  double z;
  void f();
  virtual void h();
};
```

un objeto de clase **A** aparecería en la memoria (con su tabla de función virtual) de la manera siguiente:

mientras que un objeto de clase **B** aparecería como se muestra a continuación:

Advierta cómo el apuntador de función virtual, una vez agregado a la estructura del objeto, permanece en una ubicación fija, de manera que se conoce su desplazamiento antes de la ejecución. Advierta también que la función **f** no obedece a la fijación dinámica en C++ (puesto que no es declarada "virtual"), y de esta forma no aparece en la tabla de función virtual (ni en ningún otro sitio en el ambiente); una llamada a **f** se resuelve en tiempo de compilación. §

7.4.3 Administración del apilamiento o montículo (heap)

En la sección 7.4.1 analizamos la necesidad de un ambiente de ejecución que fuera más dinámico que el ambiente basado en pila utilizado en la mayoría de los lenguajes compiladores, si queríamos que las funciones generales tuvieran soporte completo. Sin embargo, en la mayoría de los lenguajes, incluso un ambiente basado en pila necesita de algunas capacidades dinámicas con el fin de manejar la asignación y desasignación de apuntadores. La estructura de datos que maneja una asignación así se denomina *apilamiento* o *montículo (heap)*, y este apilamiento por lo regular es asignado como un bloque lineal de memoria, de tal

manera que pueda aumentar, si es necesario, al tiempo que interfiere lo menos posible con la pila. (En la página 347 mostramos el apilamiento situado en un bloque de memoria en el extremo opuesto del área de la pila.)

Hasta ahora en este capítulo nos hemos concentrado en la organización de los registros de activación y de la pila de ejecución. En esta sección queremos describir cómo se puede administrar el apilamiento, y cómo se pueden extender las operaciones de apilamiento para proporcionar la clase de asignación dinámica requerida en lenguajes con capacidades de función generales.

Un apilamiento o montículo proporciona dos operaciones, *asignación* (*allocate*) y *liberación* (*free*). La operación de *asignación* toma un parámetro de tamaño (ya sea explícita o implícitamente), por lo regular en bytes, y devuelve un apuntador a un bloque de memoria del tamaño correcto, o un apuntador nulo si no existe ninguno. La operación de *liberación* toma un apuntador a un bloque de memoria asignado y lo marca como libre de nuevo. (La operación de *liberación* también debe ser capaz de descubrir el tamaño del bloque que será liberado, ya sea de manera implícita o mediante un parámetro explícito.) Estas dos operaciones existen bajo diferentes nombres en muchos lenguajes: se denominan **new** y **dispose** en Pascal, y **new** y **delete** en C++. El lenguaje C tiene varias versiones de estas operaciones, pero las versiones básicas son conocidas como **malloc** y **free** y son parte de la biblioteca estándar (**stdlib.h**), donde tienen esencialmente las declaraciones siguientes:

```
void * malloc (unsigned nbytes);
void free (void * ap);
```

Utilizaremos estas declaraciones como la base para nuestra descripción de la administración del apilamiento.

Un método estándar para mantener el apilamiento e implementar estas funciones es emplear una lista ligada circular de bloques libres, de los cuales la memoria se toma mediante **malloc** y se devuelve mediante **free**. Esto tiene la ventaja de la simplicidad, pero también tiene desventajas. Una de éstas es que la operación **free** no puede decir si su argumento de apuntador está en realidad apuntando a un bloque legítimo que fue previamente asignado mediante **malloc**. Si el usuario llegara a pasar un apuntador inválido, entonces el apilamiento se corrompería rápida y fácilmente. Una segunda, pero mucho menos grave, desventaja es que debe tenerse cuidado de **fusionar** bloques que son devueltos a la lista libre con bloques que se encuentran adyacentes, de manera que resulte un bloque libre de tamaño máximo. Sin esta fusión, el apilamiento rápidamente se **fragmenta**, es decir, se divide en un gran número de bloques pequeños, de modo que la asignación de un bloque grande puede fallar, aun cuando haya suficiente espacio total disponible para asignarlo. (La fragmentación por supuesto puede ocurrir incluso sin que se produzca este fenómeno de fusión.)

Aquí ofrecemos una implementación hasta cierto punto diferente de **malloc** y **free** que utiliza una estructura de datos de lista ligada circular, la cual se mantiene al tanto de bloques asignados y liberados (por lo que es menos susceptible a la corrupción) y tiene también la ventaja de proporcionar bloques autofusionables. El código se proporciona en la figura 7.19 (página 379).

Este código utiliza un arreglo asignado estáticamente de tamaño **MEMSIZE** como apilamiento, pero también se podría emplear una llamada del sistema operativo para asignar el apilamiento. Definimos un tipo de datos **Header** (**"Encabezado"**) que mantendrá la información de administración de cada bloque de memoria, y definimos el arreglo del apilamiento para que tenga elementos de tipo **Header**, de manera que la información de administración pueda mantenerse fácilmente en los mismos bloques de memoria. El tipo **Header** contiene tres segmentos de información: un apuntador al siguiente bloque en la lista, el tamaño del espacio asignado actualmente (que viene enseguida en memoria) y el tamaño de cualquier espacio libre que le siga (si es que hay alguno). Por consiguiente, cada bloque en la lista es de la forma

La definición del tipo **Header** en la figura 7.19 también utiliza una declaración **union** y un tipo de datos **Align** (que hemos establecido a **double** en el código). Esto con el fin de "alinear" los elementos de memoria en un límite de byte razonable y, dependiendo del sistema, puede o no ser necesario. Esta complicación puede ignorarse sin consecuencias en el resto de esta descripción.

El único segmento adicional de datos necesario para las operaciones del apilamiento es un apuntador a uno de los bloques en la lista ligada circular. Este apuntador se denomina **memptr** y siempre apunta a un bloque que tiene algún espacio libre (por lo regular el último espacio por asignarse o liberarse). Es inicializado a **NULL**, pero en la primera llamada a **malloc** se ejecuta el código de inicialización que establece a **memptr** a principio del arreglo del apilamiento e inicializa el encabezado en el arreglo de la manera siguiente:

Este encabezado inicial que se asigna en la primera llamada a **malloc** nunca será liberado. Ahora hay un bloque en la lista, y el resto del código de **malloc** busca la lista y devuelve un nuevo bloque a partir del primer bloque que tenga suficiente espacio libre (esto es un algoritmo de **acceso primero**). De este modo, después de, digamos, tres llamadas a **malloc**, la lista tendría un aspecto como el siguiente:

Figura 7.19
Código en C para mantener un apilamiento de memoria contigua utilizando una lista de apuntadores hacia bloques tanto usados como libres

```
#define NULL 0
#define MEMSIZE 8096 /* cambio para tamaños diferentes */

typedef double Align;
typedef union header
  { struct { union header *next;
             unsigned usedsize;
             unsigned freesize;
           } s;
    Align a;
  } Header;

static Header mem[MEMSIZE];
static Header *memptr = NULL;

void *malloc(unsigned nbytes)
{ Header *p, *newp;
  unsigned nunits;
  nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1;
  if (memptr == NULL)
  { memptr->s.next = memptr = mem;
    memptr->s.usedsize = 1;
    memptr->s.freesize = MEMSIZE-1;
  }
  for(p=memptr;
      (p->s.next!=memptr) && (p->s.freesize<nunits);
      p=p->s.next);
  if (p->s.freesize < nunits) return NULL;
  /* no hay bloque suficientemente grande */
  newp = p+p->s.usedsize;
  newp->s.usedsize = nunits;
  newp->s.freesize = p->s.freesize - nunits;
  newp->s.next = p->s.next;
  p->s.freesize = 0;
  p->s.next = newp;
  memptr = newp;
  return (void *) (newp+1);
}

void free(void *ap)
{ Header *bp, *p, *prev;
  bp = (Header *) ap - 1;
  for (prev=memptr,p=memptr->s.next;
       (p!=bp) && (p!=memptr); prev=p,p=p->s.next);
  if (p!=bp) return;
  /* lista dañada, no hacer nada */
  prev->s.freesize += p->s.usedsize + p->s.freesize;
  prev->s.next = p->s.next;
  memptr = prev;
}
```

Advierta que a medida que los bloques se asignan en sucesión, se crea un nuevo bloque cada vez, y el espacio libre que deja el bloque anterior se va con él (de manera que el espacio libre del bloque desde el cual tiene lugar la asignación siempre tiene **freesize** establecido a cero). También, **memptr** sigue la construcción de los nuevos bloques, y así apunta siempre a un bloque con algo de espacio libre. Advierta también que **malloc** siempre incrementa el apuntador al bloque recientemente creado, de manera que el encabezado está protegido de ser sobreescrito mediante el programa cliente (mientras que en la memoria devuelta sólo se utilizan desplazamientos positivos).

Ahora consideremos el código para el procedimiento **free**. Primero decrementa el apuntador pasado por el usuario para encontrar el encabezado del bloque. Luego busca en la lista un apuntador que sea idéntico a éste, con lo que protege a la lista de la corrupción, y también permite que se calcule el apuntador al bloque anterior. Cuando se encuentra, el bloque es eliminado de la lista, y tanto su espacio utilizado como el libre se agregan al espacio libre del bloque anterior, fusionando así de manera automática el espacio libre. Observe que **memptr** también se establece para apuntar al bloque que contiene la memoria recién liberada.

Como ejemplo supongamos que se libera el bloque de en medio de los tres bloques utilizados en la ilustración anterior. Entonces el apilamiento y la lista de bloque asociada con él tendrían el aspecto siguiente:

7.4.4 Administración automática del apilamiento o montículo (heap)

El uso de **malloc** y **free** para realizar asignación y desasignación dinámica de apuntador es un método **manual** para la administración del apilamiento, puesto que el programador debe escribir llamadas explícitas para asignar y liberar memoria. En contraste, la pila de ejecución es administrada **automáticamente** mediante la secuencia de llamada. En un lenguaje que necesite un ambiente de ejecución completamente dinámico, el apilamiento debe ser administrado automáticamente de la misma manera. Por desgracia, mientras que las llamadas a **malloc** se pueden programar fácilmente en cada llamada de procedimiento, las llamadas a **free** no se pueden programar de manera similar al salir de escena, debido a que los registros de activación deben mantenerse hasta que hayan desaparecido todas las referencias a ellos. De este modo, la administración de memoria automática involucra el reclamo del almacenamiento previamente asignado pero ya no utilizado, tal vez tiempo después de que fue asignado, y sin una llamada explícita a **free**. Este proceso se denomina **recolección de basura**.

Reconocer cuando un bloque de almacenamiento ya no está referenciado, ya sea directa o indirectamente a través de apuntadores, es una tarea mucho más difícil que la de mantener una lista de bloques de almacenamiento en forma apilada. La técnica estándar consiste en realizar recolección de basura de **marcado y barrido**.[13] En este método no se libera memoria hasta que falla una llamada a **malloc**, en cuyo punto se activa un recolector de basura que

13. Una alternativa más simple llamada **conteo de referencia** se emplea también ocasionalmente. Véase la sección de notas y referencias.

busca todo el almacenamiento que puede ser referenciado y libera todo el almacenamiento no referenciado. El procedimiento se realiza en dos pasos. El primer paso sigue a todos los apuntadores de manera recursiva, comenzando con todos los valores de apuntador actualmente accesibles, y marca cada bloque de almacenamiento alcanzado. Este proceso requiere un bit de almacenamiento extra para el marcado. El segundo paso hace entonces un barrido lineal a través de la memoria, devolviendo los bloques no marcados a la memoria libre. Aunque este proceso por lo general encontrará suficiente memoria libre contigua para satisfacer una serie de nuevos requerimientos, es posible que la memoria esté todavía tan fragmentada que aún no pueda satisfacer un requerimiento grande de memoria, incluso después de que se haya realizado la recolección de basura. Por lo tanto, una recolección de basura por lo regular también realiza **compresión de memoria** moviendo todo el espacio asignado a un extremo del apilamiento, dejando sólo un bloque grande de espacio libre contiguo en el otro extremo. Este proceso también debe actualizar todas las referencias a aquellas áreas en la memoria que fueron desplazadas dentro del programa en ejecución.

La recolección de basura de marcado y barrido tiene varias desventajas: requiere de almacenamiento extra (para las marcas), y el doble paso a través de la memoria provoca un retardo importante en el procesamiento, en ocasiones tanto como algunos segundos, cada vez que se invoca el recolector de basura, lo que puede tomar algunos minutos. Esto evidentemente es inaceptable para muchas aplicaciones que involucran una respuesta inmediata o interactiva.

La administración de este proceso se puede mejorar dividiendo la memoria disponible en dos mitades y asignando almacenamiento sólo a partir de una mitad a la vez. Entonces, durante el paso de marcado, todos los bloques alcanzados son inmediatamente copiados a la segunda mitad del almacenamiento que no está en uso. Esto significa que no se requerirá un bit de marcado extra en el almacenamiento, y sólo se requerirá de un paso. También realiza la compresión de manera automática. Una vez que todos los bloques alcanzables en el área utilizada han sido copiados, la mitad utilizada y la no usada de la memoria son intercambiadas y el procesamiento continúa. Este método se denomina recolección de basura de **paro y copia** o de **espacio doble**. Desgraciadamente, no hace mucho por mejorar los retardos del procesamiento durante la petición de almacenamiento.

Recientemente, fue inventado un método que reduce este retardo en forma significativa. Denominado **recolección de basura generacional**, agrega un área de almacenamiento permanente al esquema de petición del párrafo anterior. Los objetos asignados que sobreviven tiempo suficiente simplemente son copiados en espacio permanente y nunca son desasignados durante las peticiones posteriores de almacenamiento. Esto significa que el recolector de basura necesita buscar sólo una sección muy pequeña de memoria para las asignaciones de almacenamiento más recientes, con lo que el tiempo para una búsqueda así se reduce a una fracción de segundo. Naturalmente, todavía es posible que la memoria permanente llegue a agotarse con el almacenamiento no alcanzable, pero esto es un problema mucho menos grave que antes, porque el almacenamiento temporal tiende a desaparecer con rapidez, mientras que el almacenamiento que permanece asignado durante algún tiempo tiende a permanecer de cualquier manera. Se ha demostrado que este proceso funciona muy bien, especialmente con un sistema de memoria virtual.

Remitimos al lector a las fuentes enumeradas en la sección de notas y referencias para más detalles acerca de éste y otros métodos de recolección de basura.

7.5 MECANISMOS DE PASO DE PARÁMETROS

Hemos visto cómo, en una llamada a procedimiento, los parámetros corresponden a ubicaciones en el registro de activación, las cuales son rellenadas con los argumentos, o valores de parámetros, mediante el elemento que llama, antes de saltar al código del procedimiento llamado. De este modo, para el código del procedimiento llamado, un parámetro representa un valor puramente formal al que no está ligado ningún código, pero que sirve sólo para establecer

una ubicación en el registro de activación, donde el código puede encontrar su valor final, el cual sólo existirá una vez que ha tenido lugar una llamada. En ocasiones se refiere al proceso de construir estos valores como a la **fijación** de los parámetros a los argumentos. Cómo interpreta el código del procedimiento los valores del argumento depende del **mecanismo o mecanismos de paso de parámetros** en particular adoptado por el lenguaje fuente. Como ya indicamos, FORTRAN77 adopta un mecanismo que fija los parámetros a ubicaciones en vez de a valores, mientras que C contempla todos los argumentos como valores. Otros lenguajes, como C++, Pascal y Ada, ofrecen una selección de mecanismos de paso de parámetros.

En esta sección comentaremos los dos mecanismos de paso de parámetros más comunes (el **paso por valor** y el **paso por referencia**, referidos a veces como *llamada por valor* y *llamada por referencia*), además de dos métodos adicionales importantes, el **paso por valor-resultado** y el **paso por nombre** (también denominado **evaluación retardada**). Algunas variaciones de éstos serán analizadas en los ejercicios.

Una cuestión que el mecanismo de paso de parámetros en sí no aborda es el orden en el que se evalúan los argumentos. En la mayoría de las situaciones este orden no es importante para la ejecución de un programa, y cualquier orden de evaluación producirá los mismos resultados. En ese caso, por eficiencia u otras razones, un compilador puede elegir variar el orden de la evaluación de los argumentos. Sin embargo, muchos lenguajes permiten argumentos para llamadas que causan efectos laterales (cambios en la memoria). Por ejemplo, la llamada de función en C

```
f(++x,x);
```

provoca un cambio en el valor de `x`, de modo que diferentes órdenes de evaluación tienen diferentes resultados. En tales lenguajes, un orden de evaluación estándar puede especificarse como el de izquierda a derecha, o puede dejarse al escritor del compilador, en cuyo caso el resultado de una llamada puede variar de una implementación a otra. Los compiladores de C por lo regular evalúan sus argumentos de derecha a izquierda, en vez de izquierda a derecha. Esto tiene en cuenta un número variable de argumentos (tal como en la función `printf`), como se comentó en la sección 7.3.1, página 361.

7.5.1 Paso por valor

En este mecanismo los argumentos son expresiones que son evaluadas en el momento de la llamada, y sus valores se convierten en los valores de los parámetros durante la ejecución del procedimiento. Éste es el único mecanismo de paso de parámetros disponible en C, y es el mecanismo predeterminado en Pascal y Ada (Ada también permite que tales parámetros sean explícitamente especificados como parámetros `in`).

En su forma más simple, esto significa que los parámetros de valor se comportan como valores constantes durante la ejecución de un procedimiento, y se puede interpretar el paso por valor como el reemplazo de todos los parámetros en el cuerpo de un procedimiento mediante los valores de los argumentos. Esta forma de paso por valor es empleada por Ada, donde tales parámetros no pueden ser asignados o utilizados de otro modo, por ejemplo como variables locales. Una visión más relajada es la de C y Pascal, donde los parámetros de valor son contemplados esencialmente como variables locales inicializadas, que se pueden utilizar como variables ordinarias, pero los cambios en ellas nunca provocan que tenga lugar algún cambio no local.

En un lenguaje como C, que ofrece sólo paso por valor, es imposible escribir directamente un procedimiento que tenga efecto haciendo cambios a sus parámetros. Por ejemplo, la siguiente función `inc2` escrita en C no logra el efecto deseado:

```
void inc2( int x)
/* ¡incorrecto! */
{ ++x;++x; }
```

Mientras que en teoría es posible, con una adecuada generalidad de funciones, realizar todos los cálculos devolviendo valores apropiados en vez de cambiando valores de parámetros, los lenguajes como C por lo regular ofrecen un método en el que se emplea el paso por valor de manera tal que obtenga cambios no locales. En C, esto toma la forma de pasar la dirección en lugar del valor (con lo que se cambia el tipo de datos del parámetro):

```
void inc2( int* x)
/* ahora está bien */
{ ++(*x);++(*x); }
```

Por supuesto, ahora, para incrementar una variable `y`, esta función debe ser llamada como `inc2(&y)`, ya que se requiere la dirección de `y` y no su valor.

Este método funciona especialmente bien en C para arreglos, ya que éstos son apuntadores de manera implícita, y de este modo el paso por valor permite que los elementos individuales del arreglo sean cambiados:

```
void init(int x[],int size)
/* esto funciona bien cuando se llama
   como init(a), donde a es un arreglo */
{ int i;
  for(i=0;i<size;++i) x[i]=0;
}
```

El paso por valor no requiere de esfuerzo especial por parte del compilador. Se implementa fácilmente tomando la visión más directa del cálculo del argumento y la construcción del registro de activación.

7.5.2 Paso por referencia

En el paso por referencia los argumentos deben (al menos en principio) ser variables con ubicaciones asignadas. En vez de pasar el valor de una variable, el paso por referencia pasa la ubicación de la variable, de manera que el parámetro se convierte en un **alias** para el argumento, y cualquier cambio que se haga al parámetro también se hace al argumento. En FORTRAN77, el paso por referencia es el único mecanismo de paso de parámetros. En Pascal, el paso por referencia se consigue utilizando la palabra reservada **var** y en C++ el símbolo especial `&` en la declaración de parámetro:

```
void inc2( int & x)
/* parámetro de referencia de C++ */
{ ++x;++x; }
```

Esta función ahora puede ser llamada sin un uso especial del operador de dirección: `inc2(y)` funciona bien.

El paso por referencia requiere que el compilador calcule la dirección del argumento (y debe tener tal dirección), la cual se almacena entonces en el registro de activación local.

El compilador también debe convertir los accesos locales a un parámetro de referencia en accesos indirectos, ya que el "valor" local es realmente la dirección en cualquier sitio del ambiente.

En lenguajes como FORTRAN77, donde sólo se dispone del paso por referencia, por lo regular se ofrece un acuerdo para argumentos que sean valores sin direcciones. En lugar de hacer una llamada como

```
p(2+3)
```

ilegal en FORTRAN77, un compilador debe "inventar" una dirección para la expresión `2 + 3`, calcular el valor en esta dirección, y posteriormente pasar la dirección a la llamada. Por lo general esto se realiza creando una localidad temporal en el registro de activación del elemento que llama (en FORTRAN77, éste será estático). Un ejemplo de esto se encuentra en el ejemplo 7.1 (página 350), donde el valor 3 se pasa como un argumento creando una localidad de memoria para el mismo en el registro de activación del procedimiento principal.

Un aspecto del paso por referencia es que no requiere que se haga una copia del valor pasado, a diferencia del paso por valor. Esto en ocasiones puede ser importante cuando el valor a ser copiado es una estructura grande (o un arreglo en otro lenguaje aparte de C o C++). En tal caso puede ser importante poder pasar un argumento por referencia, pero prohibir que se hagan cambios al valor del argumento, de esta manera se consigue el paso por valor sin el gasto de copiar el valor. Una opción así es proporcionada por C++, donde se puede escribir una llamada tal como

```
void f( const MuchData & x )
```

donde `MuchData` es un tipo de datos con una estructura grande. Esto todavía es paso por referencia, pero el compilador también debe realizar una verificación estática de que `x` nunca va a aparecer a la izquierda de una asignación o, de otro modo, puede ser cambiada.[14]

7.5.3 Paso por valor-resultado

Este mecanismo consigue un resultado similar al paso por referencia, sólo que no se establece un alias real: el valor del argumento es copiado y utilizado en el procedimiento, entonces el valor final del parámetro es copiado de vuelta a la ubicación del argumento cuando el procedimiento sale de escena. Así, este método se denomina a veces como de copia de entrada, copia de salida o copia de recuperación. Éste es el mecanismo del parámetro `in out` en Ada. (Ada también tiene simplemente un parámetro `out`, que no tiene valor inicial pasado; éste podría ser llamado paso por resultado.)

El paso por valor-resultado sólo es distinguible del paso por referencia en presencia de alias. Por ejemplo, en el código siguiente (en sintaxis de C),

```
void p(int x, int y)
{ ++x;
  ++y;
}
```

14. Esto no siempre puede hacerse de manera completamente segura.

```
main()
{ int a = 1;
  p(a,a);
  return 0;
}
```

a tiene el valor 3 después de que p es llamado si se utiliza el paso por referencia, mientras que **a** tiene el valor 2 si se emplea el paso por valor-resultado.

Cuestiones que se dejan sin especificar por este mecanismo, y que posiblemente difieran en lenguajes o implementaciones diferentes, son el orden en el que los resultados son copiados de vuelta a los argumentos y si las ubicaciones de los argumentos se calculan sólo a la entrada y se almacenan o si vuelven a calcularse a la salida.

Ada tiene otra peculiaridad: su definición establece que los parámetros `in out` pueden de hecho ser implementados como paso por referencia, y cualquier cálculo que fuera diferente bajo los dos mecanismos (lo que involucraría un alias) sería un error.

Desde el punto de vista del escritor de compiladores, el paso por valor-resultado requiere varias modificaciones a la estructura básica de la pila de ejecución y a la secuencia de llamada. En primer lugar, el registro de activación no puede ser liberado por el elemento llamado, puesto que los valores (locales) que serán copiados deben continuar disponibles para el elemento que llama. En segundo lugar, el elemento que llama debe insertar las direcciones de los argumentos como temporales en la pila antes de comenzar la construcción del nuevo registro de activación, o bien, debe volver a calcular estas direcciones al regreso del procedimiento llamado.

7.5.4 Paso por nombre

Éste es el más complejo de los mecanismos de paso de parámetros. También se le conoce como de **evaluación diferida**, puesto que la idea del paso por nombre es que el argumento no sea evaluado hasta su uso real (como un parámetro) en el programa llamado. De este modo, el nombre del argumento, o su representación textual en el punto de llamada, reemplaza al nombre del parámetro que le corresponde. Como ejemplo, en el código

```
void p(int x)
{ ++x; }
```

si se hace una llamada tal como `p(a[i])`, el efecto es el de evaluar `++(a[i])`. De esta forma, si **i** fuera a cambiar antes del uso de **x** dentro de **p**, el resultado sería diferente, ya sea del que se obtuviera mediante el paso por referencia o del que se obtuviera por medio del paso por valor-resultado. Por ejemplo, en el código (con sintaxis de C),

```
int i;
int a[10];

void p(int x)
{ ++i;
  ++x;
}
```

```
main()
{ i = 1;
  a[1] = 1;
  a[2] = 2;
  p(a[i]);
  return 0;
}
```

el resultado de la llamada a **p** es que **a[2]** se establece a 3 y **a[1]** se deja sin cambios.

La interpretación del paso por nombre es como sigue. El texto de un argumento en el punto de llamada es contemplado como una función por derecho propio, la cual es evaluada cada vez que el nombre de parámetro correspondiente se alcanza en el código del procedimiento llamado. Sin embargo, el argumento siempre será evaluado en el ambiente del elemento que llama, mientras que el procedimiento será ejecutado en su ambiente de definición.

El paso por nombre se ofreció como un mecanismo de paso del parámetro (junto con el paso por valor) en el lenguaje Algol60, pero perdió popularidad por tres razones. La primera es que puede dar resultados sorprendentes y contrarios a la intuición en la presencia de efectos colaterales (como lo muestra el ejemplo anterior). La segunda es que es difícil de implementar, puesto que cada argumento debe ser convertido en lo que es esencialmente un procedimiento (en ocasiones conocido como **suspensión** o **interrupción** [**"thunk"**]) que debe ser llamado siempre que el argumento es evaluado. La tercera es que es ineficiente, puesto que no sólo convierte una simple evaluación de argumento en una llamada de procedimiento, sino que también puede provocar que ocurran evaluaciones múltiples. Una variación de este mecanismo denominada **evaluación diferida** que se ha vuelto popular en los lenguajes puramente funcionales, consiste en evitar la reevaluación **memorizando** la suspensión con el valor calculado la primera vez que se llama. La evaluación diferida de hecho puede producir una implementación *más* eficiente, puesto que un argumento que no se utiliza nunca tampoco se evalúa nunca. Los lenguajes que ofrecen la evaluación diferida como un mecanismo de paso de parámetros son **Miranda** y **Haskell**. Remitimos al lector a la sección de notas y referencias para más información al respecto.

7.6 UN AMBIENTE DE EJECUCIÓN PARA EL LENGUAJE TINY

En esta sección final del capítulo, describiremos la estructura de un ambiente de ejecución para el lenguaje TINY, nuestro ejemplo ejecutable de un lenguaje pequeño y simple para compilación. Haremos esto aquí en una forma independiente de máquina y remitiremos al lector al siguiente capítulo para un ejemplo de una implementación en una máquina específica.

El ambiente que TINY necesita es mucho más simple que cualquiera de los ambientes comentados en este capítulo. En realidad, TINY no tiene procedimientos, y todas sus variables son globales, de manera que no hay necesidad de una pila o registros de activación, y el único almacenamiento dinámico necesario es el de temporales durante la evaluación de la expresión (éste incluso podría ser estático como en FORTRAN77: véanse los ejercicios).

Un esquema simple para un ambiente TINY es colocar las variables en direcciones absolutas en el extremo inferior de la memoria del programa, y asignar la pila temporal en el extremo superior. Así, dado un programa que utilice, digamos, cuatro variables **x**, **y**, **z** y **w**, estas variables obtendrían las direcciones absolutas del 0 hasta el 3 en la parte inferior de la memoria, y en un punto durante la ejecución donde se estén almacenando los tres temporales, el ambiente de ejecución se vería como sigue:

Dependiendo de la arquitectura, podemos necesitar establecer algunos registros de administración para apuntar a la parte inferior y/o superior de la memoria, para entonces utilizar las direcciones "absolutas" de las variables como desplazamientos desde el apuntador de la parte inferior, y emplear el apuntador de la parte superior de la memoria como el "tope (parte superior) de la pila oral", o bien, calcular los desplazamientos para los temporales desde un apuntador fijo en la parte superior. Naturalmente, también se podría utilizar la pila del procesador como la pila temporal, si está disponible.

Para implementar este ambiente de ejecución la tabla de símbolos en el compilador de TINY debe, como se describió en el último capítulo, mantener las direcciones de las variables en memoria. Se hace esto proporcionando un parámetro de ubicación en la función **st_insert** e incluyendo una función **st_lookup** que recupere la ubicación de una variable (apéndice B, líneas 1166 y 1171):

```
void st_insert( char * name, int lineno, int loc );
int st_lookup ( char * name );
```

El analizador semántico debe, a su vez, asignar direcciones a las variables cuando se hayan encontrado la primera vez. Esto se hace manteniendo un contador de ubicación de memoria estático que es inicializado para la primera dirección (apéndice B, línea 1413):

```
static int location = 0;
```

Entonces, en cualquier momento que se encuentre una variable (en una sentencia de lectura, una sentencia de asignación o una expresión de identificador), el analizador semántico ejecuta el código (apéndice B, línea 1454):

```
if (st_lookup(t->attr.name) == -1)
   st_insert(t->attr.name,t->lineno,location++);
else
   st_insert(t->attr.name,t->lineno,0);
```

Cuando **st_lookup** devuelve −1, la variable aún no está en la tabla. En ese caso, se registra una nueva ubicación, y el contador de ubicación se incrementa. De otro modo, la

variable ya está en la tabla, en cuyo caso la tabla de símbolos ignora el parámetro de ubicación (y se escribe 0 como una ubicación ficticia).

Esto maneja la asignación de las variables nombradas en un programa TINY; la asignación de las variables temporales en la parte superior de la memoria, y las operaciones necesarias para mantener esta asignación, serán responsabilidad del generador de código, el cual se analizará en el siguiente capítulo.

EJERCICIOS

7.1 Dibuje una posible organización para el ambiente de ejecución del siguiente programa en FORTRAN77, similar a la de la figura 7.2 (página 352). Asegúrese de incluir los apuntadores de memoria como se presentarían durante la llamada a **AVE**.

```
      REAL A(SIZE),AVE
      INTEGER N,I
   10 READ *, N
      IF (N.LE.0.OR.N.GT.SIZE) GOTO 99
      READ *,(A(I),I=1,N)
      PRINT *, 'AVE = ',AVE(A,N)
      GOTO 10
   99 CONTINUE
      END
      REAL FUNCTION AVE(B,N)
      INTEGER I,N
      REAL B(N),SUM
      SUM = 0.0
      DO 20 I=1,N
   20 SUM=SUM+B(I)
      AVE = SUM/N
      END
```

7.2 Dibuje una posible organización para el ambiente de ejecución del siguiente programa en C, similar a la de la figura 7.4 (página 354).

a. Después de la entrada en el bloque **A** en la función **f**.

b. Después de la entrada en el bloque **B** en la función **g**.

```
int a[10];
char * s = "hello";

int f(int i, int b[])
{ int j=i;
  A:{ int i=j;
      char c = b[i];
      ...
    }
  return 0;
}
```

```
void g(char * s)
{ char c = s[0];
  B:{ int a[5];
        ...
      }
}

main()
{ int x=1;
  x = f(x,a);
  g(s);
  return 0;
}
```

7.3 Dibuje una posible organización para el ambiente de ejecución del programa en C de la figura 4.1 (página 148) después de la segunda llamada a `factor`, dada la cadena de entrada `(2)`.

7.4 Dibuje la pila de registros de activación para el siguiente programa en Pascal, que muestre los vínculos de control y de acceso, después de la segunda llamada al procedimiento `c`. Describa cómo se tiene acceso a la variable `x` desde el interior de `c`.

```
program env;

procedure a;
var x: integer;

  procedure b;
    procedure c;
    begin
      x := 2;
      b;
    end;
  begin (* b *)
      c;
    end;

begin (* a *)
  b;
end;

begin (* main *)
  a;
end.
```

7.5 Dibuje la pila de registros de activación para el siguiente programa en Pascal

a. Después de la llamada a `a` en la primera llamada de `p`.

b. Después de la llamada a **a** en la segunda llamada de **p**.

c. ¿Qué imprime el programa y por qué?

```
program closureEx(output);
var x: integer;

procedure one;
begin
  writeln(x);
end;

procedure p(procedure a);
begin
  a;
end;

procedure q;
var x:integer;
      procedure two;
      begin
        writeln(x);
      end;
begin
  x := 2;
  p(one);
  p(two);
end; (* q *)

begin (* main *)
  x := 1;
  q;
end.
```

7.6 Considere el siguiente programa en Pascal. Suponga una entrada de usuario compuesta de los tres números 1, 2, 0 y dibuje la pila de registros de activación cuando el número 1 es impreso la primera vez. Incluya todos los vínculos de control y de acceso, además de todos los parámetros y variables globales, y suponga que todos los procedimientos son almacenados en el ambiente como cerraduras.

```
program procenv(input,output);

procedure dolist (procedure print);
var x: integer;
  procedure newprint;
  begin
    print;
    writeln(x);
  end;
```

```
begin (* dolist *)
  readln(x);
  if x = 0 then begin
    print;
    print;
  end
  else dolist(newprint);
end; (* dolist *)

procedure null;
begin
end;

begin (* main *)
  dolist(null);
end.
```

7.7 Para efectuar asignación completamente estática, un compilador FORTRAN 77 necesita constituir una estimación del número máximo de temporales requeridos para cualquier cálculo de expresión en un programa. Conciba un método para estimar el número de temporales requeridos para calcular una expresión realizando un recorrido del árbol de expresión. Suponga que las expresiones son evaluadas de izquierda a derecha y que cada subexpresión izquierda se debe guardar en un temporal.

7.8 En lenguajes que permiten números variables de argumentos en llamadas de procedimiento, una manera de encontrar el primer argumento es calcular los argumentos en orden inverso, como se describió en la sección 7.3.1, página 361.

a. Una alternativa para calcular los argumentos en reversa sería reorganizar el registro de activación para tener disponible el primer argumento incluso en la presencia de argumentos variables. Describa tal organización de registros de activación y la secuencia de llamada que necesitaría.

b. Otra alternativa para calcular los argumentos en reversa es emplear un tercer apuntador (aparte del sp y el fp), el cual por lo regular se denomina ap ("argument pointer", apuntador de argumento). Describa una estructura registro de activación que utilice un ap para hallar el primer argumento y la secuencia de llamada que necesitaría.

7.9 El texto describe cómo manejar los parámetros de longitud variable (tales como arreglos abiertos) que son pasados por valor (véase el ejemplo 7.6, página 362) y establece que una solución semejante funciona para variables locales de longitud variable. Sin embargo, existe un problema cuando se tienen presentes *tanto* parámetros de longitud variable como variables locales. Describa el problema y una solución utilizando el siguiente procedimiento de Ada como ejemplo:

```
type IntAr is Array(Integer range <>) of Integer;
...
procedure f(x: IntAr; n:Integer) is
y: Array(1..n) of Integer;
i: Integer;
begin
  ...
end f;
```

7.10 Una alternativa al encadenamiento de acceso en un lenguaje con procedimientos locales es mantener vínculos de acceso en un arreglo exterior de la pila, indicado por el nivel de anidación. Este arreglo se conoce como **despliegue** ("**display**"). Por ejemplo, la pila de ejecución de la figura 7.12 (página 369) se vería como sigue con un despliegue

mientras que la pila de ejecución de la figura 7.13 (página 370) tendría el aspecto que sigue:

a. Describa cómo un despliegue puede mejorar la eficiencia de referencias no locales de procedimientos profundamente anidados.

b. Vuelva a hacer el ejercicio 7.4 utilizando un despliegue.

c. Describa la secuencia de llamada necesaria para implementar un despliegue.

d. Existe un problema al utilizar un despliegue en un lenguaje con parámetros de procedimiento. Describa el problema utilizando el ejercicio 7.5.

7.11 Considere el siguiente procedimiento en sintaxis de C:

```
void f( char c, char s[10], double r )
{ int * x;
  int y[5];
  ...
}
```

a. Con las convenciones de paso de parámetros estándares de C, y suponiendo que los tamaños de datos son para enteros = 2 bytes, char = 1 byte, double = 8 bytes, dirección = 4 bytes, determine los desplazamientos desde el fp de los siguientes, utilizando la estructura de registros de activación descrita en este capítulo: 1) `c`, 2) `s[7]`, 3) `y[2]`.

b. Repita el inciso a suponiendo que todos los parámetros son pasados por valor (incluyendo arreglos).

c. Repita el inciso a suponiendo que todos los parámetros son pasados por referencia.

7.12 Ejecute el siguiente programa en C y explique su salida en términos del ambiente de ejecución:

```
#include <stdio.h>

void g(void)
{ {int x;
   printf("%d\n",x);
   x = 3;}
   {int y;
   printf("%d\n",y);}
}

int* f(void)
{ int x;
  printf("%d\n",x);
  return &x;
}

void main()
{ int *p;
  p = f();
  *p = 1;
  f();
  g();
}
```

7.13 Dibuje el diseño de la memoria de los objetos de las siguientes clases de C++, junto con las tablas de función virtual como se describieron en la sección 7.4.2:

```
class A
{ public:
  int a;
  virtual void f();
  virtual void g();
};
```

```
class B : public A
{ public:
  int b;
  virtual void f();
  void h();
};
class C : public B
{ public:
  int c;
  virtual void g();
}
```

7.14 Una tabla de función virtual en un lenguaje orientado a objetos ahorra recorrer la gráfica de herencia en la búsqueda de un método, pero tiene un costo. Explique cuál es este costo.

7.15 Proporcione la salida del siguiente programa (escrito con sintaxis de C) utilizando los cuatro métodos de paso de parámetros comentados en la sección 7.5:

```
#include <stdio.h>
int i=0;

void p(int x, int y)
{ x += 1;
  i += 1;
  y += 1;
}

main()
{ int a[2]={1,1};
  p(a[i],a[i]);
  printf("%d %d\n",a[0],a[1]);
  return 0;
}
```

7.16 Proporcione la salida del siguiente programa (con sintaxis de C) utilizando los cuatro métodos de paso de parámetros comentados en la sección 7.5:

```
#include <stdio.h>
int i=0;

void swap(int x, int y)
{ x = x + y;
  y = x - y;
  x = x - y;
}
```

```
main()
{ int a[3] = {1,2,0};
   swap(i,a[i]);
   printf("%d %d %d %d\n",i,a[0],a[1],a[2]);
   return 0;
}
```

7.17 Suponga que la subrutina `P` de FORTRAN77 está declarada como sigue

```
SUBROUTINE P(A)
INTEGER A
PRINT *, A
A = A + 1
RETURN
END
```

y que es llamada desde el programa principal de la manera siguiente:

```
CALL P(1)
```

En algunos sistemas FORTRAN77, esto ocasionará un error de ejecución. En otros, no ocurrirá un error de ejecución, pero si la subrutina se llama de nuevo con un 1 como su argumento, puede imprimir el valor 2. Explique cómo ambos comportamientos pueden ocurrir en términos del ambiente de ejecución.

7.18 Una variación del paso por nombre es el **paso por texto**, en el cual los argumentos son evaluados en modo retardado, justo como en el paso por nombre, pero cada argumento es evaluado en el ambiente del procedimiento llamado en vez de hacerlo en el ambiente que llama.

a. Muestre que el paso por texto puede tener diferentes resultados que el paso por nombre.

b. Describa una organización de ambiente de ejecución y secuencia de llamada que podrían utilizarse para implementar el paso por texto.

EJERCICIOS DE PROGRAMACIÓN

7.19 Como se describió en la sección 7.5, el paso por nombre, o evaluación retardada, puede ser visto como el empacamiento de un argumento en un cuerpo de función (o suspensión), el cual es llamado cada vez que el parámetro aparece en el código. Vuelva a escribir el código en C del ejercicio 7.16 para implementar los parámetros de la función `swap` de esta manera, y verifique que el resultado es en realidad equivalente al paso por nombre.

7.20 **a.** Como se describió en la sección 7.5.4, se puede conseguir un mejoramiento en la eficiencia del paso por nombre al memorizar el valor de un argumento la primera vez que es evaluado. Vuelva a escribir su código del ejercicio anterior para implementar dicha memorización, y compare los resultados con los de ese ejercicio.

b. La memorización puede provocar diferentes resultados del paso por nombre. Explique cómo puede ocurrir esto.

7.21 La compresión (sección 7.4.4) puede hacerse en un paso separado de la recolección de basura y puede ser realizada mediante `malloc` si una petición de memoria falla debido a la carencia de un bloque suficientemente largo.

a. Vuelva a escribir el procedimiento `malloc` de la sección 7.4.3 para incluir un paso de compresión.

b. La compresión requiere que la ubicación del espacio previamente asignado cambie, y esto significa que un programa debe informarse acerca del cambio. Describa cómo utilizar una tabla de apuntadores a bloques de memoria para resolver ese problema, y vuelva a escribir su código del inciso a para incluirla.

NOTAS Y REFERENCIAS

El ambiente completamente estático de FORTRAN77 (y versiones más antiguas de FORTRAN) representa un enfoque directo y natural para el diseño de ambientes y es similar a los ambientes de ensamblador. Los ambientes basados en pila se hicieron populares con la inclusión de la recursividad en lenguajes tales como Algol60 (Naur [1963]). Randell y Russell [1964] describen un temprano ambiente basado en pila de Algol60 con detalle. La organización del registro de activación y la secuencia de llamada para algunos compiladores de C se describe en Johnson y Ritchie [1981]. El uso de un despliegue en lugar de cadenas de acceso (ejercicio 7.10) se describe con detalle en Fischer y LeBlanc [1991], incluyendo los problemas al utilizarlo en un lenguaje con parámetros de procedimiento.

La administración de memoria dinámica se analiza en muchos libros acerca de estructuras de datos, tal como Aho, Hopcroft y Ullman [1983]. Una útil perspectiva general reciente se ofrece en Drozdek y Simon [1995]. El código para implementación de `malloc` y `free` que es semejante, aunque ligeramente menos sofisticado, al código dado en la sección 7.4.3, aparece en Kernighan y Ritchie [1988]. El diseño de una estructura de apilamiento para su uso en la compilación se comenta en Fraser y Hanson [1995].

Un panorama general de la colección de basura se presenta en Wilson [1992] o Cohen [1981]. Un colector de basura generacional y ambiente de tiempo de corrida para el ML del lenguaje funcional se describe en Appel [1992]. El compilador de lenguaje funcional Gofer (Jones [1984]) contiene tanto el colector de basura de marcar y barrer como el de dos espacios.

Budd [1987] describe un ambiente completamente dinámico para un pequeño sistema Smalltalk, incluyendo el uso de la gráfica de herencia, y un recolector de basura con conteos de referencia. El uso de una tabla de función virtual en C++ se describe en Ellis y Stroustrup [1990], junto con extensiones para manipular herencia múltiple.

Más ejemplos de técnicas de paso de parámetros pueden hallarse en Louden [1993], donde también se puede encontrar una descripción de la evaluación diferida. Las técnicas de implementación para la evaluación diferida pueden estudiarse en Peyton Jones [1987].

Capítulo 8

Generación de código

8.1 Código intermedio y estructuras de datos para generación de código

8.2 Técnicas básicas de generación de código

8.3 Generación de código de referencias de estructuras de datos

8.4 Generación de código de sentencias de control y expresiones lógicas

8.5 Generación de código de llamadas de procedimientos y funciones

8.6 Generación de código en compiladores comerciales: dos casos de estudio

8.7 TM: una máquina objetivo simple

8.8 Un generador de código para el lenguaje TINY

8.9 Una visión general de las técnicas de optimización de código

8.10 Optimizaciones simples para el generador de código de TINY

En este capítulo volveremos a la tarea final de un compilador, la de generar código ejecutable para una máquina objetivo que sea una fiel representación de la semántica del código fuente. La generación de código es la fase más compleja de un compilador, puesto que no sólo depende de las características del lenguaje fuente sino también de contar con información detallada acerca de la arquitectura objetivo, la estructura del ambiente de ejecución y el sistema operativo que esté corriendo en la máquina objetivo. La generación de código por lo regular implica también algún intento por **optimizar**, o mejorar, la velocidad y/o el tamaño del código objetivo recolectando más información acerca del programa fuente y adecuando el código generado para sacar ventaja de las características especiales de la máquina objetivo, tales como registros, modos de direccionamiento, distribución y memoria caché.

Debido a la complejidad de la generación del código, un compilador por lo regular divide esta fase en varios pasos, los cuales involucran varias estructuras de datos intermedias, y a menudo incluyen alguna forma de código abstracto denominada **código intermedio**. Un compilador también puede detener en breve la generación de código ejecutable real pero, en vez de esto genera alguna forma de código ensamblador que debe ser procesado adicionalmente por un ensamblador, un ligador y un cargador, los cuales pueden ser proporcionados por el sistema operativo o compactados con el compilador. En este capítulo nos concentraremos únicamente en los fundamentos de la generación del código intermedio y el código ensamblador, los cuales tienen muchas características en común. Ignoraremos el problema del procesamiento adicional del código ensamblador

en código ejecutable, el cual puede ser controlado más adecuadamente mediante un lenguaje ensamblador o sistemas de texto de programación.

En la primera sección de este capítulo analizaremos dos formas populares de código intermedio, el código de tres direcciones y el código P, y comentaremos algunas de sus propiedades. En la segunda sección describiremos los algoritmos básicos que permiten generar código intermedio o ensamblador. En secciones posteriores se analizarán técnicas de generación de código para varias características del lenguaje, incluyendo expresiones, sentencias de asignación y sentencias de flujo de control, tales como sentencias if y sentencias while y llamadas a procedimiento/función. A estas secciones les seguirán estudios de caso del código producido para estas características mediante dos compiladores comerciales: el compilador C de Borland para la arquitectura 80×86 y el compilador C de Sun para la arquitectura RISC de Sparc.

En una sección posterior aplicaremos las técnicas que permiten desarrollar un generador de código ensamblador para el lenguaje TINY estudiadas hasta aquí. Como la generación de código en este nivel de detalle requiere una máquina objetivo real, primero analizaremos una arquitectura objetivo simple y un simulador de máquina denominado TM, para el cual se proporciona un listado fuente en el apéndice C. A continuación describiremos el generador de código complejo para TINY. Finalmente, daremos una visión general de las técnicas para el mejoramiento u optimización del código estándar, y describiremos cómo algunas de las técnicas recién adquiridas pueden incorporarse en el generador de código de TINY.

8.1 CÓDIGO INTERMEDIO Y ESTRUCTURAS DE DATOS PARA GENERACIÓN DE CÓDIGO

Una estructura de datos que representa el programa fuente durante la traducción se denomina **representación intermedia**, o **IR** (por las siglas del término en inglés) para abreviar. En este texto hasta ahora hemos usado un árbol sintáctico abstracto como el IR principal. Además del IR, la principal estructura de datos utilizada durante la traducción es la tabla de símbolos, la cual se estudió en el capítulo 6.

Aunque un árbol sintáctico abstracto es una representación adecuada del código fuente, incluso para la generación de código (como veremos en una sección posterior), no se parece ni remotamente al código objetivo, en particular en su representación de construcciones de flujo de control, donde el código objetivo, como el código de máquina o código ensamblador, emplean saltos más que construcciones de alto nivel, como las sentencias if y while. Por lo tanto, un escritor de compiladores puede desear generar una nueva forma de representación intermedia del árbol sintáctico que se parezca más al código objetivo o reemplace del todo al árbol sintáctico mediante una representación intermedia de esa clase, y entonces genere código objetivo de esta nueva representación. Una representación intermedia de esta naturaleza que se parece al código objetivo se denomina **código intermedio**.

El código intermedio puede tomar muchas formas: existen casi tantos estilos de código intermedio como compiladores. Sin embargo, todos representan alguna forma de **linealización** del árbol sintáctico, es decir, una representación del árbol sintáctico en forma secuencial. El código intermedio puede ser de muy alto nivel, representar todas las operaciones de manera casi tan abstracta como un árbol sintáctico, o parecerse mucho al código objetivo. Puede o no utilizar información detallada acerca de la máquina objetivo y el ambiente de ejecución, como los tamaños de los tipos de datos, las ubicaciones de las variables y la disponibilidad de los registros. Puede o no incorporar toda la información contenida en la tabla

de símbolos, tal como los ámbitos, los niveles de anidación y los desplazamientos de las variables. Si lo hace, entonces la generación de código objetivo puede basarse sólo en el código intermedio; si no, el compilador debe retener la tabla de símbolos para la generación del código objetivo.

El código intermedio es particularmente útil cuando el objetivo del compilador es producir código muy eficiente, ya que para hacerlo así se requiere una cantidad importante del análisis de las propiedades del código objetivo, y esto se facilita mediante el uso del código intermedio. En particular, las estructuras de datos adicionales que incorporan información de un detallado análisis posterior al análisis sintáctico se pueden generar fácilmente a partir del código intermedio, aunque no es imposible hacerlo directamente desde el árbol sintáctico.

El código intermedio también puede ser útil al hacer que un compilador sea más fácilmente redirigible: si el código intermedio es hasta cierto punto independiente de la máquina objetivo, entonces generar código para una máquina objetivo diferente sólo requiere volver a escribir el traductor de código intermedio a código objetivo, y por lo regular esto es más fácil que volver a escribir todo un generador de código.

En esta sección estudiaremos dos formas populares de código intermedio: el **código de tres direcciones** y el **código P**. Ambos se presentan en muchas formas diferentes, y nuestro estudio aquí se enfocará sólo en las características generales, en lugar de presentar una descripción detallada de una versión de cada uno. Tales descripciones se pueden encontrar en la literatura que se describe en la sección de notas y referencias al final del capítulo.

8.1.1 Código de tres direcciones

La instrucción básica del código de tres direcciones está diseñada para representar la evaluación de expresiones aritméticas y tiene la siguiente forma general:

```
x = y op z
```

Esta instrucción expresa la aplicación del operador *op* a los valores de **y** y **z**, y la asignación de este valor para que sea el nuevo valor de **x**. Aquí *op* puede ser un operador aritmético como + o − o algún otro operador que pueda actuar sobre los valores de **y** y **z**.

El nombre "código de tres direcciones" viene de esta forma de instrucción, ya que por lo general cada uno de los nombres **x**, **y** y **z** representan una dirección de la memoria. Sin embargo, observe que el uso de la dirección de **x** difiere del uso de las direcciones de **y** y **z**, y que tanto **y** como **z** (pero no **x**) pueden representar constantes o valores de literales sin direcciones de ejecución.

Para ver cómo las secuencias de código de tres direcciones de esta forma pueden representar el cálculo de una expresión, considere la expresión aritmética

```
2*a+(b-3)
```

con árbol sintáctico

El código de tres direcciones correspondiente es

```
t1 = 2 * a
t2 = b - 3
t3 = t1 + t2
```

El código de tres direcciones requiere que el compilador genere nombres para elementos temporales, a los que en este ejemplo hemos llamado `t1`, `t2` y `t3`. Estos elementos temporales corresponden a los nodos interiores del árbol sintáctico y representan sus valores calculados, con el último elemento temporal (`t3`, en este ejemplo) representando el valor de la raíz.[1] La manera en que estos elementos temporales finalmente son asignados en la memoria no es especificada por este código; por lo regular serán asignados a registros, pero también se pueden conservar en registros de activación (véase el análisis de la pila temporal en el capítulo anterior).

El código de tres direcciones que se acaba de dar representa una linealización de izquierda a derecha del árbol sintáctico, debido a que el código correspondiente a la evaluación del subárbol izquierdo de la raíz se lista primero. Es posible que un compilador desee utilizar un orden diferente en ciertas circunstancias. Aquí simplemente advertimos que puede haber otro orden para este código de tres direcciones, a saber (con un significado diferente para los elementos temporales),

```
t1 = b - 3
t2 = 2 * a
t3 = t2 + t1
```

Evidentemente, la forma del código de tres direcciones que hemos mostrado no basta para representar todas las características ni siquiera del lenguaje de programación más pequeño. Por ejemplo, los operadores unitarios, como la negación, requieren de una variación del código de tres direcciones que contenga sólo dos direcciones, tal como

```
t2 = - t1
```

Si se desea tener capacidad para todas las construcciones de un lenguaje de programación estándar, será necesario variar la forma del código de tres direcciones en cada construcción. Si un lenguaje contiene características poco habituales, puede ser necesario incluso inventar nuevas formas del código de tres direcciones para expresarlas. Ésta es una de las razones por las que no existe una forma estándar para el código de tres direcciones (del mismo modo que no existe una forma estándar para árboles sintácticos).

En las siguientes secciones de este capítulo trataremos algunas construcciones comunes de lenguaje de programación de manera individual y mostraremos cómo estas construcciones son traducidas por lo común como código de tres direcciones. Sin embargo, para acostumbrarnos a lo que nos espera presentamos aquí un ejemplo completo en el que se utiliza el lenguaje TINY presentado anteriormente.

Considere el programa de muestra TINY de la sección 1.7 (capítulo 1) que calcula el máximo común divisor de dos enteros, el cual repetimos en la figura 8.1. El código de tres direcciones de muestra para este programa se ofrece en la figura 8.2. Este código contiene

1. Los nombres `t1`, `t2`, y así sucesivamente están sólo destinados a ser representativos del estilo general de tal código. De hecho, los nombres temporales en código de tres direcciones deben ser distintos de cualquier nombre que pudiera ser utilizado en el código fuente real, si se van a mezclar los nombres de código fuente, como ocurre aquí.

Figura 8.1
Programa de muestra TINY

```
{ Programa de muestra
  en lenguaje TINY--
  calcula el factorial
}
read x; { introducir un entero }
if 0 < x then { no calcular si x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { salida del factorial de x }
end
```

Figura 8.2
Código de tres direcciones para el programa TINY de la figura 8.1

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

varias formas diferentes de código de tres direcciones. En primer lugar, las operaciones integradas de entrada y salida `read` y `write` se tradujeron directamente en instrucciones de una dirección. En segundo lugar, existe una instrucción de salto condicional `if_false` que se utiliza para traducir tanto sentencias if como sentencias repeat y que contiene dos direcciones: el valor condicional que se probará y la dirección de código a la que se saltará. Las posiciones de las direcciones de salto también se indican mediante instrucciones (de una dirección) `label`. Estas instrucciones `label` pueden ser innecesarias, dependiendo de las estructuras de datos empleadas para implementar el código de tres direcciones. En tercer lugar, una instrucción `halt` (sin direcciones) sirve para marcar el final del código.

Finalmente, observamos que las asignaciones en el código fuente producen la generación de **instrucciones de copia** de la forma

```
x = y
```

Por ejemplo, la sentencia de programa de muestra

```
fact := fact * x;
```

se traduce en las dos instrucciones del código de tres direcciones

```
t2 = fact * x
fact = t2
```

aunque sería suficiente una instrucción de tres direcciones. Esto sucede por razones técnicas que se explicarán en la sección 8.2.

8.1.2 Estructuras de datos para la implementación del código de tres direcciones

El código de tres direcciones por lo regular no está implementado en forma textual como lo hemos escrito (aunque podría estarlo). En vez de eso, cada instrucción de tres direcciones está implementada como una estructura de registro que contiene varios campos, y la secuencia completa de instrucciones de tres direcciones está implementada como un arreglo o lista ligada, la cual se puede conservar en la memoria o escrita para (y leída desde) archivos temporales cuando sea necesario.

La implementación más común consiste en implementar el código de tres direcciones fundamentalmente como se muestra, lo que significa que son necesarios cuatro campos: uno para la operación y tres para las direcciones. Para las instrucciones que necesitan menos de tres direcciones, uno o más de los campos de dirección proporcionan un valor nulo o "vacío"; la selección de los campos depende de la implementación. Como son necesarios cuatro campos, una representación de código de tres direcciones de esta naturaleza se denomina **cuádruple**. Una posible implementación cuádruple del código de tres direcciones de la figura 8.2 se proporciona en la figura 8.3, donde se escribieron los cuádruples en notación matemática de "tuplas".

Los `typedef` posibles de C para implementar los cuádruples mostrados en la figura 8.3 se ofrecen en la figura 8.4. En estas definiciones permitimos que una dirección sea sólo una constante entera o una cadena (que representa el nombre de un elemento temporal o una variable). También, puesto que se utilizan nombres, éstos deben introducirse en una tabla de símbolos, y se necesitará realizar búsquedas durante procesamiento adicional. Una alternativa para conservar los nombres en los cuádruples es mantener apuntadores hacia entradas de la tabla de símbolos. Esto evita tener que realizar búsquedas adicionales y es particularmente ventajoso en un lenguaje con ámbitos anidados, donde se necesita más información de ámbito que sólo el nombre para realizar una búsqueda. Si también se introducen constantes en la tabla de símbolos, entonces no es necesaria una unión en el tipo de datos `Address`.

Una implementación diferente del código de tres direcciones consiste en utilizar las instrucciones mismas para representar los elementos temporales. Esto reduce la necesidad de campos de dirección de tres a dos, puesto que en una instrucción de tres direcciones que contiene la totalidad de las tres direcciones, la dirección objetivo es siempre un elemento temporal.[2] Una implementación del código de tres direcciones de esta clase se denomina **triple** y requiere que cada instrucción de tres direcciones sea referenciable, ya sea como

2. Esto no es una verdad inherente acerca del código de tres direcciones, pero puede asegurarse mediante la implementación. Por ejemplo, es verdadero en el código de la figura 8.2 (véase también la figura 8.3).

Figura 8.3
Implementación cuádruple para el código de tres direcciones de la figura 8.2

```
(rd,x,_,_)
(gt,x,0,t1)
(if_f,t1,L1,_)
(asn,1,fact,_)
(lab,L2,_,_)
(mul,fact,x,t2)
(asn,t2,fact,_)
(sub,x,1,t3)
(asn,t3,x,_)
(eq,x,0,t4)
(if_f,t4,L2,_)
(wri,fact,_,_)
(lab,L1,_,_)
(halt,_,_,_)
```

Figura 8.4
Código C que define estructuras de datos posibles para los cuádruples de la figura 8.3

```
typedef enum {rd,gt,if_f,asn,lab,mul,
                    sub,eq,wri,halt,. . .} OpKind;
typedef enum {Empty,IntConst,String} AddrKind;
typedef struct
            { AddrKind kind;
              union
              { int val;
                char * name;
              } contents;
            } Address;
typedef struct
            { OpKind op;
              Address addr1,addr2,addr3;
            } Quad;
```

un índice en un arreglo o como un apuntador en una lista ligada. Por ejemplo, una representación abstracta de una implementación del código de tres direcciones de la figura 8.2 como triples se proporciona en la figura 8.5 (página 404). En esa figura utilizamos un sistema de numeración que correspondería a índices de arreglo para representar los triples. Las referencias de triple también se distinguen de las constantes poniéndolas entre paréntesis en los mismos triples. Adicionalmente, en la figura 8.5 eliminamos las instrucciones `label` y las reemplazamos con referencias a los índices de triple mismos.

Los triples son una manera eficiente para representar el código de tres direcciones, ya que la cantidad de espacio se reduce y el compilador no necesita generar nombres para elementos temporales. Sin embargo, los triples tienen una importante desventaja, y ésta consiste en que, si se los representa mediante índices de arreglo, entonces cualquier movimiento de sus posiciones se vuelve difícil. Por otra parte, una representación de lista ligada no sufre de esta deficiencia. Problemas adicionales que involucran triples, y el código C apropiado para la definición de los mismos, se dejan como ejercicios.

Figura 8.5
Una representación del código de tres direcciones de la figura 8.2 como triples

```
(0)    (rd,x,_)
(1)    (gt,x,0)
(2)    (if_f,(1),(11))
(3)    (asn,1,fact)
(4)    (mul,fact,x)
(5)    (asn,(4),fact)
(6)    (sub,x,1)
(7)    (asn,(6),x)
(8)    (eq,x,0)
(9)    (if_f,(8),(4))
(10)   (wri,fact,_)
(11)   (halt,_,_)
```

8.1.3 Código P

El código P comenzó como un código ensamblador objetivo estándar producido por varios compiladores Pascal en la década de 1970 y principios de la de 1980. Fue diseñado para ser el código real de una máquina de pila hipotética, denominada **máquina P**, para la que fue escrito un intérprete en varias máquinas reales. La idea era hacer que los compiladores de Pascal se transportaran fácilmente requiriendo sólo que se volviera a escribir el intérprete de la máquina P para una nueva plataforma. El código P también ha probado ser útil como código intermedio, y se han utilizado varias extensiones y modificaciones del mismo en diversos compiladores de código nativo, la mayor parte para lenguajes tipo Pascal.

Como el código P fue diseñado para ser directamente ejecutable, contiene una descripción implícita de un ambiente de ejecución particular que incluye tamaños de datos, además de mucha información específica para la máquina P, que se debe conocer si se desea que un programa de código P sea comprensible. Para evitar este detalle describiremos aquí una versión simplificada y abstracta de código P apropiada para la exposición. Las descripciones de diversas versiones de código P real podrán encontrarse en varias referencias enumeradas al final del capítulo.

Para nuestros propósitos la máquina P está compuesta por una memoria de código, una memoria de datos no especificada para variables nombradas y una pila para datos temporales, junto con cualquier registro que sea necesario para mantener la pila y apoyar la ejecución.

Como un primer ejemplo de código P, considere la expresión

```
2*a+(b-3)
```

empleada en la sección 8.1.1, cuyo árbol sintáctico aparece en la página 399. Nuestra versión de código P para esta expresión es la que se muestra en seguida:

```
ldc 2        ; carga la constante 2
lod a        ; carga el valor de la variable a
mpi          ; multiplicación entera
lod b        ; carga el valor de la variable b
ldc 3        ; carga la constante 3
sbi          ; sustracción o resta entera
adi          ; adición de enteros
```

Estas instrucciones se ven como si representaran las siguientes operaciones en una máquina P. En primer lugar, **ldc 2** inserta el valor 2 en la pila temporal. Luego, **lod a** inserta el valor de la variable **a** en la pila. La instrucción **mpi** extrae estos dos valores de la pila, los multiplica (en orden inverso) e inserta el resultado en la pila. Las siguientes dos instrucciones (**lod b** y **ldc 3**) insertan el valor de **b** y la constante 3 en la pila (ahora tenemos tres valores en la pila). Posteriormente, la instrucción **sbi** extrae los dos valores superiores de la pila, resta el primero del segundo, e inserta el resultado. Finalmente, la instrucción **adi** extrae los dos valores restantes de la pila, los suma e inserta el resultado. El código finaliza con un solo valor en la pila, que representa el resultado del cálculo.

Como un segundo ejemplo introductorio, considere la sentencia de asignación

```
x := y + 1
```

Esto corresponde a las siguientes instrucciones en código P:

```
lda x        ; carga dirección de x
lod y        ; carga valor de y
ldc 1        ; carga constante 1
adi          ; suma
sto          ; almacena tope a dirección
             ; debajo del tope y extrae ambas
```

Advierta cómo este código calcula primero la dirección de **x**, luego el valor de la expresión que será asignada a **x**, y finalmente ejecuta un comando **sto**, el cual requiere que se encuentren dos valores en la parte superior de la pila temporal: el valor que será almacenado y, debajo de éste, la dirección en la memoria variable en la cual se almacenará. La instrucción **sto** también extrae estos dos valores (dejando la pila vacía en este ejemplo). De este modo el código P hace una distinción entre direcciones de carga (**lda**) y valores ordinarios (**lod**), que corresponden a la diferencia entre el uso de **x** en el lado izquierdo y el uso de **y** en el lado derecho de la asignación **x:=y+1**.

Como último ejemplo de código P en esta sección damos una traducción de código P en la figura 8.6 para el programa TINY de la figura 8.1, junto con comentarios que describen cada operación.

El código P de la figura 8.6 (página 406) contiene varias instrucciones nuevas de código P. En primer lugar, las instrucciones **rdi** y **wri** (sin parámetros) implementan las sentencias enteras **read** y **write** construidas en TINY. La instrucción de código P **rdi** requiere que la dirección de la variable cuyo valor va a leerse se encuentre en la parte superior de la pila, y esta dirección es extraída como parte de la instrucción. La instrucción **wri** requiere que el valor que será escrito esté en la parte superior de la pila, y este valor se extrae como parte de la instrucción. Otras instrucciones de código P que aparecen en la figura 8.6 que aún no se han comentado son: la instrucción **lab**, que define la posición de un nombre de etiqueta; la instrucción **fjp** ("false jump"), que requiere un valor booleano en el tope o parte superior de la pila (el cual es extraído); la instrucción **sbi** (de "sustracción entera", en inglés), cuya operación es semejante a la de otras instrucciones aritméticas; y las operaciones de comparación **grt** (de "greater than") y **equ** ("equal to"), que requieren dos valores enteros en el tope de la pila (los cuales son extraídos), y que insertan sus resultados booleanos. Finalmente, la instrucción **stp** ("stop") correspondiente a la instrucción **halt** del anterior código de tres direcciones.

Figura 8.6
Código P para el programa TINY de la figura 8.1

```
lda x        ; carga dirección de x
rdi          ; lee un entero, almacena a la
             ; dirección en el tope de la pila (y la extrae)
lod x        ; carga el valor de x
ldc 0        ; carga la constante 0
grt          ; extrae y compara dos valores del tope
             ; inserta resultado Booleano
fjp L1       ; extrae resultado Booleano, salta a L1 si es falso
lda fact     ; carga dirección de fact
ldc 1        ; carga la constante 1
sto          ; extrae dos valores, almacenando el primero en la
             ; dirección representada por el segundo
lab L2       ; definición de etiqueta L2
lda fact     ; carga dirección de fact
lod fact     ; carga valor de fact
lod x        ; carga valor de x
mpi          ; multiplica
sto          ; almacena el tope a dirección del segundo y extrae
lda x        ; carga dirección de x
lod x        ; carga valor de x
ldc 1        ; carga constante 1
sbi          ; resta
sto          ; almacena (como antes)
lod x        ; carga valor de x
ldc 0        ; carga constante 0
equ          ; prueba de igualdad
fjp L2       ; salto a L2 si es falso
lod fact     ; carga valor de fact
wri          ; escribe tope de pila y extrae
lab L1       ; definición de etiqueta L1
stp
```

Comparación del código P con el código de tres direcciones El código P en muchos aspectos está más cercano al código de máquina real que al código de tres direcciones. Las instrucciones en código P también requieren menos direcciones: todas las instrucciones que hemos visto son instrucciones de "una dirección" o "cero direcciones". Por otra parte, el código P es menos compacto que el código de tres direcciones en términos de números de instrucciones, y el código P no está "autocontenido" en el sentido que las instrucciones funcionen implícitamente en una pila (y las localidades de pila implícitas son de hecho las direcciones "perdidas"). La ventaja respecto a la pila es que contiene todos los valores temporales necesarios en cada punto del código, y el compilador no necesita asignar nombres a ninguno de ellos, como en el código de tres direcciones.

Implementación del código P Históricamente, el código P ha sido en su mayor parte generado como un archivo del texto, pero las descripciones anteriores de las implementaciones de estructura de datos internas para el código de tres direcciones (cuádruples y triples) también funcionarán con una modificación apropiada para el código P.

8.2 TÉCNICAS BÁSICAS DE GENERACIÓN DE CÓDIGO

En esta sección comentaremos los enfoques básicos para la generación de código en general, mientras que en secciones posteriores abordaremos la generación de código para construcciones de lenguaje individuales por separado.

8.2.1 Código intermedio o código objetivo como un atributo sintetizado

La generación de código intermedio (o generación de código objetivo directa sin código intermedio) se puede ver como un cálculo de atributo similar a muchos de los problemas de atributo estudiados en el capítulo 6. En realidad, si el código generado se ve como un atributo de cadena (con instrucciones separadas por caracteres de retorno de línea), entonces este código se convierte en un atributo sintetizado que se puede definir utilizando una gramática con atributos, y generado directamente durante el análisis sintáctico o mediante un recorrido postorden del árbol sintáctico.

Para ver cómo el código de tres direcciones, o bien, el código P se pueden definir como un atributo sintetizado, considere la siguiente gramática que representa un pequeño subconjunto de expresiones en C:

exp → **id** = *exp* | *aexp*
aexp → *aexp* + *factor* | *factor*
factor → (*exp*) | **num** | **id**

Esta gramática sólo contiene dos operaciones, la asignación (con el símbolo =) y la adición (con el símbolo +).[3] El token **id** representa un identificador simple, y el token **num** simboliza una secuencia simple de dígitos que representa un entero. Se supone que ambos tokens tienen un atributo *strval* previamente calculado, que es el valor de la cadena, o lexema, del token (por ejemplo, "42" para un **num** o "xtemp" para un **id**).

Código P Consideraremos el caso de la generación del código P en primer lugar, ya que la gramática con atributos es más simple debido a que no es necesario generar nombres para elementos temporales. Sin embargo, la existencia de asignaciones incrustadas es un factor que implica complicaciones. En esta situación deseamos mantener el valor almacenado como el valor resultante de una expresión de asignación, ya que la instrucción de código P estándar `sto` es destructiva, pues el valor asignado se pierde. (Con esto el código P muestra sus orígenes de Pascal, en el cual no existen las asignaciones incrustadas.) Resolvemos este problema introduciendo una instrucción de **almacenamiento no destructiva** `stn` en nuestro código P, la que como la instrucción `sto`, supone que se encuentra un valor en la parte superior o tope de la pila y una dirección debajo de la misma; `stn` almacena el valor en la dirección pero deja el valor en el tope de la pila, mientras que descarta la dirección. Con esta nueva instrucción, una gramática con atributos para un atributo de cadena de código

3. La asignación en este ejemplo tiene la semántica siguiente: `x = e` almacena el valor de `e` en `x` y tiene el mismo valor resultante que `e`.

P se proporciona en la tabla 8.1. En esa figura utilizamos el nombre de atributo *pcode* para la cadena de código P. También empleamos dos notaciones diferentes para la concatenación de cadena: ++ cuando las instrucciones van a ser concatenadas con retornos de línea insertados entre ellas y || cuando se está construyendo una instrucción simple y se va a insertar un espacio.

Dejamos al lector describir el cálculo del atributo *pcode* en ejemplos individuales y mostrar que, por ejemplo, la expresión `(x=x+3)+4` tiene ese atributo

```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi
```

Tabla 8.1
Gramática con atributos de código P como un atributo de cadena sintetizado

Regla gramatical	Reglas semánticas
$exp_1 \rightarrow$ **id** $= exp_2$	exp_1 *.pcode* = "**lda**" || **id** *.strval* ++ exp_2 *.pcode* ++ "**stn**"
exp → *aexp*	*exp .pcode* = *aexp .pcode*
$aexp_1 \rightarrow aexp_2$ + *factor*	$aexp_1$ *.pcode* = $aexp_2$ *.pcode* ++ *factor .pcode* ++ "**adi**"
aexp → *factor*	*aexp .pcode* = *factor .pcode*
factor → (*exp*)	*factor .pcode* = *exp .pcode*
factor → **num**	*factor .pcode* = "**ldc**" || **num** *.strval*
factor → **id**	*factor .pcode* = "**lod**" || **id** *.strval*

Código de tres direcciones Una gramática con atributos para código de tres direcciones de la gramática de expresión simple anterior se proporciona en la tabla 8.2. En esa tabla utilizamos el atributo de código que se llama *tacode* (para código de tres direcciones), y como en la tabla 8.1, ++ para concatenación de cadena con un retorno de línea y || para concatenación de cadena con un espacio. A diferencia del código P, el código de tres direcciones requiere que se generen nombres temporales para resultados intermedios en las expresiones, y esto requiere que la gramática con atributos incluya un nuevo atributo *name* para cada nodo. Este atributo también es sintetizado, pero para asignar nombres temporales recién generados a nodos interiores utilizamos una función *newtemp*( ) que se supone genera una secuencia de nombres temporales `t1`, `t2`, `t3`,. . . (se devuelve uno nuevo cada vez que se llama a *newtemp*( )). En este ejemplo simple sólo los nodos correspondientes al operador + necesitan nuevos nombres temporales; la operación de asignación simplemente utiliza el nombre de la expresión en el lado derecho.

Advierta en la tabla 8.2 que, en el caso de las producciones unitarias *exp* → *aexp* y *aexp* → *factor*, el atributo *name*, además del atributo *tacode*, se transportan de hijos a padres y que, en el caso de los nodos interiores del operador, se generan nuevos atributos *name* antes del *tacode* asociado. Observe también que, en las producciones de hoja *factor* → **num** y *factor* → **id**, el valor de cadena del token se utiliza como *factor.name*, y que (a diferencia

Tabla 8.2
Gramática con atributos para código de tres direcciones como un atributo de cadena sintetizado

Regla gramatical	Reglas semánticas
$exp_1 \rightarrow$ **id** $= exp_2$	exp_1 .*name* $= exp_2$.*name* exp_1 .*tacode* $= exp_2$.*tacode* ++ **id** .*strval* ‖ "=" ‖ exp_2 .*name*
exp → *aexp*	*exp* .*name* = *aexp* .*name* *exp* .*tacode* = *aexp* .*tacode*
$aexp_1 \rightarrow aexp_2$ + *factor*	$aexp_1$.*name* = *newtemp*( ) $aexp_1$.*tacode* = $aexp_2$.*tacode* ++ *factor* .*tacode* ++ $aexp_1$.*name* ‖ "=" ‖ $aexp_2$.*name* ‖ "+" ‖ *factor* .*name*
aexp → *factor*	*aexp* .*name* = *factor* .*name* *aexp* .*tacode* = *factor* .*tacode*
factor → (*exp*)	*factor* .*name* = *exp* .*name* *factor* .*tacode* = *exp* .*tacode*
factor → **num**	*factor* .*name* = **num** .*strval* *factor* .*tacode* = ""
factor → **id**	*factor* .*name* = **id** .*strval* *factor* .*tacode* = ""

del código P) no se genera ningún código de tres direcciones en tales nodos (utilizamos "" para representar la cadena vacía).

De nueva cuenta, dejamos al lector mostrar que, dadas las ecuaciones de atributo de la tabla 8.2, la expresión `(x=x+3)+4` tiene el atributo *tacode*

```
t1 = x + 3
x = t1
t2 = t1 + 4
```

(Esto supone que *newtemp*( ) es llamado en postorden y genera nombres temporales comenzando con `t1`.) Advierta cómo la asignación `x=x+3` genera dos instrucciones de tres direcciones utilizando un elemento temporal. Esto es una consecuencia del hecho de que la evaluación de atributos siempre crea un elemento temporal para cada subexpresión, incluyendo los lados derechos de las asignaciones.

Visualizar la generación de código como el cálculo de un atributo de cadena sintetizado es útil para mostrar claramente las relaciones entre las secuencias de código de las diferentes partes del árbol sintáctico y para comparar los diferentes métodos de generación de código, pero es poco práctico como técnica para generación de código real, por varias razones. En primer lugar, el uso de la concatenación de cadena causa que se desperdicie una cantidad desmesurada de copiado de cadenas y memoria a menos que los operadores de la concatenación sean muy complejos. En segundo, por lo regular es mucho más deseable generar pequeños segmentos de código a medida que continúa la generación de código y escribir estos segmentos en un archivo o bien insertarlos en una estructura de datos (tal como un arreglo de cuádruples), lo que requiere acciones semánticas que no se adhieren a la síntesis postorden estándar de atributos. Finalmente, aunque es útil visualizar el código como puramente sintetizado, la generación de código en general depende mucho de atributos heredados, y esto complica en gran medida las gramáticas con atributos. Es por

esto que no nos preocupamos aquí por escribir ningún código (incluso pseudocódigo) para implementar las gramáticas con atributos de los ejemplos anteriores (pero vea los ejercicios). En vez de eso, en la siguiente subsección regresaremos a técnicas de generación de código más directas.

8.2.2 Generación de código práctica

Las técnicas estándar de generación de código involucran modificaciones de los recorridos postorden del árbol sintáctico implicado por las gramáticas con atributos de los ejemplos precedentes o, si no se genera un árbol sintáctico de manera explícita, acciones equivalentes durante un análisis sintáctico. El algoritmo básico puede ser descrito com o el siguiente procedimiento recursivo (para nodos de árbol con dos hijos como máximo, pero fácilmente extensibles a más):

procedure *genCode* (*T: treenode*);
begin
  if *T no es nil* **then**
    genere código para preparar en el caso del código del hijo izquierdo de T ;
    genCode(hijo izquierdo de T) ;
    genere código para preparar en el caso del código del hijo derecho de T ;
    genCode(hijo derecho de T) ;
    genere código para implementar la acción de T ;
end;

Observe que este procedimiento recursivo no sólo tiene un componente postorden (que genera código para implementar la acción de *T*) sino también un componente de preorden y un componente enorden (que genera el código de preparación para los hijos izquierdo y derecho de *T*). En general, cada acción que representa *T* requerirá una versión un poco diferente del código de preparación preorden y enorden.

Para ver de manera detallada cómo se puede construir el procedimiento *genCode* en un ejemplo específico considere la gramática para expresiones aritméticas simples que hemos estado utilizando en esta sección (véase la gramática en la página 407). Las definiciones en C para un árbol sintáctico abstracto de esta gramática se pueden proporcionar de la manera siguiente (compare con las de la página 111 del capítulo 3):

```
typedef enum {Plus,Assign} Optype;
typedef enum {OpKind,ConstKind,IdKind} NodeKind;
typedef struct streenode
        { NodeKind kind;
          Optype op; /* usado con OpKind */
          struct streenode *lchild,*rchild;
          int val; /* usado con ConstKind */
          char * strval;
            /* usado para identificadores y números */
        } STreeNode;
typedef STreeNode *SyntaxTree;
```

Con estas definiciones se puede dar un árbol sintáctico para la expresión `(x=x+3)+4` como se ve en seguida:

Advierta que el nodo designación contiene el identificador que se está asignando (en el campo **strval**), así que un nodo designación tiene solamente un hijo (la expresión que se asigna).[4]

Basados en esta estructura para un árbol sintáctico podemos escribir un procedimiento *genCode* para generar código P como se da en la figura 8.7. En esa figura haremos los comentarios siguientes acerca del código. En primer lugar, el código utiliza la función estándar de C **sprintf** para concatenar las cadenas en el elemento temporal local **codestr**. En segundo lugar se llama el procedimiento **emitCode** para generar una línea simple de código P, ya sea en una estructura de datos o en un archivo de salida; sus detalles no se muestran. Finalmente, los dos casos de operador (**Plus** y **Assign**) requieren dos órdenes diferentes de recorridos: **Plus** necesita sólo procesamiento de postorden, mientras que **Assign** requiere cierto procesamiento tanto preorden como postorden. De este modo, las llamadas recursivas pueden no escribirse del mismo modo para todos los casos.

Para mostrar que la generación de código, incluso con la variación necesaria en orden de recorrido, todavía se puede realizar durante un análisis sintáctico (sin la generación de un árbol sintáctico), mostramos un archivo de especificación Yacc en la figura 8.8 que corresponde directamente al código de la figura 8.7. (Observe cómo el procesamiento combinado preorden y postorden de las asignaciones se traduce en secciones de acción dividida en la especificación Yacc.)

Dejamos al lector escribir un procedimiento *genCode* y especificación Yacc para generar el código de tres direcciones como se especificó en la gramática con atributos de la tabla 8.2.

Figura 8.7
Implementación de un procedimiento de generación de código para código P correspondiente a la gramática con atributos de la tabla 8.1

```
void genCode( SyntaxTree t)
{ char codestr[CODESIZE];
  /* CODESIZE = longitud máxima de 1 línea de código P  */
  if (t != NULL)
  { switch (t->kind)
    { case OpKind:
        switch (t->op)
        { case Plus:
            genCode(t->lchild);
            genCode(t->rchild);
            emitCode("adi");
            break;
```

4. También almacenamos números como cadenas en el campo **strval** en este ejemplo.

Figura 8.7 Continuación

```
                case Assign:
                  sprintf(codestr,"%s %s",
                                  "lda",t->strval);
                  emitCode(codestr);
                  genCode(t->lchild);
                  emitCode("stn");
                  break;
                default:
                  emitCode("Error");
                  break;
              }
              break;
            case ConstKind:
              sprintf(codestr,"%s %s","ldc",t->strval);
              emitCode(codestr);
              break;
            case IdKind:
              sprintf(codestr,"%s %s","lod",t->strval);
              emitCode(codestr);
              break;
            default:
              emitCode("Error");
              break;
          }
        }
      }
```

Figura 8.8
Especificación Yacc para la generación de código P de acuerdo con la gramática con atributos de la tabla 8.1

```
%{
#define YYSTYPE char *
    /* hace que Yacc utilice cadenas como valores */

/* otro código de inclusión ... */
%}

%token NUM ID

%%

exp      : ID
           { sprintf(codestr,"%s %s","lda",$1);
             emitCode(codestr); }
           '=' exp
           { emitCode("stn"); }
         | aexp
         ;
```

Figura 8.8 Continuación

```
aexp     : aexp '+' factor {emitCode("adi");}
         | factor
         ;

factor   : '(' exp ')'
         | NUM        { sprintf(codestr,"%s %s","ldc",$1);
                        emitCode(codestr); }
         | ID         { sprintf(codestr,"%s %s","lod",$1);
                        emitCode(codestr); }
         ;

%%
/* funciones de utilería ... */
```

8.2.3 Generación de código objetivo a partir del código intermedio

Si un compilador genera código intermedio, ya sea directamente durante un análisis sintáctico o desde un árbol sintáctico, entonces debe efectuarse otro paso en el código intermedio para generar el código objetivo final (por lo regular después de algún procesamiento adicional del código intermedio). Este paso puede ser bastante complejo por sí mismo, en particular si el código intermedio es muy simbólico y contiene poca o ninguna información acerca de la máquina objetivo o el ambiente de ejecución. En este caso, el paso de la generación de código final debe suministrar todas las ubicaciones reales de variables y temporales, más el código necesario para mantener el ambiente de ejecución. Una cuestión particularmente importante es la asignación apropiada de los registros y el mantenimiento de la información sobre el uso de los mismos (es decir, cuáles registros se encuentran disponibles y cuáles contienen valores conocidos). Aplazaremos un análisis detallado de tales cuestiones de asignación hasta más adelante en este capítulo. Por ahora comentaremos sólo técnicas generales para este proceso.

Por lo regular la generación de código a partir del código intermedio involucra alguna o ambas de las dos técnicas estándar: expansión de macro y simulación estática. La **expansión de macro** involucra el reemplazo de cada clase de instrucción del código intermedio con una secuencia equivalente de instrucciones de código objetivo. Esto requiere que el compilador se mantenga al tanto de las decisiones acerca de ubicaciones e idiomas de código en estructuras de datos separadas y que los procedimientos del macro varíen la secuencia de código como se requiera mediante las clases particulares de datos involucradas en la instrucción de código intermedio. De este modo, cada paso puede ser mucho más complejo que las formas simples de expansión de macro disponibles del preprocesador de C o ensambladores de macro. La **simulación estática** involucra una simulación en línea recta de los efectos del código intermedio y generación de código objetivo para igualar estos efectos. Esto también requiere de más estructuras de datos, y puede variar de formas muy simples de seguimiento empleadas en conjunto con la expansión de macro, hasta la altamente

sofisticada **interpretación abstracta** (que mantiene los valores algebraicamente a medida que son calculados).

Podemos obtener alguna idea de los detalles de estas técnicas considerando el problema de traducir desde el código P al código de tres direcciones y viceversa. Consideremos la pequeña gramática de expresión que hemos estado utilizando como ejemplo de ejecución en esta sección, y consideremos la expresión `(x=x+3)+4`, cuyas traducciones en código P y código de tres direcciones se proporcionaron en las páginas 408 y 409, respectivamente. Consideremos primero la traducción del código P para esta expresión:

```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi
```

en su correspondiente código de tres direcciones:

```
t1 = x + 3
x = t1
t2 = t1 + 4
```

Esto requiere que realicemos una simulación estática de la pila de máquina P para encontrar equivalentes de tres direcciones del código dado. Hacemos esto con una estructura de datos de pila real durante la traducción. Después de las primeras tres instrucciones de código P, no se han generado todavía instrucciones de tres direcciones, pero la pila de máquina P se ha modificado para reflejar las cargas, y la pila tiene el aspecto siguiente:

Ahora, cuando se procesa la operación `adi`, se genera la instrucción de tres direcciones

```
t1 = x + 3
```

y la pila se cambia a

La instrucción `stn` provoca entonces que se genere la instrucción de tres direcciones

```
x = t1
```

y la pila se cambie a

La instrucción siguiente inserta la constante 4 en la pila:

Finalmente, la instrucción **adi** provoca que se genere la instrucción de tres direcciones

```
t2 = t1 + 4
```

y la pila se cambie a

Esto completa la simulación estática y la traducción.

Ahora consideraremos el caso de traducir del código de tres direcciones al código P. Si ignoramos la complicación agregada de los nombres temporales, esto puede hacerse mediante expansión simple de macro. Por consiguiente, una instrucción de tres direcciones

```
a = b + c
```

siempre se puede traducir en la secuencia de código P

```
lda a
lod b ; o ldc b si b es una constante
lod c ; o ldc c si c es una constante
adi
sto
```

Esto resulta en la siguiente traducción (algo insatisfactoria) del anterior código de tres direcciones en código P:

```
lda  t1
lod  x
ldc  3
adi
sto
lda  x
lod  t1
sto
lda  t2
```

(*continúa*)

```
lod  t1
ldc  4
adi
sto
```

Si queremos eliminar los elementos temporales extra, entonces debemos utilizar un esquema más sofisticado que la expansión de macro pura. Una posibilidad es generar un nuevo árbol a partir del código de tres direcciones, indicando el efecto que el código tiene al etiquetar los nodos del árbol tanto con el operador de cada instrucción como con el nombre que se le asigna. Esto puede visualizarse como una forma de simulación estática, y el árbol resultante para el anterior código de tres direcciones es

Advierta cómo la instrucción de tres direcciones

```
x = t1
```

no provoca que se creen nodos extra en este árbol, pero sí que el nodo con nombre `t1` adquiera el nombre adicional `x`. Este árbol es similar, pero no idéntico, al árbol sintáctico de la expresión original (véase la página 411).[5] El código P se puede generar a partir de este árbol de manera muy semejante a como se genera el código P a partir de un árbol sintáctico, de la manera antes descrita, pero con elementos temporales eliminados al hacer asignaciones sólo a nombres permanentes de nodos interiores. De este modo, en el árbol de muestra, sólo se asigna `x`, los nombres `t1` y `t2` nunca se emplean en el código P generado, y el valor correspondiente al nodo raíz (con nombre `t2`) se deja en la pila de la máquina P. Esto produce exactamente la misma generación de código P que antes, aunque se utiliza `stn` en lugar de `sto` siempre que se realiza un almacenamiento. Alentamos al lector a escribir pseudocódigo o código en C para llevar a cabo este proceso.

8.3 GENERACIÓN DE CÓDIGO DE REFERENCIAS DE ESTRUCTURAS DE DATOS

8.3.1 Cálculos de direcciones

En la sección anterior vimos cómo se puede generar el código intermedio para asignaciones y expresiones aritméticas simples. En estos ejemplos todos los valores básicos eran constantes, o bien, variables simples (ya sea variables de programa tales como `x`, o temporales como `t1`). Las variables simples eran identificadas sólo por nombre: la traducción a código

5. Este árbol es un caso especial de una construcción más general denominada **DAG de un bloque básico**, la cual se describe en la sección 8.9.3.

objetivo requiere que estos nombres sean reemplazados por direcciones reales, las cuales podrían ser registros, direcciones de memoria absoluta (para globales), o desplazamientos de registro de activación (para locales, que posiblemente incluyan un nivel de anidación). Estas direcciones se pueden insertar en el momento en que es generado el código intermedio o aplazar la inserción hasta que se genere el código real (con la tabla de símbolos manteniendo las direcciones).

Sin embargo, existen muchas situaciones que requieren que se realicen los cálculos de las direcciones para localizar la dirección real en cuestión, y estos cálculos deben ser expresados directamente, incluso en código intermedio. Tales cálculos se presentan en subíndice de arreglo, campo de registro y referencias de apuntador. Comentaremos cada uno de estos casos por turno. Pero debemos comenzar por describir extensiones para el código de tres direcciones y el código P que nos permitan expresar tales cálculos de direcciones.

Código de tres direcciones para cálculos de direcciones En el código de tres direcciones, lo que se necesita no es tanto nuevas operaciones (las operaciones aritméticas habituales se pueden utilizar para calcular direcciones) sino maneras para indicar los modos de direccionamiento "dirección de" e "indirecto". En nuestra versión del código en tres direcciones utilizaremos la notación equivalente en C "&" y "*" para indicar estos modos de direccionamiento. Por ejemplo, supongamos que deseamos almacenar el valor constante 2 en la dirección de la variable **x** más 10 bytes. Expresaríamos esto en código de tres direcciones de la manera siguiente:

```
t1 = &x + 10
*t1 = 2
```

La implementación de estos nuevos modos de direccionamiento requiere que la estructura de datos para el código de tres direcciones contenga un nuevo campo o campos. Por ejemplo, la estructura de datos cuádruple de la figura 8.4 (página 403) se puede aumentar mediante un campo enumerado `AddrMode` con posibles valores `None`, `Address` e `Indirect`.

Código P para cálculos de direcciones En código P es común introducir nuevas instrucciones para expresar nuevos modos de direccionamiento (puesto que hay pocas direcciones explícitas a las cuales asignar modos de direccionamiento). Las dos instrucciones que introduciremos para este propósito son las siguientes:

1. `ind` ("indirect load"), que toma como parámetro un desplazamiento entero, supone que hay una dirección en la parte superior o tope de la pila, agrega el desplazamiento a la dirección y reemplaza a esta última en la pila con el valor de la ubicación resultante:

2. `ixa` ("indexed address"), la cual toma como parámetro un factor de escala entero, supone que se tiene un desplazamiento en la parte superior o tope de la pila y una dirección base debajo del mismo, multiplica el desplazamiento por el factor de escala, agrega la dirección base, extrae tanto el desplazamiento como la base de la pila e inserta la dirección resultante:

Estas dos instrucciones de código P, junto con la instrucción `lda` (load address) que se presentó anteriormente, permitirán realizar los mismos cálculos de dirección y las referencias a los modos de direccionamiento que para el código de tres direcciones.[6] Por ejemplo, el problema de muestra anterior (almacenar el valor constante 2 en la dirección de la variable `x` más 10 bytes) ahora se puede resolver en código P de la manera que se presenta a continuación:

```
lda x
ldc 10
ixa 1
ldc 2
sto
```

Ahora volveremos a un análisis de arreglos, registros y apuntadores, seguido por un análisis de generación de código objetivo y un ejemplo extendido.

8.3.2 Referencias de arreglo

Una referencia de arreglo involucra la subindización de una variable de arreglo mediante una expresión para obtener una referencia o valor de un simple elemento del arreglo, como en el código en C

```
int a[SIZE]; int i,j;

...
a[i+1] = a[j*2] + 3;
```

En esta asignación la subindización de `a` mediante la expresión `i+1` produce una dirección (el objetivo de la asignación), mientras que la subindización de `a` mediante la expresión `j*2` produce el valor de la dirección calculada del tipo de elemento de `a` (a saber, `int`). Como los arreglos son almacenados de manera secuencial en la memoria, cada dirección debe ser calculada desde la **dirección base** de `a` (su dirección de inicio en memoria) y un desplazamiento que depende linealmente del valor del subíndice. Cuando se deseara obtener el valor más que la dirección, se debe generar un paso de indirección extra para obtener el valor de la dirección calculada.

El desplazamiento es calculado a partir del valor de subíndice de la manera siguiente. En primer lugar, debe hacerse un ajuste al valor del subíndice si el intervalo de subíndice no comienza en 0 (esto podría ocurrir en lenguajes como Pascal y Ada, pero no en C). En segundo lugar, el valor del subíndice ajustado debe ser multiplicado por un **factor de escala**

6. De hecho, la instrucción `ixa` podría ser simulada mediante operaciones aritméticas, sólo que en el código P estas operaciones son tipeadas (`adi` = multiplicación entera sólo) y de este modo no se pueden aplicar a direcciones. No insistimos en las limitantes de tipo del código P porque involucra parámetros extra que hemos suprimido por simplicidad.

que es igual al tamaño de cada uno de los elementos del arreglo en la memoria. Finalmente, el subíndice escalado resultante se agrega a la dirección base para obtener la dirección final del elemento del arreglo.

Por ejemplo, la dirección de la referencia de arreglo en C `a[i+1]` es[7]

```
a + (i + 1) * sizeof(int)
```

De manera más general, la dirección de un elemento de arreglo `a[t]` en cualquier lenguaje es

$$\textit{dirección_base}(\texttt{a}) + (t - \textit{límite_inferior}(\texttt{a})) * \textit{tamaño_elemento}(\texttt{a})$$

Ahora volveremos a las maneras de expresar este cálculo de dirección en el código de tres direcciones y en el código P. Para hacer esto en una notación independiente de la máquina objetivo supondremos que la "dirección" de una variable de arreglo es su dirección base. De este modo, si `a` es una variable de arreglo, `&a` en código de tres direcciones es lo mismo que *dirección_base*(`a`), y en código P

```
lda a
```

carga la dirección base de `a` en la pila de máquina P. También, puesto que un cálculo de referencia de arreglo depende del tamaño del tipo de datos del elemento en la máquina objetivo, utilizaremos la expresión `elem_size(a)` para el tamaño de elemento del arreglo `a` en la máquina objetivo.[8] Como ésta es una cantidad estática (suponiendo tipeado estático), esta expresión será reemplazada por una constante en tiempo de compilación.

Código de tres direcciones para las referencias de arreglo Una manera en que se pueden expresar referencias de arreglo en código de tres direcciones es introducir dos nuevas operaciones, una que obtenga el valor de un elemento de arreglo

```
t2 = a[t1]
```

y otra que asigne la dirección de un elemento de arreglo

```
a[t2] = t1
```

(esto podría ser dado por los símbolos `=[]` y `[]=`). Si se utiliza esta terminología no es necesario expresar el cálculo de dirección real (y las dependencias de máquina tales como el tamaño del elemento desaparecen de esta notación). Por ejemplo, la sentencia de código fuente

```
a[i+1] = a[j*2] + 3;
```

se traduciría en las instrucciones de tres direcciones

```
t1 = j * 2
t2 = a[t1]
t3 = t2 + 3
t4 = i + 1
a[t5] = t3
```

7. En C, el nombre de un arreglo (tal como `a` en esta expresión) representa su propia dirección base.

8. Esto podría ser de hecho una función suministrada por la tabla de símbolos.

Sin embargo, todavía es necesario introducir modos de direccionamiento como los anteriormente descritos cuando tratamos con referencias de apuntador y campo de registro, de modo que tenga sentido tratar todos esos cálculos de dirección de manera uniforme. De esta manera, también podemos escribir los cálculos de direcciones de un elemento de arreglo directamente en código de tres direcciones. Por ejemplo, la asignación

```
t2 = a[t1]
```

también puede escribirse como (utilizando elementos temporales adicionales **t3** y **t4**)

```
t3 = t1 * elem_size(a)
t4 = &a + t3
t2 = *t4
```

y la asignación

```
a[t2] = t1
```

puede escribirse como

```
t3 = t2 * elem_size(a)
t4 = &a + t3
*t4 = t1
```

Finalmente, como un ejemplo más complejo, la sentencia de código fuente

```
a[i+1] = a[j*2] + 3;
```

se traduce en las instrucciones de tres direcciones

```
t1 = j * 2
t2 = t1 * elem_size(a)
t3 = &a + t2
t4 = *t3
t5 = t4 + 3
t6 = i + 1
t7 = t6 * elem_size(a)
t8 = &a + t7
*t8 = t5
```

Código P para referencias de arreglo Como se describió anteriormente, utilizaremos las nuevas instrucciones de dirección **ind** e **ixa**. La instrucción **ixa** de hecho fue construida precisamente con los cálculos de dirección de arreglo en mente, mientras que la instrucción **ind** se utiliza para cargar el valor de una dirección calculada con anterioridad (es decir, para implementar una carga indirecta). La referencia de arreglo

```
t2 = a[t1]
```

se escribe en código P como

```
lda t2
lda a
lod t1
ixa elem_size(a)
ind 0
sto
```

y la asignación de arreglo

```
a[t2] = t1
```

se escribe en código P como

```
lda a
lod t2
ixa elem_size(a)
lod t1
sto
```

Finalmente, el ejemplo anterior más complejo

```
a[i+1] = a[j*2] + 3;
```

se traduce en las siguientes instrucciones de código P:

```
lda a
lod i
ldc 1
adi
ixa elem_size(a)
lda a
lod j
ldc 2
mpi
ixa elem_size(a)
ind 0
ldc 3
adi
sto
```

Un procedimiento para la generación de código con referencias de arreglo Mostraremos aquí cómo las referencias de arreglo se pueden generar mediante un procedimiento de generación de código. Utilizaremos el ejemplo del subconjunto de expresiones C de la sección anterior (véase la gramática de la página 407), aumentado mediante una operación de subíndice. La nueva gramática que utilizaremos es la siguiente:

exp → *subs* = *exp* | *aexp*
aexp → *aexp* + *factor* | *factor*
factor → (*exp*) | **num** | *subs*
subs → **id** | **id** [*exp*]

Advierta que el objetivo de una asignación ahora puede ser una variable simple o una variable subindizada (ambas incluidas en el no terminal *subs*). Utilizamos la misma estructura de datos para el árbol sintáctico que antes (véanse las declaraciones en la página 410), sólo que ahora existe una operación adicional **Subs** para la subindización:

```
typedef enum {Plus,Assign,Subs} Optype;
/* otras declaraciones como antes */
```

Además, como ahora puede estar una expresión de subíndice a la izquierda de una asignación, no es posible almacenar el nombre de la variable objetivo en el mismo nodo de asignación (ya que puede no haber tal nombre). En vez de eso, los nodos de asignación ahora tienen dos hijos como nodos extra: el hijo izquierdo debe ser un identificador o una expresión de subíndice. Los subíndices mismos sólo pueden ser aplicados a identificadores, de modo que almacenamos el nombre de la variable de arreglo en nodos subíndice. Así, el árbol sintáctico para la expresión

```
(a[i+1]=2)+a[j]
```

es

Un procedimiento de generación de código que produce código P para tales árboles sintácticos se proporciona en la figura 8.9 (compare con la figura 8.7, página 411). La diferencia principal entre este código y el de la figura 8.7 es que éste necesita tener un atributo heredado **isAddr** que distinga un identificador o expresión con subíndice a la izquierda de una asignación a partir de uno a la derecha. Si **isAddr** se establece a **TRUE**, entonces la *dirección* de la expresión debe ser devuelta; de otro modo, se devuelve el *valor*. Dejamos al lector verificar que este procedimiento genera el siguiente código P para la expresión `(a[i+1]=2)+a[j]`:

```
lda a
lod i
```

```
ldc 1
adi
ixa elem_size(a)
ldc 2
stn
lda a
lod j
ixa elem_size(a)
ind 0
adi
```

También dejamos al lector la construcción de un generador de código de tres direcciones para esta gramática (véanse los ejercicios).

Figura 8.9
Implementación de un procedimiento de generación de código para el código P correspondiente a la gramática de expresión de la página 422

```
void genCode( SyntaxTree t, int isAddr)
{ char codestr[CODESIZE];
  /* CODESIZE = longitud máxima de 1 línea de código P */
  if (t != NULL)
  { switch (t->kind)
    { case OpKind:
        switch (t->op)
        { case Plus:
            if (isAddr) emitCode("Error");
            else { genCode(t->lchild,FALSE);
                   genCode(t->rchild,FALSE);
                   emitCode("adi");}
            break;
          case Assign:
            genCode(t->lchild,TRUE);
            genCode(t->rchild,FALSE);
            emitCode("stn");
            break;
          case Subs:
            sprintf(codestr,"%s %s","lda",t->strval);
            emitCode(codestr);
            genCode(t->lchild,FALSE);
            sprintf(codestr,"%s%s%s",
               "ixa elem_size(",t->strval,")");
            emitCode(codestr);
            if (!isAddr) emitCode("ind 0");
            break;
          default:
            emitCode("Error");
            break;
        }
        break;
      case ConstKind:
```

Figura 8.9 Continuación

```
            if (isAddr) emitCode("Error");
            else
            { sprintf(codestr,"%s %s","ldc",t->strval);
              emitCode(codestr);
            }
            break;
          case IdKind:
            if (isAddr)
              sprintf(codestr,"%s %s","lda",t->strval);
            else
              sprintf(codestr,"%s %s","lod",t->strval);
            emitCode(codestr);
            break;
          default:
            emitCode("Error");
            break;
        }
      }
    }
```

Arreglos multidimensionales Un factor que complica el cálculo de las direcciones de arreglo es la existencia, en la mayoría de los lenguajes, de arreglos en varias dimensiones. Por ejemplo, en C un arreglo de dos dimensiones (con diferentes tamaños de índice) puede declararse como

```
int a[15][10];
```

Tales arreglos pueden estar parcialmente subindizados, produciendo un arreglo de menos dimensiones, o subindizados por completo, produciendo un valor del tipo elemento del arreglo. Por ejemplo, dada la declaración anterior de **a** en C, la expresión **a[i]** implica subindización de manera parcial de **a**, produciendo un arreglo unidimensional de enteros, mientras que la expresión **a[i][j]** implicala subindización completa de **a** y produce un valor de tipo entero. La dirección de una variable de arreglo subindizada de manera parcial o completa se puede calcular mediante la aplicación recursiva de las técnicas recién descritas para el caso unidimensional.

8.3.3 Estructura de registro y referencias de apuntador

El cálculo de la dirección de un campo de estructura o registro presenta un problema similar al de calcular una dirección de arreglo subindizada. En primer lugar se calcula la dirección base de la variable de estructura; luego se encuentra el desplazamiento (generalmente fijo) del campo nombrado, y se suman los dos para obtener la dirección resultante. Considere, por ejemplo, las declaraciones en C

```
typedef struct rec
   { int i;
     char c;
     int j;
   } Rec;
...
Rec x;
```

Por lo general, la variable **x** está asignada en la memoria como lo muestra la siguiente ilustración, con cada campo (**i**, **c** y **j**) teniendo un desplazamiento desde la dirección base de **x** (la cual puede ser en sí misma un desplazamiento en un registro de activación):

Advierta que los campos están asignados linealmente (por lo regular desde la dirección más baja a la más alta), que el desplazamiento de cada campo es una constante, y que el primer campo (**x.i**) tiene desplazamiento cero. Advierta también que los desplazamientos de los campos dependen del tamaño de los diversos tipos de datos en la máquina objetivo, pero que no se encuentra involucrado un factor de escala como con los arreglos.

Para escribir código intermedio independiente del objetivo para cálculos de dirección de campo de estructura de registro, debemos introducir una nueva función que devuelva el desplazamiento en campo, dada una variable de estructura y el nombre del campo. Llamaremos a esta función **field_offset** y la escribiremos con dos parámetros, el primero será el nombre de la variable y el segundo el nombre del campo. De este modo, **field_offset (x,j)** devuelve el desplazamiento de **x.j**. Como con otras funciones similares, esta función puede ser suministrada mediante la tabla de símbolos. En cualquier caso, ésta es una cantidad en tiempo de compilación, de modo que el código intermedio real generado tendrá casos de llamadas a **field_offset** reemplazados por constantes.

Las estructuras de registro generalmente son utilizadas junto con apuntadores y asignación de memoria dinámica para implementar estructuras de datos dinámicas tales como listas y árboles. Así, describimos también cómo interactúan los apuntadores con cálculos de dirección de campo. Un apuntador, para los propósitos de esta exposición, establece simplemente un nivel adicional de indirección; pasamos por alto las cuestiones de asignación involucradas en la creación de valores de apuntador (éstas fueron comentadas en el capítulo anterior).

Código de tres direcciones para las referencias de apuntador y estructura Consideremos, en primer lugar, el código de tres direcciones para el cálculo de direcciones de campo: para calcular la dirección de **x.j** en un elemento temporal **t1** utilizamos la instrucción de tres direcciones

```
t1 = &x + field_offset(x,j)
```

Una asignación de campo tal como la sentencia de C

```
x.j = x.i;
```

puede ser traducida en el código de tres direcciones

```
t1 = &x + field_offset(x,j)
t2 = &x + field_offset(x,i)
 *t1 = *t2
```

Ahora consideremos los apuntadores. Supongamos, por ejemplo, que **x** es declarada como un apuntador para un entero, por ejemplo mediante la declaración en C

```
int * x;
```

Supongamos además que **i** es una variable entera normal. Entonces la asignación en C

```
*x = i;
```

puede ser traducida previamente en la instrucción de tres direcciones

```
*x = i
```

y la asignación

```
i = *x;
```

en la instrucción de tres direcciones

```
i = *x
```

Para ver cómo interactúa la indirección de apuntadores con los cálculos de dirección de campo consideremos el ejemplo siguiente de una estructura de datos de árbol y declaración de variable en C:

```
typedef struct treeNode
  { int val;
    struct treeNode * lchild, * rchild;
  } TreeNode;
...
TreeNode *p;
```

Ahora consideremos dos asignaciones típicas

```
p->lchild = p;
p = p->rchild;
```

Estas sentencias se producen en el código de tres direcciones

```
t1 = p + field_offset(*p,lchild)
*t1 = p
t2 = p + field_offset(*p,rchild)
p = *t2
```

Código P para referencias de apuntador y estructura Dada la declaración de **x** al principio de esta exposición (página 423), se puede hacer un cálculo directo de la dirección de **x.j** en código P de la manera siguiente:

```
lda x
lod field_offset(x,j)
ixa 1
```

La sentencia de asignación

```
x.j = x.i;
```

puede ser traducida en el código P

```
lda x
lod field_offset(x,j)
ixa 1
lda x
ind field_offset(x,i)
sto
```

Advierta cómo se utiliza la instrucción **ind** para obtener el valor de **x.i** sin calcular primero su dirección completa.

En el caso de los apuntadores (con **x** declarada como una **int***), la asignación

```
*x = i;
```

se traduce al código P

```
lod x
lod i
sto
```

y la asignación

```
i = *x;
```

se traduce al código P

```
lda i
lod x
ind 0
sto
```

Concluimos con el código P para las asignaciones

```
p->lchild = p;
p = p->rchild;
```

(véase la declaración de **p** en la página anterior). Esto se traduce al siguiente código P:

```
lod p
lod field_offset(*p,lchild)
ixa 1
lod p
sto
lda p
lod p
ind field_offset(*p,rchild)
sto
```

Dejamos los detalles de un procedimiento de generación de código que generará estas secuencias de código ya sea en código de tres direcciones o en código P para los ejercicios.

8.4 GENERACIÓN DE CÓDIGO DE SENTENCIAS DE CONTROL Y EXPRESIONES LÓGICAS

En esta sección describiremos la generación de código para varias formas de sentencias de control. Entre éstas se encuentran principalmente la sentencia if y la sentencia while estructuradas, las que veremos con detalle en la primera parte de esta sección. También incluimos en esta descripción el uso de la sentencia break (como en C), pero no comentaremos el control de bajo nivel como el de la sentencia goto, puesto que tales sentencias se pueden implementar con facilidad directamente en código intermedio o código objetivo. Otras formas de control estructurado, como la sentencia repeat (o la sentencia do-while), la sentencia for y la sentencia case (o sentencia switch), se dejan para los ejercicios. Otra técnica de implementación útil para la sentencia switch, denominada **tabla de salto**, también se describe en un ejercicio.

La generación de código intermedio para sentencias de control, tanto en código de tres direcciones como en código P, involucra la generación de **etiquetas** de una manera similar a la generación de nombres temporales en el código de tres direcciones, pero en este caso representan direcciones en el código objetivo a las que se harán los saltos. Si las etiquetas están por eliminarse en la generación de código objetivo, entonces surge un problema, ya que los saltos a localidades de código que ya no son conocidas se deben **reajustar** o reescribir retroactivamente. Comentaremos esto en la siguiente parte de esta sección.

Las expresiones lógicas, o booleanas, que se utilizan como pruebas de control, y que también se pueden emplear de manera independiente como datos, se analizan a continuación, particularmente respecto a la **evaluación de cortocircuito**, en la cual difieren de las expresiones aritméticas.

Finalmente, en esta sección presentaremos un procedimiento de generación de código de muestra en código P para sentencias if y while.

8.4.1 Generación de código para sentencias if y while

Consideraremos las siguientes dos formas de las sentencias if y while, que son similares en muchos lenguajes diferentes (pero que aquí daremos en una sintaxis tipo lenguaje C):

sent-if → `if` (*exp*) *sent* | `if` (*exp*) *sent* `else` *sent*
sent-while → `while` (*exp*) *sent*

El problema principal en la generación de código para tales sentencias es traducir las características de control estructurado en un equivalente "no estructurado" que involucre saltos, los cuales se pueden implementar directamente. Los compiladores se encargan de generar código para tales sentencias en un orden estándar que permite el uso eficiente de un subconjunto de los saltos posibles que una arquitectura objetivo puede permitir. Arreglos de código típicos para cada una de estas sentencias se muestran en las figuras 8.10 (más abajo en esta misma página) y 8.11 (página 430). (La figura 8.10 muestra una parte else (el caso FALSE), pero esto es opcional, de acuerdo con la regla gramatical que se dio recientemente, y el arreglo mostrado es fácil de modificar para el caso de una parte else omitida.) En cada uno de esos arreglos existen sólo dos clases de saltos (saltos incondicionales y saltos cuando la condición es falsa) y el caso verdadero es siempre un caso "que queda en la nada" que no necesita saltos. Esto reduce el número de saltos que el compilador necesita generar. También significa que sólo son necesarias dos instrucciones de salto en el código intermedio. Los saltos falsos ya aparecieron en el código de tres direcciones de la figura 8.2 (como una instrucción `if_false..goto`) y en el código P de la figura 8.6 (como una instrucción `fjp`). Resta introducir los saltos incondicionales, los cuales escribiremos

Figura 8.10
Arreglo de código típico para una sentencia if

simplemente como instrucciones **goto** en código de tres direcciones y como el **ujp** (unconditional jump) en código P.

Figura 8.11
Arreglo de código típico para una sentencia while

Código de tres direcciones para sentencias de control Supondremos que el generador de código genera una secuencia de etiquetas con nombres tales como **L1, L2**, y así sucesivamente. Para la sentencia

```
if ( E ) S1 else S2
```

se generará el siguiente patrón de código:

```
<code to evaluate E to t1>
if_false t1 goto L1
<code for S1>
goto L2
label L1
<code for S2>
label L2
```

De manera similar, una sentencia while de la forma

```
while ( E ) S
```

provocaría que fuera generado el siguiente patrón de código de tres direcciones:

```
label L1
<code to evaluate E to t1>
if_false t1 goto L2
<code for S>
goto L1
label L2
```

Código P para sentencias de control Para la sentencia

```
if ( E ) S1 else S2
```

se genera el siguiente patrón de código P:

```
<code to evaluate E>
fjp L1
<code for S1>
ujp L2
lab L1
<code for S2>
lab L2
```

y para la sentencia

```
while ( E ) S
```

se genera el siguiente patrón de código P:

```
lab L1
<code to evaluate E>
fjp L2
<code for S>
ujp L1
lab L2
```

Advierta que todas estas secuencias de código (tanto de código de tres direcciones como de código P) finalizan en una declaración de etiqueta, a la que podríamos llamar **etiqueta de salida** de la sentencia de control. Muchos lenguajes proporcionan una construcción de lenguaje que permite salir de los ciclos o bucles desde ubicaciones arbitrarias dentro del cuerpo del ciclo. Por ejemplo, C proporciona la sentencia **break** (que también se puede utilizar dentro de sentencias switch). En estos lenguajes se debe disponer de la etiqueta de salida para todas las rutinas de generación de código que se pueden llamar dentro del cuerpo de un ciclo, de manera que si se encuentra una sentencia de salida tal como un "break", se pueda generar un salto hacia la etiqueta de salida. Esto transforma a la etiqueta de salida en

un atributo heredado durante la generación del código, que debe almacenarse en una pila o bien pasarse como un parámetro a las rutinas de generación de código apropiadas. Más adelante en esta sección se ofrecerán más detalles al respecto, además de ejemplos de otras situaciones donde puede ocurrir esto último.

8.4.2 Generación de etiquetas y reajuste

Una característica de la generación de código para sentencias de control que puede provocar problemas durante la generación de código objetivo es el hecho de que, en algunos casos, deben generarse saltos a una etiqueta antes de que ésta se haya definido. Durante la generación de código intermedio, esto presenta pocos problemas, puesto que una rutina de generación de código simplemente puede llamar al procedimiento de generación de etiqueta cuando se necesita una para generar un salto hacia delante y grabar el nombre de la etiqueta (de manera local o en una pila) hasta que se conozca la ubicación de la etiqueta. Durante la generación del código objetivo, las etiquetas simplemente se pueden pasar a un ensamblador si se está generando código ensamblador, pero si se va a generar código ejecutable real, estas etiquetas deben transformarse en ubicaciones de código absolutas o relativas.

Un método estándar para generar tales saltos hacia delante consiste en dejar una brecha en el código donde el salto vaya a ocurrir, o bien, generar una instrucción de salto ficticia a una localidad falsa. Entonces, una vez que se conoce la ubicación del salto real, se utiliza para arreglar o **ajustar** el código omitido. Esto requiere que el código generado se mantenga en un "buffer" o "almacenamiento temporal" en la memoria, de manera que el ajuste se pueda hacer al vuelo, o que el código se escriba en un archivo temporal y después se reintroduzca y ajuste cuando sea necesario. En cualquier caso, puede ser necesario que los ajustes se almacenen de manera temporal en una pila o se mantengan localmente en procedimientos recursivos.

Durante el proceso de ajuste puede surgir otro problema, ya que muchas arquitecturas tienen dos variedades de saltos, un salto corto o ramificación (dentro de, digamos, 128 bytes de código) y un salto largo que requiere más espacio de código. En ese caso un generador de código puede necesitar insertar instrucciones `nop` cuando salte de manera breve, o hacer varios pasos para condensar el código.

8.4.3 Generación de código de expresiones lógicas

Hasta ahora no hemos dicho nada acerca de la generación de código para las expresiones lógicas o booleanas que se utilizan como pruebas en sentencias de control. Si el código intermedio tiene un tipo de datos booleano, y operaciones lógicas como **and** y **or**, entonces el valor de una expresión booleana se puede calcular en código intermedio exactamente de la misma manera que una expresión aritmética. Éste es el caso para el código P, y el código intermedio se puede diseñar de manera similar. Sin embargo, incluso si éste es el caso, la traducción en código objetivo por lo regular requiere que los valores booleanos se representen aritméticamente, ya que la mayoría de las arquitecturas no tienen un tipo booleano integrado. La manera estándar de hacer esto es representar el valor booleano **false** como 0 y **true** como 1. Entonces, los operadores estándar de bit **and** y **or** se pueden utilizar para calcular el valor de una expresión booleana en la mayor parte de las arquitecturas. Esto requiere que el resultado de operaciones de comparación como < se normalice a 0 o 1. En algunas arquitecturas esto requiere que 0 o 1 se carguen de manera explícita, puesto que el operador de comparación mismo sólo establece un código de condición. En ese caso necesitan generarse saltos adicionales para cargar el valor apropiado.

Si las operaciones lógicas son de **cortocircuito**, como en C, es necesario un uso adicional de los saltos. Una operación lógica es de cortocircuito si puede fallar la evaluación de su segundo argumento. Por ejemplo, si *a* es una expresión booleana que se calcula como falsa, entonces la expresión booleana *a* **and** *b* se puede determinar inmediatamente como falsa sin evaluar *b*. De manera similar, si se conoce que *a* es verdadera, entonces *a* **or** *b* se puede determinar como verdadera sin evaluar *b*. Las operaciones de cortocircuito son muy útiles para el codificador, ya que la evaluación de la segunda expresión ocasionaría un error si un operador no fuera de cortocircuito. Por ejemplo, es común escribir en C

```
if ((p!=NULL) && (p->val==0)) ...
```

donde la evaluación de `p->val` cuando `p` es nula podría causar una falla de memoria.

Los operadores booleanos de cortocircuito son semejantes a las sentencias if, sólo que éstos devuelven valores, y con frecuencia se definen utilizando **expresiones if** como

a **and** *b* ≡ **if** *a* **then** *b* **else false**

y

a **or** *b* ≡ **if** *a* **then true else** *b*

Para generar código que asegure que la segunda subexpresión será evaluada sólo cuando sea necesario, debemos emplear saltos exactamente de la misma manera que en el código para las sentencias if. Por ejemplo, el código P de cortocircuito para la expresión en C `(x!=0)&&(y==x)` es:

```
lod x
ldc 0
neq
fjp L1
lod y
lod x
equ
ujp L2
lab L1
lod FALSE
lab L2
```

8.4.4 Un procedimiento de generación de código de muestra para sentencias if y while

En esta sección mostraremos un procedimiento de generación de código para sentencias de control utilizando la siguiente gramática simplificada:

sent → *sent-if* | *sent-while* | `break` | `other`
sent-if → `if(` *exp* `)` *sent* | `if(` *exp* `)` *sent* `else` *sent*
sent-while → `while(` *exp* `)` *sent*
exp → `true` | `false`

Por simplicidad, esta gramática utiliza el token **other** para representar sentencias no incluidas en la gramática (como la sentencia de asignación). También incluye únicamente las expresiones booleanas constantes **true** y **false**. Incluye una sentencia **break** para mostrar cómo una sentencia así se puede implementar utilizando una etiqueta heredada pasada como un parámetro.

Con la siguiente declaración en C se puede implementar un árbol sintáctico abstracto para esta gramática:

```
typedef enum {ExpKind,IfKind,
      WhileKind,BreakKind,OtherKind} NodeKind;
typedef struct streenode
        { NodeKind kind;
          struct streenode * child[3];
          int val; /* usado con ExpKind */
        } STreeNode;
typedef STreeNode *SyntaxTree;
```

En esta estructura de árbol sintáctico, un nodo puede tener hasta tres hijos (un nodo if con una parte else), y los nodos de expresión son constantes con valor verdadero o falso (almacenado en el campo **val** como un 1 o un 0). Por ejemplo, la sentencia

```
if(true)while(true)if(false)break else other
```

tiene el árbol sintáctico

donde únicamente hemos mostrado los hijos no nulos de cada nodo.[9]

En la figura 8.12 se proporciona un procedimiento de generación de código que genera código P utilizando los **typedef** dados y la correspondiente estructura de árbol sintáctico. Acerca de este código hacemos las observaciones siguientes:

9. La ambigüedad del else ambiguo en esta gramática se resuelve mediante la regla estándar de la "anidación más cercana", como lo muestra el árbol sintáctico.

Figura 8.12
Procedimiento de generación de código para sentencias de control

```
void genCode( SyntaxTree t, char * label)
{ char codestr[CODESIZE];
  char * lab1, * lab2;
  if (t != NULL) switch (t->kind)
  { case ExpKind:
      if (t->val==0) emitCode("ldc false");
      else emitCode("ldc true");
      break;
    case IfKind:
      genCode(t->child[0],label);
      lab1 = genLabel();
      sprintf(codestr,"%s %s","fjp",lab1);
      emitCode(codestr);
      genCode(t->child[1],label);
      if (t->child[2] != NULL)
      { lab2 = genLabel();
        sprintf(codestr,"%s %s","ujp",lab2);
        emitCode(codestr);}
      sprintf(codestr,"%s %s","lab",lab1);
      emitCode(codestr);
      if (t->child[2] != NULL)
      { genCode(t->child[2],label);
        sprintf(codestr,"%s %s","lab",lab2);
        emitCode(codestr);}
      break;
    case WhileKind:
      lab1 = genLabel();
      sprintf(codestr,"%s %s","lab",lab1);
      emitCode(codestr);
      genCode(t->child[0],label);
      lab2 = genLabel();
      sprintf(codestr,"%s %s","fjp",lab2);
      emitCode(codestr);
      genCode(t->child[1],lab2);
      sprintf(codestr,"%s %s","ujp",lab1);
      emitCode(codestr);
      sprintf(codestr,"%s %s","lab",lab2);
      emitCode(codestr);
      break;
    case BreakKind:
      sprintf(codestr,"%s %s","ujp",label);
      emitCode(codestr);
      break;
    case OtherKind:
      emitCode("Other");
      break;
    default:
      emitCode("Error");
      break;
  }
}
```

En primer lugar, el código supone, como antes, la existencia de un procedimiento **emitCode** (este procedimiento simplemente imprimiría la cadena que se le pase). El código también supone la existencia de un procedimiento **genLabel** sin parámetros que devuelve nombres de etiqueta en la secuencia **L1**, **L2**, **L3**, y así sucesivamente.

El procedimiento **genCode** aquí tiene un parámetro **label** extra que es necesario para generar un salto absoluto en el caso de una sentencia break. Este parámetro sólo se cambia en la llamada recursiva que procesa el cuerpo de una sentencia while. De este modo, una sentencia break siempre provocará un salto fuera de la sentencia while anidada más cercana. (La llamada inicial a **genCode** puede utilizar una cadena vacía como el parámetro **label**, y cualquier sentencia break que se encuentre fuera de una sentencia while generará entonces un salto a una etiqueta vacía y, en consecuencia, un error.)

Advierta también cómo las variables locales **lab1** y **lab2** se emplean para guardar nombres de etiqueta, para los cuales existen saltos y/o definiciones aún pendientes.

Por último, puesto que una sentencia other no corresponde a ningún código real, en este caso este procedimiento simplemente genera la instrucción en código no P "**Other**".

Dejamos al lector describir la operación del procedimiento de la figura 8.12 y mostrar que para la sentencia

```
if(true)while(true)if(false)break else other
```

se genera la secuencia de código

```
ldc true
fjp L1
lab L2
ldc true
fjp L3
ldc false
fjp L4
ujp L3
ujp L5
lab L4
Other
lab L5
ujp L2
lab L3
lab L1
```

8.5 GENERACIÓN DE CÓDIGO DE LLAMADAS DE PROCEDIMIENTOS Y FUNCIONES

Las llamadas de procedimientos y funciones son el último mecanismo de lenguaje que comentamos en términos generales en este capítulo. La descripción del código intermedio y del código objetivo para este mecanismo es, mucho más que otros mecanismos de lenguaje, complicada por el hecho de que diferentes máquinas objetivo utilizan mecanismos muy diferentes para realizar llamadas y por el hecho de que las llamadas dependen mucho de la organización del ambiente de ejecución. Por consiguiente, es difícil obtener una representación

de código intermedio que sea tan general como para emplearla en cualquier ambiente de ejecución o arquitectura objetivo.

8.5.1 Código intermedio para procedimientos y funciones

Los requerimientos para las representaciones de código intermedio de llamadas de funciones pueden describirse en términos generales como sigue. En primer lugar, existen en realidad *dos* mecanismos que necesitan descripción: la **definición** de función/procedimiento (también conocida como **declaración**) y la **llamada** de función/procedimiento.[10] Una definición crea un nombre de función, parámetros y código, pero la función no se ejecuta en ese punto. Una llamada crea valores reales para los parámetros (o **argumentos** para la llamada) y realiza un salto hacia el código de la función, el cual se ejecuta entonces y regresa. El ambiente de ejecución en el que ésta tiene lugar no se conoce cuando se crea el código para la función, excepto en su estructura general. Este ambiente de ejecución es construido en parte por el elemento que llama, y en parte por el código de la función llamada; esta división de la responsabilidad forma parte de la **secuencia de llamada** estudiada en el capítulo anterior.

El código intermedio para una definición debe incluir una instrucción que marque el inicio, o **punto de entrada**, del código para la función, y una instrucción que marque el final, o **punto de retorno**, de la función. De manera esquemática podemos escribir esto de la manera siguiente:

Instrucción de entrada
<código para el cuerpo de la función>
Instrucción de retorno

De la misma manera, una llamada de función debe tener una instrucción que indique el principio del cálculo de los argumentos (en preparación para la llamada) y luego una instrucción de llamada real que indique el punto en que los argumentos han sido construidos y el salto real hacia el código de la función puede tener lugar:

Instrucción de comienzo de cálculo de argumento
<código para calcular los argumentos>
Instrucción de llamada

Diferentes versiones del código intermedio tienen versiones muy diferentes de estas cuatro instrucciones agrupadas, en particular respecto a la cantidad de información acerca del ambiente, los parámetros y la función misma que es parte de cada instrucción. Ejemplos típicos de tal información incluyen el número, tamaño y ubicación de los parámetros; el tamaño del marco de pila; el tamaño de las variables locales y el espacio temporal, así como diversas indicaciones del uso del registro por la función llamada. Como es habitual, presentaremos código intermedio que contiene una mínima cantidad de información en las instrucciones mismas, con la idea de que cualquier información necesaria puede mantenerse separadamente en una entrada de la tabla de símbolos para el procedimiento.

---

10. A lo largo de este texto adoptamos el criterio de que las funciones y procedimientos representan en esencia el mismo mecanismo y, sin abundar en el tema, consideraremos que son lo mismo para los propósitos de esta sección. La diferencia es, por supuesto, que una función debe tener disponible un valor devuelto para el elemento que llama cuando salga y el elemento que llama debe saber dónde encontrarlo.

Código de tres direcciones para procedimientos y funciones En código de tres direcciones, la instrucción de entrada necesita dar un nombre al punto de entrada de procedimiento, semejante a la instrucción **label**; de este modo, es una instrucción de una dirección a la que llamaremos simplemente **entry**. De manera similar, llamaremos **return** a la instrucción de retorno. Esta instrucción también es una instrucción de una dirección: debe dar el nombre del valor devuelto, si existe uno.

Por ejemplo, considere la definición de la función en C

```
int f(int x, int y)
{ return x+y+1; }
```

Esto se traducirá al siguiente código de tres direcciones:

```
entry f
t1 = x + y
t2 = t1 + 1
return t2
```

En el caso de una llamada, de hecho necesitamos tres diferentes instrucciones de tres direcciones: una para señalar el inicio del cálculo del argumento, a la que llamaremos **begin_args** (y la cual es una instrucción de dirección cero); una instrucción que se utiliza de manera repetida para especificar los nombres de los valores de argumento, a la que denominaremos **arg** (y que debe incluir la dirección o nombre del valor del argumento); y finalmente, la instrucción de llamada real, que denotaremos simplemente como **call**, la cual también es una instrucción de una dirección (debe proporcionarse el nombre o punto de entrada de la función que se está llamando).

Por ejemplo, supongamos que la función **f** se definió en C como en el ejemplo anterior. Entonces, la llamada

```
f(2+3,4)
```

se traduce al código de tres direcciones

```
begin_args
t1 = 2 + 3
arg t1
arg 4
call f
```

Aquí enumeramos los argumentos en orden de izquierda a derecha. El orden podría, por supuesto, ser diferente. (Véase la sección 7.3.1, página 361.)

Código P para procedimientos y funciones La instrucción de entrada en código P es **ent**, y la instrucción de retorno es **ret**. De esta manera, la definición anterior de la función **f** de C se traduce al código P

```
ent f
lod x
```

```
lod y
adi
ldc 1
adi
ret
```

Advierta que la instrucción **ret** no necesita un parámetro para indicar cuál es el valor devuelto: se supone que el valor devuelto está en el tope de la pila de la máquina P al retorno.

Las instrucciones en código P para una llamada son la instrucción **mst** y la instrucción **cup**. La instrucción **mst** viene de "mark stack" y corresponde a la instrucción en código de tres direcciones **begin_args**. La razón por la que se denomina "mark stack" es que el código objetivo generado de una instrucción de esta naturaleza está involucrado con el establecimiento del registro de activación para la nueva llamada en la pila, es decir, los primeros pasos en la secuencia de llamada. Esto generalmente significa, entre otras cosas, que el espacio debe ser asignado o "marcado" en la pila para elementos tales como los argumentos. La instrucción **cup** en código P es la instrucción de "procedimiento de llamada de usuario" y corresponde directamente a la instrucción **call** del código de tres direcciones. La razón de que reciba este nombre es que el código P distingue dos clases de llamadas: **cup** y **csp**, o "call standard procedure". Un procedimiento estándar es un procedimiento "integrado" requerido por la definición del lenguaje, como los procedimientos **sin** o **abs** de Pascal (C no tiene procedimientos integrados de los cuales hablar). Las llamadas a procedimientos integrados pueden emplear conocimiento específico acerca de su funcionamiento para mejorar la eficiencia de la llamada (o inclusive eliminarla). No consideraremos más la operación **csp**.

Advierta que no introducimos una instrucción en código P equivalente a la instrucción **arg** en código de tres direcciones. En vez de eso se supone que todos los valores de argumentos aparecen en la pila (en el orden apropiado) cuando se encuentra la instrucción **cup**. Esto puede producir un orden un poco diferente para la secuencia de llamada que el correspondiente al código de tres direcciones (véanse los ejercicios).

Nuestro ejemplo de una llamada en C (la llamada **f(2+3,4)** para la función **f** descrita anteriormente) se traduce ahora en el siguiente código P:

```
mst
ldc 2
ldc 3
adi
ldc 4
cup f
```

(De nueva cuenta, calculamos los argumentos de izquierda a derecha.)

8.5.2 Un procedimiento de generación de código para definición de función y llamada

Como en secciones anteriores, deseamos mostrar un procedimiento de generación de código para una gramática de muestra con definiciones de función y llamadas. La gramática que utilizaremos es la que se muestra a continuación:

program → *decl-list exp*
decl-list → *decl-list decl* | ε
decl → **fn id** (*param-list*) = *exp*
param-list → *param-list* , **id** | **id**
exp → *exp* + *exp* | *llamada* | **num** | **id**
llamada → **id** (*arg-list*)
arg-list → *arg-list* , *exp* | *exp*

Esta gramática define un programa como una secuencia de declaraciones de funciones, seguidas por una expresión simple. No hay variables o asignaciones en esta gramática, sólo parámetros, funciones y expresiones, las cuales pueden incluir llamadas de función. Todos los valores son enteros, todas las funciones devuelven enteros y todas las funciones deben tener por lo menos un parámetro. Existe sólo una operación numérica (aparte de la llamada de función): la adición entera. Un ejemplo de un programa definido mediante esta gramática es

```
fn f(x)=2+x
fn g(x,y)=f(x)+y
g(3,4)
```

Este programa contiene dos definiciones de funciones seguidas por una expresión que es una llamada a **g**. También hay una llamada a **f** en el cuerpo de **g**.

Queremos definir una estructura de árbol sintáctico para esta gramática. Lo haremos utilizando las siguientes declaraciones en C:

```
typedef enum
  {PrgK,FnK,ParamK,PlusK,CallK,ConstK,IdK}
  NodeKind;
typedef struct streenode
        { NodeKind kind;
          struct streenode *lchild,*rchild,
                           *sibling;
          char * name; /* usado con FnK,ParamK,
                          CallK,IdK */
          int val; /* usado con ConstK */
        } STreeNode;
typedef STreeNode *SyntaxTree;
```

Tenemos 7 diferentes clases de nodos en esta estructura de árbol. Cada árbol sintáctico tiene una raíz que es un nodo **PrgK**. Este nodo es utilizado simplemente para agrupar las declaraciones y la expresión del programa en conjunto. Un árbol sintáctico contiene exactamente un nodo así. El hijo izquierdo de este nodo es una lista de hermanos de nodos **FnK**; el hijo derecho es la expresión del programa asociado. Cada nodo **FnK** tiene un hijo izquierdo que es una lista de hermanos de nodos **ParamK**. Estos nodos definen los nombres de los parámetros. El cuerpo de cada función es el hijo derecho de su nodo **FnK**. Los nodos de expresión se representan de la manera habitual, sólo que un nodo **CallK** contiene el nombre de la función llamada y tiene un hijo derecho que es una lista de hermanos de las expresiones de argumento. Por ejemplo, el programa de muestra anterior tiene el árbol sintáctico que se da en la figura 8.13. Por claridad incluimos en esa figura la clase de nodo correspondiente a cada nodo, junto con algunos atributos nombre/valor. Los hijos y hermanos se distinguen por la dirección (hermanos a la derecha, hijos debajo).

Figura 8.13
Árbol sintáctico abstracto para el programa de muestra en la página 440

Dada esta estructura de árbol sintáctico, en la figura 8.14 se proporciona un procedimiento de generación de código que produce código P. Acerca de éste código hacemos las siguientes observaciones. En primer lugar, el código para un nodo **PrgK** simplemente recorre el resto del árbol que tiene por debajo. El código para un **IdK**, **ConstK** o **PlusK** es casi idéntico al de los ejemplos anteriores.

Figura 8.14
Procedimiento de generación de código para definición de función y llamada

```
void genCode( SyntaxTree t)
{ char codestr[CODESIZE];
  SyntaxTree p;
  if (t != NULL)
  switch (t->kind)
  { case PrgK:
      p = t->lchild;
      while (p != NULL)
      { genCode(p);
        p = p->sibling; }
      genCode(t->rchild);
      break;
    case FnK:
      sprintf(codestr,"%s %s","ent",t->name);
      emitCode(codestr);
      genCode(t->rchild);
      emitCode("ret");
      break;
    case ParamK: /* sin acciones */
      break;
    case ConstK:
      sprintf(codestr,"%s %d","ldc",t->val);
      emitCode(codestr);
      break;
    case PlusK:
      genCode(t->lchild);
      genCode(t->rchild);
      emitCode("adi");
      break;
```

Figura 8.14 Continuación

```
      case IdK:
        sprintf(codestr,"%s %s","lod",t->name);
        emitCode(codestr);
        break;
      case CallK:
        emitCode("mst");
        p = t->rchild;
        while (p!=NULL)
        { genCode(p);
          p = p->sibling; }
        sprintf(codestr,"%s %s","cup",t->name);
        emitCode(codestr);
        break;
      default:
        emitCode("Error");
        break;
    }
  }
```

Restan los casos de **FnK, ParamK** y **CallK**. El código para un nodo **FnK** simplemente rodea el código para el cuerpo de la función (el hijo derecho) con **ent** y **ret**; los parámetros de la función nunca son visitados. En realidad, los nodos de parámetro nunca provocan que se genere código: los argumentos ya han sido calculados por el elemento que llama.[11] Esto explica también por qué no hay acciones para un nodo **ParamK** en la figura 8.14; en efecto, debido al comportamiento del código para **FnK**, no deberían ser alcanzados nodos **ParamK** en el recorrido del árbol, de modo que este caso en realidad podría ser desechado.

El caso final es **Callk**. El código para este caso resulta de una instrucción **mst**, que procede a generar el código para cada argumento, y finalmente de una instrucción **cup**.

Dejamos al lector demostrar que el procedimiento de generación de código de la figura 8.14 produciría el siguiente código P, dado el programa cuyo árbol sintáctico se muestra en la figura 8.13:

```
ent f
ldc 2
lod x
adi
ret
ent g
mst
```

11. Sin embargo, los nodos de parámetro tienen un importante papel de administración en el sentido que determinan las posiciones relativas, o desplazamientos, donde se pueden encontrar los parámetros en el registro de activación. Supondremos que esto es manejado en otra parte.

```
lod x
cup f
lod y
adi
ret
mst
ldc 3
ldc 4
cup g
```

8.6 GENERACIÓN DE CÓDIGO EN COMPILADORES COMERCIALES: DOS CASOS DE ESTUDIO

En esta sección examinaremos la salida de código ensamblador producida por dos diferentes compiladores comerciales para diferentes procesadores. El primero es el compilador para C de Borland versión 3.0 para procesadores Intel 80×86. El segundo es el compilador para C de Sun versión 2.0 para SparcStations. Mostraremos la salida de ensamblador de estos compiladores para los mismos ejemplos de código que utilizamos para ilustrar el código de tres direcciones y el código P.[12] Esto debería mejorar la comprensión de las técnicas de generación de código, así como de la conversión de código intermedio a código objetivo. También debería suministrar una comparación útil para el código de máquina producido por el compilador TINY, el cual se comentará en secciones posteriores.

8.6.1 El compilador C de Borland 3.0 para la arquitectura 80 × 86

Comenzamos nuestros ejemplos de la salida de este compilador con la asignación utilizada en la sección 8.2.1:

```
(x=x+3)+4
```

Supondremos que la variable **x** en esta expresión está almacenada localmente en el marco de pila.

El código ensamblado para esta expresión tal como se produce mediante el compilador Borland versión 3.0 para la arquitectura Intel 80×86 es como se muestra a continuación:

```
mov     ax,word ptr [bp-2]
add     ax,3
mov     word ptr [bp-2],ax
add     ax,4
```

En este código el registro de acumulador **ax** se utiliza como la ubicación temporal principal para el cálculo. La ubicación de la variable local **x** es **bp-2**, lo que refleja el uso del registro **bp** (base pointer) como el apuntador de marco y el hecho de que las variables enteras ocupan dos bytes en esta máquina.

La primera instrucción mueve el valor de **x** hacia **ax** (los corchetes en la dirección **[bp-2]** indican una carga indirecta más que inmediata). La segunda instrucción agrega la

12. Para los propósitos de éste y la mayoría de otros ejemplos, las optimizaciones que estos compiladores realizan están desactivadas.

constante 3 a este registro. La tercera instrucción mueve entonces este valor a la ubicación de **x**. Finalmente, la cuarta instrucción agrega 4 a **ax**, de manera que el valor final de la expresión se deja en este registro, donde se puede utilizar para cálculos adicionales.

Observe que la dirección de **x** para la asignación en la tercera instrucción no está calculada previamente (como una instrucción **lda** de código P podría sugerir). Una simulación estática del código intermedio, junto con el conocimiento de los modos de direccionamiento disponibles, pueden aplazar el cálculo de la dirección de **x** hasta este punto.

Referencias de arreglo Utilizamos como ejemplo la expresión en C

```
(a[i+1]=2)+a[j]
```

(véase el ejemplo de la generación del código intermedio en la página 422). También suponemos que **i**, **j** y **a** son variables locales declaradas como

```
int i,j;
int a[10];
```

El compilador C de Borland genera el siguiente código ensamblado para la expresión (con el fin de facilitar la referencia numeramos las líneas del código):

```
(1)   mov   bx,word ptr [bp-2]
(2)   shl   bx,1
(3)   lea   ax,word ptr [bp-22]
(4)   add   bx,ax
(5)   mov   ax,2
(6)   mov   word ptr [bx],ax
(7)   mov   bx,word ptr [bp-4]
(8)   shl   bx,1
(9)   lea   dx,word ptr [bp-24]
(10)  add   bx,dx
(11)  add   ax,word ptr [bx]
```

Puesto que los enteros tienen un tamaño de 2 en esta arquitectura, **bp-2** es la ubicación de **i** en el registro de activación local, **bp-4** es la ubicación de **j**, y la dirección base de **a** es **bp-24** (24 = tamaño de índice de arreglo de 10 × tamaño de entero de 2 bytes + espacio de 4 bytes para **i** y **j**). De este modo, la instrucción 1 carga el valor de **i** en **bx**, y la instrucción 2 multiplica ese valor por 2 (un desplazamiento a la izquierda de 1 bit). La instrucción 3 carga la dirección base de **a** en **ax** (**lea** = load effective address), pero ya ajustada al agregar 2 bytes para la constante 1 en la expresión subindizada **i+1**. En otras palabras, el compilador ha aplicado el hecho algebraico de que

$$\begin{aligned} \mathit{address}(\texttt{a[i+1]}) &= \mathit{base_address}(\texttt{a}) + (\texttt{i}+1)*\mathit{elem_size}(\texttt{a}) \\ &= (\mathit{base_address}(\texttt{a}) + \mathit{elem_size}(\texttt{a})) + \texttt{i}*\mathit{elem_size}(\texttt{a}) \end{aligned}$$

La instrucción 4 calcula entonces la dirección resultante de **a[i+1]** en **bx**. La instrucción 5 mueve la constante 2 al registro **ax**, y la instrucción 6 almacena esto para la dirección de **a[i+1]**. La instrucción 7 carga entonces el valor de **j** en **bx**, la instrucción 8 multiplica este valor por 2, la instrucción 9 carga la dirección base de **a** en el registro **dx**, la instrucción 10 calcula la dirección de **a[j]** en **bx** y la instrucción 11 agrega el valor almacenado

en esta dirección al contenido de **ax** (a saber, 2). El valor resultante de la expresión se deja en el registro **ax**.

Referencias de apuntador y campo Tomaremos las declaraciones de ejemplos anteriores:

```
typedef struct rec
   { int i;
     char c;
     int j;
   } Rec;

typedef struct treeNode
   { int val;
     struct treeNode * lchild, * rchild;
   } TreeNode;
...
Rec x;
TreeNode *p;
```

También supondremos que **x** y **p** se declaran como variables locales y que se hizo la asignación apropiada de apuntadores.

Consideremos en primer lugar los tamaños de los tipos de datos involucrados. En la arquitectura 80×86 las variables enteras tienen un tamaño de 2 bytes, las variables carácter tienen un tamaño de 1 byte y los apuntadores (a menos que se declare de otra manera) son los denominados apuntadores "cercanos" con un tamaño de 2 bytes.[13] De este modo, la variable **x** tiene un tamaño de 5 bytes y la variable **p** de 2 bytes. Con estas dos variables declaradas localmente como las únicas variables, **x** se asigna en el registro de activación en la localidad **bp-6** (las variables locales están asignadas sólo en límites pares de bytes, una restricción típica, de modo que el byte extra queda sin utilizar), y **p** se asigna al registro **si**. Adicionalmente, en la estructura en **Rec**, **i** tiene desplazamiento 0, **c** tiene desplazamiento 2 y **j** tiene desplazamiento 3, mientras que en la estructura **TreeNode**, **val** tiene desplazamiento 0, **lchild** tiene desplazamiento 2 y **rchild** tiene un desplazamiento de 4.

El código generado para la sentencia

```
x.j = x.i;
```

es

```
mov     ax,word ptr [bp-6]
mov     word ptr [bp-3],ax
```

La primera instrucción carga **x.i** en **ax**, y la segunda almacena este valor para **x.j**. Advierta cómo el cálculo del desplazamiento para **j** ($-6 + 3 = -3$) se realiza estáticamente por el compilador.

El código generado para la sentencia

```
p->lchild = p;
```

13. La arquitectura 80×86 tiene una clase más general de apuntadores denominados apuntadores "far" ("lejanos") con un tamaño de cuatro bytes.

es

```
mov     word ptr [si+2],si
```

Advierta cómo la indirección y el cálculo del desplazamiento se combinan en una sola instrucción.

Finalmente, el código generado para la sentencia

```
p = p->rchild;
```

es

```
mov     si,word ptr [si+4]
```

Sentencias if y while Mostraremos aquí el código generado mediante el compilador C de Borland para sentencias de control típicas. Las sentencias que utilizaremos son

```
if (x>y) y++;else x--;
```

y

```
while (x<y) y -= x;
```

En ambos casos, tanto **x** como **y** son variables enteras locales.

El compilador de Borland genera el siguiente código 80×86 para la sentencia if dada, donde **x** se encuentra localizada en el registro **bx**, mientras que **y** está localizada en el registro **dx**:

```
        cmp     bx,dx
        jle     short @1@86
        inc     dx
        jmp     short @1@114
@1@86:
        dec     bx
@1@114:
```

Este código utiliza la misma organización secuencial de la figura 8.10, pero advierta que este código no calcula el valor lógico real de la expresión **x<y** sino que simplemente emplea de manera directa el código de condición.

El código generado mediante el compilador Borland para la sentencia while es el que se muestra a continuación:

```
        jmp     short @1@170
@1@142:
        sub     dx,bx
@1@170:
        cmp     bx,dx
        jl      short @1@142
```

Esto utiliza una organización secuencial diferente a la de la figura 8.11, en el sentido que la prueba se ubica al final, y se hace un salto incondicional inicial para esta prueba.

Definición de funciones y llamadas Los ejemplos que utilizaremos son la definición de la función en C

```
int f(int x, int y)
{ return x+y+1; }
```

y una llamada correspondiente

```
f(2+3,4)
```

(estos ejemplos fueron empleados en la sección 8.5.1).

Consideremos, en primer lugar, el código del compilador de Borland para la llamada `f(2+3,4)`:

```
mov     ax,4
push    ax
mov     ax,5
push    ax
call    near ptr _f
pop     cx
pop     cx
```

Advierta cómo los argumentos se insertan en la pila en orden inverso: primero el 4, luego el 5. (El valor 5 = 2 + 3 es calculado previamente por el compilador, ya que es una expresión constante.) Advierta también que el elemento que llama es responsable de eliminar los argumentos de la pila después de la llamada; ésta es la razón para las dos instrucciones **pop** después de la llamada. (El registro objetivo **cx** se emplea como un receptáculo de "basura": los valores extraídos nunca se utilizan.) La llamada misma debería ser relativamente autoexplicativa: el nombre **_f** tiene una subraya colocada antes del nombre fuente, lo que es una convención típica para compiladores C, y la declaración **near ptr** significa que la función está en el mismo segmento (y de ese modo sólo necesita una dirección de dos bytes). Por último observamos que la instrucción **call** en la arquitectura 80×86 inserta automáticamente la dirección de retorno en la pila (y una instrucción correspondiente **ret**, que se ejecute mediante la función llamada, la extraerá).

Ahora consideremos el código generado por el compilador de Borland para la definición de **f**:

```
_f      proc    near
        push    bp
        mov     bp,sp
        mov     ax,word ptr [bp+4]
        add     ax,word ptr [bp+6]
        inc     ax
        jmp     short @1@58
@1@58:
        pop     bp
        ret
_f      endp
```

Gran parte de este código debería ser claro. Como el elemento que llama no construye el registro de activación excepto para calcular e insertar los argumentos, este código debe terminar

su construcción antes de que el código del cuerpo de la función sea ejecutado. Ésta es la tarea de la segunda y tercera instrucciones, que graban el vínculo de control (**bp**) en la pila y entonces establecen el **bp** al **sp** actual (el cual se conmuta a la activación actual). Después de estas operaciones la pila tiene el aspecto que se muestra a continuación:

La dirección de retorno se encuentra en la pila entre el vínculo de control (el **bp** antiguo) y los argumentos como un resultado de la ejecución de una instrucción **call** por el elemento que llama. De este modo, el **bp** antiguo está en el tope de la pila, la dirección de retorno se encuentra en la localidad **bp+2** (las direcciones son de dos bytes en este ejemplo), el parámetro **x** está en la localidad **bp+4** y el parámetro **y** en la localidad **bp+6**. El cuerpo de **f** corresponde entonces al código que viene a continuación

```
mov     ax,word ptr [bp+4]
add     ax,word ptr [bp+6]
inc     ax
```

que carga **x** en **ax**, agrega **y** al mismo, y después lo incrementa en uno.

Finalmente, el código ejecuta un salto (que aquí es innecesario, pero de todas formas se genera porque las sentencias **return** incrustadas en el código de la función podrían necesitarlo), recupera el **bp** antiguo de la pila y regresa al elemento que llama. El valor devuelto se deja en el registro **ax**, donde estará disponible para el elemento que llama.

8.6.2 El compilador C de Sun 2.0 para las Sun SparcStations

Repetiremos los ejemplos de código utilizados en las secciones anteriores para demostrar la salida de este compilador. Del mismo modo que antes, comenzamos con la asignación

```
(x=x+3)+4
```

y suponemos que la variable **x** en esta expresión es almacenada localmente en el marco de la pila.

El compilador C de Sun produce código ensamblado que es muy similar al del código Borland:

```
ld      [%fp+-0x4],%o1
add     %o1,0x3,%o1
st      %o1,[%fp+-0x4]
ld      [%fp+-0x4],%o2
add     %o2,0x4,%o3
```

En este código, los nombres de registro comienzan con el símbolo de porcentaje, y las constantes comienzan con los caracteres **0x** (**x** = hexadecimal), de manera que, por ejemplo, **0x4** es el hexadecimal 4 (lo mismo que el decimal 4). La primera instrucción mueve el valor de **x** (en la localidad **fp-4**, porque los enteros son de cuatro bytes de longitud) al registro **o1**.[14] (Observe que las localidades fuente están a la izquierda y las ubicaciones objetivo a la derecha, de modo opuesto a la convención de la arquitectura Intel 80×86.) La segunda instrucción agrega la constante 3 a **o1**, mientras que la tercera almacena **o1** en la localidad de **x**. Finalmente, se vuelve a cargar el valor de **x**, esta vez en el registro **o2**,[15] y se le agrega 4, colocando el resultado en el registro **o3**, donde se deja como el valor final de la expresión.

Referencias de arreglo La expresión

```
(a[i+1]=2)+a[j]
```

con todas las variables locales se traduce al siguiente código ensamblado mediante el compilador de Sun (con las instrucciones numeradas para facilitar la referencia):

```
(1)   add    %fp,-0x2c,%o1
(2)   ld     [%fp+-0x4],%o2
(3)   sll    %o2,0x2,%o3
(4)   mov    0x2,%o4
(5)   st     %o4,[%o1+%o3]
(6)   add    %fp,-0x30,%o5
(7)   ld     [%fp+-0x8],%o7
(8)   sll    %o7,0x2,%l0
(9)   ld     [%o5+%l0],%l1
(10)  mov    0x2,%l2
(11)  add    %l2,%l1,%l3
```

Los cálculos realizados por este código son afectados por el hecho de que los enteros son de cuatro bytes en esta arquitectura. De esta manera, la localidad de **i** es **fp-4** (escrito como **%fp+-0x4** en este código), la localidad de **j** es **fp-8** (**%fp+-0x8**) y la dirección base de **a** es **fp-48** (escrita **%fp+-0x30**, puesto que el 48 decimal es igual al 30 hexadecimal).

---

14. En la SparcStation los registros se indican mediante una letra minúscula seguida por un número. Diferentes letras se refieren a diferentes secciones de una "ventana de registro" que puede cambiar con el contexto y que corresponde aproximadamente a usos diferentes de los registros. Las distinciones entre las diferentes letras son irrelevantes para todos los ejemplos en este capítulo, excepto para el que involucre llamadas de función (véase la sección 8.5).

15. Este paso no aparece en el código de Borland y es fácilmente optimizado. Véase la sección 8.9.

La instrucción 1 carga al registro `o1` con la dirección base de `a`, modificada (como en el código Borland) para restar 4 bytes a la constante 1 en el subíndice `i+1` (2c hex = 44 = 48 − 4). La instrucción 2 carga el valor de `i` en el registro `o2`. La instrucción 3 multiplica `o2` por 4 (un desplazamiento a la izquierda de 2 bits) y pone el resultado en `o3`. La instrucción 4 carga el registro `o4` con la constante 2, y la instrucción 5 la almacena en la dirección de `a[i+1]`. La instrucción 6 calcula la dirección base de `a` en `o5`, la instrucción 7 carga el valor de `j` en `o7`, y la instrucción 8 lo multiplica por 4, colocando el resultado en el registro `10`. Finalmente, la instrucción 9 carga el valor de `a[j]` en `11`, la instrucción 10 vuelve a cargar la constante 2 en `12`, y la instrucción 11 las suma, colocando el resultado (y el valor final de la expresión) en el registro `13`.

Referencias de apuntador y campo Consideraremos el mismo ejemplo que antes (véase la página 445). En Sun, los tamaños de los tipos de datos son los siguientes: las variables enteras tienen un tamaño de 4 bytes, las variables de carácter tienen un tamaño de 1 byte y los apuntadores tienen un tamaño de 4 bytes. Sin embargo, todas las variables, incluyendo los campos de estructura, son asignadas sólo en límites de 4 bytes. De este modo, la variable `x` tiene un tamaño de 12 bytes, la variable `p` tiene 4 bytes y los desplazamientos de `i`, `c` y `j` son 0, 4, 8, respectivamente, como lo son los desplazamientos de `val`, `lchild` y `rchild`. El compilador asigna tanto `x` como `p` en el registro de activación: `x` en la localidad `fp-0xc` (hexadecimal c = 12) y `p` en la localidad `fp-0x10` (hexadecimal 10 = 16).

El código generado para la asignación

```
x.j = x.i;
```

es

```
ld          [%fp+-0xc],%o1
st          %o1,[%fp+-0x4]
```

Este código carga el valor de `x.i` en el registro `o1` y lo almacena en `x.j` (advierta de nuevo cómo el desplazamiento de `x.j` = −12 + 8 = −4 se calcula estáticamente).

La asignación de apuntador

```
p->lchild = p;
```

produce el código objetivo

```
ld          [%fp+-0x10],%o2
ld          [%fp+-0x10],%o3
st          %o3,[%o2+0x4]
```

En este caso el valor de `p` se carga en los registros `o2` y `o3`. Una de estas copias (`o2`) se utiliza entonces como la dirección base para almacenar la otra copia en la localidad de `p ->lchild`.

Finalmente, la asignación

```
p = p->rchild;
```

produce el código objetivo

```
ld      [%fp+-0x10],%o4
ld      [%o4+0x8],%o5
st      %o5,[%fp+-0x10]
```

En este código el valor de **p** es cargado en el registro **o4** y posteriormente utilizado como la dirección base para cargar el valor de **p->rchild**. La instrucción final almacena este valor en la localidad de **p**.

Sentencias if y while Mostramos el código generado por el compilador de C de la SparcStation de Sun por las mismas sentencias de control típicas que para el compilador de Borland

```
if (x>y) y++;else x--;
```

y

```
while (x<y) y -= x;
```

con **x** y **y** como variables enteras locales.

El compilador de la SparcStation de Sun genera el siguiente código para la sentencia if, donde tanto **x** como **y** se localizan en el registro de activación local en los desplazamientos –4 y –8:

```
        ld      [%fp+-0x4],%o2
        ld      [%fp+-0x8],%o3
        cmp     %o2,%o3
        bg      L16
        nop
        b       L15
        nop
L16:
        ld      [%fp+-0x8],%o4
        add     %o4,0x1,%o4
        st      %o4,[%fp+-0x8]
        b       L17
        nop
L15:
        ld      [%fp+-0x4],%o5
        sub     %o5,0x1,%o5
        st      %o5,[%fp+-0x4]
L17:
```

Aquí las instrucciones **nop** siguen cada instrucción de ramificación debido a que la Sparc está canalizada (las ramificaciones son aplazadas y la siguiente instrucción siempre se ejecuta antes que tenga efecto la ramificación). Observe que la organización secuencial es la opuesta a la de la figura 8.10: el caso verdadero se coloca después del caso falso.

El código generado por el compilador Sun para el bucle o ciclo while es

```
        ld      [%fp+-0x4],%o7
        ld      [%fp+-0x8],%l0
        cmp     %o7,%l0
        bl      L21
        nop
        b       L20
        nop
L21:
L18:
        ld      [%fp+-0x4],%l1
        ld      [%fp+-0x8],%l2
        sub     %l2,%l1,%l2
        st      %l2,[%fp+-0x8]
        ld      [%fp+-0x4],%l3
        ld      [%fp+-0x8],%l4
        cmp     %l3,%l4
        bl      L18
        nop
        b       L22
        nop
L22:
L20:
```

Este código utiliza un arreglo similar al del compilador de Borland, sólo que el código de prueba también se duplica al principio. También, al final de este código se crea una ramificación "que no hace nada" (a la etiqueta `L22`), la cual se puede eliminar fácilmente mediante un paso de optimización.

Definición de funciones y llamadas Utilizamos la misma definición de función que antes

```
int f(int x, int y)
{ return x+y+1; }
```

y la misma llamada

```
f(2+3,4)
```

El código generado por el compilador Sun para la llamada es

```
mov     0x5,%o0
mov     0x4,%o1
call    _f,2
```

y el código generado para la definición de `f` es

```
_f:
        !#PROLOGUE# 0
        sethi     %hi(LF62),%g1
        add       %g1,%lo(LF62),%g1
        save      %sp,%g1,%sp
        !#PROLOGUE# 1
        st        %i0,[%fp+0x44]
        st        %i1,[%fp+0x48]
L64:
        .seg      "text"
        ld        [%fp+0x44],%o0
        ld        [%fp+0x48],%o1
        add       %o0,%o1,%o0
        add       %o0,0x1,%o0
        b         LE62
        nop
LE62:
        mov       %o0,%i0
        ret
        restore
```

No analizaremos este código con detalle, pero haremos los comentarios siguientes. En primer lugar, la llamada pasa los argumentos en los registros `o0` y `o1`, en vez de en la pila. La llamada indica con el número `2` cuántos registros se utilizan para este propósito. La llamada también realiza varias funciones de administración, las cuales no describiremos, excepto para decir que los registros "o" (como `o0` y `o1`) se convierten en los registros "i" (por ejemplo, `i0` e `i1`) después de la llamada (y los registros "i" vuelven a ser los registros "o" al retorno de la llamada).[16]

El código para la definición de `f` también comienza con algunas instrucciones de administración que finalizan la secuencia de llamada (entre los comentarios `!#PROLOGUE#`), las cuales tampoco describiremos. El código almacena entonces los valores de parámetro para la pila (de los registros "i", que son los mismos que los registros "o" del elemento que llama). Después que se calcula el valor devuelto, se coloca en el registro `i0` (donde estará disponible después del retorno como registro `o0`).

8.7 TM: UNA MÁQUINA OBJETIVO SIMPLE

En la siguiente sección presentaremos un generador de código para el lenguaje TINY. Para hacer de ésta una tarea significativa, generamos código objetivo directamente para una máquina muy simple que se pueda simular con facilidad. Llamaremos a esta máquina TM (por Tiny Machine) y en el apéndice C proporcionaremos un listado de programa en C de un simulador TM completo que se puede emplear para ejecutar el código producido por el generador de código TINY. En la sección actual describiremos la arquitectura TM completa y el conjunto de instrucciones, además del uso del simulador del apéndice C. Con el fin de

16. Hablando libremente, la "o" es para "output" y la "i" es para "input". Esta conmutación de registros a través de las llamadas se realiza mediante un mecanismo de **ventana de registro** de la arquitectura Sparc.

facilitar la comprensión emplearemos fragmentos de código en C para ayudar a esta descripción, y las mismas instrucciones TM siempre se darán en formato de código ensamblado en vez de hacerlo en códigos binario o hexadecimal. (En cualquier caso el simulador sólo lee código ensamblado.)

8.7.1 Arquitectura básica de la máquina TINY

TM se compone de una memoria de instrucción sólo de lectura, una memoria de datos, y un conjunto de ocho registros de propósito general. Todos éstos utilizan direcciones enteras no negativas que comienzan en 0. El registro 7 es el contador del programa y es el único registro especial, como se describe a continuación. Las declaraciones en C

```
#define IADDR_SIZE ...
      /* tamaño de la memoria de la instrucción */
#define DADDR_SIZE ...
      /* tamaño de la memoria de datos */
#define NO_REGS 8 /* número de registros */
#define PC_REG 7

Instruction iMem[IADDR_SIZE];
int dMem[DADDR_SIZE];
int reg[NO_REGS];
```

se utilizarán en las descripciones que siguen.

La TM realiza un ciclo convencional de obtención-ejecución:

```
do
  /* obtención */
  currentInstruction = iMem[reg[pcRegNo]++] ;
  /* ejecuta la instrucción actual */
  ...
while (!(halt||error)) ;
```

Al comienzo, la máquina TINY establece todos los registros y memoria de datos a 0, luego carga el valor de la dirección legal más alta (a saber, **DADDR_SIZE** – 1) en **dMem[0]**. (Esto permite que la memoria se agregue fácilmente a la TM, puesto que los programas pueden averiguar durante la ejecución de cuánta memoria se dispone.) La TM comienza entonces a ejecutar las instrucciones comenzando en **iMem[0]**. La máquina se detiene cuando se ejecuta una instrucción **HALT**. Las condiciones de error posibles incluyen a **IMEM_ERR**, que se presenta si **reg[PC_REG]** < 0 o si **reg[PC_REG]** ≥ **IADDR_SIZE** en el paso de obtención anterior, y las dos condiciones **DMEM_ERR** y **ZERO_DIV**, las cuales se presentan durante la ejecución de instrucciones como se describió anteriormente.

El conjunto de instrucciones de la TM se proporciona en la figura 8.15, junto con una breve descripción del efecto de cada instrucción. Existen dos formatos de instrucción básicos: sólo de registro, o instrucciones RO y de memoria de registro, o instrucciones RM. Una instrucción sólo de registro tiene el formato

```
opcode r,s,t
```

Figura 8.15
Conjunto completo de instrucciones de la máquina TINY

Instrucciones RO

Formato: *opcode r,s,t*

Opcode	Efecto
HALT	detener ejecución (operandos ignorados)
IN	reg[r] ← lectura de valor entero desde la entrada estándar (s y t ignorados)
OUT	reg[r] → la salida estándar (s y t ignorados)
ADD	reg[r] = reg[s] + reg[t]
SUB	reg[r] = reg[s] - reg[t]
MUL	reg[r] = reg[s] * reg[t]
DIV	reg[r] = reg[s] / reg[t] (puede generar ZERO_DIV)

Instrucciones RM

Formato: *opcode r,d(s)*

(a = d + reg[s]; cualquier referencia a dMem[a] genera DMEM_ERR si a < 0 o a ≥ DADDR_SIZE)

Opcode	Efecto
LD	reg[r] = dMem[a] (carga r con valor de la memoria en a)
LDA	reg[r] = a (carga dirección a directamente en r)
LDC	reg[r] = d (carga constante d directamente en r—s es ignorada)
ST	dMem[a] = reg[r] (almacena valor en r a localidad de memoria a)
JLT	if (reg[r]<0) reg[PC_REG] = a (salta a instrucción a si r es negativa, de manera similar para el siguiente)
JLE	if (reg[r]<=0) reg[PC_REG] = a
JGE	if (reg[r]>=0) reg[PC_REG] = a
JGT	if (reg[r]>0) reg[PC_REG] = a
JEQ	if (reg[r]==0) reg[PC_REG] = a
JNE	if (reg[r]!=0) reg[PC_REG] = a

donde los operandos r, s, t son registros legales (verificados en tiempo de carga). De este modo, tales instrucciones son tres direcciones, y las tres direcciones deben ser registros. Todas las instrucciones aritméticas se encuentran limitadas a este formato, como lo están las dos instrucciones primitivas de entrada/salida.

Una instrucción de memoria de registro tiene el formato

opcode r,d(s)

En este código r y s deben ser registros legales (verificados en tiempo de carga), y d es un entero positivo o negativo que representa un desplazamiento. Esta instrucción es una instrucción de dos direcciones, donde la primera dirección es siempre un registro y la segunda dirección es una dirección de memoria a dada por a = d + reg[r], donde a debe ser una dirección legal (0 ≤ a < DADDR_SIZE). Si a está fuera del intervalo legal, entonces se genera DMEM_ERR durante la ejecución.

Las instrucciones RM incluyen tres instrucciones de carga diferentes que corresponden a los tres modos de direccionamiento "carga constante" (**LDC**), "dirección de carga" (**LDA**) y "memoria de carga" (**LD**). Además, existe una instrucción de almacenamiento y seis instrucciones de salto condicional.

Tanto en las instrucciones RO como en las RM, los tres operandos deben estar presentes, aun cuando algunos de ellos pueden ser ignorados. Esto se debe a la naturaleza simple del cargador, el cual sólo distingue las dos clases de instrucciones (RO y RM) y no permite diferentes formatos de instrucción dentro de cada clase.[17]

La figura 8.15 y el análisis de la TM hasta este punto representan la arquitectura completa TM. En particular, no hay pila de hardware u otras facilidades de ninguna clase. Ningún registro, excepto el pc, es especial en modo alguno (no hay sp o fp). Un compilador para la TM debe, por lo tanto, mantener alguna organización del ambiente de ejecución de manera enteramente manual. Aunque esto puede ser algo poco realista, tiene la ventaja de que todas las operaciones se deben generar explícitamente a medida que sean necesarias.

Debido a que el conjunto de instrucciones es mínimo, algunos comentarios están dispuestos acerca de cómo se puede utilizar para conseguir casi todas las operaciones estándar de un lenguaje de programación (en realidad ésta es una máquina objetivo adecuado, si no es que hasta cómodo incluso para lenguajes muy sofisticados).

1. El registro objetivo en las operaciones aritméticas, **IN**, y de carga viene en primer lugar, y el (los) registro(s) fuente vienen en segundo, de manera similar a la arquitectura 80×86 y diferente a la de la SparcStation de Sun. No hay restricción sobre el uso de los registros para fuentes y objetivos; en particular, los registros fuente y objetivo pueden ser los mismos.
2. Todas las operaciones aritméticas están restringidas a registros. Ninguna operación (excepto las operaciones de carga y almacenamiento) actúa directamente en la memoria. En esto, la TM se parece a las máquinas RISC, tales como la SparcStation de Sun. Por otra parte, la TM tiene sólo 8 registros, mientras que la mayoría de los procesadores RISC tienen por lo menos 32.[18]
3. No hay operaciones de punto flotante o registros de punto flotante. Aunque no sería muy difícil agregar un coprocesador a la TM con registros y operaciones de punto flotante, la traducción de los valores de punto flotante hacia y desde los registros regulares y la memoria requerirían de alguna precaución. Remitiremos al lector a los ejercicios.
4. No existen modos de direccionamiento especificables en los operandos como en algún código ensamblado (tal como **LD #1** para modo inmediato, o **LD @a** para indirecto). En vez de eso existen diferentes instrucciones para cada modo: **LD** es indirecto, **LDA** es directo y **LDC** es inmediato. En realidad, la TM tiene muy pocas opciones de direccionamiento.
5. No existe restricción para el uso del pc en ninguna de las instrucciones. Efectivamente, puesto que no hay instrucción de salto incondicional, ésta debe ser simulada mediante el uso del pc como el registro objetivo en una instrucción **LDA**:

```
LDA 7, d(s)
```

Esta instrucción tiene el efecto de saltar a la localidad $a = d + reg[s]$.

---

17. También facilita el trabajo de un generador de código, ya que sólo serán necesarias dos rutinas por separado para las dos clases de instrucciones.

18. Aunque sería fácil incrementar el número de registros TM, aquí no lo haremos, ya que no se necesitan en la generación de código básico. Véanse los ejercicios al final del capítulo.

6. Tampoco hay instrucción de salto indirecto, pero también se puede imitar, si es necesario, mediante el uso de una instrucción **LD**. Por ejemplo,

```
LD 7,0(1)
```

salta a la instrucción cuya dirección está almacenada en la memoria en la localidad a la que apunta el registro 1.

7. Las instrucciones de salto condicional (**JLT**, etc.) pueden hacerse relativas a la posición actual en el programa mediante el uso del pc como el segundo registro. Por ejemplo,

```
JEQ 0,4(7)
```

provoca que la TM salte cinco instrucciones hacia delante en el código si el registro 0 es 0. Un salto incondicional también puede hacerse relativo al pc utilizando dos veces el pc en una instrucción **LDA**. De este modo,

```
LDA 7, -4(7)
```

realiza un salto incondicional de tres instrucciones hacia atrás.

8. No existe llamada de procedimiento o instrucción **JSUB**. En su lugar debemos escribir

```
LD 7,d(s)
```

que tiene el efecto de saltar al procedimiento cuya dirección de entrada es **dMem-[d+reg[s]]**. Naturalmente, debemos recordar grabar la dirección de retorno ejecutando primero algo como

```
LDA 0,1(7)
```

que coloca el valor actual del pc más uno en **reg[0]** (que es a donde queremos regresar, suponiendo que la instrucción siguiente es el salto real al procedimiento).

8.7.2 El simulador de TM

El simulador de máquina acepta archivos de texto que contienen instrucciones de TM de la manera en que se describieron anteriormente, con las convenciones siguientes:

1. Una línea completamente en blanco se ignora.
2. Una línea que comienza con un asterisco se considera un comentario y se ignora.
3. Cualquier otra línea debe contener una localidad de instrucción entera seguida por un signo de dos puntos seguido por una instrucción legal. Cualquier texto que se presente después de la instrucción se considera un comentario y se ignora.

El simulador de TM no contiene otras características, en particular no hay etiquetas simbólicas ni facilidades de macro. En la figura 8.16 se proporciona un programa de muestra TM escrito manualmente que corresponde al programa TINY de la figura 8.1.

Estrictamente hablando, la instrucción **HALT** al final del código de la figura 8.16 no es necesaria, puesto que el simulador de TM establece todas las localidades de instrucción a **HALT** antes de cargar el programa. Sin embargo, es útil conservarla como un recordatorio, y como un objetivo para saltos que deseen salir del programa.

Figura 8.16
Un programa TM que muestra las convenciones de formato

```
* Este programa introduce un entero, calcula
* su factorial si es positivo,
* e imprime el resultado

0:     IN   0,0,0        r0 = read
1:     JLE  0,6(7)       if 0 < r0 then
2:     LDC  1,1,0          r1 = 1
3:     LDC  2,1,0          r2 = 1
                         * repite
4:     MUL 1,1,0            r1 = r1 * r0
5:     SUB 0,0,2            r0 = r0 - r2
6:     JNE  0,-3(7)        until r0 == 0
7:     OUT  1,0,0          write r1
8:     HALT 0,0,0        halt
* fin del programa
```

Tampoco es necesario que aparezcan localidades en secuencia ascendente como hicimos en la figura 8.16. Cada línea de entrada es efectivamente una directiva del tipo "almacena esta instrucción en esta localidad": si un programa TM fuera perforado en tarjetas, sería aceptable tirarlas y que se desordenaran en el piso antes de leerlas en la TM. Aunque esta propiedad del simulador de TM podría causar alguna confusión al lector de un programa, facilita ajustar saltos cuando no hay etiquetas simbólicas, ya que el código se puede ajustar sin apoyarse en el archivo del código. Por ejemplo, es probable que un generador de código genere el código de la figura 8.16 en la secuencia que se muestra a continuación:

```
0:   IN   0,0,0
2:   LDC  1,1,0
3:   LDC  2,1,0
4:   MUL  1,1,0
5:   SUB  0,0,2
6:   JNE  0,-3(7)
7:   OUT  1,0,0
1:   JLE  0,6(7)
8:   HALT 0,0,0
```

Esto se debe a que el salto hacia adelante en la instrucción 1 no se puede generar hasta que se conozca la ubicación después del cuerpo de la sentencia if.

Si el programa de la figura 8.16 está en el archivo **fact.tm**, entonces este archivo se puede cargar y ejecutar como en la siguiente sesión de muestra (el simulador de TM automáticamente presupone una extensión de archivo **.tm** si no se proporciona alguna otra):

```
tm fact
TM simulation (enter h for help) ...
Enter command: g
Enter value for IN instruction: 7
OUT instruction prints: 5040
```

```
HALT: 0,0,0
Halted
Enter command: q
Simulation done.
```

El comando **g** representa "ir a", lo que significa que el programa se ejecuta comenzando en el contenido actual del pc (que es 0 justo después de la carga), hasta que se detecta una instrucción **HALT**. La lista completa de comandos del simulador se puede obtener mediante el uso del comando **h**, el cual imprime la lista que se muestra a continuación:

```
Los comandos son:*
   s(tep <n>        Execute n (default 1) TM instructions
   g(o              Execute TM instructions until HALT
   r(egs            Print the contents of the registers
   i(Mem <b <n>>    Print n iMem locations starting at b
   d(Mem <b <n>>    Print n dMem locations starting at b
   t(race           Toggle instruction trace
   p(rint           Toggle print of total instructions executed
                   ('go' only)
   c(lear           Reset simulator for new execution of program
   h(elp            Cause this list of commands to be printed
   q(uit            Terminate the simulation
```

El paréntesis izquierdo en cada comando indica el mnemónico a partir del cual se deriva la letra del comando (también es aceptable utilizar más de una letra, pero el simulador sólo examinará la primera). Los picoparéntesis < > señalan los parámetros opcionales.

8.8 UN GENERADOR DE CÓDIGO PARA EL LENGUAJE TINY

Ahora deseamos describir un generador de código para el lenguaje TINY. Suponemos que el lector está familiarizado con los pasos previos del compilador TINY, en particular con la estructura de árbol sintáctico generada por el analizador sintáctico como se describió en el capítulo 3, la construcción de la tabla de símbolos como fue descrita en el capítulo 6 y el ambiente de ejecución descrito en el capítulo 7.

En esta sección describiremos primero la interfaz del generador de código TINY para la TM, junto con las funciones utilitarias necesarias para la generación del código. Posteriormente describiremos los pasos del generador de código apropiado. En tercer lugar describiremos el uso del compilador TINY en combinación con el simulador TM. Finalmente, analizaremos el archivo del código objetivo para el programa TINY de muestra que se ha estado utilizando en este libro.

8.8.1 La interfaz TM del generador de código TINY

Encapsulamos algo de la información que el generador de código necesita conocer acerca de la TM en los archivos **code.h** y **code.c**, los cuales se listan en el apéndice B, líneas 1600-1685 y 1700-1796, respectivamente. También ponemos el código que emite funciones en estos archivos. Naturalmente, el generador de código todavía necesitará conocer los nombres de instrucción de TM, pero estos archivos separan los detalles del formateo de

* Traducción por línea: Ejecuta n (por omisión 1) instrucciones TM. Ejecuta las instrucciones TM hasta que encuentra a HALT. Imprime el contenido de los registros. Imprime n localidades de iMem iniciando en b. Imprime n localidades de dMem iniciando en b. Fija la instrucción de trazado. Activa la impresión del total de las instrucciones ejecutadas (sólo con 'go'). Reinicia el simulador para una nueva ejecución del programa. Presenta esta lista de comandos. Termina la simulación.

instrucción, la ubicación del archivo del código objetivo y el uso de registros particulares en el ambiente de ejecución. De manera importante, el archivo **code.c** encapsula completamente la secuencia de instrucciones en localidades **iMem** particulares, así que el generador de código no tiene que mantenerse al tanto de este detalle. Si se fuera a mejorar el cargador de TM, para permitir etiquetas simbólicas y para eliminar la necesidad de números de localidad por ejemplo, entonces sería fácil incorporar la generación de etiquetas y cambios de formato en el archivo **code.c**.

Revisaremos aquí algunas de las características de las definiciones de función y constantes en el archivo **code.h**. En primer lugar consideraremos las declaraciones de los valores de registro (líneas 1612, 1617, 1623, 1626 y 1629). Evidentemente, el pc debe ser conocido por el generador de código y las utilidades de emisión de código. Más allá de esto, el ambiente de ejecución para el lenguaje TINY, como se describió en el capítulo anterior, asigna localidades en la parte superior de la memoria de datos para elementos temporales (de una manera tipo pila) y las localidades en la parte inferior de la memoria de datos a las variables. Como no hay registros de activación (y, por lo tanto, tampoco fp) en TINY (debido a que no hay ámbitos o llamadas de procedimiento), las localidades para variables y elementos temporales se pueden visualizar como direcciones absolutas. Sin embargo, la operación **LD** de la máquina TM no permite el modo de direccionamiento absoluto, sino que debe tener un valor base de registro en su cálculo de la dirección para una carga de memoria. Así, asignamos dos registros, a los que denominamos el mp (memory pointer) y gp (global pointer), para apuntar a la parte superior o tope de la memoria y a la parte inferior de ésta, respectivamente. El mp será utilizado para acceso de la memoria a los elementos temporales y siempre contendrá la localidad más alta de memoria legal, mientras que el gp será utilizado para todos los accesos de memoria variable nombrada y siempre contendrá 0, de manera que las direcciones absolutas calculadas mediante la tabla de símbolos se pueden emplear como desplazamientos relativos al gp. Por ejemplo, si un programa tiene dos variables **x** y **y**, y se tienen dos valores temporales actualmente almacenados en memoria, entonces **dMem** tendría un aspecto como el siguiente:

En esta ilustración **t1** tiene dirección **0(mp)**, **t2** tiene dirección **-1(mp)**, **x** tiene dirección **0(gp)** y la **y** tiene dirección **1(gp)**. En esta implementación gp es el registro 5 y mp es el registro 6.

Los otros dos registros que serán utilizados por el generador de código son los registros 0 y 1, los cuales son llamados "acumuladores" y a los que se les asignan los nombres ac y ac1. Éstos se emplearán como registros abiertos. En particular, los resultados de los cálculos siempre se dejarán en ac. Advierta que los registros 2, 3 y 4 no están nombrados (¡y nunca serán utilizados!).

Volveremos ahora a comentar las siete funciones de emisión de código cuyos prototipos están dados en el archivo **code.h**. La función **emitComment** imprime su cadena de parámetro en el formato de un comentario en una línea por separado en el archivo del código, si la señal o "bandera" **TraceCode** está establecida. Las siguientes dos funciones **emitRO** y **emitRM** son las funciones estándar de emisión de código que se emplean por las clases RO y RM de instrucciones, respectivamente. Además de la cadena de instrucción y tres operandos, cada función toma un parámetro de cadena adicional, el cual se agrega a la instrucción como un comentario (si la señal **TraceCode** está establecida).

Las siguientes tres funciones se emplean para saltos de generación y ajuste. La función **emitSkip** se utiliza para saltar un número de localidades que se ajustarán posteriormente, además de regresar a la localidad de instrucción actual, la cual se mantiene de manera interna dentro de **code.c**. Por lo común esto se utiliza sólo en las llamadas **emitSkip(1)**, que salta una localidad simple que posteriormente será rellenada con una instrucción de salto, y **emitSkip(0)**, que no salta localidades, sino que sólo es llamada para obtener la localidad de la instrucción actual a fin de grabarla para una referencia posterior en un salto hacia atrás. La función **emitBackup** se utiliza para establecer la localidad de la instrucción actual a una localidad anterior para ajuste, y **emitRestore** se emplea para devolver la localidad de la instrucción actual al valor antes de una llamada a **emitBackup**. Generalmente, estas instrucciones se utilizan juntas como

```
emitBackup(savedLoc) ;
/* genera instrucción de salto de ajuste aquí */
emitRestore() ;
```

La función de emisión de código final (**emitRM_Abs**) es necesaria para generar el código para un salto de ajuste o cualquier salto a una localidad de código devuelta por una llamada a **emitSkip**. Regresa una dirección de código absoluta en una dirección relativa al pc al restar la localidad de instrucción actual más 1 (que es lo que será el pc durante la ejecución) desde el parámetro de localidad pasado, y utilizando el pc como el registro fuente. Por lo general esta función sólo será utilizada con una instrucción de salto condicional, como **JEQ**, o para generar un salto incondicional empleando **LDA** y el pc como registro objetivo, de la manera en que se describió en la subsección anterior.

Esto completa la descripción de las utilidades de generación de código TINY, y avanzaremos a una descripción del generador de código TINY en sí.

8.8.2 El generador de código TINY

El generador de código TINY está contenido en el archivo **cgen.c**, con su única interfaz hacia el compilador TINY, la función **codeGen**, con prototipo

```
void codeGen(void);
```

dada como la única declaración en su archivo de interfaz **cgen.h**. Un listado completo del archivo **cgen.c** se proporciona en el apéndice B, líneas 1900-2111.

La función **codeGen** en sí (líneas 2095-2111) hace muy poco: genera unos cuantos comentarios e instrucciones (llamados **preludio estándar**) que establecen el ambiente de ejecución al inicio, posteriormente llama a la función **cGen** en el árbol sintáctico, y por último genera una instrucción **HALT** para finalizar el programa. El preludio estándar consta de dos instrucciones: la primera carga el registro mp con la localidad de memoria legal

más alta (que el simulador TM ha colocado en la localidad cero al inicio), mientras que la segunda borra la localidad 0. (El gp no necesita ser establecido a 0, puesto que todos los registros se establecen a 0 en el inicio.)

La función **cGen** (líneas 2070-2084) es responsable de llevar a cabo el recorrido del árbol sintáctico de TINY que genera código en postorden modificado, de la manera descrita en secciones anteriores. Recuerde que un árbol sintáctico de TINY tiene una forma dada por las declaraciones

```
typedef enum {StmtK,ExpK} NodeKind;
typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK} StmtKind;
typedef enum {OpK,ConstK,IdK} ExpKind;

#define MAXCHILDREN 3
typedef struct treeNode
   { struct treeNode * child[MAXCHILDREN];
     struct treeNode * sibling;
     int lineno;
     NodeKind nodekind;
     union { StmtKind stmt; ExpKind exp;} kind;
     union { TokenType op;
             int val;
             char * name; } attr;
     ExpType type;
   } TreeNode;
```

Existen dos clases de nodos de árbol: los nodos de sentencia y los nodos de expresión. Si un nodo es un nodo de sentencia, entonces representa una de las cinco clases diferentes de sentencias de TINY (la condicional if, la de ciclo repeat, la de asignación assign, la de lectura read o la de escritura write), y si un nodo es un nodo de expresión, entonces representa una de las tres clases de expresiones (un identificador, una constante entera o un operador). La función **cGen** prueba sólo si un nodo es un nodo de sentencia o de expresión (o nulo), llamando a la función apropiada **genStmt** o **genExp**, y llamándose más adelante a sí misma de manera recursiva en los nodos hermanos (de manera que las listas hermanas tendrán código generado en orden de izquierda a derecha).

La función **genStmt** (líneas 1924-1994) contiene una gran sentencia **switch** que distingue entre las cinco clases de sentencias, generando el código apropiado y llamadas recursivas a **cGen** en cada caso, y de manera semejante para la función **genExp** (líneas 1997-2065). En todos los casos se supone que el código para una subexpresión deja un valor en el ac, donde se puede tener acceso al mismo mediante un código posterior. En los casos en que es necesario el acceso a una variable (sentencias de asignación y lectura y expresiones de identificador), se tiene acceso a la tabla de símbolos mediante la operación

```
loc = lookup(tree->attr.name);
```

El valor de **loc** es la dirección de la variable en cuestión y se utiliza como el desplazamiento con el registro gp para cargar o almacenar su valor.

El otro caso en el cual son necesarios los accesos a memoria es en el cálculo del resultado de una expresión de operador, donde el operando a la izquierda debe ser almacenado

en una localidad temporal antes de calcular el operando de la derecha. De este modo, el código para una expresión de operador incluye la siguiente secuencia de generación de código antes que se pueda aplicar el operador (líneas 2021-2027):

```
cGen(p1); /* p1 = hijo izquierdo */
emitRM("ST",ac,tmpOffset--,mp,"op: push left");
cGen(p2); /* p2 = hijo derecho */
emitRM("LD",ac1,++tmpOffset,mp,"op: load left");
```

Aquí **tmpOffset** es una variable estática que es inicializada a 0 y empleada como el desplazamiento de la siguiente localidad temporal disponible desde la parte superior o tope de la memoria (a la que se apunta mediante el registro mp). Advierta cómo **tmpOffset** se decrementa después de cada almacenamiento y se incrementa después de cada carga. De esta forma, **tmpOffset** se puede ver como un apuntador hacia el tope de la "pila temporal", y las llamadas a la función **emitRM** corresponden a una inserción y extracción de esta pila. Esto protege cada elemento temporal mientras se mantiene en la memoria. Después de que el código generado por las acciones anteriores es ejecutado, el operando del lado izquierdo estará en el registro 1 (ac1) y el operando del lado derecho en el registro 0 (ac). Puede entonces generarse la operación apropiada RO, en el caso de las operaciones aritméticas.

El caso de los operadores de comparación es un poco diferente. La semántica del lenguaje TINY (como se implementó mediante el analizador semántico; véase el capítulo anterior) permite a los operadores de comparación sólo en las expresiones de prueba de las sentencias if y las sentencias while. No hay variables booleanas u otros valores además de en tales pruebas, y los operadores de comparación se podrían abordar con ellas en la generación de código de esas sentencias. No obstante, aquí utilizaremos un enfoque más general, el cual es más ampliamente aplicable a lenguajes con operaciones lógicas y/o valores booleanos, e implementaremos el resultado de una prueba como un 0 (para falso) o 1 (para verdadero), como en C. Esto requiere que la constante 0 o 1 sea cargada de manera explícita en el ac, lo que se consigue haciendo saltos para ejecutar la carga correcta. Por ejemplo, en el caso del operador "menor que" se genera el código siguiente, una vez que se ha generado el código para calcular el operando del lado izquierdo en el registro 1 y el operando del lado derecho en el registro 0:

```
SUB   0,1,0
JLT   0,2(7)
LDC   0,0(0)
LDA   7,1(7)
LDC   0,1(0)
```

La primera instrucción resta el operando derecho del izquierdo, y deja al resultado en el registro 0. Si < es verdadero, este resultado será negativo, y la instrucción **JLT 0,2(7)** provocará un salto sobre dos instrucciones hasta la última instrucción, la cual carga el valor 1 en el ac. Si < es falso, entonces la tercera y la cuarta instrucciones se ejecutan, lo que carga 0 en el ac y entonces salta sobre la última instrucción (recuerde de la descripción de la TM que cuando **LDA** utiliza el pc como ambos registros provoca un salto incondicional).

Finalizamos la descripción del generador de código TINY con un análisis de la sentencia if (líneas 1930-1954). Los casos restantes se dejan al lector.

La primera acción del generador de código para una sentencia if es generar código para la expresión de prueba. El código para la prueba, como acaba de describirse, deja el 0 en el ac en el caso falso y 1 en el caso verdadero. El generador de código debe generar a continuación una **JEQ** para el código para la parte else de la sentencia if. Sin embargo, la ubicación de este código aún no se conoce, puesto que el código para la parte "then" todavía debe ser generado. Por lo tanto, el generador de código utiliza la utilidad **emitSkip** para saltar la sentencia siguiente y grabar su ubicación para ajuste posterior:

```
savedLoc1 = emitSkip(1) ;
```

La generación de código continúa con la generación del código para la parte then de la sentencia if. Después de ello, se debe generar un salto incondicional sobre el código de la parte else, pero nuevamente esa ubicación no se conoce, de modo que la localidad para este salto también se debe omitir y su ubicación se debe grabar:

```
savedLoc2 = emitSkip(1) ;
```

Ahora, sin embargo, el siguiente paso es generar código para la parte else, de modo que la ubicación del código actual es el objetivo correcto para el salto en el caso falso, el cual ahora se debe ajustar a la ubicación **savedLoc1**, que es lo que realiza el código siguiente:

```
currentLoc = emitSkip(0) ;
emitBackup(savedLoc1) ;
emitRM_Abs("JEQ",ac,currentLoc,"if: jmp to else");
emitRestore() ;
```

Advierta cómo la llamada **emitSkip(0)** se utiliza para obtener la ubicación de la instrucción actual, y cómo el procedimiento **emitRM_Abs** se emplea para convertir el salto a esta localidad absoluta en un salto relativo al pc, el cual es requerido por la instrucción **JEQ**. Después de esto, finalmente se puede generar el código para la parte else, y entonces se ajusta un salto absoluto (**LDA**) a **savedLoc2** mediante una secuencia de código semejante.

8.8.3 Generación y uso de archivos de código TM con el compilador de TINY

El generador de código TINY es afinado para trabajar limpiamente con el simulador de TM. Cuando todas las marcas **NO_PARSE**, **NO_ANALYZE** y **NO_CODE** se establecen como falsas en el programa principal, el compilador crea un archivo de código con el sufijo o extensión **.tm** (suponiendo que no hay errores en el código fuente) y escribe las instrucciones de TM en este archivo en el formato requerido de simulador de TM. Por ejemplo, para compilar y ejecutar el programa **sample.tny**, sólo se necesita emitir los comandos siguientes:

```
tiny sample
<listado producido en la salida estándar>
tm sample
<ejecución del simulador tm>
```

Para propósitos de rastreo existe una marca **TraceCode** declarada en **globals.h** y definida en **main.c**. Si se establece como **TRUE**, la información de rastreo se genera mediante el generador de código, la cual aparece como comentarios en el archivo de código, indicando dónde se generó cada instrucción o secuencia de instrucciones en el generador de código, y por qué razón.

8.8.4 Un archivo de código TM de muestra generado por el compilador de TINY

Para mostrar con mayor detalle cómo funciona el generador de código, presentamos en la figura 8.17 el archivo de código realizado por el generador de código TINY para el programa de muestra de la figura 8.1, con **TraceCode = TRUE**, de manera que también sean generados comentarios de código. Este archivo de código tiene 42 instrucciones, incluyendo las dos instrucciones del preludio estándar. Al comparar esto con las nueve instrucciones del programa escrito a mano de la figura 8.16, podemos ver desde luego varias ineficiencias. En particular, el programa de la figura 8.16 utiliza los registros de manera muy eficiente: no se utiliza en absoluto ninguna otra memoria más que la de los registros. Por otra parte, el código de la figura 8.17 (páginas 466-467), nunca emplea más de dos registros y realiza muchos almacenamientos y cargas que son innecesarios. Particularmente torpe es la manera en que trata los valores de las variables, los que son cargados sólo para ser almacenados otra vez como elementos temporales, como ocurre en el código

```
16:     LD   0,1(5) load id value
17:     ST   0,0(6) op: push left
18:     LD   0,0(5) load id value
19:     LD   1,0(6) op: load left
```

el cual se puede reemplazar mediante las dos instrucciones

```
LD   1,1(5) load id value
LD   0,0(5) load id value
```

que tienen exactamente el mismo efecto.

Otras ineficiencias sustanciales se provocan mediante el código generado para las pruebas y saltos. Un ejemplo particularmente ridículo es la instrucción

```
40:     LDA 7,0(7) jmp to end
```

que es un **NOP** elaborado (una instrucción de "no operación").

No obstante, el código de la figura 8.17 tiene una propiedad importante: es correcto. En la prisa por mejorar la eficiencia del código generado, los escritores de compiladores en ocasiones olvidan la necesidad de esta propiedad y permiten que se genere código que, aunque eficiente, no siempre se ejecuta de manera correcta. Tal comportamiento, si no se encuentra bien documentado y es predecible (y a veces incluso así), puede conducir al desastre.

Aunque estudiar todas las maneras en que se puede mejorar el código producido por un compilador rebasa el alcance de este texto, en las dos secciones finales de este capítulo examinaremos las áreas principales en las que se pueden hacer tales mejoras, además de las técnicas que se pueden utilizar para implementarlas, y describiremos brevemente cómo se pueden aplicar algunas de ellas al generador de código TINY con el fin de mejorar el código que genera.

Figura 8.17
Salida de código TM para el programa de muestra TINY de la figura 8.1 (página 401)

```
* Compilación TINY para código TM
* Archivo: sample.tm
* Preludio estándar:
  0:     LD  6,0(0)     load maxaddress from location 0
  1:     ST  0,0(0)     clear location 0
* Fin del preludio estándar.
  2:     IN  0,0,0      read integer value
  3:     ST  0,0(5)     read: store value
* -> if
* -> Op
* -> Const
  4:    LDC  0,0(0)     load const
* <- Const
  5:     ST  0,0(6)     op: push left
* -> Id
  6:     LD  0,0(5)     load id value
* <- Id
  7:     LD  1,0(6)     op: load left
  8:    SUB  0,1,0      op <
  9:    JLT  0,2(7)     br if true
 10:    LDC  0,0(0)     false case
 11:    LDA  7,1(7)     unconditional jmp
 12:    LDC  0,1(0)     true case
* <- Op
* if: el salto al else pertenece aquí
* -> assign
* -> Const
 14:    LDC  0,1(0)     load const
* <- Const
 15:     ST  0,1(5)     assign: store value
* <- assign
* -> repeat
* repeat: el salto después del cuerpo regresa aquí
* -> assign
* -> Op
* -> Id
 16:     LD  0,1(5)     load id value
* <- Id
 17:     ST  0,0(6)     op: push left
* -> Id
 18:     LD  0,0(5)     load id value
* <- Id
 19:     LD  1,0(6)     op: load left
 20:    MUL  0,1,0      op *
```

Figura 8.17 Continuación

```
* <- Op
  21:       ST  0,1(5)     assign: store value
* <- assign
* -> assign
* -> Op
* -> Id
  22:       LD  0,0(5)     load id value
* <- Id
  23:       ST  0,0(6)     op: push left
* -> Const
  24:      LDC  0,1(0)     load const
* <- Const
  25:       LD  1,0(6)     op: load left
  26:      SUB  0,1,0      op -
* <- Op
  27:       ST  0,0(5)     assign: store value
* <- assign
* -> Op
* -> Id
  28:       LD  0,0(5)     load id value
* <- Id
  29:       ST  0,0(6)     op: push left
* -> Const
  30:      LDC  0,0(0)     load const
* <- Const
  31:       LD  1,0(6)     op: load left
  32:      SUB  0,1,0      op ==
  33:      JEQ  0,2(7)     br if true
  34:      LDC  0,0(0)     false case
  35:      LDA  7,1(7)     unconditional jmp
  36:      LDC  0,1(0)     true case
* <- Op
  37:      JEQ  0,-22(7)   repeat: jmp back to body
* <- repeat
* -> Id
  38:       LD  0,1(5)     load id value
* <- Id
  39:      OUT  0,0,0      write ac
* if: el salto al final pertenece aquí
  13:      JEQ  0,27(7)    if: jmp to else
  40:      LDA  7,0(7)     jmp to end
* <- if
* Fin de la ejecución.
  41:     HALT  0,0,0
```

8.9 UNA VISIÓN GENERAL DE LAS TÉCNICAS DE OPTIMIZACIÓN DE CÓDIGO

Desde los primeros compiladores de la década de 1950, la calidad del código generado por un compilador ha sido de gran importancia. Esta calidad puede ser mensurada tanto por la velocidad como por el tamaño del código objetivo, aunque generalmente se ha dado más importancia a la velocidad. Los compiladores modernos abordan el problema de la calidad de código al efectuar, en varios puntos durante el proceso de compilación, una serie de pasos que incluyen el obtener información acerca del código fuente y el uso posterior de ésta para realizar **transformaciones de mejoramiento de código** sobre las estructuras de datos que representan el código. Un gran número y variedad de técnicas para mejorar la calidad del código se han desarrollado a través de los años, y se han llegado a conocer como **técnicas de optimización de código**. Sin embargo, esta terminología es engañosa, puesto que sólo en situaciones muy especiales cualquiera de estas técnicas puede producir código optimizado en el sentido matemático. No obstante, el nombre es tan común que continuará utilizándose.

Existen tantas y diversas técnicas de optimización de código que aquí sólo podemos dar una pequeña visión fugaz de las más importantes y las más ampliamente utilizadas, e incluso para éstas no proporcionaremos los detalles de cómo implementarlas. Al final del capítulo se dan referencias de fuentes de información adicional al respecto. Sin embargo, es importante enfatizar que un escritor de compiladores no puede esperar incluir cada técnica de optimización simple que se haya desarrollado, pero debe considerar, para el lenguaje en cuestión, cuáles técnicas tienen más probabilidad de producir una mejora de código importante con el menor incremento de la complejidad del compilador. Se han escrito muchos artículos describiendo técnicas de optimización que requieren de implementaciones muy complejas, aunque las mejoras, en promedio, que producen en el código objetivo son relativamente pequeñas (digamos, una disminución de un pequeño porcentaje en el tiempo de ejecución). Sin embargo, la experiencia generalmente ha demostrado que unas cuantas técnicas básicas, aun cuando se apliquen en apariencia de manera muy simple, pueden conducir a mejoras más significativas, en ocasiones recortando el tiempo de ejecución a la mitad o incluso a más de la mitad.

Al juzgar si la implementación de una técnica de optimización en particular es demasiado compleja respecto a sus resultados en la mejora real del código, es importante determinar no sólo la complejidad de la implementación en términos de las estructuras de datos y código de compilador extra, sino también en el efecto que el paso de optimización pueda tener sobre la velocidad de ejecución del compilador mismo. Todas las técnicas de análisis sintáctico y análisis que hemos estudiado han sido dirigidas a un tiempo de ejecución lineal para el compilador; es decir, la velocidad de compilación es directamente proporcional al tamaño del programa que se está compilando. Muchas técnicas de optimización pueden incrementar el tiempo de compilación a una función cuadrática o cúbica del tamaño del programa, de manera que la compilación de un programa de proporciones considerables puede tomar minutos (o en el peor de los casos, horas) o mucho tiempo con optimización completa. Esto puede provocar que los usuarios eviten el uso de las optimizaciones (o eviten el compilador en conjunto), y que se desperdicie gran parte del tiempo que tomó implementar las optimizaciones.

En las secciones siguientes describiremos primero las fuentes principales de optimización y posteriormente las diversas clasificaciones de optimizaciones, para después inspeccionar de manera breve algunas de las técnicas más importantes junto con las principales estructuras de datos que se emplearon para implementarlas. Todo el tiempo proporcionaremos ejemplos simples de optimizaciones para ilustrar la exposición. En la sección siguiente daremos ejemplos más detallados de cómo se pueden aplicar algunas de las técnicas comentadas al generador de código TINY de la sección anterior, con sugerencias para métodos de implementación.

8.9.1 Fuentes principales de optimizaciones de código

En el texto que sigue describiremos algunas de las áreas en las cuales un generador de código puede fallar en producir código correcto, aproximadamente en orden decreciente de "liquidación", es decir, cuánto mejoramiento de código se puede obtener en cada área.

Asignación de registro El buen uso de los registros es la característica más importante del código eficiente. A lo largo de la historia el número de registros disponibles había estado severamente limitado: por lo regular a 8 o 16 registros, incluyendo los registros de propósito especial como el pc, sp y fp. Esto dificultaba la buena asignación de registros debido a que la competencia por el espacio de registros entre las variables y los elementos temporales era intensa. Esta situación persiste en algunos procesadores, particularmente en los microprocesadores. Un método para resolver este problema ha sido incrementar el número y velocidad de las operaciones que se pueden realizar directamente en la memoria, de manera que un compilador, una vez que ha agotado el espacio de registros, puede evitar el gasto de tener que reclamar registros almacenando valores de registro en localidades temporales y cargar nuevos valores (denominadas operaciones de **desbordamiento de registro**). Otro método (el denominado método RISC) consiste en *disminuir* el número de operaciones que se pueden realizar directamente en la memoria (a menudo a 0), pero al mismo tiempo incrementar el número de registros disponibles a 32, 64 o 128. En tales arquitecturas la asignación apropiada de registros se vuelve aún más crítica, puesto que se podrían mantener todas, o casi todas, las variables simples de un programa entero en registros. La penalización para el fracaso al asignar registros apropiadamente en tales arquitecturas se incrementa por la necesidad de cargar y almacenar valores en forma constante, de manera que se les puedan aplicar las operaciones. Al mismo tiempo, la labor de asignar registros se vuelve más fácil, ya que se dispone de muchos. De este modo, la buena asignación de registros debe ser la meta de cualquier esfuerzo serio por mejorar la calidad del código.

Operaciones innecesarias La segunda fuente principal de mejoramiento de código es el evitar la generación de código para las operaciones que sean redundantes o innecesarias. Esta clase de optimización puede variar desde una búsqueda muy simple de código localizado hasta el análisis complejo de las propiedades semánticas del programa completo. El número de posibilidades para la identificación de tales operaciones es grande, y existe un gran número correspondiente de técnicas empleadas para hacerlo. Un ejemplo típico de una oportunidad de optimización de esta clase es una expresión que aparece repetidamente en el código, y cuyo valor permanece igual. La evaluación repetida se puede eliminar guardando el primer valor para su uso posterior (denominada **eliminación de subexpresión común**).[19] Un segundo ejemplo es evitar almacenar el valor de una variable o elemento temporal que no sea utilizado posteriormente (esto va de la mano con la anterior optimización).

Una clase completa de oportunidades de optimización involucra la identificación del **código inalcanzable** o **código muerto**. Un ejemplo típico de código inalcanzable es el uso de una marca o "bandera" constante para activar y desactivar la información de depuración:

---

19. Mientras que un buen programador puede evitar, hasta cierto punto, expresiones comunes en el código fuente, el lector no debería suponer que esta optimización existe sólo para ayudar a los malos programadores. Muchas subexpresiones comunes resultan de cálculos de dirección que son generados por el mismo compilador y no pueden ser eliminadas al escribir un mejor código fuente.

```
#define DEBUG 0
...
if (DEBUG)
{...}
```

Si **DEBUG** se establece a 0 (como en este código), entonces el código entre llaves dentro de la sentencia if es inalcanzable, y el código objetivo no necesita generarse, ya sea para aquél o para la sentencia if encerrada. Otro ejemplo de código inalcanzable es un procedimiento que nunca es llamado (o que sólo es llamado desde un código que él mismo es inalcanzable). La eliminación del código inalcanzable por lo regular no afecta de manera importante la velocidad de ejecución, pero puede reducir de manera sustancial el tamaño del código objetivo, y vale la pena, en particular si sólo se utiliza un pequeño esfuerzo extra en análisis para identificar los casos más evidentes.

En ocasiones, al identificar operaciones innecesarias es más fácil continuar con la generación de código y luego probar la redundancia del código objetivo. Un caso en el que a veces es más difícil probar la redundancia por adelantado se encuentra en la generación de saltos para representar sentencias de control estructuradas. Tal código puede incluir saltos a la sentencia más próxima o saltos hacia saltos. Un paso de **optimización de saltos** puede eliminar estos saltos innecesarios.

Operaciones costosas Un generador de código no sólo debería buscar operaciones innecesarias, sino que debería tomar ventaja de las oportunidades para reducir el costo de operaciones que son necesarias, pero que se pueden implementar de maneras más económicas que lo que puede indicar el código fuente o una implementación simple. Un ejemplo típico de esto es el reemplazo de las operaciones aritméticas por operaciones más económicas. Por ejemplo, la multiplicación por 2 puede implementarse como una operación de desplazamiento, y una potencia entera pequeña, tal como x^3, puede implementarse como una multiplicación, tal como $x * x * x$. Esta optimización se conoce como **reducción de potencia**. Esto puede extenderse de varias formas, como el reemplazo de multiplicaciones que involucren pequeñas constantes enteras mediante desplazamientos y sumas (por ejemplo, reemplazando $5*x$ por $2*2*x + x$: dos desplazamientos y una suma).

Una optimización relacionada es emplear información acerca de constantes para eliminar tantas operaciones como sea posible o para precalcular tantas operaciones como sea posible. Por ejemplo, la suma de dos constantes, como 2 + 3, se puede calcular mediante el compilador y reemplazarse por el valor constante 5 (esto se conoce como **incorporación de constante**). En ocasiones vale la pena intentar determinar si una variable puede tener también un valor constante para una parte o la totalidad de un programa, y tales transformaciones entonces se pueden aplicar también a expresiones que involucren a esa variable (esto se denomina **propagación de constante**).

Una operación que en ocasiones puede ser relativamente costosa es la llamada de procedimiento, donde se deben realizar muchas operaciones de secuencia de llamada. Los procesadores modernos han reducido este costo de manera sustancial, ofreciendo apoyo de hardware para secuencias de llamada estándar, pero la eliminación de llamadas frecuentes a procedimientos pequeños aún puede producir aumentos de velocidad mensurables. Existen dos maneras estándar para eliminar llamadas de procedimiento. Una es reemplazar la llamada de procedimiento con el código para el cuerpo del procedimiento (con un reemplazo adecuado de parámetros por argumentos). Esto se denomina **descobertura de procedimiento**, y en ocasiones incluso es una opción de lenguaje (como en C++). Otra forma de eliminar las llamadas de procedimiento es reconocer la **recursividad de cola**, es decir, cuando la última operación de un procedimiento es para llamarse a sí mismo. Por ejemplo, el procedimiento

```
int gcd( int u, int v)
{ if (v==0) return u;
  else return gcd(v,u % v); }
```

es recursivo de cola, pero el procedimiento

```
int fact( int n )
{ if (n==0) return 1;
  else return n * fact(n-1); }
```

no lo es. La recursividad de cola es equivalente a asignar los valores de los nuevos argumentos a los parámetros y efectuar un salto hasta el inicio del cuerpo del procedimiento. Por ejemplo, el procedimiento recursivo de cola **gcd** se puede volver a escribir mediante el compilador para el código equivalente

```
int gcd( int u, int v)
{ begin:
    if (v==0) return u;
    else
    { int t1 = v, t2 = u%v;
      u = t1; v = t2;
      goto begin;
    }
}
```

(Advierta la sutil necesidad de elementos temporales en este código.) Este proceso se conoce como **eliminación de recursividad de cola**.

Una cuestión respecto a la llamada de procedimiento se relaciona con la asignación del registro. Debe hacerse la previsión en cada llamada de procedimiento para grabar y recuperar registros que permanezcan en uso durante la duración de la llamada. Si se proporciona una fuerte asignación de registros, se puede incrementar el costo de las llamadas de procedimiento, puesto que proporcionalmente será necesario grabar y recuperar más registros. En ocasiones este costo se puede reducir incluyendo las consideraciones de llamadas en el elemento que asigna el registro. El hardware de apoyo para llamadas con muchos registros en uso también puede reducir este costo.[20] Pero éste es un caso de un fenómeno común: a veces las optimizaciones pueden provocar que se invierta el efecto deseado, y deben hacerse convenios al respecto.

Durante la generación de código final se pueden encontrar algunas últimas oportunidades para reducir el costo de ciertas operaciones utilizando instrucciones especiales disponibles en la máquina objetivo. Por ejemplo, muchas arquitecturas incluyen operaciones de desplazamiento de bloques que son más rápidas que la copia de elementos individuales de cadenas o arreglos. También, en ocasiones, los cálculos de direcciones pueden ser optimizados cuando la arquitectura permite que se combinen diversos modos de direccionamiento o cálculos de desplazamiento dentro de una instrucción simple. De manera similar, a veces existen modos de autoincremento y autodecremento para su uso en la indización (la arquitectura VAX tiene incluso una instrucción de incremento-comparación-ramificación para bucles o ciclos). Tales optimizaciones vienen bajo el encabezado **selección de instrucciones** o **uso de idiomas de máquina**.

---

20. Las ventanas de registro en las SparcStation de Sun representan un ejemplo de soporte de hardware para asignación de registros a través de llamadas de procedimiento. Véase la página 453.

Comportamiento de programa predictivo Para realizar algunas de las optimizaciones previamente descritas, un compilador debe recolectar información acerca de los usos de las variables, valores y procedimientos en los programas: si las expresiones son reutilizadas (y así se vuelven subexpresiones comunes), si es que las variables cambian sus valores o permanecen constantes y cuándo ocurre esto, y si los procedimientos son o no llamados. Un compilador debe, dentro del alcance de sus técnicas de cálculo, hacer suposiciones basado en el peor de los casos acerca de la información que recolecta o arriesgarse a generar código incorrecto: si una variable puede o no ser constante en un punto particular, el compilador debe suponer que no es constante. Esta propiedad se denomina **estimación conservadora** de la información del programa. Significa que un compilador debe hacerlo con información poco menos que perfecta, y con frecuencia incluso de manera rudimentaria, acerca del comportamiento del programa. Naturalmente, cuanto más sofisticado se vuelva el análisis del programa, mejor será la información que un optimizador de código puede utilizar. No obstante, incluso en los compiladores más avanzados de la actualidad, puede haber propiedades comprobables de programas que no sean descubiertas por el compilador.

Un método diferente es tomado por algunos compiladores en que se obtiene un comportamiento estadístico acerca de un programa aparte de las ejecuciones reales, y posteriormente utilizado para predecir qué trayectorias tienen más probabilidad de ser tomadas, qué procedimientos tienen más probabilidad de ser llamados con frecuencia y qué secciones de código tienen mayores probabilidades de ser ejecutadas con más frecuencia. Esta información puede entonces utilizarse para ajustar la estructura de saltos, bucles y código de procedimientos con el fin de minimizar la velocidad de ejecución en la mayoría de las ejecuciones que se presentan comúnmente. Este proceso requiere, por supuesto, que un **compilador de rendimiento** así tenga acceso a los datos apropiados y que (por lo menos algo de) el código ejecutable contenga código de instrumentación para generarlos.

8.9.2 Clasificación de optimizaciones

Como hay tantas oportunidades de optimización y técnicas, vale la pena reducir la complejidad de su estudio adoptando varios esquemas de clasificación para enfatizar las calidades diferentes de las optimizaciones. Dos clasificaciones útiles son el tiempo durante el proceso de compilación en que se puede aplicar una optimización y el área del programa sobre la cual se aplica la optimización.

En primer lugar, consideraremos el tiempo de aplicación durante la compilación. Las optimizaciones se pueden realizar prácticamente en cualquier etapa de la compilación. Por ejemplo, la incorporación de constante se puede realizar en una etapa tan temprana como durante el análisis sintáctico (aunque por lo regular es aplazada, de manera que el compilador puede obtener una representación del código fuente exactamente como fue escrito). Por otra parte, algunas optimizaciones se pueden retrasar hasta después que se haya generado el código objetivo; el código objetivo es examinado y vuelto a escribir para reflejar la optimización. Por ejemplo, las optimizaciones de saltos se podrían realizar de esta manera. (En ocasiones las optimizaciones realizadas en el código objetivo se denominan **optimizaciones de mirilla**, ya que el compilador generalmente examina pequeñas secciones de código objetivo para descubrir estas optimizaciones.)

Por lo común, la mayoría de las optimizaciones se realizan durante la generación de código intermedio, justo después de la generación de código intermedio, o bien, durante la generación del código objetivo. En el punto en que una optimización no depende de las características de la máquina objetivo (denominadas **optimizaciones a nivel de fuente**), pueden realizarse antes de las que dependen de la arquitectura objetivo (**optimizaciones a nivel de objetivo**). A veces una optimización puede tener tanto un componente al nivel de fuente como un componente al nivel de objetivo. Por ejemplo, en la asignación de registros

es común contar el número de referencias a cada variable y preferir aquellas variables con conteos más altos de referencias para la asignación a registros. Esta tarea se puede dividir en un componente al nivel de fuente, en el cual las variables son seleccionadas para ser asignadas en registros sin conocimiento específico de cuántos registros están disponibles. Luego, un paso de **asignación de registro** que es dependiente de la máquina objetivo asigna registros reales a aquellas variables etiquetadas como asignadas a registros, o a localidades de memoria denominadas **pseudorregistros**, si no se dispone de registros.

Al ordenar las diversas optimizaciones es importante considerar el efecto que una optimización tiene sobre otra. Por ejemplo, tiene sentido propagar constantes antes de realizar eliminación de código inalcanzable, puesto que algún código se puede volver inalcanzable con base en el hecho de que se hayan encontrado variables de prueba que sean constantes. Ocasionalmente, puede surgir un **problema de fase** en el que cada una de dos optimizaciones puede revelar oportunidades adicionales para la otra. Por ejemplo, considere el código

```
x = 1;
...
y = 0;
...
if (y) x = 0;
...
if (x) y = 1;
```

Un primer paso para propagación de constantes puede producir el código

```
x = 1;
...
y = 0;
...
if (0) x = 0;
...
if (x) y = 1;
```

Ahora, el cuerpo del primer `if` es código inalcanzable; al eliminarlo obtenemos

```
x = 1;
...
y = 0;
...
if (x) y = 1;
```

lo que puede beneficiarse ahora de un paso adicional de propagación de constante y de eliminación de código inalcanzable. Por esta razón algunos compiladores efectúan varias iteraciones de un grupo de optimizaciones, para asegurarse de que se hayan encontrado la mayor parte de las oportunidades para la aplicación de las optimizaciones.

El segundo esquema de clasificación para optimizaciones que consideramos es por el área del programa sobre la que se aplica la optimización. Las categorías para esta clasificación se denominan optimizaciones **local, global** e **interprocedural**. Las optimizaciones locales están definidas como aquellas optimizaciones que se aplican a **segmentos lineales**

de código, es decir, secuencias de código sin saltos dentro o fuera de la secuencia.[21] Una secuencia máxima de código lineal se conoce como **bloque básico**, y por definición las optimizaciones locales son aquellas que se encuentran restringidas a bloques básicos. Las optimizaciones que se extienden más allá de los bloques básicos, pero que están confinadas a un procedimiento individual, se denominan **optimizaciones globales** (aun cuando no sean verdaderamente "globales", puesto que se encuentran confinadas a un procedimiento). Las optimizaciones que se extienden más allá de las limitaciones de los procedimientos hasta todo el programa se conocen como **optimizaciones interprocedurales**.

Las optimizaciones locales son relativamente fáciles de realizar, puesto que la naturaleza lineal del código permite que la información sea propagada de forma simple recorriendo la secuencia de código. Por ejemplo, puede suponerse que una variable que fue cargada en un registro mediante una instrucción anterior en un bloque básico, está todavía en ese registro en un punto posterior en el bloque, siempre que el registro no fuera el objetivo de alguna carga intermedia. Esta conclusión no sería correcta si se pudiera hacer un salto dentro del código intermedio, como se indica en el diagrama siguiente:

Las optimizaciones globales, en contraste, son mucho más difíciles de realizar: por lo general requieren de una técnica denominada **análisis de flujo de datos**, la cual intenta recolectar información a través de los límites de saltos. La optimización interprocedural es aún más difícil, puesto que involucra posiblemente varios mecanismos diferentes de paso de parámetros, la posibilidad de acceso a variables no locales y la necesidad de calcular información simultánea en todos los procedimientos que puedan llamarse entre sí. Una complicación más para la optimización interprocedural es la posibilidad de que se puedan compilar muchos procedimientos de manera separada y sólo se puedan ligar entre sí en un punto posterior. El compilador no puede entonces efectuar ninguna optimización interprocedural sin involucrar en una forma especializada al ligador que lleva a cabo las optimizaciones basado en información que el compilador ha obtenido. Por esa razón muchos compiladores sólo realizan el análisis interprocedural básico, o bien no realizan ninguno.

Una clase especial de optimizaciones globales es la que se aplica a los ciclos o bucles. Como éstos por lo regular se ejecutan un gran número de veces, es importante poner especial atención al código dentro de los ciclos, en particular respecto a la reducción de la complejidad de los cálculos. Las estrategias típicas de optimización de ciclos o bucles se enfocan en la identificación de las variables que se incrementan mediante cantidades fijas con cada iteración (denominadas, por lo tanto, ***variables de inducción***). Éstas incluyen

21. Las llamadas de procedimiento representan una clase especial de salto, de modo que normalmente rompen el código lineal. Sin embargo, como siempre regresan a la instrucción inmediata posterior, a menudo se pueden incluir en código lineal y tratarse más adelante en el proceso de generación de código.

variables de control de ciclos, así como otras variables que dependen de las anteriores (las variables de control de ciclos) de formas fijas. Las variables de inducción seleccionadas se pueden colocar en registros y simplificar su cálculo. Este código vuelto a escribir puede incluir cálculos de eliminación de constante del ciclo (denominado **movimiento de código**). En realidad, el reacomodo del código también puede ser útil para mejorar la eficiencia del código dentro de los bloques básicos.

A lo largo de la historia, una tarea adicional en la optimización de ciclos ha sido el problema de identificar realmente los ciclos o bucles en un programa, de manera que puedan ser optimizados. Este **descubrimiento de ciclos o bucles** era necesario debido a la ausencia del control estructurado y el uso de las sentencias goto para implementar ciclos. Aunque el descubrimiento de ciclos permanece como algo necesario en algunos lenguajes como FORTRAN, en la mayoría de los lenguajes se puede utilizar la sintaxis misma para localizar estructuras de ciclo significativas.

8.9.3 Estructuras de datos y técnicas de implementación para optimizaciones

Algunas optimizaciones se pueden hacer mediante transformaciones en el mismo árbol sintáctico. Éstas pueden incluir tanto incorporación de constantes como eliminación de código inalcanzable, donde los subárboles apropiados son eliminados o reemplazados por otros más simples. La información a utilizar en optimizaciones posteriores también se puede obtener durante la construcción o recorrido del árbol sintáctico, tal como el conteo de referencia u otra información de uso, y conservarla como atributos en el árbol o introducirla en la tabla de símbolos.

Sin embargo, para muchas de las optimizaciones mencionadas anteriormente, el árbol sintáctico es una estructura inadecuada o difícil de manejar para recolectar información y realizar optimizaciones. En su lugar, un optimizador que realiza optimizaciones globales construirá a partir del código intermedio de cada procedimiento una representación gráfica del código denominada **gráfica de flujo**. Los nodos de una gráfica de flujo son los bloques básicos, y los bordes son formados a partir de los saltos condicionales e incondicionales (los que deben tener como sus objetivos los inicios de otros bloques básicos). Cada nodo de bloque básico contiene la secuencia de instrucciones de código intermedio del bloque. Como ejemplo, en la figura 8.18 (página 476) se muestra la gráfica de flujo correspondiente al código intermedio de la figura 8.2 (página 401). (Los bloques básicos que se muestran en esta figura están etiquetados para futuras referencias.)

Una gráfica de flujo, junto con cada uno de sus bloques básicos, puede construirse mediante un simple paso sobre el código intermedio. Cada nuevo bloque básico se identifica de la manera siguiente:[22]

1. La primera instrucción comienza un nuevo bloque básico.
2. Cada etiqueta que es el objetivo de un salto comienza un nuevo bloque básico.
3. Cada instrucción que sigue a un salto comienza un nuevo bloque básico.

Para saltos hacia delante hacia etiquetas que todavía no se alcanzan, se puede construir un nuevo nodo de bloque vacío e insertarlo en la tabla de símbolos bajo el nombre de la etiqueta, para buscarlo una vez que la etiqueta sea alcanzada.

22. Estos criterios permiten que las llamadas de procedimiento se incluyan dentro de bloques básicos. Como las llamadas no contribuyen con nuevas trayectorias a la gráfica de flujo, esto es razonable. Más adelante, cuando los bloques básicos se procesan individualmente, las llamadas se pueden separar para un procesamiento especial si es necesario.

Figura 8.18
La gráfica de flujo del código intermedio de la figura 8.2

La gráfica de flujo es la principal estructura de datos que necesita el análisis de flujo de datos, el cual la utiliza para acumular información que será empleada en las optimizaciones. Diferentes clases de información pueden requerir diferentes clases de procesamiento de la gráfica de flujo, y la información obtenida puede ser muy variada, dependiendo de las clases de optimizaciones deseadas. Aunque no tenemos espacio en esta breve perspectiva general para describir la técnica de análisis de flujo de datos con detalle (véase la sección de notas y referencias al final del capítulo), puede valer la pena describir un ejemplo de la clase de datos que se pueden acumular mediante este proceso.

Un problema estándar de análisis de flujo de datos es calcular, para cada variable, el conjunto de las denominadas **definiciones de alcance** de esa variable al principio de cada bloque básico. Aquí una **definición** es una instrucción de código intermedio que puede establecer el valor de la variable, tal como una asignación o una lectura.[23] Por ejemplo, las definiciones de la variable `fact` en la figura 8.18 son la instrucción simple del bloque básico *B2* (`fact=1`) y la tercera instrucción del bloque *B3* (`fact=t2`). Llamemos a estas definiciones *d1* y *d2*. Se dice que una definición **alcanza** un bloque básico si al principio del bloque la variable todavía puede tener el valor establecido por esta definición. En la gráfica de flujo de la figura 8.18 se puede establecer que ninguna definición de `fact` alcanza *B1* o *B2*, que tanto *d1* como *d2* alcanzan *B3* y que sólo *d2* alcanza *B4* y *B5*. Las definiciones de alcance se pueden utilizar en diversas optimizaciones: en la propagación de constante, por ejemplo, si las únicas definiciones que alcanzan un bloque representan un valor constante

23. Esto no debe confundirse con una definición de C, la cual es una clase de declaración.

simple, entonces la variable se puede reemplazar por este valor (por lo menos hasta que se alcance otra definición dentro del bloque).

La gráfica de flujo es útil para representar la información global acerca del código de cada procedimiento, pero los bloques básicos todavía son representados como secuencias de código simples. Una vez que se realiza el análisis de flujo de datos, y que se va a generar el código para cada bloque básico, con frecuencia se construye otra estructura de datos para cada bloque, llamada **DAG de un bloque básico** (DAG = Directed Acyclic Graph). (Se puede construir una DAG para cada bloque básico incluso sin construir la gráfica de flujo.)

Una estructura de datos DAG describe el cálculo y reasignación de valores y variables en un bloque básico de la manera siguiente. Los valores que son utilizados en el bloque que vienen de cualquier parte se representan como nodos de hoja. Las operaciones sobre aquéllos y otros valores se representan mediante nodos interiores. La asignación de un valor nuevo se representa adjuntando el nombre de la variable o elemento temporal objetivo, al nodo que representa el valor asignado. (Un caso especial de esta construcción se describió en la página 416.)[24]

Por ejemplo, el bloque básico *B3* de la figura 8.18 se puede representar mediante la DAG de la figura 8.19. (Las etiquetas al principio de los bloques básicos y saltos al final por lo regular no se incluyen en la DAG.) Observe en esta DAG que las operaciones de copia tales como `fact=t2` y `x=t3` no crean nuevos nodos, sino simplemente agregan nuevas etiquetas a los nodos con las etiquetas `t2` y `t3`. Advierta también que el nodo de hoja etiquetado **x** tiene dos padres, lo que resulta del hecho que el valor de entrada de **x** es utilizado en dos instrucciones por separado. De este modo, el uso repetido del mismo valor también es representado en la estructura DAG. Esta propiedad de una DAG permite representar el uso repetido de las subexpresiones comunes. Por ejemplo, la asignación en C

```
x = (x+1)*(x+1)
```

se traduce en las instrucciones de tres direcciones

```
t1 = x + 1
t2 = x + 1
t3 = t1 * t2
x = t3
```

y la DAG para esta secuencia de instrucciones está dada en la figura 8.20, que muestra el uso repetido de la expresión **x+1**.

Figura 8.19
La DAG del bloque básico B3 de la figura 8.18

24. Esta descripción de la estructura DAG está adaptada respecto al uso del código de tres direcciones como código intermedio, pero una estructura DAG semejante se puede definir para el código P, o incluso para el código ensamblado objetivo.

Figura 8.20
La DAG de las instrucciones de tres direcciones correspondientes a la asignación en C
`x = (x+1)*(x+1)`

La DAG de un bloque básico se puede construir manteniendo dos diccionarios. El primero es una tabla que contiene nombres de variable y constantes, con una operación de búsqueda que devuelve el nodo DAG para el cual se encuentra actualmente asignado un nombre de variable (la tabla de símbolos se podría utilizar como esta tabla). El segundo es una tabla de nodos DAG, con una operación de búsqueda que, dada una operación y nodos hijos, devuelve el nodo con esa operación y los hijos o el elemento nulo si no hay tal nodo. Esta operación permite la búsqueda de valores existentes sin construir un nuevo nodo en la DAG. Por ejemplo, una vez que se construye el nodo + de la figura 8.20 con hijos `x` y `1`, y se le asigna el nombre `t1` (como resultado de procesar la instrucción de tres direcciones `t1=x+1`), una búsqueda de (`+`, `x`, `1`) en la segunda tabla devolverá este nodo ya construido, y la instrucción de tres direcciones `t2=x+1` simplemente provocará que `t2` también sea asignado a este nodo. Los detalles de esta construcción se pueden encontrar en otra parte (véase la sección de notas y referencias).

El código objetivo, o una versión revisada de código intermedio, se puede generar a partir de una DAG mediante un recorrido según cualquiera de las posibles clases topológicas de los nodos no hoja. (Una **clase topológica** de una DAG es un recorrido tal que los hijos de nodos son visitados antes que todos los padres.) Puesto que, en general, existen muchas clases topológicas, es posible generar muchas secuencias de código diferentes a partir de una DAG. Cuáles pueden ser mejores que otras depende de varios factores, que incluyen detalles de la arquitectura de la máquina objetivo. Por ejemplo, una de las secuencias de recorrido legales de los tres nodos no hoja de la figura 8.19 daría como resultado la siguiente nueva secuencia de las instrucciones de tres direcciones, que podrían reemplazar el bloque básico original:

```
t3 = x - 1
t2 = fact * x
x = t3
t4 = x == 0
fact = t2
```

Naturalmente, es probable que deseemos evitar el uso innecesario de elementos temporales, y, en consecuencia, generar el siguiente código equivalente de tres direcciones, cuyo orden debe permanecer fijo:

```
fact = fact * x
x = x - 1
t4 = x == 0
```

Un recorrido semejante de la DAG de la figura 8.20 produce el siguiente código de tres direcciones revisado:

```
t1 = x + 1
x = t1 * t1
```

Al utilizar una DAG para generar código objetivo para un bloque básico, automáticamente obtenemos la eliminación de subexpresión común local. La representación DAG también permite eliminar almacenamientos (asignaciones) redundantes e indica cuántas referencias existen para cada valor (el número de padres de un nodo es el número de referencias). Esto nos da información que permite una buena asignación de registros (por ejemplo, si un valor tiene muchas referencias, déjelo en un registro; si se han visto todas las referencias, el valor está muerto y ya no es necesario mantenerlo; y así sucesivamente).

Un método final que se utiliza con frecuencia para ayudar a la asignación de registros a medida que continúa la generación de código involucra el mantenimiento de datos denominados **descriptores de registro** y **descriptores de dirección**. Los descriptores de registro asocian con cada registro una lista de los nombres de variable cuyo valor está actualmente en ese registro (claro que todos ellos deben tener el mismo valor en ese punto). Los descriptores de dirección asocian con cada nombre de variable las localidades en la memoria donde se va a encontrar su valor. Éstas podrían ser registros (en cuyo caso, la variable se va a encontrar en el descriptor o descriptores de registro correspondientes) o la memoria o ambos (si la variable acaba de ser cargada desde la memoria en un registro, pero su valor no ha cambiado). Tales descriptores permiten el seguimiento del movimiento de los valores entre la memoria y los registros y tienen en cuenta la reutilización de los valores ya cargados en los registros, además de la reclamación de los mismos, ya sea al descubrir que ya no contienen el valor de ninguna variable con usos posteriores o al vaciar los valores a las localidades de memoria apropiadas (operaciones de derramamiento).

Tome, por ejemplo, la DAG de bloque básico de la figura 8.19, y considere la generación del código de TM de acuerdo con un recorrido de izquierda a derecha de los nodos interiores, utilizando los tres registros 0, 1 y 2. Suponga también que existen cuatro descriptores de dirección: `inReg(reg_no)`, `isGlobal(global_offset)`, `isTemp(temp_offset)` e `isConst(value)` (esto corresponde a la organización del ambiente de ejecución de TINY sobre la máquina TM comentada en la sección anterior). Suponga, además, que `x` está en la localidad global 0, que `fact` está en la localidad global 1 y que se tiene acceso a las localidades globales mediante el registro gp, y que se tiene acceso a las localidades temporales mediante el registro mp. Finalmente, suponga también que ninguno de los registros comienza con valor alguno en ellos. Entonces, antes que comience la generación de código para los bloques básicos, los descriptores de dirección para las variables y constantes estarían de la manera siguiente:

Variable/Constante	Descriptores de dirección
fact	isGlobal(1)
x	isGlobal(0)
t2	-
t3	-
t4	-
1	isConst(1)
0	isConst(0)

La tabla del descriptor de registro estaría vacía, y no la imprimimos.

Ahora suponga que se genera el código siguiente:

```
LD 0,1(gp) load fact into reg 0
LD 1,0(gp) load x into reg 1
MUL 0,0,1
```

Los descriptores de dirección ahora serían:

Variable/Constante	Descriptores de dirección
fact	inReg(0)
x	isGlobal(0),inReg(1)
t2	inReg(0)
t3	-
t4	-
1	isConst(1)
0	isConst(0)

y los descriptores de registro serían

Registro	Variables/Constantes
0	fact, t2
1	x
2	-

Ahora, dado el código subsiguiente

```
LDC 2,1(0) load constant 1 into reg 2
ADD 1,1,2
```

los descriptores de dirección se convertirían en:

Variable/Constante	Descriptores de dirección
fact	inReg(0)
x	inReg(1)
t2	inReg(0)
t3	inReg(1)
t4	-
1	isConst(1),inReg(2)
0	isConst(0)

y los descriptores de registro se convertirían en:

Registro	Variables/Constantes
0	fact, t2
1	x, t3
2	1

Dejamos al lector proporcionar el código adecuado para el cálculo del valor en el nodo restante DAG y describir los descriptores de registro y dirección resultantes.

Esto concluye nuestro breve recorrido a través de las técnicas de optimización de código.

8.10 OPTIMIZACIONES SIMPLES PARA EL GENERADOR DE CÓDIGO DE TINY

El generador de código para el lenguaje TINY, como se proporcionó en la sección 8.8, produce un código muy ineficiente, como se demostró mediante una comparación de las 42 instrucciones de la figura 8.17 (página 466) respecto a las nueve instrucciones codificadas a mano del programa equivalente de la figura 8.16 (página 458). En primer lugar, las ineficiencias se deben a dos fuentes:

1. El generador de código TINY hace un uso muy pobre de los registros de la máquina TM (de hecho, nunca utiliza los registros 2, 3 o 4).
2. El generador de código TINY genera innecesariamente los valores lógicos 0 y 1 para las pruebas, cuando éstas sólo aparecen en sentencias if y sentencias while, donde lo haría un código más simple.

Deseamos señalar en esta sección cómo incluso las técnicas relativamente rudimentarias pueden mejorar de manera sustancial el código generado por el compilador de TINY. En realidad, no genera ni bloques básicos ni gráficas de flujo, pero continúa generando código directamente del árbol sintáctico. La única maquinaria necesaria es un segmento adicional de datos de atributo y algún código de compilador un poco más complejo. No damos detalles completos de implementación para las mejoras descritas aquí, pero los dejamos como ejercicios para el lector.

8.10.1 Conservación de elementos temporales en registros

La primera optimización que deseamos describir es un método fácil para mantener elementos temporales en registros en lugar de almacenarlos y volverlos a cargar constantemente desde la memoria. En el generador de código TINY los elementos temporales siempre eran almacenados en la localidad

```
tmpOffset(mp)
```

donde `tmpOffset` es una variable estática inicializada a 0, que se decrementa cada vez que se almacena un elemento temporal, y se incrementa cada vez que se vuelve a cargar (véase el apéndice B, líneas 2023 y 2027). Una manera simple de utilizar los registros como localidades temporales es interpretar `tmpOffset` haciendo referencia inicialmente a los registros, y sólo después que se han agotado los registros disponibles, utilizarlo como un desplazamiento real en la memoria. Por ejemplo, si queremos emplear todos los registros

disponibles como elementos temporales (después del pc, gp y mp), entonces los valores desde 0 hasta −4 de `tmpOffset` se pueden interpretar como referencias a los registros 0 hasta el 4, mientras que los valores que comienzan con −5 se emplean como desplazamientos (con el 5 agregado a sus valores). Este mecanismo se puede implementar directamente en el generador de código mediante pruebas apropiadas o encapsulado en procedimientos auxiliares (los cuales podrían denominarse `saveTmp` y `loadTmp`). También debe prevenirse el hecho de que el resultado de un cálculo de subexpresión se pueda encontrar en otro registro aparte del 0 después de la generación de código recursivo.

Con esta mejora el generador de código TINY ahora genera la secuencia de código TM dada en la figura 8.21 (compare con la figura 8.17). Este código ahora es 20% más corto y no contiene almacenamiento de elementos temporales (y el mp no utiliza el registro 5). A pesar de todo, los registros 2, 3 y 4 nunca se utilizan. Esto no es inusual: las expresiones en los programas rara vez son lo suficientemente complejas para requerir más de dos o tres elementos temporales a la vez.

8.10.2 Conservación de variables en registros

Se puede hacer una mejora adicional en el uso de los registros TM reservando algunos de los registros para su uso como localidades de variables. Esto requiere un poco más de trabajo que la optimización anterior, ya que la ubicación de una variable debe ser determinada antes de la generación de código y almacenada en la tabla de símbolos. Un esquema básico consiste simplemente en seleccionar unos cuantos registros y asignar éstos como las localidades para las variables más utilizadas del programa. Para determinar cuáles variables son las "más utilizadas", debe hacerse un conteo de referencias (usos y asignaciones). Las variables que son referencias durante los ciclos (ya sea en el cuerpo del ciclo o la expresión de prueba) deberían tener preferencia, puesto que las referencias tienen la probabilidad de ser repetidas a medida que se ejecuta el ciclo o bucle. Un enfoque simple que ha demostrado funcionar bien en muchos compiladores existentes es multiplicar por 10 el conteo de todas las referencias dentro de un ciclo, por 100 dentro de un ciclo doblemente anidado, y así sucesivamente. Este conteo de referencias se puede hacer durante el análisis semántico, y una vez que se ha hecho, se realiza un paso de asignación de variable por separado. El atributo de localidad que se almacena en la tabla de símbolos debe ahora tener en cuenta aquellas variables que están asignadas a registros en contraposición a la memoria. Un esquema simple utiliza un tipo enumerado para indicar dónde está localizada una variable; en este caso existen sólo dos posibilidades: `inReg` e `inMem`. Adicionalmente, se debe conservar el número de registro en el primero de los casos y la dirección de la memoria en el segundo. (Éste es un ejemplo simple de descriptores de dirección para variables; los descriptores de registro no son necesarios porque permanecen fijos durante el transcurso de la generación del código.)

Con estas modificaciones el código del programa de muestra puede ahora utilizar el registro 3 para variable `x` y el registro 4 para la variable `fact` (hay sólo dos variables, de modo que todos pueden caber en los registros), suponiendo que los registros del 0 al 2 todavía están reservados para elementos temporales. Las modificaciones al código de la figura 8.21 están dadas en la figura 8.22. Este código es de nueva cuenta considerablemente menor que el código anterior, pero todavía mucho más largo que el código escrito a mano.

8.10.3 Optimización de expresiones de prueba

La optimización final que comentamos aquí es simplificar el código generado para pruebas en sentencias if y sentencias while. El código generado para estas expresiones es muy general, implementa los valores booleanos verdadero y falso como 0 y 1, aun cuando TINY no

Figura 8.21
Código TM para el programa TINY de muestra con elementos temporales conservados en registros

```
 0:    LD   6,0(0)      17:    ST   0,1(5)
 1:    ST   0,0(0)      18:    LD   0,0(5)
 2:    IN   0,0,0       19:   LDC   1,1(0)
 3:    ST   0,0(5)      20:   SUB   0,0,1
 4:   LDC   0,0(0)      21:    ST   0,0(5)
 5:    LD   1,0(5)      22:    LD   0,0(5)
 6:   SUB   0,0,1       23:   LDC   1,0(0)
 7:   JLT   0,2(7)      24:   SUB   0,0,1
 8:   LDC   0,0(0)      25:   JEQ   0,2(7)
 9:   LDA   7,1(7)      26:   LDC   0,0(0)
10:   LDC   0,1(0)      27:   LDA   7,1(7)
11:   JEQ   0,21(7)     28:   LDC   0,1(0)
12:   LDC   0,1(0)      29:   JEQ   0,-16(7)
13:    ST   0,1(5)      30:    LD   0,1(5)
14:    LD   0,1(5)      31:   OUT   0,0,0
15:    LD   1,0(5)      32:   LDA   7,0(7)
16:   MUL   0,0,1       33:  HALT   0,0,0
```

Figura 8.22
Código TM para el programa TINY de muestra con elementos temporales y variables conservados en registros

```
 0:    LD   6,0(0)      13:   LDC   0,1(0)
 1:    ST   0,0(0)      14:   SUB   0,3,0
 2:    IN   3,0,0       15:   LDA   3,0(0)
 3:   LDC   0,0(0)      16:   LDC   0,0(0)
 4:   SUB   0,0,3       17:   SUB   0,3,0
 5:   JLT   0,2(7)      18:   JEQ   0,2(7)
 6:   LDC   0,0(0)      19:   LDC   0,0(0)
 7:   LDA   7,1(7)      20:   LDA   7,1(7)
 8:   LDC   0,1(0)      21:   LDC   0,1(0)
 9:   JEQ   0,15(7)     22:   JEQ   0,-12(7)
10:   LDC   4,1(0)      23:   OUT   4,0,0
11:   MUL   0,4,3       24:   LDA   7,0(7)
12:   LDA   4,0(0)      25:  HALT   0,0,0
```

tiene variables booleanas y no necesita este nivel de generalidad. Esto también produce muchas cargas adicionales de las constantes 0 y 1, además de pruebas adicionales que se generan por separado mediante el código **genStmt** para las sentencias de control.

La mejora que describimos aquí depende del hecho que un operador de comparación debe aparecer como el nodo raíz de la expresión de prueba. El código de **genExp** para este operador simplemente generará código para sustraer el valor del operando de la derecha del operando de la izquierda, dejando el resultado en el registro 0. El código para la sentencia if o la sentencia while probará entonces para cuál operador de comparación es aplicado y generará el código apropiado del salto condicional.

De este modo, en el caso del código de la figura 8.22, el código TINY

```
if 0 < x then ...
```

que ahora corresponde al código TM

```
4:     SUB   0,0,3
5:     JLT   0,2(7)
6:     LDC   0,0(0)
7:     LDA   7,1(7)
8:     LDC   0,1(0)
9:     JEQ   0,15(7)
```

generará en su lugar el código TM más simple

```
4:     SUB   0,0,3
5:     JGE   0,10(7)
```

(Advierta cómo el salto del caso falso debe ser el condicional complementario **JGE** para el operador de prueba <.)

Con esta optimización el código generado para el programa de prueba se convierte en el que se proporciona en la figura 8.23. (También incluimos en este paso la eliminación del salto vacío al final del código, que corresponde a la parte else vacía de la sentencia if en el código TINY. Esto requiere únicamente la adición de una prueba simple al código **genStmt** para la sentencia if.)

El código de la figura 8.23 está ahora relativamente cercano al del código generado a mano de la página 458. Aun así, todavía hay oportunidad para unas cuantas optimizaciones más de caso especial, las cuales dejaremos para los ejercicios.

Figura 8.23
Código TM para el programa TINY de muestra con elementos temporales y variables conservados en registros, y con pruebas de saltos optimizadas

```
0:     LD    6,0(0)        9:     LDC   0,1(0)
1:     ST    0,0(0)       10:     SUB   0,3,0
2:     IN    3,0,0        11:     LDA   3,0(0)
3:     LDC   0,0(0)       12:     LDC   0,0(0)
4:     SUB   0,0,3        13:     SUB   0,3,0
5:     JGE   0,10(7)      14:     JNE   0,-8(7)
6:     LDC   4,1(0)       15:     OUT   4,0,0
7:     MUL   0,4,3        16:    HALT   0,0,0
8:     LDA   4,0(0)
```

EJERCICIOS

8.1 Proporcione la secuencia de instrucciones de código de tres direcciones correspondiente a cada una de las siguientes expresiones aritméticas:

a. 2+3+4+5

b. 2+(3+(4+5))

c. a*b+a*b*c

8.2 Proporcione la secuencia de instrucciones de código P correspondiente a cada una de las instrucciones aritméticas del ejercicio anterior.

8.3 Proporcione las instrucciones de código P correspondientes a las siguientes expresiones en C:

a. `(x = y = 2)+3*(x=4)`

b. `a[a[i]]=b[i=2]`

c. `p->next->next = p->next`

(Suponga una declaración `struct` apropiada.)

8.4 Proporcione instrucciones en código de tres direcciones para las expresiones del ejercicio anterior.

8.5 Proporcione la secuencia de **a**) código de tres direcciones o **b**) código P correspondiente al siguiente programa de TINY:

```
{ Programa gcd en lenguaje TINY }
read u;
read v; { entrada de dos enteros }
if v = 0 then v := 0 { no hacer nada }
else
  repeat
    temp := v;
    v := u - u/v*v; { calcula u mod v }
    u := temp
  until v = 0
end;
write u { salida gcd de la u y v originales }
```

8.6 Proporcione las declaraciones de estructura de datos en C apropiadas para los triples de la figura 8.5 (sección 8.1.2), similares a las dadas para cuádruples en la figura 8.4.

8.7 Extienda la gramática con atributos para el código P de la tabla 8.1 (sección 8.2.1, página 408) a **a**) la gramática de subíndice de la sección 8.3.2, página 422; **b**) la gramática de estructura de control de la sección 8.4.4, página 433.

8.8 Repita el ejercicio anterior para la gramática con atributos del código de tres direcciones de la tabla 8.2, página 409.

8.9 Describa cómo se puede adaptar el procedimiento de recorrido genérico de la sección 6.5.2 (página 337) para generación de código. ¿Valdría esto la pena?

8.10 Agregue los operadores de dirección unitarios `&` y `*` (con semántica de C), así como el operador de selección de campo de estructura binaria `.` para

a. la gramática de expresión de la sección 8.2.1 (página 407);

b. la estructura de árbol sintáctico de la sección 8.2.2 (página 410).

8.11 **a.** Agregue una sentencia repeat-until o do-while a la gramática de control de la sección 8.4.4 (página 433), y dibuje un diagrama de control adecuado correspondiente al de la figura 8.11 (página 431).

b. Vuelva a escribir las declaraciones de la estructura de árbol sintáctico para la gramática (página 434) con el fin de incluir su nueva estructura del inciso a.

8.12 **a.** Describa cómo una sentencia for se puede convertir sistemáticamente en una sentencia while correspondiente. ¿Tiene sentido utilizar esto para generar código?

b. Describa cómo una sentencia case o switch se puede convertir sistemáticamente en una secuencia de sentencias anidadas if correspondientes. ¿Tiene sentido utilizar esto para generar código?

8.13 **a.** Dibuje un diagrama de control correspondiente a la figura 8.11, página 431, para la organización de ciclo o bucle exhibida por el compilador Borland de C para la arquitectura 80×86 de la sección 8.6.1.

b. Dibuje un diagrama de control correspondiente a la figura 8.11 para la organización de ciclo o bucle exhibida por el compilador SparcStation de Sun de C de la sección 8.6.2.

c. Suponga que una instrucción de salto condicional toma tres veces el tiempo para ejecutarse si el salto está seguido en lugar de que el código "caiga en la nada" (es decir, la condición es falsa). ¿Tienen las organizaciones de salto de los incisos a y b alguna ventaja en el tiempo sobre la de la figura 8.11?

8.14 Una alternativa para implementar sentencias case o switch como una secuencia de pruebas para cada caso es la denominada **tabla de salto**, donde el índice de caso se utiliza como un desplazamiento para un salto indizado en una tabla de saltos absolutos.

a. Este método de implementación es ventajoso sólo si existen muchos casos diferentes que se presentan de manera muy densa sobre un intervalo bastante compacto de índices. ¿Por qué?

b. Los generadores de código tienden a generar esta clase de código sólo cuando hay más de aproximadamente 10 casos. Determine si su compilador de C genera una tabla de salto para una sentencia switch y si hay un número mínimo de casos para hacerlo así.

8.15 a. Desarrolle una fórmula para el cálculo de dirección de un elemento de arreglo multidimensional semejante al de la sección 8.3.2 (página 419). Establezca cualquier suposición que haga.

b. Suponga que una variable de arreglo **a** está definida mediante el código en C

```
int a[12][100][5];
```

Suponiendo que un entero ocupa dos bytes en memoria, use su fórmula del inciso a) para determinar el desplazamiento desde la dirección base de **a** de la variable subindizada

```
a[5][42][2]
```

8.16 Dado que el programa siguiente escrito de acuerdo con la gramática de definición/llamada de función en la página 440:

```
fn f(x)=x+1
fn g(x,y)=x+y
g(f(3),4+5)
```

a. Describa la secuencia de instrucciones en código P que serían generadas por este programa mediante el procedimiento **genCode** de la figura 8.14, página 441.

b. Escriba una secuencia de instrucciones de tres direcciones que sería generada por este programa.

8.17 El texto no especificó cuándo se emitía la instrucción de tres direcciones **arg** durante las llamadas de función: algunas versiones de código de tres direcciones requieren que todas las sentencias **arg** se reúnan inmediatamente antes de la llamada asociada, mientras que otras permiten que se entremezclen el cálculo de los argumentos y las sentencias **arg**. (Véase la página 438). Discuta los pros y los contras de estos dos métodos.

8.18 a. Enumere todas las instrucciones en código P utilizadas en este capítulo, junto con una descripción de su significado y uso.

b. Enumere todas las instrucciones en código de tres direcciones utilizadas en este capítulo, junto con una descripción de su significado y uso.

8.19 Escriba un programa TM equivalente al programa gcd de TINY del ejercicio 8.5.

8.20 a. La TM no tiene instrucción de movimiento de registro a registro. Describa cómo se efectúa esto.

b. La TM no tiene instrucción de llamada o retorno. Describa cómo se pueden imitar.

8.21 Diseñe un coprocesador de punto flotante para la máquina TINY que se pueda agregar sin modificar ninguna de las declaraciones de memoria o registro existentes del apéndice C.

8.22 Escriba la secuencia de instrucciones TM generadas por el compilador de TINY para las siguientes expresiones y asignaciones de TINY:

a. `2+3+4+5`

b. `2+(3+(4+5))`

c. `x:= x+(y+2*z)`, suponiendo que `x`, `y` y `z` tengan localidades `dMem` 0, 1 y 2, respectivamente.

d. `v := u - u/v*v;`

(Una línea del programa gcd de TINY del ejercicio 8.5; suponga el ambiente de ejecución TINY estándar.)

8.23 Diseñe un ambiente de ejecución TM para la gramática de llamada de función de la sección 8.5.2.

8.24 El compilador Borland 3.0 genera el código 80×86 siguiente para calcular el resultado lógico de una comparación `x<y`, suponiendo que `x` y `y` son enteros a desplazamientos −2 y −4 en el registro de activación local:

```
        mov     ax,word ptr [bp-2]
        cmp     ax,word ptr [bp-4]
        jge     short @1@86
        mov     ax,1
        jmp     short @1@114
@1@86:
        xor     ax,ax
@1@114:
```

Compare esto con el código TM producido por el compilador TINY para la misma expresión.

8.25 Examine cómo su compilador de C implementa operaciones booleanas de cortocircuito, y compare la implementación con las estructuras de control de la sección 8.4.

8.26 Dibuje una gráfica de flujo para el código de tres direcciones correspondiente al programa gcd de TINY del ejercicio 8.5.

8.27 Dibuje una DAG para el bloque básico correspondiente al cuerpo de la sentencia repeat del programa gcd de TINY del ejercicio 8.5.

8.28 Considere la DAG de la figura 8.19, página 477. Si se supone que el operador de igualdad del nodo de extrema derecha es imitado en la máquina TM mediante sustracción, el código TM correspondiente a este nodo puede ser de la manera siguiente:

```
LDC 2,0(0) load constant 0 into reg 2
SUB 2,1,2
```

Escriba descriptores de registro y de dirección (basados en los de la página 480) como podrían llegar a ser después de la ejecución del código anterior.

8.29 Determine las optimizaciones que realiza su compilador de C, y compárelas con las descritas en la sección 8.9.

8.30 Dos optimizaciones adicionales que pueden ser implementadas en el generador de código TINY son las siguientes:

1. Si uno de los operandos de una expresión de prueba es la constante 0, entonces no necesita realizarse ninguna sustracción antes de generar el salto condicional.
2. Si el objetivo de una asignación está ya en un registro, entonces la expresión del lado derecho puede calcularse en este registro, ahorrando así un movimiento de registro a registro.

Muestre el código para el programa factor y al TINY de muestra que sería generado si estas dos optimizaciones se agregaran al generador de código que produce el código de la figura 8.23. ¿Cómo se compara este código con el código generado a mano de la figura 8.16 (página 458)?

EJERCICIOS DE PROGRAMACIÓN

8.31 Vuelva a escribir el código de la figura 8.7 (sección 8.2.2, página 412) para producir código P como un atributo de cadena sintetizado de acuerdo con la gramática con atributos de la tabla 8.1 (página 408), y compare la complejidad del código con el de la figura 8.7.

8.32 Vuelva a escribir cada uno de los siguientes procedimientos de generación de código P para producir en su lugar código de tres direcciones:

a. Figura 8.7, página 412 (expresiones simples tipo C).

b. Figura 8.9, página 424 (expresiones con arreglos).

c. Figura 8.12, página 435 (sentencias de control).

d. Figura 8.14, página 441 (funciones).

8.33 Escriba una especificación Yacc similar a la de la figura 8.8, página 413, correspondiente a los procedimientos de generación de código de la

a. Figura 8.9, página 424 (expresiones con arreglos).

b. Figura 8.12, página 435 (sentencias de control).

c. Figura 8.14, página 441 (funciones).

8.34 Vuelva a escribir la especificación Yacc de la figura 8.8 para producir código de tres direcciones en lugar de código P.

8.35 Agregue los operadores del ejercicio 8.10 al procedimiento de generación de código de la figura 8.7, página 412.

8.36 Vuelva a escribir el procedimiento de generación de código de la figura 8.12, página 435, para incluir las nuevas estructuras de control del ejercicio 8.11.

8.37 Vuelva a escribir el código de la figura 8.7, página 412, para producir código TM en vez de código P. (Usted puede tomar las utilidades de generación de código del archivo `code.h` del compilador TINY.)

8.38 Vuelva a escribir el código de la figura 8.14 (página 441) para generar código TM, utilizando su diseño de ambiente de ejecución del ejercicio 8.23.

8.39 **a.** Agregue arreglos simples al compilador y lenguaje TINY. Esto requiere que las declaraciones de arreglo se agreguen antes de las sentencias mismas, como en

```
array a[10];
i := 1;
repeat
  read a[i];
  i := i + 1;
until 10 < i
```

b. Agregue verificaciones de límites a su código del inciso a, de manera que los subíndices fuera de límites provoquen un alto en la máquina TM.

8.40 **a.** Implemente su diseño del coprocesador de punto flotante TM del ejercicio 8.21.

b. Utilice su capacidad TM de punto flotante para reemplazar los enteros por números reales en el compilador y lenguaje TINY.

c. Vuelva a escribir el compilador y lenguaje TINY para incluir tanto valores enteros como valores de punto flotante.

8.41 Escriba un traductor de código P a código de tres direcciones.

8.42 Escriba un traductor de código de tres direcciones a código P.

8.43 Vuelva a escribir el generador de código TINY para generar código P.

8.44 Escriba un traductor de código P a código de máquina TM, tomando el generador de código P del ejercicio anterior y el ambiente de ejecución TINY descrito en el texto.

8.45 Vuelva a escribir el generador de código TINY para generar código de tres direcciones.

8.46 Escriba un traductor del código de tres direcciones a código de máquina TM, tomando el generador de código de tres direcciones del ejercicio anterior y el ambiente de ejecución TINY descrito en el texto.

8.47 Implemente las tres optimizaciones del generador de código TINY descritas en la sección 8.10:

a. Utilice los primeros tres registros TM como localidades temporales.

b. Emplee los registros 3 y 4 como localidades para las variables más utilizadas.

c. Optimice el código para las expresiones de prueba de manera que ya no generen los valores booleanos 0 y 1.

8.48 Implemente la incorporación de constante en el compilador TINY.

8.49 **a.** Implemente la optimización 1 del ejercicio 8.30.

b. Implemente la optimización 2 del ejercicio 8.30.

NOTAS Y REFERENCIAS

Existe una variedad enorme de técnicas de generación de código y optimización; este capítulo representa sólo una introducción. Una buena visión general de tales técnicas (en particular el análisis de flujo de datos), desde un punto de vista algo teórico, está contenida en Aho, Sethi y Ullman [1986]. Varios temas prácticos se tratan con más detalle en Fischer y LeBlanc [1991]. En ambos se describen las tablas de salto para sentencias case/switch (ejercicio 8.14). Para ejemplos de generación de código en procesadores particulares (MIPS, Sparc y PC) véase Fraser y Hanson [1995]. La generación de código como análisis de atributo se trata en Slonneger y Kurtz [1995].

La variabilidad del código intermedio de compilador a compilador ha sido una fuente de problemas de transportabilidad desde que los primeros compiladores fueron escritos. Originalmente, se pensaba que se podría desarrollar un código intermedio universal que serviría para todos los compiladores y resolvería el problema de la transportabilidad (Strong [1958], Steel [1961]). Desgraciadamente, éste no ha probado ser el caso. El código de tres direcciones o cuádruples es una forma tradicional de código intermedio y se utiliza en muchos textos de compiladores. El código P se describe con detalle en Nori *et al.* [1981]. Una forma un poco más sofisticada del código P, llamada código U, que permite una mejor optimización del código objetivo, se describe en Perkins y Sites [1977]. Una versión similar de código P fue utilizada en un compilador de optimización de Modula-2 (Powell [1984]). Un código intermedio especializado para compiladores Ada, llamado Diana, se describe en Goos y Wulf [1981]. Un código intermedio que utiliza expresiones de prefijo tipo LISP se conoce como lenguaje de transferencia de registro, o RTL, por sus siglas en inglés, y se utiliza en los compiladores Gnu (Stallman [1994]); se describe en Davidson y Fraser [1984a,b]. Para un ejemplo de un código intermedio basado en C que se puede compilar utilizando un compilador de C, véase Holub [1990].

No hay una referencia detallada actualizada para técnicas de optimización, aunque las referencias estándar de Aho, Sethi y Ullman [1986] y de Fischer y LeBlanc [1991] contienen buenos resúmenes. Muchas técnicas poderosas y útiles han sido publicadas en el *ACM Programming Languages Design and Implementation Conference Proceedings* (anteriormente conocida como la Compiler Construction Conference), que aparecen como parte de las *ACM SIGPLAN Notices*. Fuentes adicionales para muchas técnicas de optimización son los *ACM Principles of Programming Languages Conference Proceedings* y las *ACM Transactions on Programming Languages and Systems.*

Un aspecto de la generación de código que no hemos mencionado es la generación automática de un generador de código utilizando una descripción formal de la arquitectura de la máquina, de una manera semejante a la forma en que los analizadores sintácticos y analizadores semánticos se pueden generar de manera automática. Tales métodos varían desde los puramente sintácticos (Glanville y Graham [1978]) hasta los basados en atributos (Ganapathi y Fischer [1985]) y los basados en el código intermedio (Davidson y Fraser [1984a]). Una visión general de éstos y otros métodos se puede encontrar en Fischer y LeBlanc [1991].

Apéndice A

Proyecto de compilador

A.1 Convenciones léxicas de C−
A.2 Sintaxis y semántica de C−
A.3 Programas de muestra en C−
A.4 Un ambiente de ejecución de la Máquina TINY para el lenguaje C−
A.5 Proyectos de programación utilizando C− y TM

Definimos aquí un lenguaje de programación denominado **C− Minus** o C−, para abreviar, el cual es un lenguaje adecuado para un proyecto de compilador que es más complejo que el lenguaje TINY debido a que incluye funciones y arreglos. En esencia es un subconjunto del lenguaje C, pero sin algunas partes importantes de éste, de donde se deduce su nombre. Este apéndice consta de cinco secciones. En la primera enumeramos las convenciones léxicas del lenguaje, incluyendo una descripción de los tokens del lenguaje. En la segunda proporcionamos una descripción BNF de cada construcción del lenguaje, junto con una descripción en idioma inglés de la semántica asociada. En la tercera sección ofrecemos dos programas de muestra en C−. En la cuarta describimos un ambiente de ejecución de la máquina TINY para C−. La última sección describe varios proyectos de programación utilizando C− y TM apropiados para un curso de compiladores.

A.1 CONVENCIONES LÉXICAS DE C−

1. Las palabras clave o reservadas del lenguaje son las siguientes:

```
else if int return void while
```

   Todas las palabras reservadas o clave están reservadas, y deben ser escritas en minúsculas.

2. Los símbolos especiales son los siguientes:

```
+ - * / < <= > >= == != = ; , ( ) [ ] { } /* */
```

3. Otros tokens son *ID* y *NUM*, definidos mediante las siguientes expresiones regulares:

```
ID = letra letra*
NUM = dígito dígito*
letra = a|..|z|A|..|Z
dígito = 0|..|9
```

   Se distingue entre letras minúsculas y mayúsculas.
4. Los espacios en blanco se componen de blancos, retornos de línea y tabulaciones. El espacio en blanco es ignorado, excepto cuando deba separar *ID*, *NUM* y palabras reservadas.
5. Los comentarios están encerrados entre las anotaciones habituales del lenguaje C `/*...*/`. Los comentarios se pueden colocar en cualquier lugar donde pueda aparecer un espacio en blanco (es decir, los comentarios no pueden ser colocados dentro de los token) y pueden incluir más de una línea. Los comentarios no pueden estar anidados.

A.2 SINTAXIS Y SEMÁNTICA DE C−

Una gramática BNF para C− es como se describe a continuación:

1. *program* → *declaration-list*
2. *declaration-list* → *declaration-list declaration* | *declaration*
3. *declaration* → *var-declaration* | *fun-declaration*
4. *var-declaration* → *type-specifier* **ID** ; | *type-specifier* **ID** [**NUM**] ;
5. *type-specifier* → **int** | **void**
6. *fun-declaration* → *type-specifier* **ID** (*params*) *compound-stmt*
7. *params* → *param-list* | **void**
8. *param-list* → *param-list* , *param* | *param*
9. *param* → *type-specifier* **ID** | *type-specifier* **ID** [ ]
10. *compound-stmt* → { *local-declarations statement-list* }
11. *local-declarations* → *local-declarations var-declaration* | *empty*
12. *statement-list* → *statement-list statement* | *empty*
13. *statement* → *expression-stmt* | *compound-stmt* | *selection-stmt*
    | *iteration-stmt* | *return-stmt*
14. *expression-stmt* → *expression* ; | ;
15. *selection-stmt* → **if** (*expression*) *statement*
    | **if** (*expression*) *statement* **else** *statement*
16. *iteration-stmt* → **while** (*expression*) *statement*
17. *return-stmt* → **return** ; | **return** *expression* ;
18. *expression* → *var* = *expression* | *simple-expression*
19. *var* → **ID** | **ID** [*expression*]
20. *simple-expression* → *additive-expression relop additive-expression*
    | *additive-expression*
21. *relop* → <= | < | > | >= | == | !=
22. *additive-expression* → *additive-expression addop term* | *term*
23. *addop* → +|-
24. *term* → *term mulop factor* | *factor*
25. *mulop* → * | /
26. *factor* → (*expression*) | *var* | *call* | **NUM**

27. *call* → `ID` (*args*)
28. *args* → *arg-list* | *empty*
29. *arg-list* → *arg-list* , *expression* | *expression*

Proporcionamos una breve explicación de la semántica asociada para cada una de estas reglas gramaticales.

1. *program* → *declaration-list*
2. *declaration-list* → *declaration-list declaration* | *declaration*
3. *declaration* → *var-declaration* | *fun-declaration*

Un programa (*program*) se compone de una lista (o secuencia) de declaraciones (*declaration-list*), las cuales pueden ser declaraciones de variable o función, en cualquier orden. Debe haber al menos una declaración. Las restricciones semánticas son como sigue (éstas no se presentan en C). Todas las variables y funciones deben ser declaradas antes de utilizarlas (esto evita las referencias de retroajuste). La última declaración en un programa debe ser una declaración de función con el nombre `main`. Advierta que C− carece de prototipos, de manera que no se hace una distinción entre declaraciones y definiciones (como en el lenguaje C).

4. *var-declaration* → *type-specifier* `ID` ; | *type-specifier* `ID` [*NUM*] ;
5. *type-specifier* → `int` | `void`

Una declaración de variable declara una variable simple de tipo entero o una variable de arreglo cuyo tipo base es entero, y cuyos índices abarcan desde 0 . . *NUM*−1. Observe que en C− los únicos tipos básicos son entero y vacío ("void"). En una declaración de variable sólo se puede utilizar el especificador de tipo `int`. `Void` es para declaraciones de función (véase más adelante). Advierta también que sólo se puede declarar una variable por cada declaración.

6. *fun-declaration* → *type-specifier* `ID` (*params*) *compound-stmt*
7. *params* → *param-list* | `void`
8. *param-list* → *param-list* , *param* | *param*
9. *param* → *type-specifier* `ID` | *type-specifier* `ID` [ ]

Una declaración de función consta de un especificador de tipo (*type-specifier*) de retorno, un identificador y una lista de parámetros separados por comas dentro de paréntesis, seguida por una sentencia compuesta con el código para la función. Si el tipo de retorno de la función es `void`, entonces la función no devuelve valor alguno (es decir, es un procedimiento). Los parámetros de una función pueden ser `void` (es decir, sin parámetros) o una lista que representa los parámetros de la función. Los parámetros seguidos por corchetes son parámetros de arreglo cuyo tamaño puede variar. Los parámetros enteros simples son pasados por valor. Los parámetros de arreglo son pasados por referencia (es decir, como apuntadores) y deben ser igualados mediante una variable de arreglo durante una llamada. Advierta que no hay parámetros de tipo "función". Los parámetros de una función tienen un ámbito igual a la sentencia compuesta de la declaración de función, y cada invocación de una función tiene un conjunto separado de parámetros. Las funciones pueden ser recursivas (hasta el punto en que la declaración antes del uso lo permita).

10. *compound-stmt* → { *local-declarations statement-list* }

Una sentencia compuesta se compone de llaves que encierran un conjunto de declaraciones y sentencias. Una sentencia compuesta se realiza al ejecutar la secuencia de sentencias

en el orden dado. Las declaraciones locales tienen un ámbito igual al de la lista de sentencias de la sentencia compuesta y reemplazan cualquier declaración global.

11. *local-declarations* → *local-declarations var-declaration* | *empty*
12. *statement-list* → *statement-list statement* | *empty*

Advierta que tanto la lista de declaraciones como la lista de sentencias pueden estar vacías. (El no terminal *empty* representa la cadena vacía, que se describe en ocasiones como ε.)

13. *statement* → *expression-stmt*
| *compound-stmt*
| *selection-stmt*
| *iteration-stmt*
| *return-stmt*
14. *expression-stmt* → *expression* **;** | **;**

Una sentencia de expresión tiene una expresión opcional seguida por un signo de punto y coma. Tales expresiones por lo regular son evaluadas por sus efectos colaterales. Por consiguiente, esta sentencia se utiliza para asignaciones y llamadas de función.

15. *selection-stmt* → `if` (*expression*) *statement*
| `if` (*expression*) *statement* `else` *statement*

La sentencia if tiene la semántica habitual: la expresión es evaluada; un valor distinto de cero provoca la ejecución de la primera sentencia; un valor de cero ocasiona la ejecución de la segunda sentencia, si es que existe. Esta regla produce la ambigüedad clásica del else ambiguo, la cual se resuelve de la manera estándar: la parte else siempre se analiza sintácticamente de manera inmediata como una subestructura del if actual (la regla de eliminación de ambigüedad "de anidación más cercana").

16. *iteration-stmt* → `while` (*expression*) *statement*

La sentencia while es la única sentencia de iteración en el lenguaje C−. Se ejecuta al evaluar de manera repetida la expresión y al ejecutar entonces la sentencia si la expresión evalúa un valor distinto de cero, finalizando cuando la expresión se evalúa a 0.

17. *return-stmt* → `return` **;** | `return` *expression* **;**

Una sentencia de retorno puede o no devolver un valor. Las funciones no declaradas como `void` deben devolver valores. Las funciones declaradas `void` no deben devolver valores. Un retorno provoca la transferencia del control de regreso al elemento que llama (o la terminación del programa si está dentro de `main`).

18. *expression* → *var* = *expression* | *simple-expression*
19. *var* → *ID* | *ID* [*expression*]

Una expresión es una referencia de variable seguida por un símbolo de asignación (signo de igualdad) y una expresión, o solamente una expresión simple. La asignación tiene la semántica de almacenamiento habitual: se encuentra la localidad de la variable representada por *var*, luego se evalúa la subexpresión a la derecha de la asignación, y se almacena el valor de la subexpresión en la localidad dada. Este valor también es devuelto como el valor de la expresión completa. Una *var* es una variable (entera) simple o bien una variable de arreglo subindizada. Un subíndice negativo provoca que el programa se detenga (a diferencia de C). Sin embargo, no se verifican los límites superiores de los subíndices.

Las variables representan una restricción adicional en C− respecto a C. En C el objetivo de una asignación debe ser un **valor l**, y los valores l son direcciones que pueden ser obtenidas mediante muchas operaciones. En C− los únicos valores l son aquellos dados por la sintaxis de *var*, y así esta categoría es verificada sintácticamente, en vez de hacerlo durante la verificación de tipo como en C. Por consiguiente, en C− está prohibida la aritmética de apuntadores.

20. *simple-expression* → *additive-expression relop additive-expression*
| *additive-expression*

21. *relop* → <= | < | > | >= | == | !=

Una expresión simple se compone de operadores relacionales que no se asocian (es decir, una expresión sin paréntesis puede tener solamente un operador relacional). El valor de una expresión simple es el valor de su expresión aditiva si no contiene operadores relacionales, o bien, 1 si el operador relacional se evalúa como verdadero, o 0 si se evalúa como falso.

22. *additive-expression* → *additive-expression addop term* | *term*
23. *addop* → + | -
24. *term* → *term mulop factor* | *factor*
25. *mulop* → * | /

Los términos y expresiones aditivas representan la asociatividad y precedencia típicas de los operadores aritméticos. El símbolo / representa la división entera; es decir, cualquier residuo es truncado.

26. *factor* → (*expression*) | *var* | *call* | **NUM**

Un factor es una expresión encerrada entre paréntesis, una variable, que evalúa el valor de su variable; una llamada de una función, que evalúa el valor devuelto de la función; o un NUM, cuyo valor es calculado por el analizador léxico. Una variable de arreglo debe estar subindizada, excepto en el caso de una expresión compuesta por una ID simple y empleada en una llamada de función con un parámetro de arreglo (véase a continuación).

27. *call* → **ID** (*args*)
28. *args* → *arg-list* | *empty*
29. *arg-list* → *arg-list* , *expression* | *expression*

Una llamada de función consta de un **ID** (el nombre de la función), seguido por sus argumentos encerrados entre paréntesis. Los argumentos pueden estar vacíos o estar compuestos por una lista de expresiones separadas mediante comas, que representan los valores que se asignarán a los parámetros durante una llamada. Las funciones deben ser declaradas antes de llamarlas, y el número de parámetros en una declaración debe ser igual al número de argumentos en una llamada. Un parámetro de arreglo en una declaración de función debe coincidir con una expresión compuesta de un identificador simple que representa una variable de arreglo.

Finalmente, las reglas anteriores no proporcionan sentencia de entrada o salida. Debemos incluir tales funciones en la definición de C−, puesto que a diferencia del lenguaje C, C− no tiene facilidades de ligado o compilación por separado. Por lo tanto, consideraremos dos funciones por ser **predefinidas** en el ambiente global, como si tuvieran las declaraciones indicadas:

```
int input(void) { . . . }
void output(int x) { . . . }
```

La función **input** no tiene parámetros y devuelve un valor entero desde el dispositivo de entrada estándar (por lo regular el teclado). La función **output** toma un parámetro entero, cuyo valor imprime a la salida estándar (por lo regular la pantalla), junto con un retorno de línea.

A.3 PROGRAMAS DE MUESTRA EN C–

El siguiente es un programa que introduce dos enteros, calcula su máximo común divisor y lo imprime:

```
/* Un programa para realizar el algoritmo
   de Euclides para calcular mcd. */

int gcd (int u, int v)
{ if (v == 0) return u ;
  else return gcd(v,u-u/v*v);
  /* u-u/v*v == u mod v */
}

void main(void)
{ int x; int y;
  x = input(); y = input();
  output(gcd(x,y));
}
```

A continuación tenemos un programa que introduce una lista de 10 enteros, los clasifica por orden de selección, y los exhibe otra vez:

```
/* Un programa para realizar ordenación por
   selección en un arreglo de 10 elementos. */

int x[10];

int minloc ( int a[], int low, int high )
{ int i; int x; int k;
  k = low;
  x = a[low];
  i = low + 1;
  while (i < high)
    { if (a[i] < x)
        { x = a[i];
          k = i; }
      i = i + 1;
    }
  return k;
}

void sort( int a[], int low, int high)
```

```
{ int i; int k;
  i = low;
  while (i < high-1)
    { int t;
      k = minloc(a,i,high);
      t = a[k];
      a[k] = a[i];
      a[i] = t;
      i = i + 1;
    }
}

void main(void)
{ int i;
  i = 0;
  while (i < 10)
    { x[i] = input();
      i = i + 1; }
  sort(x,0,10);
  i = 0;
  while (i < 10)
    { output(x[i]);
      i = i + 1; }
}
```

A.4 UN AMBIENTE DE EJECUCIÓN DE LA MÁQUINA TINY PARA EL LENGUAJE C−

La descripción siguiente supone que se conoce la máquina TINY como se expuso en la sección 8.7 y se comprenden los ambientes de ejecución basados en pila expuestos en el capítulo 7. Como C− (a diferencia de TINY) tiene procedimientos recursivos, el ambiente de ejecución debe estar basado en pilas. El ambiente consta de un área global en la parte superior o tope de dMem, y la pila justo debajo de ella, creciendo hacia abajo en dirección al 0. Puesto que C− no contiene apuntadores o asignación dinámica, no hay necesidad de un apilamiento. La organización básica de cada registro de activación (o marco de pila) en C− es

Aquí, fp es el **apuntador de marco actual**, que se mantiene en un registro para facilidad de acceso. El ofp (por las siglas del término apuntador de marco antiguo en inglés), como se comentó en el capítulo 7, es el **vínculo de control**. Las constantes para la terminación derecha en FO (por las siglas del término desplazamiento de marco en inglés) son los desplazamientos en los que se almacena cada una de las cantidades indicadas. El valor initFO es el desplazamiento en el que los parámetros y variables locales comienzan sus áreas de almacenamiento en un registro de activación. Como la máquina TINY no contiene apuntador de pila, todas las referencias a campos dentro de un registro de activación utilizarán el fp, con desplazamientos de marco negativos.

Por ejemplo, si tenemos la siguiente declaración de función en C−,

```
int f(int x, int y)
{ int z;
  . . .
}
```

entonces **x**, **y** y **z** deben asignarse en el marco actual, y el desplazamiento de marco en el principio de la generación de código para el cuerpo de **f** será −5 (una localidad para cada una de **x**, **y** y **z** y dos localidades para la información de administración del registro de activación). Los desplazamientos de **x**, **y** y **z** son −2, −3 y −4, respectivamente.

Pueden encontrarse referencias globales en localidades absolutas en memoria. No obstante, como con TINY, también preferimos referenciar estas variables mediante desplazamientos desde un registro. Hacemos esto manteniendo un registro fijado, al cual llamaremos gp, que siempre apunta a la dirección máxima. Puesto que el simulador de TM almacena esta dirección en la localidad 0 antes que comience la ejecución, el gp puede ser cargado desde la localidad 0 en el arranque, y el siguiente preludio estándar inicializa el ambiente de ejecución:

```
0: LD gp,    0(ac)   * carga gp con maxaddress
1: LDA fp,   0(gp)   * copia gp a fp
2: ST ac,    0(ac)   * limpia la localidad 0
```

Las llamadas de función también requieren que la localidad del código de inicio para sus cuerpos se utilice en una secuencia de llamada. También preferimos llamar funciones efectuando un salto relativo utilizando el valor actual del pc en vez de un salto absoluto. (Esto hace al código potencialmente relocalizable.) El procedimiento de utilería **emitRAbs** en **code.h/code.c** se puede emplear para este propósito. (Toma una localidad de código absoluta y lo hace relativo al utilizar la localidad de generación de código actual.)

Por ejemplo, suponga que deseamos llamar una función f cuyo código comienza en la localidad 27, y que estamos actualmente en la localidad 42. Entonces, en lugar de generar el salto absoluto

```
42: LDC pc, 27(*)
```

generaríamos

```
42: LDA pc, -16(pc)
```

puesto que 27 − (42 + 1) = −16.

La secuencia de llamada Una división del trabajo razonable entre el elemento que llama y el que es llamado es consentir que el elemento que llama almacene los valores de los argumentos en el nuevo marco y crear el nuevo marco, excepto para el almacenamiento del apuntador

de retorno en la posición retFO. En vez de almacenar el apuntador de retorno mismo, el elemento que llama lo deja en el registro ac, y el elemento llamado lo almacena en el nuevo marco. De este modo, todo cuerpo de función debe comenzar con código para almacenar ese valor en el (ahora actual) marco:

```
ST ac, retFO(fp)
```

Esto guarda una instrucción para cada sitio de llamada. Al regreso, cada función carga entonces el pc con esta dirección de retorno al ejecutar la instrucción

```
LD pc, retFO(fp)
```

De manera correspondiente, el elemento que llama calcula los argumentos uno por uno, e inserta cada uno de ellos en la pila en su posición apropiada antes de insertar el nuevo marco. El elemento que llama también debe guardar el fp actual en el marco del ofpFO antes de insertar el nuevo marco. Después de un retorno desde el elemento llamado, el elemento que llama descarta entonces el nuevo marco al cargar el fp con el fp antiguo. De esta forma, una llamada a una función de dos parámetros provocará la generación del código siguiente:

```
<código para calcular el primer arg>
ST ac, frameoffset+initFO (fp)
<código para calcular el segundo arg>
ST ac, frameoffset+initFO-1 (fp)
ST fp, frameoffset+ofpFO (fp) * almacena el fp actual
LDA fp frameOffset(fp) * inserta nuevo marco
LDA ac,1(pc) * guarda el retorno en ac
LDA pc, . . .(pc) * salto relativo a la entrada de función
LD fp, ofpFO(fp) * extrae el marco actual
```

Cálculos de dirección Puesto que se permiten tanto variables como arreglos subindizados en el lado izquierdo de la asignación, debemos distinguir entre direcciones y valores durante la compilación. Por ejemplo, en la sentencia

```
a[i] := a[i+1];
```

la expresión **a[i]** hace referencia a la dirección de **a[i]**, mientras que la expresión **a[i+1]** hace referencia al valor de **a** en la localidad **i+1**. Esta distinción se puede obtener utilizando un parámetro **isAddress** para el procedimiento **cGen**. Cuando este parámetro es verdadero, **cGen** genera código para calcular la dirección de una variable más que de su valor. En el caso de una variable simple, esto significa sumar el desplazamiento al gp (en el caso de una variable global) o al fp (en el caso de una variable local) y cargar el resultado en el ac:

```
LDA ac, offset(fp) ** pone la dirección de la variable local
en ac
```

En el caso de una variable de arreglo, esto significa sumar el valor del índice a la dirección base del arreglo y cargar el resultado en el ac, como se describe a continuación.

Arreglos Los arreglos se asignan en la pila comenzando en el desplazamiento de marco actual y extendiéndose hacia abajo en la memoria a fin de incrementar el subíndice, de la manera que se muestra a continuación:

Advierta que las localidades de arreglos se calculan restando el valor del índice de la dirección base.

Cuando un arreglo se pasa a una función, solamente se pasa la dirección base. La asignación del área para los elementos base se hace sólo una vez y permanece fija durante la vida del arreglo. Los argumentos de la función no incluyen los elementos reales de un arreglo, solamente la dirección. De este modo, los parámetros de arreglo se convierten en parámetros de referencia. Esto causa una anomalía cuando los parámetros de arreglo son referenciados dentro de una función, puesto que deben ser tratados como si tuvieran sus direcciones base en vez de sus valores almacenados en memoria. Así, un parámetro de arreglo tiene su dirección base calculada utilizando una operación LD en vez de una LDA.

A.5 PROYECTOS DE PROGRAMACIÓN UTILIZANDO C− Y TM

No es irracional en un curso de compiladores con duración de un semestre requerir como proyecto un compilador completo para el lenguaje C− basado en el compilador TINY que se analizó en este texto (y cuyo listado se puede encontrar en el apéndice B). Esto se puede coordinar de manera que se implemente cada fase del compilador a medida que se estudia la teoría asociada. Otra manera de hacerlo sería que el instructor suministrara una o más partes del compilador C− y pidiera a los estudiantes que completaran las partes restantes. Esto es especialmente útil cuando se tiene poco tiempo (por ejemplo en un curso trimestral) o cuando los estudiantes estarán generando código ensamblador para una máquina "real", como la Sparc o la PC (lo cual requiere más detalle durante la fase de generación de código). Implementar sólo una parte del compilador C− es menos útil, puesto que entonces se restringen las interacciones entre las partes y la capacidad para probar el código. La siguiente lista de tareas separadas se suministra como una manera de facilitar las cosas, con la advertencia de que cada tarea no puede ser independiente de las otras y que probablemente es mejor que todas las tareas se terminen en orden para obtener una experiencia completa de la escritura de un compilador.

PROYECTOS

A.1. Implemente una utilería de tabla de símbolos adecuada para el lenguaje C−. Esto requerirá una estructura de tabla que incorpore información de ámbito, ya sea como tablas separadas vinculadas en conjunto o con un mecanismo de eliminación que funcione de manera basada en pilas, como se describió en el capítulo 6.

A.2. Implemente un analizador léxico de C−, ya sea a mano como un DFA o utilizando Lex, como se describió en el capítulo 2.

A.3. Diseñe una estructura de árbol sintáctico para C− apropiada para la generación mediante un analizador sintáctico.

A.4. Implemente un analizador sintáctico de C− (esto requiere un analizador léxico de C−), ya sea a mano utilizando descendentes recursivos o mediante el uso de Yacc, como se describió en el capítulo 4 o 5. El analizador sintáctico debería generar un árbol sintáctico apropiado (véase el proyecto A.3).

A.5. Implemente un analizador semántico para C−. El requerimiento principal del analizador, además de obtener información en la tabla de símbolos, es realizar verificación de tipo en el uso de variables y funciones. Como no hay apuntadores o estructuras, y el único tipo básico es el entero, los tipos que necesitan ser tratados mediante el verificador de tipo son los tipos void, integer, array y function.

A.6. Implemente un generador de código para C−, de acuerdo con el ambiente de ejecución descrito en la sección anterior.

Apéndice B

Listado del compilador TINY

```
1  /*********************************************************/
2  /* Archivo: main.c                                       */
3  /* Programa principal para el compilador TINY            */
4  /* Construcción de compiladores: principios y práctica   */
5  /* Kenneth C. Louden                                     */
6  /*********************************************************/
7
8  #include "globals.h"
9
10 /* establece NO_PARSE a TRUE para obtener un compilador sólo
      de análisis léxico */
11 #define NO_PARSE FALSE
12
13 /* establece NO_ANALYZE a TRUE para obtener un compilador
      sólo de análisis sintáctico */
14 #define NO_ANALYZE FALSE
15
16 /* establece NO_CODE a TRUE para obtener un compilador que no
17  * genere código
18  */
19 #define NO_CODE FALSE
20
21 #include "util.h"
22 #if NO_PARSE
23 #include "scan.h"
24 #else
25 #include "parse.h"
26 #if !NO_ANALYZE
27 #include "analyze.h"
28 #if !NO_CODE
29 #include "cgen.h"
30 #endif
31 #endif
32 #endif
33
34 /* asignación de variables globales */
```

```
35 int lineno = 0;
36 FILE * source;
37 FILE * listing;
38 FILE * code;
39
40 /* asignar y establecer marcas de rastreo */
41 int EchoSource = TRUE;
42 int TraceScan = TRUE;
43 int TraceParse = TRUE;
44 int TraceAnalyze = TRUE;
45 int TraceCode = TRUE;
46
47 int Error = FALSE;
48
49 main( int argc, char * argv[] )
50 { TreeNode * syntaxTree;
51   char pgm[20]; /* nombre de archivo de código fuente */
52   if (argc != 2)
53     { fprintf(stderr,"usage: %s <filename>\n",argv[0]);
54       exit(1);
55     }
56   strcpy(pgm,argv[1]) ;
57   if (strchr (pgm, '.') == NULL)
58       strcat(pgm,".tny");
59   source = fopen(pgm,"r");
60   if (source==NULL)
61   { fprintf(stderr,"File %s not found\n",pgm);
62     exit(1);
63   }
64   listing = stdout; /* enviar listado a la pantalla */
65   fprintf(listing,"\nTINY COMPILATION: %s\n",pgm);
66 #if NO_PARSE
67   while (getToken()!=ENDFILE);
68 #else
69   syntaxTree = parse();
70   if (TraceParse) {
71     fprintf(listing,"\nSyntax tree:\n");
72     printTree(syntaxTree);
73   }
74 #if !NO_ANALYZE
75   if (! Error)
76   { fprintf(listing,"\nBuilding Symbol Table...\n");
77     buildSymtab(syntaxTree);
78     fprintf(listing,"\nChecking Types...\n");
79     typeCheck(syntaxTree);
80     fprintf(listing,"\nType Checking Finished\n");
```

```
81    }
82  #if !NO_CODE
83    if (! Error)
84    { char * codefile;
85      int fnlen = strcspn(pgm,".");
86      codefile = (char *) calloc(fnlen+4, sizeof(char));
87      strncpy(codefile,pgm,fnlen);
88      strcat(codefile,".tm");
89      code = fopen(codefile,"w");
90      if (code == NULL)
91      { printf("Unable to open %s\n",codefile);
92        exit(1);
93      }
94      codeGen(syntaxTree,codefile);
95      fclose(code);
96    }
97  #endif
98  #endif
99  #endif
100   return 0;
101 }
```

```
150 /*************************************************************/
151 /* Archivo: globals.h                                        */
152 /* Los tipos globales y variables para el compilador TINY    */
153 /* deben aparecer antes de otros archivos "include"          */
154 /* Construcción de compiladores: principios y práctica       */
155 /* Kenneth C. Louden                                         */
156 /*************************************************************/
157
158 #ifndef _GLOBALS_H_
159 #define _GLOBALS_H_
160
161 #include <stdio.h>
162 #include <stdlib.h>
163 #include <ctype.h>
164 #include <string.h>
165
166 #ifndef FALSE
167 #define FALSE 0
168 #endif
169
170 #ifndef TRUE
171 #define TRUE 1
172 #endif
```

```
173
174 /* MAXRESERVED = el número de palabras reservadas */
175 #define MAXRESERVED 8
176
177 typedef enum
178     /* tokens de administración */
179    {ENDFILE,ERROR,
180     /* palabras reservadas */
181     IF,THEN,ELSE,END,REPEAT,UNTIL,READ,WRITE,
182     /* Tokens de caracteres múltiples */
183     ID,NUM,
184     /* símbolos especiales */
185     ASSIGN,EQ,LT,PLUS,MINUS,TIMES,OVER,LPAREN,RPAREN,SEMI
186    } TokenType;
187
188 extern FILE* source; /* archivo de texto de código fuente */
189 extern FILE* listing; /* archivo de texto de salida de listado */
190 extern FILE* code; /* archivo de texto de código para el simulador TM */
191
192 extern int lineno; /* número de línea fuente para el listado */
193
194 /*****************************************************/
195 /** árbol sintáctico para el análisis sintáctico **/
196 /*****************************************************/
197
198 typedef enum {StmtK,ExpK} NodeKind;
199 typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK} StmtKind;
200 typedef enum {OpK,ConstK,IdK} ExpKind;
201
202 /* ExpType es utilizado para verificación de tipo */
203 typedef enum {Void,Integer,Boolean} ExpType;
204
205 #define MAXCHILDREN 3
206
207 typedef struct treeNode
208    { struct treeNode * child[MAXCHILDREN];
209      struct treeNode * sibling;
210      int lineno;
211      NodeKind nodekind;
212      union { StmtKind stmt; ExpKind exp;} kind;
213      union { TokenType op;
214              int val;
215              char * name; } attr;
216      ExpType type; /* para verificación de tipo de expresiones */
217    } TreeNode;
218
```

```
219 /**************************************************/
220 /***********     Marcas para rastreo    ************/
221 /**************************************************/
222
223 /* EchoSource = TRUE causa que el programa fuente se
224  * replique en el archivo de listado con números de línea
225  * durante el análisis sintáctico
226  */
227 extern int EchoSource;
228
229 /* TraceScan = TRUE ocasiona que la información de token sea
230  * impresa al archivo de listado a medida que cada token sea
231  * reconocido por el analizador léxico
232  */
233 extern int TraceScan;
234
235 /* TraceParse = TRUE provoca que el árbol sintáctico sea
236  * impreso en el archivo del listado en forma linealizada
237  * (utilizando sangrías para los hijos)
238  */
239 extern int TraceParse;
240
241 /* TraceAnalyze = TRUE provoca que la tabla de símbolos inserte
242  * y busque para informarlo al archivo de listado
243  */
244 extern int TraceAnalyze;
245
246 /* TraceCode = TRUE causa que se escriban comentarios
247  * al archivo de código TM a medida que se genera el código
248  */
249 extern int TraceCode;
250
251 /* Error = TRUE evita pasos adicionales si se presenta un error */
252 extern int Error;
253 #endif
```

```
300 /*******************************************************/
301 /* Archivo: util.h                                     */
302 /* Funciones de utilidad para el compilador TINY       */
303 /* Construcción de compiladores: principios y práctica */
304 /* Kenneth C. Louden                                   */
305 /*******************************************************/
306
307 #ifndef _UTIL_H_
308 #define _UTIL_H_
309
```

```
310 /* El procedimiento printToken imprime un token
311  * y su lexema al archivo del listado
312  */
313 void printToken( TokenType, const char* );
314
315 /* La función newStmtNode crea un nuevo nodo de sentencia
316  * para la construcción del árbol sintáctico
317  */
318 TreeNode * newStmtNode(StmtKind);
319
320 /* La función newExpNode crea un nuevo nodo de expresión
321  * para la construcción del árbol sintáctico
322  */
323 TreeNode * newExpNode(ExpKind);
324
325 /* La función copyString asigna y crea una nueva
326  * copia de una cadena existente
327  */
328 char * copyString( char * );
329
330 /* el procedimiento printTree imprime un árbol sintáctico para el
331  * archivo de listado en el que los subárboles se indican mediante sangrías
332  */
333 void printTree( TreeNode * );
334
335 #endif
```

```
350 /************************************************************/
351 /* Archivo: util.c                                          */
352 /* Implementación de función de utilidad                    */
353 /* para el compilador TINY                                  */
354 /* Construcción de compiladores: principios y práctica      */
355 /* Kenneth C. Louden                                        */
356 /************************************************************/
357
358 #include "globals.h"
359 #include "util.h"
360
361 /* El procedimiento printToken imprime un token
362  * y su lexema al archivo de listado
363  */
364 void printToken( TokenType token, const char* tokenString )
365 { switch (token)
366   { case IF:
367     case THEN:
368     case ELSE:
```

```
369      case END:
370      case REPEAT:
371      case UNTIL:
372      case READ:
373      case WRITE:
374        fprintf(listing,
375           "reserved word: %s\n",tokenString);
376        break;
377      case ASSIGN: fprintf(listing,":=\n"); break;
378      case LT: fprintf(listing,"<\n"); break;
379      case EQ: fprintf(listing,"=\n"); break;
380      case LPAREN: fprintf(listing,"(\n"); break;
381      case RPAREN: fprintf(listing,")\n"); break;
382      case SEMI: fprintf(listing,";\n"); break;
383      case PLUS: fprintf(listing,"+\n"); break;
384      case MINUS: fprintf(listing,"-\n"); break;
385      case TIMES: fprintf(listing,"*\n"); break;
386      case OVER: fprintf(listing,"/\n"); break;
387      case ENDFILE: fprintf(listing,"EOF\n"); break;
388      case NUM:
389        fprintf(listing,
390           "NUM, val= %s\n",tokenString);
391        break;
392      case ID:
393        fprintf(listing,
394           "ID, name= %s\n",tokenString);
395        break;
396      case ERROR:
397        fprintf(listing,
398           "ERROR: %s\n",tokenString);
399        break;
400      default: /* nunca debería ocurrir */
401        fprintf(listing,"Unknown token: %d\n",token);
402    }
403  }
404
405  /* La función newStmtNode crea un nuevo nodo de sentencia
406   * para la construcción del árbol sintáctico
407   */
408  TreeNode * newStmtNode(StmtKind kind)
409  { TreeNode * t = (TreeNode *) malloc(sizeof(TreeNode));
410    int i;
411    if (t==NULL)
412      fprintf(listing,"Out of memory error at line %d\n",lineno);
413    else {
414      for (i=0;i<MAXCHILDREN;i++) t->child[i] = NULL;
415      t->sibling = NULL;
```

```
416      t->nodekind = StmtK;
417      t->kind.stmt = kind;
418      t->lineno = lineno;
419    }
420    return t;
421  }
422
423  /* La función newExpNode crea un nuevo nodo de expresión
424   * para la construcción de árbol sintáctico
425   */
426  TreeNode * newExpNode(ExpKind kind)
427  { TreeNode * t = (TreeNode *) malloc(sizeof(TreeNode));
428    int i;
429    if (t==NULL)
430      fprintf(listing,"Out of memory error at line %d\n",lineno);
431    else {
432      for (i=0;i<MAXCHILDREN;i++) t->child[i] = NULL;
433      t->sibling = NULL;
434      t->nodekind = ExpK;
435      t->kind.exp = kind;
436      t->lineno = lineno;
437      t->type = Void;
438    }
439    return t;
440  }
441
442  /* La función copyString asigna y hace una nueva
443   * copia de una cadena existente
444   */
445  char * copyString(char * s)
446  { int n;
447    char * t;
448    if (s==NULL) return NULL;
449    n = strlen(s)+1;
450    t = malloc(n);
451    if (t==NULL)
452      fprintf(listing,"Out of memory error at line %d\n",lineno);
453    else strcpy(t,s);
454    return t;
455  }
456
457  /* printTree utiliza la variable indentno para
458   * almacenar el número actual de espacios para sangría
459   */
460  static indentno = 0;
461
462  /* macros para incrementar/decrementar la sangría */
```

```
463 #define INDENT indentno+=2
464 #define UNINDENT indentno-=2
465
466 /* printSpaces realiza las sangrías mediante la impresión de espacios */
467 static void printSpaces(void)
468 { int i;
469   for (i=0;i<indentno;i++)
470     fprintf(listing," ");
471 }
472
473 /* el procedimiento printTree imprime un árbol sintáctico al
474  * archivo de listado utilizando las sangrías para indicar los subárboles
475  */
476 void printTree( TreeNode * tree )
477 { int i;
478   INDENT;
479   while (tree != NULL) {
480     printSpaces();
481     if (tree->nodekind==StmtK)
482     { switch (tree->kind.stmt) {
483         case IfK:
484           fprintf(listing,"If\n");
485           break;
486         case RepeatK:
487           fprintf(listing,"Repeat\n");
488           break;
489         case AssignK:
490           fprintf(listing,"Assign to: %s\n",tree->attr.name);
491           break;
492         case ReadK:
493           fprintf(listing,"Read: %s\n",tree->attr.name);
494           break;
495         case WriteK:
496           fprintf(listing,"Write\n");
497           break;
498         default:
499           fprintf(listing,"Unknown ExpNode kind\n");
500           break;
501       }
502     }
503     else if (tree->nodekind==ExpK)
504     { switch (tree->kind.exp) {
505         case OpK:
506           fprintf(listing,"Op: ");
507           printToken(tree->attr.op,"\0");
508           break;
509         case ConstK:
```

```
510            fprintf(listing,"const: %d\n",tree->attr.val);
511            break;
512          case IdK:
513            fprintf(listing,"Id: %s\n",tree->attr.name);
514            break;
515          default:
516            fprintf(listing,"Unknown ExpNode kind\n");
517            break;
518        }
519      }
520      else fprintf(listing,"Unknown node kind\n");
521      for (i=0;i<MAXCHILDREN;i++)
522           printTree(tree->child[i]);
523      tree = tree->sibling;
524    }
525    UNINDENT;
526  }
```

```
550  /*********************************************************************/
551  /* Archivo: scan.h                                                   */
552  /* La interfaz del analizador léxico para el compilador TINY         */
553  /* Construcción de compiladores: principios y práctica               */
554  /* Kenneth C. Louden                                                 */
555  /*********************************************************************/
556
557  #ifndef _SCAN_H_
558  #define _SCAN_H_
559
560  /* MAXTOKENLEN es el tamaño máximo de un token */
561  #define MAXTOKENLEN 40
562
563  /* el arreglo tokenString almacena el lexema de cada token */
564  extern char tokenString[MAXTOKENLEN+1];
565
566  /* la función getToken devuelve el
567   * token siguiente en el archivo fuente
568   */
569  TokenType getToken(void);
570
571  #endif
```

```
600  /*********************************************************************/
601  /* Archivo: scan.c                                                   */
602  /* La implementación del analizador léxico para el compilador TINY   */
603  /* Construcción de compiladores: principios y práctica               */
```

```
604 /* Kenneth C. Louden                                    */
605 /*****************************************************/
606
607 #include "globals.h"
608 #include "util.h"
609 #include "scan.h"
610
611 /* estados en analizador léxico DFA */
612 typedef enum
613    { START,INASSIGN,INCOMMENT,INNUM,INID,DONE }
614    StateType;
615
616 /* lexema de identificador o palabra reservada */
617 char tokenString[MAXTOKENLEN+1];
618
619 /* BUFLEN = longitud del buffer de entrada para
620    las líneas del código fuente */
621 #define BUFLEN 256
622
623 static char lineBuf[BUFLEN]; /* mantiene la línea actual */
624 static int linepos = 0; /* posición actual en LineBuf */
625 static int bufsize = 0; /* tamaño actual de la cadena del buffer */
626
627 /* getNextChar obtiene el siguiente carácter distinto del blanco
628    de lineBuf leyendo en una nueva línea si lineBuf está
629    agotado */
630 static char getNextChar(void)
631 { if (!(linepos < bufsize))
632   { lineno++;
633     if (fgets(lineBuf,BUFLEN-1,source))
634     { if (EchoSource) fprintf(listing,"%4d: %s",lineno,lineBuf);
635   bufsize = strlen(lineBuf);
636   linepos = 0;
637   return lineBuf[linepos++];
638     }
639     else return EOF;
640   }
641   else return lineBuf[linepos++];
642 }
643
644 /* ungetNextChar reajusta un carácter
645    en lineBuf */
646 static void ungetNextChar(void)
647 { linepos-- ;}
648
649 /* tabla de búsqueda de palabras reservadas */
650 static struct
```

```
651      { char* str;
652        TokenType tok;
653      } reservedWords[MAXRESERVED]
654    =   {{"if",IF},{"then",THEN},{"else",ELSE},{"end",END},
655        {"repeat",REPEAT},{"until",UNTIL},{"read",READ},
656        {"write",WRITE}};
657
658 /* busca un identificador para ver si es una palabra reservada */
659 /* utiliza búsqueda lineal */
660 static TokenType reservedLookup (char * s)
661 { int i;
662   for (i=0;i<MAXRESERVED;i++)
663      if (!strcmp(s,reservedWords[i].str))
664        return reservedWords[i].tok;
665   return ID;
666 }
667
668 /***********************************************/
669 /* la función principal del analizador léxico */
670 /***********************************************/
671 /* la función getToken devuelve el
672  * token siguiente en el archivo fuente
673  */
674 TokenType getToken(void)
675 {  /* índice para almacenamiento en tokenString */
676    int tokenStringIndex = 0;
677    /* mantiene el token actual que se devolverá */
678    TokenType currentToken;
679    /* estado actual - siempre comienza en START */
680    StateType state = START;
681    /* marca para indicar grabar en tokenString */
682    int save;
683    while (state != DONE)
684    { char c = getNextChar();
685      save = TRUE;
686      switch (state)
687      { case START:
688          if (isdigit(c))
689            state = INNUM;
690          else if (isalpha(c))
691            state = INID;
692          else if (c == ':')
693            state = INASSIGN;
694          else if ((c == ' ') || (c == '\t') || (c == '\n'))
695            save = FALSE;
696          else if (c == '{')
697          { save = FALSE;
```

```
698             state = INCOMMENT;
699           }
700           else
701           { state = DONE;
702             switch (c)
703             { case EOF:
704                 save = FALSE;
705                 currentToken = ENDFILE;
706                 break;
707               case '=':
708                 currentToken = EQ;
709                 break;
710               case '<':
711                 currentToken = LT;
712                 break;
713               case '+':
714                 currentToken = PLUS;
715                 break;
716               case '-':
717                 currentToken = MINUS;
718                 break;
719               case '*':
720                 currentToken = TIMES;
721                 break;
722               case '/':
723                 currentToken = OVER;
724                 break;
725               case '(':
726                 currentToken = LPAREN;
727                 break;
728               case ')':
729                 currentToken = RPAREN;
730                 break;
731               case ';':
732                 currentToken = SEMI;
733                 break;
734               default:
735                 currentToken = ERROR;
736                 break;
737             }
738           }
739           break;
740         case INCOMMENT:
741           save = FALSE;
742           if (c == '}') state = START;
743           break;
744         case INASSIGN:
```

```
745          state = DONE;
746          if (c == '=')
747            currentToken = ASSIGN;
748          else
749          { /* respaldo en la entrada */
750            ungetNextChar();
751            save = FALSE;
752            currentToken = ERROR;
753          }
754          break;
755        case INNUM:
756          if (!isdigit(c))
757          { /* respaldo en la entrada */
758            ungetNextChar();
759            save = FALSE;
760            state = DONE;
761            currentToken = NUM;
762          }
763          break;
764        case INID:
765          if (!isalpha(c))
766          { /* respaldo en la entrada */
767            ungetNextChar();
768            save = FALSE;
769            state = DONE;
770            currentToken = ID;
771          }
772          break;
773        case DONE:
774        default: /* nunca debería ocurrir */
775          fprintf(listing,"Scanner Bug: state= %d\n",state);
776          state = DONE;
777          currentToken = ERROR;
778          break;
779      }
780      if ((save) && (tokenStringIndex <= MAXTOKENLEN))
781        tokenString[tokenStringIndex++] = c;
782      if (state == DONE)
783      { tokenString[tokenStringIndex] = '\0';
784        if (currentToken == ID)
785          currentToken = reservedLookup(tokenString);
786      }
787    }
788    if (TraceScan) {
789      fprintf(listing,"\t%d: ",lineno);
790      printToken(currentToken,tokenString);
791    }
```

```
792     return currentToken;
793 } /* fin de getToken */
```

```
850 /*****************************************************************/
851 /* Archivo: parse.h                                              */
852 /* La interfaz del analizador sintáctico para el compilador TINY  */
853 /* Construcción de compiladores: principios y práctica           */
854 /* Kenneth C. Louden                                             */
855 /*****************************************************************/
856
857 #ifndef _PARSE_H_
858 #define _PARSE_H_
859
860 /* La función parse regresa el
861  * árbol sintáctico recién construido
862  */
863 TreeNode * parse(void);
864
865 #endif
```

```
900 /*****************************************************************/
901 /* Archivo: parse.c                                              */
902 /* La implementación del analizador sintáctico para el compilador TINY */
903 /* Construcción de compiladores: principios y práctica           */
904 /* Kenneth C. Louden                                             */
905 /*****************************************************************/
906
907 #include "globals.h"
908 #include "util.h"
909 #include "scan.h"
910 #include "parse.h"
911
912 static TokenType token; /* mantiene el token actual */
913
914 /* prototipos de función para llamadas recursivas */
915 static TreeNode * stmt_sequence(void);
916 static TreeNode * statement(void);
917 static TreeNode * if_stmt(void);
918 static TreeNode * repeat_stmt(void);
919 static TreeNode * assign_stmt(void);
920 static TreeNode * read_stmt(void);
921 static TreeNode * write_stmt(void);
922 static TreeNode * exp(void);
923 static TreeNode * simple_exp(void);
924 static TreeNode * term(void);
925 static TreeNode * factor(void);
```

```
926
927 static void syntaxError(char * message)
928 { fprintf(listing,"\n>>> ");
929   fprintf(listing,"Syntax error at line %d: %s",lineno,message);
930   Error = TRUE;
931 }
932
933 static void match(TokenType expected)
934 { if (token == expected) token = getToken();
935   else {
936     syntaxError("unexpected token -> ");
937     printToken(token,tokenString);
938     fprintf(listing,"      ");
939   }
940 }
941
942 TreeNode * stmt_sequence(void)
943 { TreeNode * t = statement();
944   TreeNode * p = t;
945   while ((token!=ENDFILE) && (token!=END) &&
946          (token!=ELSE) && (token!=UNTIL))
947   { TreeNode * q;
948     match(SEMI);
949     q = statement();
950     if (q!=NULL) {
951       if (t==NULL) t = p = q;
952       else /* ahora p ya no puede ser NULL */
953       { p->sibling = q;
954         p = q;
955       }
956     }
957   }
958   return t;
959 }
960
961 TreeNode * statement(void)
962 { TreeNode * t = NULL;
963   switch (token) {
964     case IF : t = if_stmt(); break;
965     case REPEAT : t = repeat_stmt(); break;
966     case ID : t = assign_stmt(); break;
967     case READ : t = read_stmt(); break;
968     case WRITE : t = write_stmt(); break;
969     default : syntaxError("unexpected token -> ");
970               printToken(token,tokenString);
971               token = getToken();
972               break;
973   } /* fin del case */
```

```
974     return t;
975   }
976
977   TreeNode * if_stmt(void)
978   { TreeNode * t = newStmtNode(IfK);
979     match(IF);
980     if (t!=NULL) t->child[0] = exp();
981     match(THEN);
982     if (t!=NULL) t->child[1] = stmt_sequence();
983     if (token==ELSE) {
984       match(ELSE);
985       if (t!=NULL) t->child[2] = stmt_sequence();
986     }
987     match(END);
988     return t;
989   }
990
991   TreeNode * repeat_stmt(void)
992   { TreeNode * t = newStmtNode(RepeatK);
993     match(REPEAT);
994     if (t!=NULL) t->child[0] = stmt_sequence();
995     match(UNTIL);
996     if (t!=NULL) t->child[1] = exp();
997     return t;
998   }
999
1000  TreeNode * assign_stmt(void)
1001  { TreeNode * t = newStmtNode(AssignK);
1002    if ((t!=NULL) && (token==ID))
1003      t->attr.name = copyString(tokenString);
1004    match(ID);
1005    match(ASSIGN);
1006    if (t!=NULL) t->child[0] = exp();
1007    return t;
1008  }
1009
1010  TreeNode * read_stmt(void)
1011  { TreeNode * t = newStmtNode(ReadK);
1012    match(READ);
1013    if ((t!=NULL) && (token==ID))
1014      t->attr.name = copyString(tokenString);
1015    match(ID);
1016    return t;
1017  }
1018
1019  TreeNode * write_stmt(void)
1020  { TreeNode * t = newStmtNode(WriteK);
```

```
1021    match(WRITE);
1022    if (t!=NULL) t->child[0] = exp();
1023    return t;
1024 }
1025
1026 TreeNode * exp(void)
1027 { TreeNode * t = simple_exp();
1028    if ((token==LT)||(token==EQ)) {
1029      TreeNode * p = newExpNode(OpK);
1030      if (p!=NULL) {
1031        p->child[0] = t;
1032        p->attr.op = token;
1033        t = p;
1034      }
1035      match(token);
1036      if (t!=NULL)
1037        t->child[1] = simple_exp();
1038    }
1039    return t;
1040 }
1041
1042 TreeNode * simple_exp(void)
1043 { TreeNode * t = term();
1044    while ((token==PLUS)||(token==MINUS))
1045    { TreeNode * p = newExpNode(OpK);
1046      if (p!=NULL) {
1047        p->child[0] = t;
1048        p->attr.op = token;
1049        t = p;
1050        match(token);
1051        t->child[1] = term();
1052      }
1053    }
1054    return t;
1055 }
1056
1057 TreeNode * term(void)
1058 { TreeNode * t = factor();
1059    while ((token==TIMES)||(token==OVER))
1060    { TreeNode * p = newExpNode(OpK);
1061      if (p!=NULL) {
1062        p->child[0] = t;
1063        p->attr.op = token;
1064        t = p;
1065        match(token);
1066        p->child[1] = factor();
1067      }
```

```
1068    }
1069    return t;
1070 }
1071
1072 TreeNode * factor(void)
1073 { TreeNode * t = NULL;
1074   switch (token) {
1075      case NUM :
1076        t = newExpNode(ConstK);
1077        if ((t!=NULL) && (token==NUM))
1078          t->attr.val = atoi(tokenString);
1079        match(NUM);
1080        break;
1081      case ID :
1082        t = newExpNode(IdK);
1083        if ((t!=NULL) && (token==ID))
1084          t->attr.name = copyString(tokenString);
1085        match(ID);
1086        break;
1087      case LPAREN :
1088        match(LPAREN);
1089        t = exp();
1090        match(RPAREN);
1091        break;
1092      default:
1093        syntaxError("unexpected token -> ");
1094        printToken(token,tokenString);
1095        token = getToken();
1096        break;
1097      }
1098   return t;
1099 }
1100
1101 /*****************************************************/
1102 /* la función principal del analizador sintáctico */
1103 /*****************************************************/
1104 /* La función parse devuelve el
1105  * árbol sintáctico recién construido
1106  */
1107 TreeNode * parse(void)
1108 { TreeNode * t;
1109   token = getToken();
1110   t = stmt_sequence();
1111   if (token!=ENDFILE)
1112     syntaxError("Code ends before file\n");
```

```
1113   return t;
1114 }

1150 /*****************************************************************/
1151 /* Archivo: symtab.h                                             */
1152 /* Interfaz de la tabla de símbolos para el compilador TINY      */
1153 /* (permite solamente una tabla de símbolos)                     */
1154 /* Construcción de compiladores: principios y práctica           */
1155 /* Kenneth C. Louden                                             */
1156 /*****************************************************************/
1157
1158 #ifndef _SYMTAB_H_
1159 #define _SYMTAB_H_
1160
1161 /* El procedimiento st_insert inserta números de línea y
1162  * localidades de memoria en la tabla de símbolos
1163  * loc = localidad de memoria que se inserta únicamente
1164  * la primera vez, de otro modo se ignora
1165  */
1166 void st_insert( char * name, int lineno, int loc );
1167
1168 /* La función st_lookup devuelve la localidad de
1169  * memoria de una variable o -1 si no se encuentra
1170  */
1171 int st_lookup ( char * name );
1172
1173 /* El procedimiento printSymTab imprime un listado
1174  * formateado del contenido de la tabla de símbolos
1175  * al archivo de listado
1176  */
1177 void printSymTab(FILE * listing);
1178
1179 #endif

1200 /*****************************************************************/
1201 /* Archivo: symtab.c                                             */
1202 /* Implementación de tabla de símbolos para el compilador TINY */
1203 /* (permite solamente una tabla de símbolos)                     */
1204 /* La tabla de símbolos es implementada como una                 */
1205 /* tabla de dispersión encadenada                                */
1206 /* Construcción de compiladores: principios y práctica           */
1207 /* Kenneth C. Louden                                             */
1208 /*****************************************************************/
1209
1210 #include <stdio.h>
```

```
1211 #include <stdlib.h>
1212 #include <string.h>
1213 #include "symtab.h"
1214
1215 /* SIZE es el tamaño de la tabla de dispersión */
1216 #define SIZE 211
1217
1218 /* SHIFT es la potencia de dos empleada como multiplicador
1219    en la función de dispersión  */
1220 #define SHIFT 4
1221
1222 /* la función de dispersión */
1223 static int hash ( char * key )
1224 { int temp = 0;
1225   int i = 0;
1226   while (key[i] != '\0')
1227   { temp = ((temp << SHIFT) + key[i]) % SIZE;
1228     ++i;
1229   }
1230   return temp;
1231 }
1232
1233 /* la lista de números de línea del
1234  * código fuente en el que es referenciada una variable
1235  */
1236 typedef struct LineListRec
1237    { int lineno;
1238      struct LineListRec * next;
1239    } * LineList;
1240
1241 /* El registro en las listas de cubetas para
1242  * cada variable, incluyendo nombre,
1243  * localidad de memoria asignada, y
1244  * la lista de números de línea en los que
1245  * aparece en el código fuente
1246  */
1247 typedef struct BucketListRec
1248    { char * name;
1249      LineList lines;
1250    int memloc; /* localidad de memoria para variable */
1251      struct BucketListRec * next;
1252    } * BucketList;
1253
1254 /* la tabla de dispersión */
1255 static BucketList hashTable[SIZE];
1256
1257 /* El procedimiento st_insert inserta números de línea y
```

```
1258  * localidades de memoria en la tabla de símbolos
1259  * loc = localidad de memoria que se inserta solamente la
1260  * primera vez, de otro modo se ignora
1261  */
1262 void st_insert( char * name, int lineno, int loc )
1263 { int h = hash(name);
1264   BucketList l =  hashTable[h];
1265   while ((l != NULL) && (strcmp(name,l->name) != 0))
1266     l = l->next;
1267   if (l == NULL) /* variable que todavía no está en la tabla */
1268   { l = (BucketList) malloc(sizeof(struct BucketListRec));
1269     l->name = name;
1270     l->lines = (LineList) malloc(sizeof(struct LineListRec));
1271     l->lines->lineno = lineno;
1272     l->memloc = loc;
1273     l->lines->next = NULL;
1274     l->next = hashTable[h];
1275     hashTable[h] = l; }
1276   else /* está en la tabla, de modo que sólo se agrega el número de línea*/
1277   { LineList t = l->lines;
1278     while (t->next != NULL) t = t->next;
1279     t->next = (LineList) malloc(sizeof(struct LineListRec));
1280     t->next->lineno = lineno;
1281     t->next->next = NULL;
1282   }
1283 } /* de st_insert */
1284
1285 /* La función st_lookup devuelve la localidad de
1286  * memoria de una variable o -1 si no se encuentra
1287  */
1288 int st_lookup ( char * name )
1289 { int h = hash(name);
1290   BucketList l =  hashTable[h];
1291   while ((l != NULL) && (strcmp(name,l->name) != 0))
1292     l = l->next;
1293   if (l == NULL) return -1;
1294   else return l->memloc;
1295 }
1296
1297 /* El procedimiento printSymTab imprime un listado
1298  * formateado del contenido de la tabla de símbolos
1299  * para el archivo de listado
1300  */
1301 void printSymTab(FILE * listing)
1302 { int i;
1303   fprintf(listing,"Variable Name  Location   Line Numbers\n");
1304   fprintf(listing,"------------  -------   -----------\n");
```

```
1305   for (i=0;i<SIZE;++i)
1306   { if (hashTable[i] != NULL)
1307     { BucketList l = hashTable[i];
1308       while (l != NULL)
1309       { LineList t = l->lines;
1310         fprintf(listing,"%-14s ",l->name);
1311         fprintf(listing,"%-8d  ",l->memloc);
1312         while (t != NULL)
1313         { fprintf(listing,"%4d ",t->lineno);
1314           t = t->next;
1315         }
1316         fprintf(listing,"\n");
1317         l = l->next;
1318       }
1319     }
1320   }
1321 } /* de printSymTab */
```

```
1350 /*****************************************************************/
1351 /* Archivo: analyze.h                                            */
1352 /* Interfaz del analizador semántico para el compilador TINY     */
1353 /* Construcción de compiladores: principios y práctica           */
1354 /* Kenneth C. Louden                                             */
1355 /*****************************************************************/
1356
1357 #ifndef _ANALYZE_H_
1358 #define _ANALYZE_H_
1359
1360 /* La función buildSymtab construye la tabla de
1361  * símbolos mediante recorrido preorden del árbol sintáctico
1362  */
1363 void buildSymtab(TreeNode *);
1364
1365 /* El procedimiento typeCheck realiza verificación de tipo
1366  * mediante un recorrido postorden del árbol sintáctico
1367  */
1368 void typeCheck(TreeNode *);
1369
1370 #endif
```

```
1400 /*****************************************************************/
1401 /* Archivo: analyze.c                                            */
1402 /* Implementación del analizador semántico                       */
1403 /* para el compilador TINY                                       */
1404 /* Construcción de compiladores: principios y práctica           */
```

```
1405 /* Kenneth C. Louden                                    */
1406 /*****************************************************/
1407
1408 #include "globals.h"
1409 #include "symtab.h"
1410 #include "analyze.h"
1411
1412 /* contador para localidades de memoria de variable */
1413 static int location = 0;
1414
1415 /* El procedimiento traverse es una rutina
1416  * de recorrido del árbol sintáctico recursiva genérica:
1417  * aplica preProc en preorden y postProc
1418  * en postorden para el árbol al que apunta mediante t
1419  */
1420 static void traverse( TreeNode * t,
1421                 void (* preProc) (TreeNode *),
1422                 void (* postProc) (TreeNode *) )
1423 { if (t != NULL)
1424   { preProc(t);
1425     { int i;
1426       for (i=0; i < MAXCHILDREN; i++)
1427         traverse(t->child[i],preProc,postProc);
1428     }
1429     postProc(t);
1430     traverse(t->sibling,preProc,postProc);
1431   }
1432 }
1433
1434 /* nullProc es un procedimiento que no hace nada para
1435  * generar recorridos solamente preorden o solamente postorden
1436  * a partir de traverse
1437  */
1438 static void nullProc(TreeNode * t)
1439 { if (t==NULL) return;
1440   else return;
1441 }
1442
1443 /* El procedimiento insertNode inserta
1444  * identificadores almacenados en t dentro de
1445  * la tabla de símbolos
1446  */
1447 static void insertNode( TreeNode * t)
1448 { switch (t->nodekind)
1449   { case StmtK:
1450       switch (t->kind.stmt)
1451       { case AssignK:
```

```
1452          case ReadK:
1453 /* todavía no está en la tabla, de modo que se trata como nueva definición */
1454            if (st_lookup(t->attr.name) == -1)
1455              st_insert(t->attr.name,t->lineno,location++);
1456            else
1457            /* ya está en la tabla, así que se ignora la ubicación,
1458               se agrega el número de línea de uso solamente */
1459              st_insert(t->attr.name,t->lineno,0);
1460            break;
1461          default:
1462            break;
1463        }
1464        break;
1465      case ExpK:
1466        switch (t->kind.exp)
1467        { case IdK:
1468 /* todavía no está en la tabla, de modo que se trata como nueva definición */
1469            if (st_lookup(t->attr.name) == -1)
1470              st_insert(t->attr.name,t->lineno,location++);
1471            else
1472            /* ya está en la tabla, así que se ignora la ubicación,
1473               se agrega el número de línea de uso solamente */
1474              st_insert(t->attr.name,t->lineno,0);
1475            break;
1476          default:
1477            break;
1478        }
1479        break;
1480      default:
1481        break;
1482    }
1483 }
1484
1485 /* La función buildSymtab construye la tabla de símbolos
1486  * mediante recorrido preorden del árbol sintáctico
1487  */
1488 void buildSymtab(TreeNode * syntaxTree)
1489 { traverse(syntaxTree,insertNode,nullProc);
1490   if (TraceAnalyze)
1491   { fprintf(listing,"\nSymbol table:\n\n");
1492     printSymTab(listing);
1493   }
1494 }
1495
1496 static void typeError(TreeNode * t, char * message)
1497 { fprintf(listing,"Type error at line %d: %s\n",t->lineno,message);
1498   Error = TRUE;
```

```
1499 }
1500
1501 /* El procedimiento checkNode efectúa
1502  * verificación de tipo para un solo nodo de árbol
1503  */
1504 static void checkNode(TreeNode * t)
1505 { switch (t->nodekind)
1506   { case ExpK:
1507       switch (t->kind.exp)
1508       { case OpK:
1509           if ((t->child[0]->type != Integer) ||
1510               (t->child[1]->type != Integer))
1511             typeError(t,"Op applied to non-integer");
1512           if ((t->attr.op == EQ) || (t->attr.op == LT))
1513             t->type = Boolean;
1514           else
1515             t->type = Integer;
1516           break;
1517         case ConstK:
1518         case IdK:
1519           t->type = Integer;
1520           break;
1521         default:
1522           break;
1523       }
1524       break;
1525     case StmtK:
1526       switch (t->kind.stmt)
1527       { case IfK:
1528           if (t->child[0]->type == Integer)
1529             typeError(t->child[0],"if test is not Boolean");
1530           break;
1531         case AssignK:
1532           if (t->child[0]->type != Integer)
1533             typeError(t->child[0],"assignment of non-integer value");
1534           break;
1535         case WriteK:
1536           if (t->child[0]->type != Integer)
1537             typeError(t->child[0],"write of non-integer value");
1538           break;
1539         case RepeatK:
1540           if (t->child[1]->type == Integer)
1541             typeError(t->child[1],"repeat test is not Boolean");
1542           break;
1543         default:
1544           break;
1545       }
```

```
1546        break;
1547      default:
1548        break;
1549
1550    }
1551  }
1552
1553  /* El procedimiento typeCheck realiza verificación de tipo
1554   * mediante un recorrido postorden de árbol sintáctico
1555   */
1556  void typeCheck(TreeNode * syntaxTree)
1557  { traverse(syntaxTree,nullProc,checkNode);
1558  }


1600  /************************************************************/
1601  /* Archivo: code.h                                          */
1602  /* Utilidades de emisión de código para el compilador TINY */
1603  /* e interfaz para la máquina TINY                          */
1604  /* Construcción de compiladores: principios y práctica      */
1605  /* Kenneth C. Louden                                        */
1606  /************************************************************/
1607
1608  #ifndef _CODE_H_
1609  #define _CODE_H_
1610
1611  /* pc = contador de programa */
1612  #define  pc 7
1613
1614  /* mp = el "apuntador de memoria" apunta
1615   * a la parte superior de la memoria (para almacenamiento temporal)
1616   */
1617  #define  mp 6
1618
1619  /* gp = el "apuntador global" apunta
1620   * a la parte inferior de la memoria para
1621   * almacenamiento de variable (global)
1622   */
1623  #define gp 5
1624
1625  /* acumulador */
1626  #define  ac 0
1627
1628  /* segundo acumulador */
1629  #define  ac1 1
1630
1631  /* utilidades de emisión de código */
```

```
1632
1633 /* El procedimiento emitRO emite una
1634  * instrucción TM sólo de registro
1635  * op = el opcode
1636  * r = registro objetivo
1637  * s = 1er. registro fuente
1638  * t = 2do. registro fuente
1639  * c = un comentario para ser impreso si TraceCode es TRUE
1640  */
1641 void emitRO( char *op, int r, int s, int t, char *c);
1642
1643 /* El procedimiento emitRM emite una instrucción TM
1644  * de registro-a-memoria
1645  * op = el opcode
1646  * r = registro objetivo
1647  * d = el desplazamiento
1648  * s = el registro base
1649  * c = un comentario para imprimirse si TraceCode es TRUE
1650  */
1651 void emitRM( char * op, int r, int d, int s, char *c);
1652
1653 /* La función emitSkip salta las localidades de código "howMany"
1654  * para reajuste posterior. También
1655  * devuelve la posición del código actual
1656  */
1657 int emitSkip( int howMany);
1658
1659 /* El procedimiento emitBackup respalda a
1660  * loc = una localidad previamente saltada
1661  */
1662 void emitBackup( int loc);
1663
1664 /* El procedimiento emitRestore restablece la posición
1665  * del código actual a la más alta
1666  * posición no emitida previamente
1667  */
1668 void emitRestore(void);
1669
1670 /* El procedimiento emitComment imprime una línea de comentario
1671  * con el comentario c en el archivo de código
1672  */
1673 void emitComment( char * c );
1674
1675 /* El procedimiento emitRM_Abs convierte una referencia absoluta
1676  * en una referencia relativa al pc cuando se emite una
1677  * instrucción TM de registro a memoria
1678  * op = el opcode (código operacional)
```

```
1679  * r = registro objetivo
1680  * a = la localidad absoluta en memoria
1681  * c = un comentario para imprimirse si TraceCode es TRUE
1682  */
1683 void emitRM_Abs( char *op, int r, int a, char * c);
1684
1685 #endif


1700 /*********************************************************/
1701 /* Archivo: code.c                                       */
1702 /* Implementación de utilidades de emisión de código TM */
1703 /* para el compilador TINY                               */
1704 /* Construcción de compiladores: principios y práctica   */
1705 /* Kenneth C. Louden                                     */
1706 /*********************************************************/
1707
1708 #include "globals.h"
1709 #include "code.h"
1710
1711 /* Número de localidad TM para la emisión de la instrucción actual */
1712 static int emitLoc = 0 ;
1713
1714 /* Localidad TM más alta emitida hasta ahora
1715    Para su uso en conjunto con emitSkip,
1716    emitBackup y emitRestore */
1717 static int highEmitLoc = 0;
1718
1719 /* El procedimiento emitComment imprime una línea de comentario
1720  * con comentario c en el archivo de código
1721  */
1722 void emitComment( char * c )
1723 { if (TraceCode) fprintf(code,"* %s\n",c);}
1724
1725 /* El procedimiento emitRO emite una
1726  * instrucción TM sólo de registro
1727  * op = el opcode
1728  * r = registro objetivo
1729  * s = 1er. registro fuente
1730  * t = 2do. registro fuente
1731  * c = un comentario para imprimirse si TraceCode es TRUE
1732  */
1733 void emitRO( char *op, int r, int s, int t, char *c)
1734 { fprintf(code,"%3d:  %5s  %d,%d,%d ",emitLoc++,op,r,s,t);
1735   if (TraceCode) fprintf(code,"\t%s",c) ;
1736   fprintf(code,"\n") ;
1737   if (highEmitLoc < emitLoc) highEmitLoc = emitLoc ;
```

```
1738 } /* de emitRO */
1739
1740 /* Procedimiento emitRM emite una instrucción TM
1741  * de registro-a-memoria
1742  * op = el opcode
1743  • r = registro objetivo
1744  * d = el desplazamiento
1745  * s = registro base
1746  * c = un comentario para imprimirse si TraceCode es TRUE
1747  */
1748 void emitRM( char * op, int r, int d, int s, char *c)
1749 { fprintf(code,"%3d:  %5s  %d,%d(%d) ",emitLoc++,op,r,d,s);
1750   if (TraceCode) fprintf(code,"\t%s",c) ;
1751   fprintf(code,"\n") ;
1752   if (highEmitLoc < emitLoc)  highEmitLoc = emitLoc ;
1753 } /* de emitRM */
1754
1755 /* la función emitSkip salta las localidades del código "howMany"
1756  * para un reajuste posterior. También
1757  * devuelve la posición del código actual
1758  */
1759 int emitSkip( int howMany)
1760 {  int i = emitLoc;
1761    emitLoc += howMany ;
1762    if (highEmitLoc < emitLoc)  highEmitLoc = emitLoc ;
1763    return i;
1764 } /* de emitSkip */
1765
1766 /* El procedimiento emitBackup respalda a
1767  * loc = una localidad previamente saltada
1768  */
1769 void emitBackup( int loc)
1770 { if (loc > highEmitLoc) emitComment("BUG in emitBackup");
1771   emitLoc = loc ;
1772 } /* de emitBackup */
1773
1774 /* El procedimiento emitRestore restablece la
1775  * posición de código actual a la posición más alta
1776  * no emitida previamente
1777  */
1778 void emitRestore(void)
1779 { emitLoc = highEmitLoc;}
1780
1781 /* El procedimiento emitRM_Abs convierte una referencia absoluta
1782  * a una referencia relativa al pc cuando se emite una
1783  * instrucción TM de registro-a-memoria
1784  * op = el opcode
```

```
1785  * r = registro objetivo
1786  * a = la localidad absoluta en memoria
1787  * c = un comentario para imprimirse si TraceCode es TRUE
1788  */
1789 void emitRM_Abs( char *op, int r, int a, char * c)
1790 { fprintf(code,"%3d:  %5s  %d,%d(%d) ",
1791                emitLoc,op,r,a-(emitLoc+1),pc);
1792   ++emitLoc ;
1793   if (TraceCode) fprintf(code,"\t%s",c) ;
1794   fprintf(code,"\n") ;
1795   if (highEmitLoc < emitLoc) highEmitLoc = emitLoc ;
1796 } /* de emitRM_Abs */
```

```
1850 /*****************************************************************/
1851 /* Archivo: cgen.h                                               */
1852 /* La interfaz del generador de código para el compilador TINY */
1853 /* Construcción de compiladores: principios y práctica           */
1854 /* Kenneth C. Louden                                             */
1855 /*****************************************************************/
1856
1857 #ifndef _CGEN_H_
1858 #define _CGEN_H_
1859
1860 /* El procedimiento codeGen genera código hacia un
1861  * archivo de código por recorrido del árbol sintáctico. El
1862  * segundo parámetro es el nombre de archivo
1863  * del archivo de código, y se utiliza para imprimir el
1864  * nombre de archivo como un comentario en el archivo de código
1865  */
1866 void codeGen(TreeNode * syntaxTree, char * codefile);
1867
1868 #endif
```

```
1900 /*****************************************************************/
1901 /* Archivo: cgen.c                                               */
1902 /* La implementación del generador de código                     */
1903 /* para el compilador TINY                                       */
1904 /* (genera código para la máquina TM)                            */
1905 /* Construcción de compiladores: principios y práctica           */
1906 /* Kenneth C. Louden                                             */
1907 /*****************************************************************/
1908
1909 #include "globals.h"
1910 #include "symtab.h"
1911 #include "code.h"
```

```
1912 #include "cgen.h"
1913
1914 /* tmpOffset es el desplazamiento de memoria para elementos temporales
1915    Se decrementa cada vez que un elemento temporal es
1916    almacenado, y se incrementa cuando se carga de nuevo
1917 */
1918 static int tmpOffset = 0;
1919
1920 /* prototipo para generador de código recursivo interno */
1921 static void cGen (TreeNode * tree);
1922
1923 /* El procedimiento genStmt genera código para un nodo de sentencia */
1924 static void genStmt( TreeNode * tree)
1925 { TreeNode * p1, * p2, * p3;
1926   int savedLoc1,savedLoc2,currentLoc;
1927   int loc;
1928   switch (tree->kind.stmt) {
1929
1930       case IfK :
1931          if (TraceCode) emitComment("-> if") ;
1932          p1 = tree->child[0] ;
1933          p2 = tree->child[1] ;
1934          p3 = tree->child[2] ;
1935          /* genera código para expresión de prueba */
1936          cGen(p1);
1937          savedLoc1 = emitSkip(1) ;
1938          emitComment("if: jump to else belongs here");
1939          /* recursividad en la parte then */
1940          cGen(p2);
1941          savedLoc2 = emitSkip(1) ;
1942          emitComment("if: jump to end belongs here");
1943          currentLoc = emitSkip(0) ;
1944          emitBackup(savedLoc1) ;
1945          emitRM_Abs("JEQ",ac,currentLoc,"if: jmp to else");
1946          emitRestore() ;
1947          /* recursividad en la parte else */
1948          cGen(p3);
1949          currentLoc = emitSkip(0) ;
1950          emitBackup(savedLoc2) ;
1951          emitRM_Abs("LDA",pc,currentLoc,"jmp to end") ;
1952          emitRestore() ;
1953          if (TraceCode)  emitComment("<- if") ;
1954          break; /* if_k */
1955
1956       case RepeatK:
1957          if (TraceCode) emitComment("-> repeat") ;
1958          p1 = tree->child[0] ;
```

```
1959            p2 = tree->child[1] ;
1960            savedLoc1 = emitSkip(0);
1961            emitComment("repeat: jump after body comes back here");
1962            /* genera código para el cuerpo */
1963            cGen(p1);
1964            /* genera código para prueba */
1965            cGen(p2);
1966            emitRM_Abs("JEQ",ac,savedLoc1,"repeat: jmp back to body");
1967            if (TraceCode)  emitComment("<- repeat") ;
1968            break; /* del repeat */
1969
1970         case AssignK:
1971            if (TraceCode) emitComment("-> assign") ;
1972            /* genera código para rhs */
1973            cGen(tree->child[0]);
1974            /* ahora almacena valor */
1975            loc = st_lookup(tree->attr.name);
1976            emitRM("ST",ac,loc,gp,"assign: store value");
1977            if (TraceCode)  emitComment("<- assign") ;
1978            break; */ de assign_k */
1979
1980         case ReadK:
1981            emitRO("IN",ac,0,0,"read integer value");
1982            loc = st_lookup(tree->attr.name);
1983            emitRM("ST",ac,loc,gp,"read: store value");
1984            break;
1985         case WriteK:
1986            /* genera código para la expresión a escribir */
1987            cGen(tree->child[0]);
1988            /* ahora la extrae */
1989            emitRO("OUT",ac,0,0,"write ac");
1990            break;
1991         default:
1992            break;
1993      }
1994 } /* de genStmt */
1995
1996 /* El procedimiento genExp genera código en un nodo de expresión */
1997 static void genExp( TreeNode * tree)
1998 { int loc;
1999   TreeNode * p1, * p2;
2000   switch (tree->kind.exp) {
2001
2002      case ConstK :
2003         if (TraceCode) emitComment("-> Const") ;
2004         /* genera código para cargar constante entera utilizando LDC */
2005         emitRM("LDC",ac,tree->attr.val,0,"load const");
```

```
2006        if (TraceCode)  emitComment("<- Const") ;
2007        break; /* de ConstK */
2008
2009      case IdK :
2010        if (TraceCode) emitComment("-> Id") ;
2011        loc = st_lookup(tree->attr.name);
2012        emitRM("LD",ac,loc,gp,"load id value");
2013        if (TraceCode)  emitComment("<- Id") ;
2014        break; /* de IdK */
2015
2016      case OpK :
2017        if (TraceCode) emitComment("-> Op") ;
2018        p1 = tree->child[0];
2019        p2 = tree->child[1];
2020        /* genera código para ac = argumento izquierdo */
2021        cGen(p1);
2022        /* genera código para insertar operando izquierdo */
2023        emitRM("ST",ac,tmpOffset--,mp,"op: push left");
2024        /* genera código para ac = operando derecho */
2025        cGen(p2);
2026        /* ahora carga el operando izquierdo */
2027        emitRM("LD",ac1,++tmpOffset,mp,"op: load left");
2028        switch (tree->attr.op) {
2029           case PLUS :
2030              emitRO("ADD",ac,ac1,ac,"op +");
2031              break;
2032           case MINUS :
2033              emitRO("SUB",ac,ac1,ac,"op -");
2034              break;
2035           case TIMES :
2036              emitRO("MUL",ac,ac1,ac,"op *");
2037              break;
2038           case OVER :
2039              emitRO("DIV",ac,ac1,ac,"op /");
2040              break;
2041           case LT :
2042              emitRO("SUB",ac,ac1,ac,"op <") ;
2043              emitRM("JLT",ac,2,pc,"br if true") ;
2044              emitRM("LDC",ac,0,ac,"false case") ;
2045              emitRM("LDA",pc,1,pc,"unconditional jmp") ;
2046              emitRM("LDC",ac,1,ac,"true case") ;
2047              break;
2048           case EQ :
2049              emitRO("SUB",ac,ac1,ac,"op ==") ;
2050              emitRM("JEQ",ac,2,pc,"br if true");
2051              emitRM("LDC",ac,0,ac,"false case") ;
2052              emitRM("LDA",pc,1,pc,"unconditional jmp") ;
```

```
2053            emitRM("LDC",ac,1,ac,"true case") ;
2054            break;
2055         default:
2056            emitComment("BUG: Unknown operator");
2057            break;
2058       } /* de case op */
2059       if (TraceCode)  emitComment("<- Op") ;
2060       break; /* de OpK */
2061
2062     default:
2063       break;
2064   }
2065 } /* de genExp */
2066
2067 /* El procedimiento cGen genera código recursivamente mediante
2068  * el recorrido del árbol
2069  */
2070 static void cGen( TreeNode * tree)
2071 { if (tree != NULL)
2072   { switch (tree->nodekind) {
2073       case StmtK:
2074         genStmt(tree);
2075         break;
2076       case ExpK:
2077         genExp(tree);
2078         break;
2079       default:
2080         break;
2081   }
2082   cGen(tree->sibling);
2083  }
2084 }
2085
2086 /**************************************************/
2087 /* la función principal del generador de código */
2088 /**************************************************/
2089 /* El procedimiento codeGen genera código a un archivo
2090  * de código mediante recorrido del árbol sintáctico. El
2091  * segundo parámetro (codefile) es el nombre de archivo
2092  * que contiene el código, y se emplea para imprimir el
2093  * nombre del archivo como un comentario en el archivo de código
2094  */
2095 void codeGen(TreeNode * syntaxTree, char * codefile)
2096 {  char * s = malloc(strlen(codefile)+7);
2097    strcpy(s,"File: ");
2098    strcat(s,codefile);
2099    emitComment("TINY Compilation to TM Code");
```

```
2100     emitComment(s);
2101     /* genera preludio estándar */
2102     emitComment("Standard prelude:");
2103     emitRM("LD",mp,0,ac,"load maxaddress from location 0");
2104     emitRM("ST",ac,0,ac,"clear location 0");
2105     emitComment("End of standard prelude.");
2106     /* genera código para el programa TINY */
2107     cGen(syntaxTree);
2108     /* final */
2109     emitComment("End of execution.");
2110     emitRO("HALT",0,0,0,"");
2111 }


3000 /*********************************************************/
3001 /* Archivo: tiny.l                                       */
3002 /* Especificación Lex para TINY                          */
3003 /* Construcción de compiladores: principios y práctica */
3004 /* Kenneth C. Louden                                     */
3005 /*********************************************************/
3006
3007 %{
3008 #include "globals.h"
3009 #include "util.h"
3010 #include "scan.h"
3011 /* lexema del identificador o palabra reservada */
3012 char tokenString[MAXTOKENLEN+1];
3013 %}
3014
3015 digit          [0-9]
3016 number         {digit}+
3017 letter         [a-zA-Z]
3018 identifier     {letter}+
3019 newline        \n
3020 whitespace     [ \t]+
3021
3022 %%
3023
3024 "if"           {return IF;}
3025 "then"         {return THEN;}
3026 "else"         {return ELSE;}
3027 "end"          {return END;}
3028 "repeat"       {return REPEAT;}
3029 "until"        {return UNTIL;}
3030 "read"         {return READ;}
3031 "write"        {return WRITE;}
3032 ":="           {return ASSIGN;}
```

```
3033 "="              {return EQ;}
3034 "<"              {return LT;}
3035 "+"              {return PLUS;}
3036 "-"              {return MINUS;}
3037 "*"              {return TIMES;}
3038 "/"              {return OVER;}
3039 "("              {return LPAREN;}
3040 ")"              {return RPAREN;}
3041 ";"              {return SEMI;}
3042 {number}         {return NUM;}
3043 {identifier}     {return ID;}
3044 {newline}        {lineno++;}
3045 {whitespace}     {/* salta espacio en blanco */}
3046 "{"              { char c;
3047                    do
3048                    { c = input();
3049                      if (c == '\n') lineno++;
3050                    } while (c != '}');
3051                  }
3052 .                {return ERROR;}
3053
3054 %%
3055
3056 TokenType getToken(void)
3057 { static int firstTime = TRUE;
3058   TokenType currentToken;
3059   if (firstTime)
3060   { firstTime = FALSE;
3061     lineno++;
3062     yyin = source;
3063     yyout = listing;
3064   }
3065   currentToken = yylex();
3066   strncpy(tokenString,yytext,MAXTOKENLEN);
3067   if (TraceScan) {
3068     fprintf(listing,"\t%d: ",lineno);
3069     printToken(currentToken,tokenString);
3070   }
3071   return currentToken;
3072 }
```

```
4000 /*****************************************************/
4001 /* Archivo: tiny.y                                   */
4002 /* El archivo de especificación Yacc/Bison de TINY   */
4003 /* Construcción de compiladores: principios y práctica */
4004 /* Kenneth C. Louden                                 */
```

```
4005 /*******************************************************/
4006 %{
4007 #define YYPARSER /* distingue la salida Yacc de otros archivos de código */
4008
4009 #include "globals.h"
4010 #include "util.h"
4011 #include "scan.h"
4012 #include "parse.h"
4013
4014 #define YYSTYPE TreeNode *
4015 static char * savedName; /* para su uso en asignaciones */
4016 static int savedLineNo;  /* ídem */
4017 static TreeNode * savedTree; /* almacena el árbol sintáctico para un retorno
     posterior */
4018
4019 %}
4020
4021 %token IF THEN ELSE END REPEAT UNTIL READ WRITE
4022 %token ID NUM
4023 %token ASSIGN EQ LT PLUS MINUS TIMES OVER LPAREN RPAREN SEMI
4024 %token ERROR
4025
4026 %% /* Gramática para TINY */
4027
4028 program      : stmt_seq
4029                   { savedTree = $1;}
4030               ;
4031 stmt_seq      : stmt_seq SEMI stmt
4032                   { YYSTYPE t = $1;
4033                     if (t != NULL)
4034                     { while (t->sibling != NULL)
4035                          t = t->sibling;
4036                       t->sibling = $3;
4037                       $$ = $1; }
4038                       else $$ = $3;
4039                   }
4040               | stmt  { $$ = $1; }
4041               ;
4042 stmt          : if_stmt { $$ = $1; }
4043               | repeat_stmt { $$ = $1; }
4044               | assign_stmt { $$ = $1; }
4045               | read_stmt { $$ = $1; }
4046               | write_stmt { $$ = $1; }
4047               | error  { $$ = NULL; }
4048               ;
4049 if_stmt       : IF exp THEN stmt_seq END
4050                   { $$ = newStmtNode(IfK);
4051                     $$->child[0] = $2;
```

```
4052                    $$->child[1] = $4;
4053                  }
4054            | IF exp THEN stmt_seq ELSE stmt_seq END
4055                  { $$ = newStmtNode(IfK);
4056                    $$->child[0] = $2;
4057                    $$->child[1] = $4;
4058                    $$->child[2] = $6;
4059                  }
4060            ;
4061 repeat_stmt  : REPEAT stmt_seq UNTIL exp
4062                  { $$ = newStmtNode(RepeatK);
4063                    $$->child[0] = $2;
4064                    $$->child[1] = $4;
4065                  }
4066            ;
4067 assign_stmt  : ID { savedName = copyString(tokenString);
4068                     savedLineNo = lineno; }
4069                ASSIGN exp
4070                  { $$ = newStmtNode(AssignK);
4071                    $$->child[0] = $4;
4072                    $$->attr.name = savedName;
4073                    $$->lineno = savedLineNo;
4074                  }
4075            ;
4076 read_stmt    : READ ID
4077                  { $$ = newStmtNode(ReadK);
4078                    $$->attr.name =
4079                      copyString(tokenString);
4080                  }
4081            ;
4082 write_stmt   : WRITE exp
4083                  { $$ = newStmtNode(WriteK);
4084                    $$->child[0] = $2;
4085                  }
4086            ;
4087 exp          : simple_exp LT simple_exp
4088                  { $$ = newExpNode(OpK);
4089                    $$->child[0] = $1;
4090                    $$->child[1] = $3;
4091                    $$->attr.op = LT;
4092                  }
4093            | simple_exp EQ simple_exp
4094                  { $$ = newExpNode(OpK);
4095                    $$->child[0] = $1;
4096                    $$->child[1] = $3;
4097                    $$->attr.op = EQ;
4098                  }
```

```
4099                | simple_exp { $$ = $1; }
4100                ;
4101 simple_exp     : simple_exp PLUS term
4102                    { $$ = newExpNode(OpK);
4103                      $$->child[0] = $1;
4104                      $$->child[1] = $3;
4105                      $$->attr.op = PLUS;
4106                    }
4107                | simple_exp MINUS term
4108                    { $$ = newExpNode(OpK);
4109                      $$->child[0] = $1;
4110                      $$->child[1] = $3;
4111                      $$->attr.op = MINUS;
4112                    }
4113                | term { $$ = $1; }
4114                ;
4115 term           : term TIMES factor
4116                    { $$ = newExpNode(OpK);
4117                      $$->child[0] = $1;
4118                      $$->child[1] = $3;
4119                      $$->attr.op = TIMES;
4120                    }
4121                | term OVER factor
4122                    { $$ = newExpNode(OpK);
4123                      $$->child[0] = $1;
4124                      $$->child[1] = $3;
4125                      $$->attr.op = OVER;
4126                    }
4127                | factor { $$ = $1; }
4128                ;
4129 factor         : LPAREN exp RPAREN
4130                    { $$ = $2; }
4131                | NUM
4132                    { $$ = newExpNode(ConstK);
4133                      $$->attr.val = atoi(tokenString);
4134                    }
4135                | ID { $$ = newExpNode(IdK);
4136                      $$->attr.name =
4137                            copyString(tokenString);
4138                    }
4139                | error { $$ = NULL; }
4140                ;
4141
4142 %%
4143
4144 int yyerror(char * message)
4145 { fprintf(listing,"Syntax error at line %d: %s\n",lineno,message);
```

```
4146   fprintf(listing,"Current token: ");
4147   printToken(yychar,tokenString);
4148   Error = TRUE;
4149   return 0;
4150 }
4151
4152 /* yylex llama a getToken para hacer la salida Yacc/Bison
4153  * compatible con versiones más antiguas del
4154  * analizador léxico TINY
4155  */
4156 static int yylex(void)
4157 { return getToken(); }
4158
4159 TreeNode * parse(void)
4160 { yyparse();
4161   return savedTree;
4162 }
```

```
4200 /*************************************************************/
4201 /* Archivo: globals.h                                        */
4202 /* Versión Yacc/Bison                                        */
4203 /* Las variables y tipos globales para el compilador TINY */
4204 /* deben venir antes de otros archivos "include"             */
4205 /* Construcción de compiladores: principios y práctica       */
4206 /* Kenneth C. Louden                                         */
4207 /*************************************************************/
4208
4209 #ifndef _GLOBALS_H_
4210 #define _GLOBALS_H_
4211
4212 #include <stdio.h>
4213 #include <stdlib.h>
4214 #include <ctype.h>
4215 #include <string.h>
4216
4217 /* Yacc/Bison genera internamente sus propios valores
4218  * para los tokens. Otros archivos pueden tener acceso a estos valores
4219  * incluyendo el archivo tab.h generado utilizando la
4220  * opción -d de Yacc/Bison ("generar encabezado")
4221  *
4222  * La marca o bandera YYPARSER previene la inclusión del tab.h
4223  * en la salida Yacc/Bison misma
4224  */
4225
4226 #ifndef YYPARSER
4227
```

```
4228 /* el nombre del archivo siguiente puede cambiar */
4229 #include "y.tab.h"
4230
4231 /* ENDFILE está definido implícitamente por medio de Yacc/Bison,
4232  * y no se incluye en el archivo tab.h
4233  */
4234 #define ENDFILE 0
4235
4236 #endif
4237
4238 #ifndef FALSE
4239 #define FALSE 0
4240 #endif
4241
4242 #ifndef TRUE
4243 #define TRUE 1
4244 #endif
4245
4246 /* MAXRESERVED = el número de palabras reservadas */
4247 #define MAXRESERVED 8
4248
4249 /* Yacc/Bison genera sus propios valores enteros
4250  * para tokens
4251  */
4252 typedef int TokenType;
4253
4254 extern FILE* source; /* archivo de texto del código fuente */
4255 extern FILE* listing; /* archivo de texto de salida del listado */
4256 extern FILE* code; /* archivo de texto del código para el simulador TM */
4257
4258 extern int lineno; /* número de línea fuente para el listado */
4259
4260 /*****************************************************/
4261 /** Árbol sintáctico para el análisis sintáctico **/
4262 /*****************************************************/
4263
4264 typedef enum {StmtK,ExpK} NodeKind;
4265 typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK} StmtKind;
4266 typedef enum {OpK,ConstK,IdK} ExpKind;
4267
4268 /* ExpType se utiliza para verificación de tipo */
4269 typedef enum {Void,Integer,Boolean} ExpType;
4270
4271 #define MAXCHILDREN 3
4272
4273 typedef struct treeNode
4274    { struct treeNode * child[MAXCHILDREN];
```

```
4275      struct treeNode * sibling;
4276      int lineno;
4277      NodeKind nodekind;
4278      union { StmtKind stmt; ExpKind exp;} kind;
4279      union { TokenType op;
4280              int val;
4281              char * name; } attr;
4282      ExpType type; /* para verificación de tipo de expresiones */
4283    } TreeNode;
4284
4285 /**************************************************/
4286 /************   Marcas para rastreo   ************/
4287 /**************************************************/
4288
4289 /* EchoSource = TRUE causa que el programa fuente sea
4290  * repetido en el archivo de listado con números de línea
4291  * durante el análisis sintáctico
4292  */
4293 extern int EchoSource;
4294
4295 /* TraceScan = TRUE provoca que la información del token sea
4296  * impresa en el archivo de listado a medida que cada token es
4297  * reconocido por el analizador léxico
4298  */
4299 extern int TraceScan;
4300
4301 /* TraceParse = TRUE provoca que el árbol sintáctico sea
4302  * impreso al archivo de listado en forma lineal
4303  * (utilizando sangrías para los hijos)
4304  */
4305 extern int TraceParse;
4306
4307 /* TraceAnalyze = TRUE provoca que la tabla de símbolos inserte
4308  * y busque para informar al archivo del listado
4309  */
4310 extern int TraceAnalyze;
4311
4312 /* TraceCode = TRUE causa que los comentarios sean escritos
4313  * al archivo de código TM a medida que se genera el código
4314  */
4315 extern int TraceCode;
4316
4317 /* Error = TRUE evita pasos adicionales si se presenta un error */
4318 extern int Error;
4319 #endif
4320
```

Apéndice C

Listado del simulador de la máquina TINY

```
/*********************************************************/
/* Archivo: tm.c                                         */
/* La computadora TM ("Tiny Machine")                    */
/* Construcción de compiladores: principios y práctica */
/* Kenneth C. Louden                                     */
/*********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#ifndef TRUE
#define TRUE 1
#endif
#ifndef FALSE
#define FALSE 0
#endif

/******* const *******/
#define    IADDR_SIZE  1024 /* incremente para programas grandes */
#define    DADDR_SIZE  1024 /* incremente para programas grandes */
#define    NO_REGS 8
#define    PC_REG 7

#define    LINESIZE  121
#define    WORDSIZE  20

/******* type *******/

typedef enum {
   opclRR,      /* reg operandos r,s,t */
   opclRM,      /* reg r, mem d+s */
   opclRA       /* reg r, int d+s */
   } OPCLASS;
```

```
typedef enum {
   /* Instrucciones RR */
   opHALT,     /* RR     alto, los operandos deben ser cero */
   opIN,       /* RR     lee en reg(r); s y t son ignorados/
   opOUT,      /* RR     escribe de reg(r); s y t son ignorados/
   opADD,     /* RR     reg(r) = reg(s)+reg(t) */
   opSUB,     /* RR     reg(r) = reg(s)-reg(t) */
   opMUL,     /* RR     reg(r) = reg(s)*reg(t) */
   opDIV,     /* RR     reg(r) = reg(s)/reg(t) */
   opRRLim,    /* límite de opcodes RR */

   /* Instrucciones RM */
   opLD,       /* RM     reg(r) = mem(d+reg(s)) */
   opST,       /* RM     mem(d+reg(s)) = reg(r) */
   opRMLim,    /* Límite de opcodes RM */

   /* Instrucciones RA */
   opLDA,      /* RA     reg(r) = d+reg(s) */
   opLDC,      /* RA     reg(r) = d ; reg(s) es ignorado */
   opJLT,      /* RA     if reg(r)<0 then reg(7) = d+reg(s) */
   opJLE,      /* RA     if reg(r)<=0 then reg(7) = d+reg(s) */
   opJGT,      /* RA     if reg(r)>0 then reg(7) = d+reg(s) */
   opJGE,      /* RA     if reg(r)>=0 then reg(7) = d+reg(s) */
   opJEQ,      /* RA     if reg(r)==0 then reg(7) = d+reg(s) */
   opJNE,      /* RA     if reg(r)!=0 then reg(7) = d+reg(s) */
   opRALim     /* Límite de opcodes RA */
   } OPCODE;

typedef enum {
   srOKAY,
   srHALT,
   srIMEM_ERR,
   srDMEM_ERR,
   srZERODIVIDE
   } STEPRESULT;

typedef struct {
      int iop  ;
      int iarg1  ;
      int iarg2  ;
      int iarg3  ;
   } INSTRUCTION;
```

```
/******** vars ********/
int iloc = 0 ;
int dloc = 0 ;
int traceflag = FALSE;
int icountflag = FALSE;

INSTRUCTION iMem [IADDR_SIZE];
int dMem [DADDR_SIZE];
int reg [NO_REGS];

char * opCodeTab[]
       = {"HALT","IN","OUT","ADD","SUB","MUL","DIV","????",
           /* RR opcodes */
          "LD","ST","????", /* RM opcodes */
          "LDA","LDC","JLT","JLE","JGT","JGE","JEQ","JNE","????"
          /* RA opcodes */
         };

char * stepResultTab[]
       = {"OK","Halted","Instruction Memory Fault",
          "Data Memory Fault","Division by 0"
         };

char pgmName[20];
FILE *pgm ;

char in_Line[LINESIZE] ;
int lineLen ;
int inCol ;
int num ;
char word[WORDSIZE] ;
char ch ;
int done ;

/*********************************************/
int opClass( int c )
{ if      ( c <= opRRLim) return ( opclRR );
  else if ( c <= opRMLim) return ( opclRM );
  else                    return ( opclRA );
} /* de opClass */

/*********************************************/
void writeInstruction ( int loc )
{ printf( "%5d: ", loc) ;
  if ( (loc >= 0) && (loc < IADDR_SIZE) )
  { printf("%6s%3d,", opCodeTab[iMem[loc].iop], iMem[loc].iarg1);
    switch ( opClass(iMem[loc].iop) )
```

```
    { case opclRR: printf("%1d,%1d", iMem[loc].iarg2, iMem[loc].iarg3);
                   break;
      case opclRM:
      case opclRA: printf("%3d(%1d)", iMem[loc].iarg2, iMem[loc].iarg3);
                   break;
    }
    printf ("\n") ;
  }
} /* de writeInstruction */

/*********************************************/
void getCh (void)
{ if (++inCol < lineLen)
  ch = in_Line[inCol] ;
  else ch = ' ' ;
} /* de getCh */

/*********************************************/
int nonBlank (void)
{ while ((inCol < lineLen)
         && (in_Line[inCol] == ' ') )
    inCol++ ;
  if (inCol < lineLen)
  { ch = in_Line[inCol] ;
    return TRUE ; }
  else
  { ch = ' ' ;
    return FALSE ; }
} /* de nonBlank */

/*********************************************/
int getNum (void)
{ int sign;
  int term;
  int temp = FALSE;
  num = 0 ;
  do
  { sign = 1;
    while ( nonBlank() && ((ch == '+') || (ch == '-')) )
    { temp = FALSE ;
      if (ch == '-') sign = - sign ;
      getCh();
    }
    term = 0 ;
    nonBlank();
    while (isdigit(ch))
```

```
      { temp = TRUE ;
        term = term * 10 + ( ch - '0' ) ;
        getCh();
      }
      num = num + (term * sign) ;
    } while ( (nonBlank()) && ((ch == '+') || (ch == '-')) ) ;
    return temp;
  } /* de getNum */

/*********************************************/
int getWord (void)
{ int temp = FALSE;
  int length = 0;
  if (nonBlank ())
  { while (isalnum(ch))
    { if (length < WORDSIZE-1) word [length++] = ch ;
      getCh() ;
    }
    word[length] = '\0';
    temp = (length != 0);
  }
  return temp;
} /* de getWord */

/*********************************************/
int skipCh ( char c )
{ int temp = FALSE;
  if ( nonBlank() && (ch == c) )
  { getCh();
    temp = TRUE;
  }
  return temp;
} /* de skipCh */

/*********************************************/
int atEOL(void)
{ return ( ! nonBlank ());
} /* de atEOL */

/*********************************************/
int error( char * msg, int lineNo, int instNo)
{ printf("Line %d",lineNo);
  if (instNo >= 0) printf(" (Instruction %d)",instNo);
  printf("   %s\n",msg);
  return FALSE;
} /* de error */
```

```
/*****************************************/
int readInstructions (void)
{ OPCODE op;
  int arg1, arg2, arg3;
  int loc, regNo, lineNo;
  for (regNo = 0 ; regNo < NO_REGS ; regNo++)
      reg[regNo] = 0 ;
  dMem[0] = DADDR_SIZE - 1 ;
  for (loc = 1 ; loc < DADDR_SIZE ; loc++)
      dMem[loc] = 0 ;
  for (loc = 0 ; loc < IADDR_SIZE ; loc++)
  { iMem[loc].iop = opHALT ;
    iMem[loc].iarg1 = 0 ;
    iMem[loc].iarg2 = 0 ;
    iMem[loc].iarg3 = 0 ;
  }
  lineNo = 0 ;
  while (! feof(pgm))
  { fgets( in_Line, LINESIZE-2, pgm ) ;
    inCol = 0 ;
    lineNo++;
    lineLen = strlen(in_Line)-1 ;
    if (in_Line[lineLen]=='\n') in_Line[lineLen] = '\0' ;
    else in_Line[++lineLen] = '\0';
    if ( (nonBlank()) && (in_Line[inCol] != '*') )
    { if (! getNum())
        return error("Bad location", lineNo,-1);
      loc = num;
      if (loc > IADDR_SIZE)
        return error("Location too large",lineNo,loc);
      if (! skipCh(':'))
        return error("Missing colon", lineNo,loc);
      if (! getWord ())
        return error("Missing opcode", lineNo,loc);
      op = opHALT ;
      while ((op < opRALim)
             && (strncmp(opCodeTab[op], word, 4) != 0) )
          op++ ;
      if (strncmp(opCodeTab[op], word, 4) != 0)
          return error("Illegal opcode", lineNo,loc);
      switch ( opClass(op) )
      { case opclRR :
        /***********************************/
        if ( (! getNum ()) || (num < 0) || (num >= NO_REGS) )
             return error("Bad first register", lineNo,loc);
        arg1 = num;
        if ( ! skipCh(','))
             return error("Missing comma", lineNo, loc);
```

```
        if ( (! getNum ()) || (num < 0) || (num >= NO_REGS) )
             return error("Bad second register", lineNo, loc);
        arg2 = num;
        if ( ! skipCh(','))
             return error("Missing comma", lineNo,loc);
        if ( (! getNum ()) || (num < 0) || (num >= NO_REGS) )
             return error("Bad third register", lineNo,loc);
        arg3 = num;
        break;

        case opclRM :
        case opclRA :
        /***********************************/
        if ( (! getNum ()) || (num < 0) || (num >= NO_REGS) )
             return error("Bad first register", lineNo,loc);
        arg1 = num;
        if ( ! skipCh(','))
             return error("Missing comma", lineNo,loc);
        if (! getNum ())
             return error("Bad displacement", lineNo,loc);
        arg2 = num;
        if ( ! skipCh('(') && ! skipCh(',') )
             return error("Missing LParen", lineNo,loc);
        if ( (! getNum ()) || (num < 0) || (num >= NO_REGS))
             return error("Bad second register", lineNo,loc);
        arg3 = num;
        break;
        }
      iMem[loc].iop = op;
      iMem[loc].iarg1 = arg1;
      iMem[loc].iarg2 = arg2;
      iMem[loc].iarg3 = arg3;
    }
  }
  return TRUE;
} /* de readInstructions */



/*******************************************/
STEPRESULT stepTM (void)
{ INSTRUCTION currentinstruction ;
  int pc  ;
  int r,s,t,m ;
  int ok ;
```

```
pc = reg[PC_REG] ;
if ( (pc < 0) || (pc > IADDR_SIZE) )
    return srIMEM_ERR ;
reg[PC_REG] = pc + 1 ;
currentinstruction = iMem[ pc ] ;
switch (opClass(currentinstruction.iop) )
{ case opclRR :
  /***********************************/
    r = currentinstruction.iarg1 ;
    s = currentinstruction.iarg2 ;
    t = currentinstruction.iarg3 ;
    break;

  case opclRM :
  /***********************************/
    r = currentinstruction.iarg1 ;
    s = currentinstruction.iarg3 ;
    m = currentinstruction.iarg2 + reg[s] ;
    if ( (m < 0) || (m > DADDR_SIZE) )
       return srDMEM_ERR ;
    break;

  case opclRA :
  /***********************************/
    r = currentinstruction.iarg1 ;
    s = currentinstruction.iarg3 ;
    m = currentinstruction.iarg2 + reg[s] ;
    break;
} /* de case */

switch ( currentinstruction.iop)
{ /* Instrucciones RR */
  case opHALT :
  /***********************************/
    printf("HALT: %1d,%1d,%1d\n",r,s,t);
    return srHALT ;
    /* break; */

  case opIN :
  /***********************************/
    do
    { printf("Enter value for IN instruction: ") ;
      fflush (stdin);
      gets(in_Line);
      inCol = 0;
      ok = getNum();
      if ( ! ok ) printf ("Illegal value\n");
      else reg[r] = num;
    }
```

```
      while (! ok);
      break;

    case opOUT :
      printf ("OUT instruction prints: %d\n", reg[r] ) ;
      break;
    case opADD :  reg[r] = reg[s] + reg[t] ;  break;
    case opSUB :  reg[r] = reg[s] - reg[t] ;  break;
    case opMUL :  reg[r] = reg[s] * reg[t] ;  break;

    case opDIV :
    /***********************************/
      if ( reg[t] != 0 ) reg[r] = reg[s] / reg[t];
      else return srZERODIVIDE ;
      break;

    /**************  Instrucciones RM  ********************/
    case opLD :     reg[r] = dMem[m] ;  break;
    case opST :     dMem[m] = reg[r] ;  break;

    /*************** Instrucciones RA *********************/
    case opLDA :     reg[r] = m ; break;
    case opLDC :     reg[r] = currentinstruction.iarg2 ;  break;
    case opJLT :     if ( reg[r] < 0 ) reg[PC_REG] = m ; break;
    case opJLE :     if ( reg[r] <= 0 ) reg[PC_REG] = m ; break;
    case opJGT :     if ( reg[r] > 0 ) reg[PC_REG] = m ; break;
    case opJGE :     if ( reg[r] >= 0 ) reg[PC_REG] = m ; break;
    case opJEQ :     if ( reg[r] == 0 ) reg[PC_REG] = m ; break;
    case opJNE :     if ( reg[r] != 0 ) reg[PC_REG] = m ; break;

    /* fin de las instrucciones legales */
  } /* de case */
  return srOKAY ;
} /* de stepTM */

/*********************************************/
int doCommand (void)
{ char cmd;
  int stepcnt=0, i;
  int printcnt;
  int stepResult;
  int regNo, loc;
  do
  { printf ("Enter command: ");
    fflush (stdin);
    gets(in_Line);
    inCol = 0;
```

```
  }
  while (! getWord ());

  cmd = word[0] ;
  switch ( cmd )
  { case 't' :
    /***********************************/
      traceflag = ! traceflag ;
      printf("Tracing now ");
      if ( traceflag ) printf("on.\n"); else printf("off.\n");
      break;

    case 'h' :
    /***********************************/
      printf("Commands are:\n");
      printf("   s(tep <n>      "\
             "Execute n (default 1) TM instructions\n");
      printf("   g(o            "\
             "Execute TM instructions until HALT\n");
      printf("   r(egs          "\
             "Print the contents of the registers\n");
      printf("   i(Mem <b <n>>  "\
             "Print n iMem locations starting at b\n");
      printf("   d(Mem <b <n>>  "\
             "Print n dMem locations starting at b\n");
      printf("   t(race         "\
             "Toggle instruction trace\n");
      printf("   p(rint         "\
             "Toggle print of total instructions executed"\
             " ('go' only)\n");
      printf("   c(lear         "\
             "Reset simulator for new execution of program\n");
      printf("   h(elp          "\
             "Cause this list of commands to be printed\n");
      printf("   q(uit          "\
             "Terminate the simulation\n");
      break;

    case 'p' :
    /***********************************/
      icountflag = ! icountflag ;
      printf("Printing instruction count now ");
      if ( icountflag ) printf("on.\n"); else printf("off.\n");
      break;

    case 's' :
    /***********************************/
      if ( atEOL ())  stepcnt = 1;
```

```
    else if ( getNum ()) stepcnt = abs(num);
    else   printf("Step count?\n");
    break;

  case 'g' :   stepcnt = 1 ;    break;

  case 'r' :
  /***********************************/
    for (i = 0; i < NO_REGS; i++)
    { printf("%1d: %4d     ", i,reg[i]);
      if ( (i % 4) == 3 ) printf ("\n");
    }
    break;

  case 'i' :
  /***********************************/
    printcnt = 1 ;
    if ( getNum ())
    { iloc = num ;
      if ( getNum ()) printcnt = num ;
    }
    if ( ! atEOL ())
      printf ("Instruction locations?\n");
    else
    { while ((iloc >= 0) && (iloc < IADDR_SIZE)
              && (printcnt > 0) )
      { writeInstruction(iloc);
        iloc++ ;
        printcnt-- ;
      }
    }
    break;

  case 'd' :
  /***********************************/
    printcnt = 1 ;
    if ( getNum ())
    { dloc = num ;
      if ( getNum ()) printcnt = num ;
    }
    if ( ! atEOL ())
      printf("Data locations?\n");
    else
    { while ((dloc >= 0) && (dloc < DADDR_SIZE)
              && (printcnt > 0))
      { printf("%5d: %5d\n",dloc,dMem[dloc]);
        dloc++;
```

```
          printcnt--;
        }
      }
      break;

    case 'c' :
    /***********************************/
      iloc = 0;
      dloc = 0;
      stepcnt = 0;
      for (regNo = 0; regNo < NO_REGS ; regNo++)
            reg[regNo] = 0 ;
      dMem[0] = DADDR_SIZE - 1 ;
      for (loc = 1 ; loc < DADDR_SIZE ; loc++)
            dMem[loc] = 0 ;
      break;

    case 'q' : return FALSE; /* break; */

    default : printf("Command %c unknown.\n", cmd); break;
  }  /* de case */
  stepResult = srOKAY;
  if ( stepcnt > 0 )
  { if ( cmd == 'g' )
    { stepcnt = 0;
      while (stepResult == srOKAY)
      { iloc = reg[PC_REG] ;
        if ( traceflag ) writeInstruction( iloc ) ;
        stepResult = stepTM ();
        stepcnt++;
      }
      if ( icountflag )
        printf("Number of instructions executed = %d\n", stepcnt);
    }
    else
    { while ((stepcnt > 0) && (stepResult == srOKAY))
      { iloc = reg[PC_REG] ;
        if ( traceflag ) writeInstruction( iloc ) ;
        stepResult = stepTM ();
        stepcnt-- ;
      }
    }
    printf( "%s\n",stepResultTab[stepResult] );
  }
  return TRUE;
} /* de doCommand */
```

```
main( int argc, char * argv[] )
{ if (argc != 2)
  { printf("usage: %s <filename>\n" ,argv[0]);
    exit(1);
  }
  strcpy(pgmName,argv[1]) ;
  if (strchr (pgmName, '.') == NULL)
     strcat(pgmName,".tm");
  pgm = fopen(pgmName,"r");
  if (pgm == NULL)
  { printf("file '%s' not found\n",pgmName);
    exit(1);
  }

  /* lee el programa */
  if ( ! readInstructions ())
         exit(1) ;
  /* archivo de entrada conmuta a terminal */
  /* reset (input); */
  /* lee-eval-imprime */
  printf("TM simulation (enter h for help)...\n");
  do
     done = ! doCommand ();
  while (! done );
  printf("Simulation done.\n");
  return 0;
}
```

Índice

Los números de páginas seguidos por una *e* indican que la página corresponde a ejercicios; los que se encuentran seguidos por una *n* corresponden a las páginas con notas a pie de página; los seguidos por *r* indican que son páginas de las secciones de notas y referencias; los seguidos por una *t* señalan páginas con tablas, y la abreviatura *ilus.* hace referencia a las páginas del texto que contienen figuras.

Acceso encadenado, 368-370, 392*e*
Acción de reducción, 198-199, 206-209
Acción Generate, en análisis sintáctico descendente, 153
Acción "por default" (predeterminada), 86
Acción Shift, 198-199, 206-209
Acciones embebidas, en código Yacc, 229, 241-242, 242*t*
Acciones Goto, 232, 233-234*ilus.*, 246-247
Acciones. *Véase también* Acción predeterminada; Acciones embebidas; Generación de acción; Acciones Goto; Instrucciones de saltos; Acción de reducción; Acción de desplazamiento
  de autómatas finitos determinísticos, 53-56
  en código Lex, 85, 90
  en código Yacc, 229-230
action *default*, en código Yacc, 234
Adición (suma), 117-118, 118-119
  ambigüedad no esencial con, 123
Administración automática de estructuras dinámicas (heap), 380-381
Administración de memoria, 345, 346-349
Administración manual de la estructura dinámica, 377-380
Administradores de proyecto, 6
Alfabetos, 34-35, 49-50, 57-58
Algoritmo de análisis sintáctico general LR(1), 221
Algoritmo de construcción de subconjuntos, 69, 70-72
Algoritmo de Euclides para el máximo común divisor, 353-354
Algoritmos
  autómatas finitos determinísticos para, 53-56
  de primer ajuste, 378
  para análisis sintáctico LALR(1), 224-225
  para calcular conjuntos Primero, 186-169, 168*ilus*, 169*ilus*.
  para calcular conjuntos Siguiente, 173-174, 174*ilus*.
  para cálculo de atributo, 270-295
  para construir autómatas finitos determinísticos, 252*e*
  para determinar equivalencia de tipos, 322-329
  para recolección de basura, 17
Algoritmos, para análisis sintáctico descendente, 197-198
Alias, 300, 328, 383
ambiente de ejecución basado en pilas para, 353-354, 354*ilus.*
Ambientes completamente dinámicos, 345. *Véase también* Ambientes de ejecución
Ambientes de desarrollo interactivos (IDE, por sus siglas en inglés), 4, 6
Ambientes de ejecución, 301, 345-388, 388-391*e*. *Véase también* Ambientes de ejecución dinámicos; Ambientes de ejecución basados en pilas; Ambientes de ejecución estáticos
  basados en pilas, 352-373
  completamente estáticos, 349-352, 352*ilus.*, 349-352, 352*ilus.*
  construcción de compiladores y, 17
  dinámicos, 373-381
  mecanismos de paso de parámetro en, 381-386, 394-395*e*
  Organización de memoria en, 346-349
  para el lenguaje TINY, 386-388
  para la máquina TM, 456
  tipos de, 345-346
Ambientes de ejecución basados en pilas, 17, 345, 352-373
  acceso a nombres en, 356-359
  con parámetros de procedimiento, 370-373
  con procedimientos locales, 365-370
  datos de longitud variable en, 361-363
  declaraciones anidadas en, 363, 364-365
  organización de, 352-353
  secuencias de llamadas de procedimiento en, 359-361
  sin procedimientos locales, 353-365
  temporales locales en, 363
Ambientes de ejecución completamente estáticos, 345, 349-352, 350*ilus.*, 352*ilus*. *Véase también* Ambientes de ejecución estáticos
Ambientes de ejecución dinámicos, 17, 373-381. *Véase también* Ambientes completamente dinámicos
Ambientes de ejecución estáticos, 17, 391*e*. *Véase también* Ambientes de ejecución completamente estáticos
Ambientes en ventanas, 4
Ambigüedad, 114-123, 141*e*, 201*n*, 253*e*. *Véase también* Problema de else ambiguo; Reglas de no ambigüedad
  ámbito dinámico y, 306
  de gramáticas con atributos, 270
  definición formal de, 129
  en análisis sintáctico LR(1), 221
  en expresiones de C, 342*e*
  en gramáticas de expresión aritmética simple, 308-309, 341*e*
  en gramáticas LL(1), 155
  en gramáticas LR(0), 207-208
  en gramáticas SLR(1), 210-211, 213-215
  en sentencias "if", 120-122
  lenguajes con, inherente, 133
  no esencial, 122-123
  resolución de, en Lex, 87, 89
  simplificación de gramáticas con atributos con, 269
Ámbito, 300, 301-308
  dinámico, 305-306, 365-366
  generación de código intermedio y, 398-399
  para declaraciones al mismo nivel de anidación, 306-308
  para declaraciones de expresión *let*, 309-310
  reglas del, 301-302, 303, 305-306, 365-366
Ámbito estático, 305-306

Ámbito léxico, 305
Ámbitos anidados, 302*ilus.*, 302-305, 303*ilus.*, 306-308, 308-313, 367
alternativas para, 304
en gramática de expresión aritmética simple, 308-313
Análisis de flujo de datos, 474
Análisis gramaticales, gramáticas ambiguas y, 116
Análisis léxico, 8, 31. *Véase también* Rastreo
Análisis, operaciones de compilador para, 15
Análisis semántico, 257-338
atributos y, 259-261
cálculo de atributos en, 270-295
categorías y naturaleza de, 257-259
de lenguaje TINY, 334-338
gramáticas con atributos y, 261-270
tablas de símbolos en, 295-313
tipos de datos y verificación de tipo en, 313-334
Análisis semántico estático, 257, 258
Análisis sintáctico, 95-97, 106-128, 343*r*. *Véase también* Análisis sintáctico ascendente; Análisis sintáctico LALR(1); Análisis sintáctico LL(1); Análisis sintáctico LR(0); Análisis sintáctico LR(1); Análisis sintáctico SLR(1); Análisis sintáctico descendente
cálculo de atributos durante, 289-293
generación de código intermedio y objeto durante, 407-410
proceso del, 96-97
Análisis sintáctico ascendente, 96, 197-250
análisis sintáctico LALR(1), 224-242
análisis sintáctico LR(0), 201-210
análisis sintáctico LR(1), 217-223
análisis sintáctico SLR(1), 210-217
descripción, 198-201
generador de analizadores sintácticos Yacc para, 226-245
para sintaxis de TINY, 243-245
recuperación de errores durante, 245-250
Análisis sintáctico descendente, 96, 143-189, 152*t*, 196*r*
análisis sintáctico LL(1), 152-180
de lenguaje TINY, 179-182
mediante recursivo descendente, 144-152
recuperación de errores durante, 183-189
Análisis sintáctico fuerte LL(*k*), 180
Análisis sintáctico LALR(1), 197, 224-226, 250-251*e*, 252*e*, 256*r*, 290
detección de errores en, 245
Yacc y, 226-242
Análisis sintáctico LL(1), 143-144, 152-180, 190-193*e*, 196*r*, 197
algoritmo y tabla para, 154-157
búsqueda hacia delante extendida para, 180
conjuntos Primero para, 168-173
conjuntos Siguiente para, 168, 173-177
construcción de árbol sintáctico en, 166-167
eliminación de recursión por la izquierda en, 157-162
factorización por la izquierda en, 157, 162-166
método básico de, 152-154
recuperación de errores en, 186-187
Análisis sintáctico LR, 289-290, 343*r*
Análisis sintáctico LR(0), 197, 201, 206-210, 245
Análisis sintáctico LR(1), 197, 224-225, 245, 256*r*
de búsqueda hacia delante. *Véase* Análisis sintáctico LALR(1)
Análisis sintáctico recursivo descendente, 143-144, 144-152, 196*r*
analizadores sintácticos LL(k) y, 180
de lenguaje TINY, 180-182
eliminación de recursión por la izquierda y, 161-162
forma Backus-Naur extendida en, 145-151
método básico del, 144-145
modificación de analizadores sintácticos para, 193-194*e*
problemas con, 151-152
recuperación de errores en, 183-185
retroseguimiento en, 192*e*
Análisis sintáctico simple LR(1). *Véase* Análisis sintáctico SLR(l)
Análisis sintáctico SLL(*k*), 180
Análisis sintáctico SLR(1), 197, 210-217, 251*e*, 252*e*, 256*r*
algoritmo para, 210-211
análisis sintáctico LALR(1) y, 224-225
análisis sintáctico SLR(*k*) y, 216-217
conflictos de no ambigüedad en, 213-215
detección de errores en, 245
ejemplos de, 211-213, 341*e*
limitaciones de, 215-216
Analizador léxico TINY
archivo de entrada de Lex para, 90-91
autómata finito determinístico para, 77*ilus.*
implementación, 32, 75-80
modificación, 93-94*e*
Analizador sintáctico TINY, 187-189, 243-245, 255-256*e*
Analizadores semánticos, 9-10
Analizadores sintácticos, 8-9. *Véase también* Árboles de análisis sintáctico
abstractos construidos mediante, 109-111
contra analizadores léxicos, 95-96
después de eliminación de recursión por la izquierda, 160-162
ejecución de rastreo de, 238-239
generados por Yacc, 238-239
manejo de errores por, 96-97
Analizadores sintácticos con retroseguimiento, 143
Analizadores sintácticos LL(*k*), 143, 180
Analizadores sintácticos predictivos, 143
Analizadores sintácticos Shift-reduce, 199, 200-201. *Véase también* Analizadores sintácticos ascendentes
ANSI (American National Standards Institute), lenguajes estándar adoptados por, 16
Antlr, generador de analizador sintáctico, 196*r*
Apuntador de argumento (ap, por sus siglas en inglés), 348-349, 361, 391*e*
Apuntador de entorno (ep por sus siglas en inglés), 371-373, 372*ilus.*, 373*ilus.*
Apuntador de instrucción (ip, por sus siglas en inglés), 371-373, 372*ilus.*, 373*ilus.*
Apuntador de marco (fp, por sus siglas en inglés), 348-349, 353, 354*ilus.*, 356*ilus.*, 357*ilus.*, 358*ilus.*, 392*e*
con desplazamientos, 357, 358, 358*ilus.*
en pilas de ejecución con vínculos de acceso, 367*ilus.*, 369*ilus.*, 370*ilus.*, 372*ilus.*, 373*ilus.*
en secuencias de llamada de procedimiento, 359-361, 359*ilus.*, 360*ilus.*, 361*ilus.*, 362*ilus.*, 363*ilus.*, 364*ilus.*, 365*ilus.*, 366*ilus.*
Apuntador de pila (sp, por sus siglas en inglés), 348, 353, 354*ilus.*, 356*ilus.*, 392*e*
Apuntador de tabla de función virtual, 376*ilus.*
Apuntador de tope de pila (tos, por sus siglas en inglés), 353
Apuntador `memptr`, 378-380, 378*ilus.*, 379*ilus.*, 380*ilus.*
Apuntadores, 322. *Véase también* Apuntadores de acceso; apuntador de argumento (ap, por sus siglas en inglés); Registro de apuntador base; Carácter caret (^); Apuntador de ambiente (ep, por sus siglas en inglés); Apuntadores remotos; Apuntador de marco (fp, por sus siglas en inglés); Apuntador de instrucción (ip, por sus siglas en inglés); Apuntadores cercanos; Desplazamientos; Apuntador de pila (sp, por sus siglas en inglés); tope del apuntador de pila (tos, por sus siglas en inglés); apuntador de tablas de función virtual

Apuntadores cercanos, 319
Apuntadores de acceso, 304
Apuntadores lejanos, 319
árbol de análisis gramatical recorrido de izquierda a derecha y, 289-290, 291-292
Árboles binarios enteros, gramática para, 339*e*
Árboles de activación, 356, 357*ilus.*
Árboles de análisis gramatical, 8-9, 9*ilus.*, 10*ilus.*, 11*ilus.*, 96, 106-109. *Véase también* Árboles de análisis sintáctico
análisis sintáctico descendente y, 153-154
árboles de análisis sintáctico abstractos y, 109-114
atributos sintetizados y, 277
definición formal de, 129
dependencias "hacia atrás" en, 289-290
después de eliminación de recursión por la izquierda, 160-161
especificación de asociatividad y precedencia en, 118-119
para expresiones aritméticas, 109*ilus.*, 272-273
para expresiones aritméticas enteras, 265*ilus.*, 265
para gramática de declaración de variables, 266*ilus.*
para gramáticas ambiguas, 114-116
para gramáticas de números con base, 273-275
para gramáticas de números sin signo, 263-264, 264*ilus.*, 267-268, 268*ilus.*, 271-272, 276-277
para gramáticas libres de contexto, 95
para sentencias "if", 120-122
para tipos de valor de punto flotante, 294*ilus.*
representación de hijo extremo izquierdo con hermanos a la derecha, 114
Árboles de búsqueda binarios, en lenguaje ML, 321-322
Árboles sintácticos, 8, 11*ilus.*, 13, 96, 193*e*, 194*e*. *Véase también* Árboles sintácticos abstractos
atributos heredados en, 280-281
como atributos sintetizados, 289
con código P, 411, 416
de Yacc, 243
en análisis sintáctico LL(1), 166-167
especificación de asociatividad y precedencia en, 118-119
linealización de, 398, 399-400
para arreglos con subíndice, 422-423, 424-425*ilus.*
para compilador TINY, 134-138, 136-138*ilus.*, 182*ilus.*, 462
para expresiones de tipo, 323-324
para gramáticas ambiguas, 114-118
para gramáticas libres de contexto, 95
recorrido postorden de, 277
recorrido preorden de, 279
recorrido recursivo de, 410-411
simplificación, 114
Árboles sintácticos abstractos, 9, 106, 109-114, 258. *Véase también* Árboles sintácticos
con llamadas y definiciones de función, 440-441, 441*ilus.*
generación de código en tres direcciones de, 399-400
generación de código intermedio de, 398-399
para gramáticas de control de sentencias, 434-436
simplificación de las gramáticas con atributos con, 269-270, 269*ilus.*, 270*t*
árboles sintácticos como, 289
Archivo `analyze.c`, 337
Archivo `analyze.h`, 336
Archivo `calc.y`, opciones de Yacc con, 230-231
archivo `code.c`, 459-460
Archivo `code.h`, 459-460
definiciones de función y constantes en, 460-461
Archivo de código fuente `tm.c`, 26
Archivo de código `sample.tm`, 25, 26, 464-465
Archivo de texto `sample.tiny`, 24
Archivo `main.c`
para el analizador léxico de TINY, 78
para el compilador de TINY, 23, 24
Archivo `parse.c`, 181
Archivo `parse.h`, 181
Archivo `scan.c`, 77
Archivo `scan.h`, 77
Archivo `symtab.e`, 336
Archivo `symtab.h`, 336
Archivo `tiny.l`, 90
Archivo `tiny.y`, 243, 250
Archivo `util.c`, 181, 182
Archivo `util.h`, 181
Archivo `y.output` para Yacc, 231-232, 233-234*ilus.*, 237*ilus.*, 238
Archivo `y.tab.c`, 226, 238-239
Archivo `y.tab.h`, opciones de Yacc con, 230-231
Archivo `yyin`, 89
Archivo `yyout`, 89
Archivos `analyze`, para compilador TINY, 23
Archivos C, para el compilador de TINY, 23
Archivos `cgen`, para el compilador de TINY, 23
archivos `code`, para el compilador TINY, 23-24
Archivos de cabecera, para el compilador TINY, 23-24
Archivos de código TM, generación de, 464-465
Archivos de entrada Lex, formato de los, 83-89
Archivos de especificación, para Yacc, 226-230
Archivos de salida. *Véase también* Archivo `y.output`.
de Yacc, 226-227
Archivos `#include`, para especificaciones Yacc, 227-228
Archivos `parse`, para el compilador TINY, 23
Archivos `scan`, 23, 77
Archivos `symtab`, para el compilador TINY, 23-24
Archivos temporales, 14
Archivos `util` para el compilador TINY, 23-24
Archivos. *Véase* archivos de C; Archivos de cabecera; Archivos `#include`; Archivos de entrada Lex; Archivos de salida; Archivos `parse`; Archivos `scan`; Archivos de especificación; Archivos `symtab`; Archivos temporales; Archivos de código de TM; Archivos `util`; Archivo `yyin`; Archivo `yyout`
Área de código, de memoria, 346-347, 350*ilus.*
Área de datos, de memoria, 346-347, 350*ilus.*
Área de pila, en la asignación de memoria, 347-348
Área de registro, 346, 348-349. *Véase también* Procesador
Argumentos. *Véase también* Parámetros en llamadas de función
y procedimiento, 437
Arquitectura Sparc RISC, 398
Arquitectura VAX, 349*n*, 361
Arranque automático, de transferencia, ("Bootstrapping"), 18-21, 20-21*ilus.*
Arreglos, 488*e*
como funciones, 319
constructores de tipos para, 315-316
direcciones base de, 418
en ambientes de ejecución basados en pilas, 357-358
multidimensionales, 316, 423, 486*e*
no acotado, 362-363
pasos por valor de, 383
tablas de transición como, 61-64
Arreglos completamente subindizados, 423
Arreglos de índice abierto, 316
Arreglos locales, en ambientes de ejecución basados en pila, 357-358
Arreglos multidimensionales, 423, 486*e*
Arreglos parcialmente subindizados, 423
Arreglos sin limitación, en ambientes de ejecución basados en pilas, 362-363

Asignación de memoria, 346-349. *Véase también* Memoria dinámica; Estructuras dinámicas, en asignación de memoria
en ambientes de ejecución basados en pilas, 352-373
en ambientes de ejecución dinámicos, 373-381
para arreglos, 316
para el lenguaje FORTRAN77, 349-352, 350*ilus.*
para estructuras y apuntador de referencia, 423-424, 424*ilus.*
para la máquina TM, 460
para lenguaje TINY, 386-388
para registros y estructuras, 317
para tipo de datos apuntador, 319
para tipo de datos de unión, 318
para tipos de datos recursivos, 322
para variables, 259, 260, 300-301
por declaración, 301
Asignación de registro, 469, 471
optimización de código, 473
Asignación de variables, 259, 260, 300-301, 349-352. *Véase también* Asignación de memoria
Asociatividad, 253*e*
en expresiones aritméticas, 117-118
en notación de la forma Backus-Naur extendida, 124
en Yacc, 236-238, 238*ilus.*
especificación mediante paréntesis, 118
resolución de ambigüedad con, 118-119
Asociatividad por la derecha, 119, 124, 125, 238
Asociatividad por la izquierda, 117, 118-119, 167
en la notación de la forma Backus-Naur extendida, 124
en Yacc, 237-238
recursión por la izquierda y, 157-158
Atributo *base*, 266-268, 267*t*, 268*ilus.*, 273-275, 275*ilus.*, 276, 278, 279, 282-283, 285-286, 287-288
Atributo *dtype*, 265-266, 266*t*, 266*ilus.*, 272-273, 278-282, 288-289, 292-293, 294, 294*ilus.*
Atributo *err*, 310-313, 311*t*
Atributo *etype*, 284
Atributo `isAddr`, 422
Atributo *isFloat*, 284
Atributo *nestlevel*, 310-312, 311*t*
Atributo *pcode*, 408, 408*t*
Atributo *postfix*, 339*e*
Atributo *symtab*, 310-312, 311*t*
Atributo *val*, 263-265, 263*t*, 264*ilus.*, 264*t*, 265*ilus.*, 268-269, 269*t*, 269*ilus.*, 271-272, 273-275, 275*ilus.*, 276, 277-278, 285-286, 290
Atributos, 10, 259-261
aumento de la eficiencia de, 72-74
Atributos dinámicos, 260-261
Atributos estáticos, 260-261
árbol de análisis gramatical recorrido de izquierda a derecha y, 289-290, 292
cálculo de, 291-293
como parámetros y valores devueltos, 285-286
definidos, 278
gráficas de dependencia para, 278-279
heredados en, 278-281
heredados y, 283, 283-284
métodos algorítmicos para evaluación de, 279-284
modificación en atributos sintetizados, 293-295
sintetizados y, 283, 283-284
tablas de símbolos como, 295
Atributos sintetizados, 283-284
cálculo de, durante el análisis sintáctico, 289-295
como parámetros y valores devueltos, 285-286
de nodos de árbol, 134-135
de tokens, 33
definidos, 259
en el análisis semántico, 258, 259
estructuras de datos externas para almacenamiento de, 286-289
heredados, 277, 278-284
orden de evaluación para, 271, 276-277, 343*r*
sintetizados, 277-278, 283-284
Autómata(s) finito(s) determinísticos (DFA, por sus siglas en inglés), 91-93*e*, 94*r*, 250-252*e*
acciones sobre tokens de, 53-56
análisis sintáctico ascendente y, 197, 198
análisis sintáctico SLR(1) y, 213-214
definición, 49-51
ejemplos de, 51-53
elementos LR(0) como, 204-206, 205*ilus.*, 206*ilus.*
elementos LR(1) como, 217-220, 220*ilus.*
implementación de, 59-63
minimización del número de estados en, 72-74
para análisis sintáctico LALR(1), 224-225, 225*ilus.*
para análisis sintáctico LR(1), 220-223, 223*ilus.*
para analizador sintáctico de sentencia if, 214*ilus.*
para el analizador lexicográfico de TINY, 75-77, 77*ilus.*
traducción de expresiones regulares en, 64-74
traducción de los autómatas finitos no determinísticos en, 69-72
Autómata(s) finito(s) no determinístico(s) (NFA, por sus siglas en inglés), 3, 31-32,47-64, 53, 56-59, 92*e*, 94*r*
construcción de autómatas finitos determinísticos a partir de, 69-72
definición, 57-58
ejemplos de, 58-59
elementos LR(0) como, 202-204, 204*ilus.*
implementación, 59, 63-64
simulación utilizando construcción de subconjuntos, 72
traducción de expresiones regulares en, 64-69
Autómatas finitos, 3, 31-32, 47-64. *Véase también* Autómata(s) finitos determinísticos (DFA por sus siglas en inglés); Autómata(s) finito(s) no determinístico(s) (NFA, por sus siglas en inglés)
de elementos, 202-206
implementación en código, 59-64
para identificadores con delimitador y valor devuelto, 54*ilus.*
para identificadores con transiciones de error, 50*ilus.*
para números de punto flotante, 52
tablas de transición para, 61-64
teoría matemática de, 94*r*

Backpatching, 14, 428, 432, 458
Backus, John, 3, 98
Bases, para código relocalizable, 346*n*
Bloques, 302. *Véase también* Bloques básicos de código
de estructura dinámica de memoria (heap), 377-380, 378*ilus.*, 380*ilus.*, 380-381
en código, 364-365, 474
Bloques anidados, en código, 364-365
Bloques básicos de código 477-481, 478*ilus.*
como nodos de gráfica de flujo, 475-477
gráficas acíclicas dirigidas de, 416*n*, 477-481, 478*ilus.*
llamadas de procedimiento, 475*n*
optimización de, 474
Bloques de memoria fragmentados, 377
Bloques *let*, cálculo de niveles de anidación en, 310-313, 311*t*
Bloques no básicos de código, 474
`BucketList`, 336
Búsqueda hacia delante, 53-54, 184-185, 197, 200, 216-217
de un carácter simple, 46
de un símbolo simple, 13
elementos LR(0) y, 202
elementos LR(1) y, 217-218
en análisis sintáctico descendente, 143-144, 180

en análisis sintáctico LR(1) contra análisis sintáctico SLR(1), 222-223
para analizadores sintácticos SLR(*k*), 216-217
problema de, 46
producciones de error en Yacc y, 247-250
Búsqueda *lineal*, 80
Búsquedas binarias, haciendo los analizadores léxicos más eficientes con, 80
Búsquedas hacia delante
combinación, en análisis sintáctico LALR(1), 224
en listados Yacc, 232-234
espontáneamente generadas, 226
generadas espontáneamente, 226
propagación, 225-226

Cadena vacía (ε), 35
autómatas finitos no determinísticos y, 56-57, 64-65
excluyendo repeticiones, 41-42
gramáticas para lenguajes incluyendo, 105
Cadena `yytext`, 85, 86
Cadenas, 31. *Véase también* Caracteres; Letras; Metacaracteres
árboles múltiples de análisis gramatical para, 114-116
autómatas finitos no determinísticos y, 56-59
cálculo de dirección en enteros, 297-298
código intermedio y objetivo como, 407-410, 408*t*, 409*t*
como reglas de gramática libre de contexto BNF, 98
complementarias, 42-43
concatenación de, 36-37
derivaciones múltiples para, 106-107
expresiones regulares como, 34-35, 38-41, 41-43
formas de sentencia de, 129
gráficas de dependencia para, 271
literal, 43-44
números binarios como, 41-42
principio de la subcadena más larga, 46
cálculo de, 290-293
Cálculo de atributo, 270-295
Cálculo Lambda, 371*n*
Cálculos de dirección, en generación de código, 416-418
cambio de atributos heredados en, 293-295
Campo `AddrMode`, 417
Campo `type`, 135
Campos de atributos, en árboles sintácticos, 96
Campos de registro
cálculo de referencias para, en generación de código, 417, 423-428
código del compilador C de Borland para, 445-446
código del compilador C de Sun para, 450-451
Campos `strval`, 410-411, 411*n*
Carácter asterisco (*), 35, 36-37, 97. *Véase también* Operación de repetición
Carácter caret (^), 42-43
como constructor de tipo apuntador en Pascal, 319
Carácter de barra vertical (|), 35-36. *Véase también* Selección entre alternativas en gramáticas libres de contexto
en especificaciones de reglas de Yacc, 227
tipo de datos unión definido por, 319
Carácter de diagonal inversa (\), 81, 82, 90
Carácter de espacio en blanco, 46
Carácter de signo monetario ($)
cálculo de conjuntos Siguiente y, 173-174, 176-177
marcación del final del archivo de entrada con, 152-153, 154
Carácter de subraya (_), en la convención de nombrado de funciones del compilador C, 447
Carácter guión (-), 42, 82
Carácter Nueva-línea, 46
Carácter Tab, 46
Carácter tilde (~), 42
Caracteres, 78. *Véase también* Letras; Metacaracteres; Cadenas
Caracteres de escape, 35
Caracteres en mayúsculas, conversión a minúsculas, 87-88
Caracteres no alfanuméricos, 50*n*
Caracteres y símbolos legales, 34-35
Cargadores, 5
Caso base, en definiciones recursivas, 102-103
Cerradura, 35, 36, 371*n*. *Véase también* cerradura ε
Cerradura (como parámetro de procedimiento), 371-373, 372*ilus.*, 373*ilus.*
Cerradura de Kleene, 36, 371*n*
Cerradura ε, 69-70, 92*e*, 204-205, 217, 226, 371*n*
Chomsky, Noam, 3, 132-133, 142*r*
cicle Fetch-execute, de la máquina TM, 454
Ciclos, 474-475. *Véase también* Ciclos infinitos
en gramáticas, 159
Ciclos infinitos, 159. *Véase también ciclos*
Clases de caracteres, convención de Lex para, 82-83
Clasificaciones topológicas, 274-275, 276-277
de gráficas acíclicas dirigidas, 478
Código, 5, 59-64, 346*n*, 469-470
Código de máquina, 1
Código Diana, 489*r*
Código en C, 84-87, 89, 90-91, 227-228, 229-230
Código en tres direcciones, 11, 398, 399-402, 477*n*, 486*e*, 488-489*e*
como un atributo sintetizado, 407, 408-410
contracódigo P, 406-407
estructuras de datos para, 402-404
gramática con atributos para, 407, 408-410, 409*t*
para cálculos de dirección, 417
para expresiones aritméticas, 484*e*, 485*e*
para funciones y procedimientos, 438
para referencias de apuntador y estructura, 425-426
para referencias de arreglo, 419-420
para sentencias de control, 430
programa TINY en, 400-402
traducción de código P en, y viceversa, 414-416
Código ensamblador, 397-398, 447-448, 452-453
para expresiones aritméticas, 443-444, 448-449
para llamadas y definiciones de función, 447-448, 452-453
para referencias de arreglos, 444-445, 449-450
para referencias de campos y apuntadores, 445-446, 450-451
para sentencias "if" y "while", 446, 451-452
Código inalcanzable, 469-470
Código intermedio, 11, 14, 397-398, 398-399, 489*r*. *Véase también* Código P; Código en tres direcciones
código P como, 404
como un atributo sintetizado, 407-410
construcción de gráficas de flujo a partir de, 475-476, 476*ilus.*
generación de código objeto desde, 411-416
para funciones y procedimientos, 437-439
y objetivo como, 407-410
Código muerto, 469-470
Código objeto (objetivo), 1, 4, 259, 260, 399
como un atributo sintetizado, 407-410
de gráficas acíclicas dirigidas, 478-479
generación de, 398-399
generación de, desde código intermedio, 411-416
Código P, 11, 398, 399, 404-407, 477*n*, 486*e*, 488*e*, 489*r*
como un atributo sintetizado, 407-408
contracódigo en tres direcciones, 406-407
gramática con atributos para, 407-408, 408*t*, 412*ilus.*

para cálculos de dirección, 417-418
para definiciones y llamadas de función, 441-443
para expresiones aritméticas, 484-485*e*
para funciones y procedimientos, 438-439
para referencias de apuntador y estructura, 427-428
para referencias de arreglo, 420-421, 422-423, 424-425*ilus.*
para referencias de estructura, 427-428
para sentencias de control, 430-432, 433, 434-436
procedimiento *genCode* en, 412*ilus.*
procedimiento para generación de, 411
programas TINY en, 406*ilus.*
Traducción de código en tres direcciones en, y viceversa, 414-416
Código relocalizable, 5, 346*n*
Código U, 489*r*
Coerción, 332-333
Colisiones, en tablas de cálculo de dirección, 296, 297-298
Comando `ADD`, 26
Comando `g`, para el simulador TM, 459
Comando `h`, para el simulador TM, 459
Comando `halt`, en código de tres direcciones, 401
Comentarios
autómatas finitos determinísticos para, 52-53
como pseudotokens, 44, 46
en lenguaje TINY, 22, 75, 90
expresiones regulares para, 44-45
Comentarios estilo lenguaje C, 44-45, 53*ilus.*, 61, 62, 87-88, 92-93*e*
como parámetros y valores devueltos, 285-286
Compactación de memoria, 381, 395-396*e*
Compactación. *Véase* Compactación de memoria
Compilación TINY, 24-25, 79
Compilador Algol60, 30*r*
Compilador C Sun 2.0 para estaciones de trabajo Sun SparcStations, 16, 29*r*, 396*r*, 398, 443-453, 487*e*, 489*r*
Compilador de C Borland 3.0 para procesadores 80 × 86 de Intel, 398, 443-448, 485*e*, 487*e*
Compilador de compiladores. *Véase* Generadores de analizadores sintácticos
Compilador Gofer, 396*r*
Compilador TINY, 141-142*e*
archivos y marcas para, 23-25
estructura de árbol sintáctico para, 134-138, 136-138*ilus.*
lenguaje TINY y, 22
listado para, 502-544
recuperación de errores en, 250
Compiladores, 1-2. *Véase también* Compilador de Algol60; Compiladores de C; Compiladores cruzados; Compilador de FORTRAN; Paquete de compiladores Gnu; Compilador de Gofer; Compilador de Modula-2; Compilador de Pascal; Generadores de analizador sintáctico; Compilador de TINY; Yacc (yet another compiler-compiler)
ambiente de ejecución para, 345-346
análisis semántico mediante, 258
analizadores sintácticos para, 96
cuestiones estructurales respecto a los, 14-18
de una pasada, 16
definición del lenguaje y, 16-17
diagramas T para, 18-21
directivas de depuración para, 17
efectos de la optimización sobre el tiempo de ejecución de, 468
estructuras de datos principales en, 13-14
etapa inicial y etapa final de, 15
fases de, 7*ilus.*, 7-12
historia de los, 2-4
manejo de errores y excepciones en, 18
múltiples de pasadas, 258-259
partes de análisis y síntesis de, 15
pasadas de, 15-16
portátiles, 15
programas de prueba estándar para, 16
programas relacionados con, 4-6
referencias sobre, 29-30*r*
transportación de, 18-21
Compiladores cruzados, 18, 20, 21
Compiladores de C, 16, 29*r*, 396*r*, 398, 443-453, 487*e*, 489*r*. *Véase también* Compilador C Borland 3.0 para procesadores Intel 80 × 86;
Compiladores de Modula-2, 16
Compiladores de múltiples pasadas, 258-259
Compiladores de Pascal, 11, 16
Compiladores de perfil, 472
Compiladores de una pasada, 16
Compiladores FORTRAN, 30*r*
Completar las gramáticas con atributos, 270
Computadora con conjunto de instrucciones reducido (RISC, por sus siglas en inglés), 25, 456, 469
Computadoras, John von Neumann y las, 2
con atributos S, 277
Conflictos de análisis sintáctico. *Véase también* Problema del else ambiguo; Conflictos Reduce-reduce; Conflictos Shift-reduce
Reglas de no ambigüedad para, en Yacc, 235-238
Conflictos Reduce-reduce, 207, 211
con Yacc, 236-238
reglas de no ambigüedad para, 213
Conflictos Shift-reduce, 207, 211, 213, 235-236
Conjunto de caracteres ASCII (American Standard Code for Information Interchange), 34
Conjunto de herramientas de construcción para compilador de Purdue (PCCTS, por sus siglas en inglés), 196*r*
Conjunto de programas de prueba, 16
Conjuntos de potencia, 57
Conjuntos Primero, 144, 168-173, 190-191*e*
definición, 151, 168-169
para analizadores sintácticos LL(*k*), 180
para gramática de expresión de enteros, 170-171, 171*t*, 178-179
para gramática de secuencia de sentencias, 173, 179-180
para gramática de sentencia "if", 171-172, 172*t*, 179
conjuntos regulares, 39
Conjuntos Siguiente, 144, 168, 173-177, 190-191*e*
definición, 152
en análisis sintáctico SLR(1), 210, 211
para analizadores sintácticos LL(*k*), 180
para gramática de expresión de enteros, 174-176, 176*t*, 178-179
para gramática de secuencia de sentencias, 177, 179-180
para gramática de sentencia "if", 179
Consistencia de gramáticas con atributos, 270
Constante `false`, 139*e*, 432-433, 433-434
Constante `true`, 139*e*, 432-433, 433-434
Constantes, 43
Constantes `nat`, 51-52
Constantes `number`, 51-52
Constantes numéricas, autómatas finitos determinísticos para, 51-52
Constantes `signedNat`, 51-52, 82-83
asignación de memoria para, 347
descriptores de registro y dirección para, 479-481, 479*t*, 480*t*, 481*t*
en el archivo `code.h`, 460-461
constructores de tipo cartesiano en, 317
Constructores de tipos, 314, 315-320
Constructores de valor, en lenguaje ML, 319
Contador de programa (pc, por sus siglas en inglés), 348
Contexto, definición en, 132
controlado por tabla, 63, 154-157
Convenciones de Lex, 81-83
Convenciones de metacarácter, 81-83, 83*t*, 98-99
Convenciones léxicas, para el lenguaje TINY, 75
Convenciones. *Véase* Convenciones léxicas; Convenciones de metacaracteres
Conversión de tipo, 332-333
Corchetes o paréntesis cuadrados ([]), 42, 82, 124-125
Corrección de errores, 183

Corrección de errores de distancia mínima, 183
Cuádruples, para código en tres direcciones, 402, 403*ilus.*
Cubos, en tablas de cálculo de dirección, 296-297, 297*ilus.*
Cuerpo de una expresión *let*, 309

Datos de longitud variable, en ambientes de ejecución basados en pilas, 361-363, 391*e*
Datos de memoria para la máquina TM, 454
Datos estáticos, asignación de memoria para, 347
Datos globales, asignación de memoria para, 347
Declaración antes del uso, 132, 301-302
Declaración automática, 301
Declaración `char`, 240, 314
Declaración Class, 320, 376, 376*ilus.*, 393-394*e*
Declaración `COMMON`, 350*n*, 350-351
Declaración `const`, 299, 314, 347
Declaración `constant`, 43
Declaración `datatype`, 328-329
Declaración de constantes, 299, 300
Declaración `double`, 240, 314
Declaración `ExpType`, 135
definición, 277
Declaración `extern`, 301
Declaración `literal`, 43
Declaración por uso, 300
Declaración `static`, 301
Declaración `struct`, 299
nombres de tipo y declaraciones en lenguaje C asociados con, 321-322
Declaración `TokenType`, 33
Declaración `type`, 299, 328
Declaración `typedef`, 299
Declaración `union`, 111, 299, 317-319
administradores de proyecto en, 6
declaración `%union`, en Yacc, 240-241
declaraciones y nombres de tipo en lenguaje C asociados con, 321-322
producciones de unidad, 141*e*
sistema Unix, 4
Yacc incluido en, 256*r*
Declaraciones, 10, 139*e*, 140*e*. *Véase también* Declaraciones automáticas; Declaración de clase; Declaraciones colaterales; Declaraciones de constante; Declaraciones de tipo de datos; Definiciones; Declaraciones explícitas; Declaraciones por adelantado; Declaraciones de función; Declaraciones de prototipo de función; Declaraciones globales; Declaraciones implícitas; Declaraciones locales; Declaraciones anidadas; Declaraciones de procedimiento; Declaraciones recursivas; Declaraciones secuenciales; Declaraciones estáticas; Declaraciones de tipo; Declaraciones de variable
al mismo nivel de anidación, 306-308
comportamiento de tablas de símbolos y, 298-301
con expresiones *let*, 309-310
de tipo de datos, 111
en código Yacc, 229
en gramáticas de expresión aritmética simple, 308-313
en tablas de símbolos, 295
extensión de, 300-301
tiempo de vida de, 300
verificación de tipos de, 329-330
Declaraciones anidadas, 363, 364-365, 368-369
Declaraciones colaterales, 307
Declaraciones de función, 299-300, 307-308, 341-342*e*, 437-439
Declaraciones de procedimiento, 141*e*, 299-300, 307-308, 437-439
Declaraciones de prototipo de función, 308
Declaraciones de tipo, *Véase también* Declaraciones de tipo de datos
Declaraciones de tipos de datos, 111, 259, 260, 313-314. *Véase también* Declaraciones de tipo
en tablas de símbolos, 295
en Yacc, 239-241
gramática simple de, 265-266, 288-289
Declaraciones de variables, 299, 300-301
gramática simple de, 265-266, 288-289, 294
tipo de datos en, 314
Declaraciones estáticas, 27, 300-301
Declaraciones estilo Pascal, gramática para, 339-340*e*
Declaraciones explícitas, 299
Declaraciones Forward, 308
Declaraciones globales, en el lenguaje C-Minus, 27
Declaraciones implícitas, 299-300
Declaraciones locales, en el lenguaje C-Minus, 27
Declaraciones recursivas, 307-308
Declaraciones secuenciales, 307, 310
Definición de lenguaje, 16-17
Definición del ambiente de un procedimiento, 367
Definiciones, 301, 476*n*, 476-477
del código objetivo, 399
Definiciones de alcance, 476-477
Definiciones de función, 486*e*
código intermedio para, 437-439
compilador C de Sun, código para, 452-453
compilador de C Borland, código para, 447-448
en el archivo `code.h`, 460-461
procedimiento de generación de código para, 439-443
simplificación de gramáticas con atributos con, 268
Definiciones de tipo, 320
Definiciones formales, de lenguajes de computadora, 16
Definiciones regulares, 37
Delimitadores, 44-45, 46, 54
Delimitadores de comentario, 44-45
Delimitadores de signo porcentual (%%), 83-84, 84, 85
Delimitadores de token, 46
Depuradores, 6, 238-239
Derivaciones, 100-101
árboles de análisis gramatical y, 106-109
contra reglas gramaticales, 101
definición formal de, 129
por la derecha, 109, 114, 129
por la izquierda, 108-109, 114-116, 129, 153-154
por la izquierda y por la derecha, 108-109
reglas gramaticales y, 102-106
Descriptor de dirección, 479*t*, 479-481, 480*t*
Descriptores de registro, 479-481, 480*t*, 481*t*
Desplazamientos (Offsets)
cálculo de direcciones con, 356-358, 357*ilus.*, 358*ilus.*, 365-366, 393*e*
generación de código intermedio y, 399
para referencias de apuntador y estructuras de registro, 423-425, 424*ilus.*
subíndices de arreglo y, 418-419
Detección de ciclos, 475
Detección de errores, en análisis sintáctico ascendente, 245
DFA de estados mínimos, 73, 74
Diagramas de sintaxis para análisis sintáctico, 123, 125-128
estructuras de datos basadas en, 194-195*e*
para expresiones aritméticas, 126-127, 127*ilus.*
para sentencias "if", 127-128, 128*ilus.*
Diagramas T, 18-19, 19-21*ilus.*, 28-29*e*
Dígitos, 43. *Véase también* Enteros; Números
en constantes numéricas, 51-52, 259, 260
en expresiones regulares, 131-133
en gramáticas con atributos, 268
en gramáticas con tipos de valor de punto flotante y enteros mezclados, 283-284
en gramáticas de números con base, 273-275, 282-283
en gramáticas de números sin signo, 262-264, 266-268, 271-272, 276-277
en identificadores, 44, 47-48
en Lex, 82
significativos, 259, 260
Dirección base, de un arreglo, 418
Direccionamiento abierto, 356-359
en ambientes de ejecución basados en pilas, 356-359

Direccionamiento abierto, en tablas de cálculo de dirección, 296
Direccionamiento de byte, 12
Direccionamiento indirecto, 417-418
de registro, 12
Direccionamiento. *Véase también* Direccionamiento de byte; Direccionamiento de registro indirecto
Directiva `%type`, 240-241
Directivas de depuración, 17
Directivas del compilador, 17
Discriminante, de tipo de datos de unión, 318-319
División, 117, 193*e*, 283-284
División de punto flotante, 283-284
División entera, 283-284

Ecuaciones de atributos, 258, 261*n*, 270-271. *Véase también* Reglas semánticas
definidas, 261
llamadas de procedimiento en lugar de, 287
metalenguajes y, 268
para expresiones aritméticas, 272-273, 284
para números con base, 273-275
para números sin signo, 263-264, 271-272
Ecuaciones, reglas gramaticales como, 129-130
Editores, 6
Editores basados en estructura, 6
Eficiencia
de los autómatas finitos determinísticos, 72-74
Ejecución de programa, organización de la memoria durante, 346-349
Elección entre alternativas, 35-36
código para, 145-151
con gramáticas libres de contexto, 97
en análisis sintáctico LL(1), 157
en expresiones regulares, 38-41, 41-43
implementación para autómatas finitos no determinísticos, 65-66
precedencia de, 37, 38
Elementos completos, 202
Elementos de cerradura, 206, 232
Elementos de núcleo (kernel), 206, 232-234
Elementos iniciales, 202
Elementos LALR(1), 224-225, 250*e*, 251*e*, 252*e*
Elementos LR(0), 201-206, 250*e*, 251*e*, 252*e*
análisis sintáctico LALR(1) y, 224
autómatas finitos de, 202-206
Elementos LR(1), 250*e*, 251*e*
análisis sintáctico LALR(1) y, 224
autómatas finitos de, 217-220, 224, 224*n*
Elementos SLR(1), 250*e*
Elementos. *Véase* Elementos cerrados; Elementos completos; Elementos iniciales; Elementos de núcleo (Kernel); Elementos LALR(1); Elementos LR(0); Elementos LR(1); Elementos SLR(1)
Eliminación de subexpresión común, 469, 469*n*
Encadenamiento, 365. *Véase también* Encadenamiento de acceso; Encadenamiento por separado
Encadenamiento por separado, en tablas de cálculo de dirección, 296-297, 297*ilus.*
Ensambladores y lenguaje ensamblador, 3, 5
Enteros. *Véase también* Dígitos;
asignación de memoria para grandes, 347
cadenas de caracteres de cálculo de dirección en, 297-298
gramática de números tipo, 339*e*
máximo común divisor de, 353-354
Entradas predeterminadas, en tablas de análisis sintáctico LL(1), 191*e*
Equivalencia de declaración, 328-329
Equivalencia de nombre, 324-329, 344*r*
Equivalencia de tipos, 322-329, 344*r*
Equivalencia estructural, 324, 327, 328-329, 344*r*
Errores de compilación, 27-28*e*
Errores de semántica, 27-28*e*
Errores de sintaxis, 27-28*e*
Errores en tiempo de compilación, 18
Errores estáticos, 18
Escritura polimórfica, 333-334, 344*r*
Esquemas de tipo, 333
Estado de error. *Véase también* Estados de autómatas finitos determinísticos, 50-51, 54, 73
Estado `DONE`, en autómata finito determinístico de TINY, 76
Estado `error`, 50
de Yacc, 247, 248-249, 250
Estados de aceptación. *Véase también* Estados
en autómatas finitos, 48, 49-51
Estados de inicio. *Véase también* Estado de autómatas finitos determinísticos, 49, 54-55
en autómatas finitos, 48
para gramáticas aumentadas, 203
Estados. *Véase también* Estados de aceptación; Estados de error; Estados de start
cerradura ε de, 69-70
de autómatas finitos, 48, 49-51, 57-58
de elementos LR(1), 218-220
en análisis sintáctico LR(1), 221, 222-223
en listados de Yacc, 232-235, 233-234*ilus.*
en tablas de transición, 61-64
minimización del número de, 72-74
Estimación conservativa de información de programa, 472
Estimulación estática, 413, 416
Estructura de datos de pantalla, 369, 392*e*, 396*r*
Estructura de datos del diccionario, de tablas de símbolos, 295-296
Estructura *factor*, 125, 127
Estructura *program*, 101-102
Estructura `program`, en el analizador sintáctico TINY, 243-244
Estructura `treeNode`, 462
Estructuras, 357-358, 423-428. *Véase también* Estructuras de datos; Tipo de datos; Campos de registros; Registros
Estructuras de árbol de búsqueda, 296
Estructuras de datos. *Véase también* Árboles sintácticos abstractos; Arreglos; Pilas de llamada; Tipos de datos; Gráficas de flujo; Métodos; Objetos; Campos de registro; Registros; Cadenas; Tablas de símbolos; Árboles sintácticos
basadas en diagramas de sintaxis, 194-195*e*
de longitud variable, en ambientes de ejecución basados en pilas, 361-363
generación de código de referencias para, 416-428
gráficas de flujo como, 475-477, 476*ilus.*
para almacenar valores de atributo, 286-289
para generación de código, 398-407
para implementación del código en tres direcciones, 403-404
para optimización de código, 475-481
para tablas de palabras reservadas, 80
Estructuras dinámicas (heap), en asignación de memoria, 17, 347*n*, 347-348, 378*ilus.*, 380*ilus.*, 396*r*. *Véase también* Asignación de memoria
administración automática de, 380-381
administración de, 376-380, 379*ilus.*
Estructuras locales, en ambientes de ejecución basados en pilas, 357-358
Etiquetas, 428, 432
Etiquetas de salida, de sentencias de control, 431-432
Evaluación de cortocircuito, 428, 433, 487*e*
Evaluación descuidada, 386, 396*r*
Evaluadores de atributo
construcción automática de, 344*r*
postorden recursivos, 277-278
Expansión de macro, 413, 415-416
Exponentes
en constantes numéricas, 51-52
multiplicación y, 470

notación para, 43
suma de, para expresiones aritméticas enteras, 193*e*
Expresión *let*, 309-310, 311*t*, 312-313
Expresiones aritméticas, 140*e*, 255*e*. *Véase también* Expresiones aritméticas enteras
ambigüedad no esencial en, 123
analizador sintáctico ascendente para, 199, 200*t*
analizador sintáctico SLR(1) para, 211*t*, 211-212, 212*t*
autómata finito determinístico para gramática de, 205-206, 206*ilus.*
autómata finito no determinístico para gramática de, 204, 204*ilus.*
calculadora recursiva descendente para, 146-151, 148-149*ilus.*
cálculo de conjuntos Primero para, 170-171, 171*t*
cálculo de conjuntos Siguiente para, 174-176, 176*t*
código en tres direcciones para, 399-402, 414-416, 484*e*
código P para, 404-405, 414-416, 484-485*e*
código para, 146-147
código para el compilador de Borland C, 443-444
código para el Compilador Sun C, 448-449
construcción de tablas de análisis sintáctico LL(1) para, 178-179
declaraciones de tipo en Yacc para, 239-241
diagramas sintácticos para, 126-127, 127*ilus.*
división en, 283-284
en análisis sintáctico LL(1), 157
en código Yacc, 239-241, 241-242
en lenguaje TINY, 22, 133-134
en notación extendida de la forma Backus-Naur, 123-125
evaluador de atributos recursivo de orden posterior para, 277-278, 278*ilus.*
expresiones booleanas en, 192*e*
gráficas de dependencia para, 272-273
gramática LL(1) para, 160-162, 160*ilus.*, 163*t*, 165, 166-167
gramática para, 202
gramáticas con atributos para, 264-265, 277-278, 283-284, 284*t*, 290-291, 291-293, 294, 308-313
modificación de la calculadora recursiva descendente de, 193-194*e*
números de punto flotante en, 283-284
precedencia en, 116-118
reglas de semántica y gramática para, 269-270, 269*t*, 270*t*, 272-273
Simplificación de la gramática con atributos para, 269-270
tipos mezclados en, 332-333
Yacc generador de analizadores sintácticos para la gramática de, 227-230
Expresiones booleanas, 22, 133-134, 139*e*, 192*e*, 428, 463. *Véase también* Operadores de comparación; Expresiones de prueba
Expresiones completamente entre paréntesis, 118
Expresiones constantes, en el lenguaje TINY, 134
Expresiones de estructura (*exp*), 130
en árboles de análisis gramatical, 107-109
en especificaciones de Yacc, 229
en expresiones *let*, 309
en gramáticas ambiguas, 114-115
en gramáticas libres de contexto, 98-100, 100-101
en listados de Yacc, 234
en sintaxis abstracta, 110-111
Expresiones de operador, en lenguaje TINY, 134, 137*ilus.*
Expresiones de prueba. *Véase también* Expresiones booleanas; Optimización
de sentencias de control, 482-484
Expresiones de reconocimiento, en el lenguaje TINY, 134
Expresiones de subíndice, 421-423, 424-425*ilus.*, 485*e*
Expresiones de tipo, 313-314, 314-320
con declaraciones de tipo, 324-326, 326*ilus.*
determinación de equivalencia de, 322-329
gramática simple para, 323*ilus.*, 323-324
Expresiones "If", 433
Expresiones if-then-else, en ecuaciones de atributos, 268
Expresiones lógicas, generación de código para, 428, 432-433
Expresiones regulares, 3, 31-32, 34-47, 91*e*, 138*e*, 255*e*
básicas, 35, 38-41, 64-65
extendidas, 41-43
cadenas como, 34-35, 38-41, 41-43
contra gramáticas libres de contexto, 95, 97-98
convenciones de Lex para, 81-83
definición, 35-41
en lenguaje TINY, 75-76
extensiones para, 41-43
formas de, 38-41
gramáticas libres de contexto para, 104-105
lenguajes generados por, 34-35
nombres para, 37-38
operaciones en, 35
para tokens de lenguaje de programación, 43-47
teoría matemática de, 94*r*
traducción en autómatas finitos determinísticos, 64-74
Expresiones. *Véase también* Expresiones aritméticas; Expresiones regulares básicas; Expresiones Booleanas; Expresiones constantes; Expresiones de identificador; Expresiones "if"; Expresiones "if-then-else"; Expresiones aritméticas enteras; Expresiones *let*; Expresiones lógicas; Expresiones de operador; Expresiones regulares; Subexpresiones; Expresiones con subíndice; Expresiones de tipo
en lenguaje TINY, 22, 133-134, 134-137
evaluación, 259, 260
verificación de tipo de, 331
Extensión de declaraciones, 300-301
Extracción, de pilas, 153, 167, 186-187, 352

Factor de escala, 358, 418-419
Factorización por la izquierda, 157, 162-166
False, Saltos de, 429
Fases del compilador, 7-12,
Fijación, 259-260, 375, 376, 381-382
Fijación dinámica, 375, 376
Flex, 81
Forma Backus-Naur (BNF, por sus siglas en inglés), 140-141*e*, 258. *Véase también* Forma Backus-Naur Extendida (EBNF, por sus siglas en inglés)
de gramáticas LL(1), 178
de las gramáticas libres de contexto, 98
del lenguaje C-Minus, 492-496
jerarquía de Chomsky y, 131-133
para la gramática del lenguaje TINY, 133-134, 134*ilus.*
para sintaxis abstracta y concreta, 111
problemas al traducir la BNF a la forma Backus-Naur extendida, 151-152
reglas de no ambigüedad y, 121-122
reglas gramaticales para, 98-100
Forma Backus-Naur Extendida (EBNF), 123-128
análisis sintáctico recursivo descendente y, 196*r*
Calculadora recursiva descendente para aritmética entera simple y, 148-149*ilus.*
diagramas de sintaxis de, 125-128
elección de traducción y operaciones de repetición en código con, 145-151
Jerarquía de Chomsky y, 131-133
para el lenguaje gramatical TINY, 180-181
Forma de columna principal de un arreglo, 316
Forma sentencial, de una cadena, 129

Formas de sentencias por la derecha, 200-201, 252*e*
fp antiguo, 353
Función booleana *isen*, 310, 312
Función `cGen`, para la máquina TM, 461-462
Función `codeGen` para el compilador TINY, 461-462
Función `emitBackup`, 461
Función `emitComment`, 461
Función `emitRestore`, 461, 464
Función `emitRM`, 461, 463
Función `emitRM_Abs`, 461, 464
Función `emitRO`, 461
Función `emitSkip`, 461, 464
Función factorial
  programa de muestra en C-Minus para, 27
  programa de muestra en TINY para, 23, 78, 79, 137-138*ilus.*, 335, 487*e*
  programa simulador de TM para, 458-459
Función `field_offset`, 425-428
Función `genExp`, 462, 483
Función `genStmt`, 462, 483
Función `getNextChar` para analizador léxico de TINY, 78
Función *isArrayType*, 331
Función `main`, en el lenguaje C-Minus, 27
Función `mod`, 297, 298. *Véase también* Módulo entero
Función módulo, 297-298. *Véase también* módulo entero
Función *newtemp*, 408-409
Función `ord`, 297
Función `printf`, 27, 89, 361
Función `read`, 27, 400-401
Función `realloc`, 80
Función `scanf`, 27
Función `sprintf`, 411
Función `SQRT`, 350*n*, 352
Función `st_insert`, 387-388
Función `st_lookup`, 387-388
Función `traverse`, 337
Función *typeEqual*, 323-327, 342*e*
  pseudocódigo para, 325*ilus.*, 326, 341*e*
  verificación de tipo con, 329-331
Función `void`, en el lenguaje C-Minus, 27*n*
Función `write`, 27, 400-401
Funciones de cálculo de dirección de mínimo perfecto, 80, 94*r*, 296, 297-298, 298*ilus.*, 336
Funciones de conjuntos, 130
Funciones de transición, 49, 50
  en autómatas finitos no determinísticos, 57-58
Funciones miembro, 320
Funciones recursivas, 27, 307-308, 353-354
Funciones. *Véase también* Secuencias de llamada; Función factorial; Funciones de cálculo de dirección; Funciones miembro; Función módulo; Procedimientos; Funciones recursivas; Funciones de conjunto; Funciones de transición
  anidadas, 303
  constructores de tipo para, 319-320
  generación de código de llamadas a, 436-443
  secuencias de llamadas de, 349, 359-361, 369-370
Fusión, de bloques de memoria, 377

GAG, 344*r*
Gamas de caracteres, en expresiones regulares, 42
Generación automática de generadores de código, 490*r*
Generación de analizadores léxicos, con Lex, 81-91
Generación de código, 4, 11-12, 260-261, 345, 397-484, 490*r*. *Véase también* Código ensamblador; Código C; Código intermedio; Código objeto; Código P; Código objetivo; Código en tres direcciones
  de código ensamblador, 397-398
  de llamadas de procedimiento y de función, 436-443
  de referencias de estructura de datos, 416-428
  durante la optimización, 397, 468-481
  en compiladores comerciales, 443-453
  estructuras de datos para, 398-407
  exactitud de, 465
  optimizaciones para el generador de código TINY, 481-484
  para la máquina TINY (TM), 453-459
  para secuencias de llamada de procedimiento, 349
  para sentencias de control y expresiones lógicas, 428-436
  práctica, 410-411
  técnicas básicas para, 407-416
Generador de analizador sintáctico Gnu Bison, 226, 232*n*, 256*r*
Generador de analizadores sintácticos LLGen, 196*r*
Generador de código TINY
  archivo de código muestra TM generado por, 465-467, 466-467*ilus.*
  generación archivos de código TM de, 464-465
  ineficiencias de, 481
  interfaz TM para, 459-461
  optimizaciones simples para, 481-484
Generador de sintetizador, 344r
Generadores de analizadores léxicos, 4
  Analizadores léxicos (rastreadores), 8
  comparados con analizadores sintácticos, 95-96
  desarrollo de código para, 75-80
  reconocimiento de palabras reservadas mediante, 80
Generadores de analizadores semánticos, 259
Generadores de analizadores sintácticos, 4, 196*r*, 226. *Véase también* generador de analizador sintáctico Antlr; generador de analizador sintáctico Bison; generador de analizadores sintácticos Bison Gnu; generador de analizadores sintácticos LLGen; Yacc (yet another compiler-compiler, "otro compilador de compilador más")
`globals.h` archivo de cabecera, 23, 77, 78, 135, 244-245
Gráficas acíclicas dirigidas (DAG, por sus siglas en inglés), 274-275, 275*ilus.*, 276-277, 487*e*
  de bloques básicos, 416*n*, 477-481, 478*ilus.*
Gráficas de dependencia, 271-277, 340*e*
  asociadas, 271
  asociadas para reglas de gramática, 271
  atributos sintetizados y, 277
  de gramáticas de números con base, 273-275, 282-283
  de gramáticas de números sin signo, 271-272, 276-277
  dependencias "en reversa" en, 289-290
  nodos raíz de, 275*ilus.*, 275*n*, 275-276
  para atributos heredados, 278-279
  para cadenas, 271
  para expresiones aritméticas, 272-273
Gráficas de flujo, 475-477, 476*ilus.*
Gráficas de herencia, 375, 394*e*, 396*r*
Gramáticas, 3, 138-140*e*, 189-190*e*, 194*e*. *Véase también* Gramáticas ambiguas; Gramáticas con atributos; Gramáticas aumentadas; Gramáticas libres de contexto; Gramáticas cíclicas; Gramáticas LALR(1); Lenguajes; Gramáticas LL(1); Gramáticas LR(0); Gramáticas LR(1); Gramáticas regulares; Gramáticas SLR(1); Gramáticas no restringidas
  ambiguas, 114-118
  con atributos L, 290
  construcción de autómatas finitos determinísticos para, 250-252*e*
  de declaración de variables, 265-266
  de números con base, 273-275, 285-286, 287-288
  de números sin signo, 262-264, 266-268, 271-272, 276-277
  dependencia del cálculo de atributos en la sintaxis de, 293-295
  eliminación de recursión por la izquierda en, 157-162
  factorización por la izquierda en, 162-166

modificación de, 293-295
para arreglos subindizados, 421-423
para expresiones de tipo, 323*ilus.*, 323-324
para lenguajes de programación, 101-102
para llamadas y definiciones de función, 439-440
para verificación de tipo, 329*ilus.*, 330*t*
Gramáticas ambiguas, 114-118
con Yacc, 234-238, 237*ilus.*, 238*ilus.*
definidas, 116
Gramáticas aumentadas, 199, 203
Gramáticas cíclicas, 141*e*, 159
con atributos con, 308-313
declaraciones y, 295, 298-301
Gramáticas con atributos, 258, 259, 261-270, 339-340*e*, 340-341*e*, 342*e*, 343-344*r*. *Véase también* Gramáticas; Gramáticas con atributos L; Gramáticas con atributos S
cálculo de atributos con, 270-271
con tablas de símbolos, 308-313
de declaraciones de variable, 265-266
de expresiones aritméticas enteras, 264-265
de números con base, 273-275, 282-283, 285-286, 287-288
de números sin signo, 263-264, 266-268, 267*t*
definición de código intermedio y objetivo con, 407-410
definidas, 261-262
extensiones y simplificaciones de, 268-270
no circular y altamente no circular, 276
para código en tres direcciones, 407, 408-410, 409*t*, 485*e*
para código P, 407-408, 408*t*, 412*ilus.*, 485*e*
para verificación de tipos, 329-330, 330*t*
Gramáticas con atributos
fuertemente no circulares, 276
L, 290, 291-293
no circular, 276
S, 277
Gramáticas LALR(1), 225
Gramáticas libres de contexto, 3, 95, 96, 97-106, 128-138, 142*r*
definición formal de las, 128-129
elementos LR(0) para, 201-206
elementos LR(1) para, 217-220
lenguajes definidos por, 100-106, 128-130
notaciones para, 97-98
para lenguaje TINY, 133-134
reglas gramaticales para, 98-100
Gramáticas LL(1), 155-156, 178
Gramáticas LR(0), 207
Gramáticas LR(1), 221
Gramáticas regulares, 131
Gramáticas sin restricción, 132-133
Gramáticas SLR(1), 210-211

Herencia, 375, 375*n*
Herencia múltiple, 375*n*, 396*r*
Herencia simple, 375*n*
Huecos de ámbito, 302

Identificador DFA, implementación de, usando variables de estado y sentencias "case" anidadas, 60-61
Idiomas de máquina, optimización de código al utilizar, 471
Indizado de arreglos, 12, 486*e*
Inferencia de tipo, 313, 329-331, 343-344*r*
Inferencia de tipos de Hindley-Milner, 344*r*
Información de tipo dinámico, 313
Información de tipo estático, 313
Información de tipo implícita, 314
Información explícita de tipo, 314
Inserción, de pilas, 152-153, 167, 186-187, 352
Instrucción `adi`, 405, 414-415, 416
Instrucción `arg`, 438, 486*e*
Instrucción `begin_args`, 438
Instrucción `call`, 438, 447
Instrucción `csp`, 439
Instrucción `cup`, 439, 442
Instrucción `ent`, 438-439
Instrucción `entry`, en código de tres direcciones, 438
Instrucción `equ`, 406
Instrucción `fjp`, 405, 430-431
Instrucción `goto`, 430
Instrucción `grt`, 406
Instrucción `HALT`, en el simulador TM, 457
Instrucción `if_false`, en código de tres direcciones, 401, 429-430
Instrucción `ind`, 417-418, 420-421
Instrucción `ixa`, 417-418, 420-421
Instrucción `JEQ`, 457
Instrucción `JLT`, 457
Instrucción `label`, en el código en tres direcciones, 401, 438
Instrucción `LD`, para la máquina TM, 25, 456, 457
Instrucción `lda`, 414, 415, 418, 419
Instrucción `LDA`, para la máquina TM Machine, 25, 456, 457
Instrucción `ldc`, 404-405
Instrucción `LDC`, para la máquina TM, 25, 26*n*, 456
Instrucción `lod`, 405, 415-416
Instrucción Marcar-pila (Mark-stack), 439
Instrucción `mst`, 439, 442
Instrucción `nop`, 451-452
Instrucción `rdi`, 405
Instrucción `ret`, 438-439
Instrucción `return`, en código de tres direcciones, 438
Instrucción `sbi`, 405-406
Instrucción `stn`, 407-408, 416
Instrucción `sto`, 407, 416
Instrucción `stp`, 406
Instrucción `ujp`, 430-431
Instrucción `wri`, 405
Instrucciones Copy, en código de tres direcciones, 401-402
Instrucciones de almacenamiento no destructivas, 407
Instrucciones de salto condicional, 429, 429*ilus.*, 431*ilus.*, 486*e*. *Véase también* Instrucciones de salto; Instrucciones de salto incondicional
en código de tres direcciones, 401
optimización de código con, 482-484, 484*ilus.*
para la máquina TM, 457
Instrucciones para la máquina TM, 454-457, 455*t*
Instrucciones para saltos, 428-430, 448. *Véase también* Sentencias de ruptura; Instrucciones de salto condicional; Sentencias de control; Acciones Goto; Sentencias Goto; Instrucciones de salto incondicional
en el simulador TM, 458
optimización de secuencias de código sin, 474
separación de bloques básicos de código mediante, 475-476, 476*ilus.*
Interfaces, entre compiladores y sistemas operativos, 17
Interfaz TM, para generador de código TINY, 459-461
Interpretación abstracta, 413
Intérpretes, 4-5,
Intérpretes de comandos, 1
Interrupción ("Thunk"), 386
ISO (International Organization for Standardization), estándares del lenguaje adoptados por, 16

Jerarquía de Chomsky, 3, 131-133, 142*r*
Jerarquía de clase, 320
Johnson, Steve, 4

Lema de la extracción (pumping lemma), 39
Lenguaje Ada, 16, 141*e*, 489*r*
ambiente de ejecución basado en pilas para, 362-363, 365
ambiente de ejecución de, 345
ámbito en, 305
análisis semántico en, 257
Arreglos con subíndices en, 418-419
comentarios en, 44-45
declaración de constante en, 300
declaraciones al mismo nivel de anidación en, 306

equivalencia de tipo en, 344*r*
información de tipo estático en, 313
lenguaje TINY y, 22
mecanismo de paso de parámetros de, 382
mecanismo de paso por valor de, 382, 391*e*
mecanismo de paso por valor-resultado de, 384-385
operadores sobrecargados en, 332
pasando procedimientos como parámetros en, 373
problema del else ambiguo con, 122
procedimientos y funciones anidados en, 303
tipo de datos de función en, 319
tipo de datos de unión en, 319
Lenguaje Algol60, 142*r*
ambiente de ejecución basado en pilas para, 396*r*
equivalencia estructural en, 344*r*
mecanismo de paso por nombre de, 386
problema de else ambiguo con, 122
recursión en, 396*r*
reglas gramaticales para, 98
Lenguaje Algol68, 122, 344*r*
Lenguaje BASIC, 4-5, 299
Lenguaje C, 139*e*, 140*e*
ambiente de ejecución basado en pilas para, 353-361, 363-365
ambiente de ejecución para, 17, 345, 388-389*e*, 393*e*, 394-395*e*
ámbito dinámico contra ámbito estático en, 305-306, 306*ilus.*
ámbitos de anidación en, 302*ilus.*
análisis semántico en, 257
arreglos con subíndices en, 421-423
asignación de memoria en, 347
comentarios en, 44-45
constructores de tipo arreglo en, 315-316
conversión y coerción de tipo en, 332-333
declaración antes del uso en, 301-302
declaración de constantes en, 299, 300
declaración de equivalencia en, 328
declaración de tipo de datos en, 111
declaraciones al mismo nivel de anidación en, 306-307
declaraciones de tipo en, 299, 320-322
declaraciones de variables en, 299, 300-301
declaraciones para funciones recursivas en, 307-308
definiciones contra declaraciones en, 301
equivalencia de tipos en, 344*r*
estructura de bloques de, 302
estructuras de datos para código en tres direcciones cuádruples en, 403*ilus.*
forma Backus-Naur para, 102
función de tipo de datos en, 320
funciones y procedimientos anidados en, 303
gramática de sentencias de control en, 434-436
información de tipo estático en, 313
lenguaje TINY y, 133-134
Lex y, 81
mecanismo de paso de parámetros de, 382
mecanismo de paso de parámetros de, mecanismo de paso por valor-resultado, 384-385
mecanismo de paso de parámetros de, paso por nombre, 385-386, 395*e*
mecanismo de paso de parámetros de, paso por valor, 382-383
nombres de tipo en, 320-322
operaciones con estructuras dinámicas (heap) en, 377-380, 379*ilus.*
operaciones lógicas de cortocircuito en, 433
operadores sobrecargados en, 332
paso de procedimientos como parámetros en, 373
referencias colgantes en, 373-374
referencias de apuntador y estructuras de registro en, 423-425, 426
registro de activación de procedimiento en, 348
subíndices de arreglo en, 418-419
tipo de datos apuntador en, 319
tipo de datos de unión en, 317-318
tipos de datos indirectamente recursivos en, 322
verificación de tipo en, 260
Lenguaje C++
ambiente de ejecución del, 345, 393-394e
ámbito de resolución del operador en, 305
coacción en, 333
mecanismo de paso por referencia en, 383-384
mecanismos de paso de parámetros, 382
memoria dinámica para, 375-376
operadores sobrecargados en, 332
tablas de función virtual en, 396*r*
Lenguaje C-Minus, 2, 26-27, 491-500
ambiente de ejecución de la máquina "TM" para, 497-500
convenciones léxicas de, 491-492
gramática de la forma Backus-Naur para, 492-496
programas de muestra en, 496-497
propósito del, 491
proyectos de programación en, 500-501*e*
sintaxis y semántica de, 492-496
Lenguaje de alto nivel, 1
Lenguaje de máquina, 2
Lenguaje ensamblador TM, 25-26
Lenguaje estándar, 16
Lenguaje FORTRAN, 16, 47
declaraciones implícitas en, 299-300
desarrollo de, 3
descubrimiento de ciclo en, 475
preprocesador para, 5
problemas de buscar hacia adelante con, 47
Lenguaje FORTRAN77
ambiente de ejecución para, 17, 345, 349-352, 388*e*, 395*e*, 396*r*
asignación de memoria en, 347
asignación de variables en, 260, 391*e*
equivalencia estructural en, 344*r*
mecanismo de paso de parámetros, 382
mecanismo de paso por referencia, 383-384
programa de muestra en, 350-352
registro de activación de procedimiento en, 348
Lenguaje fuente, 1
Lenguaje Haskell, 140*e*, 344*r*, 386
Lenguaje LISP, 5, 139*e*, 140*e*, 489*r*
ambiente completamente dinámico para, 374
ambiente de ejecución para, 17
análisis semántico en, 257
información de tipos dinámicos en, 313
memoria dinámica para, 375
registro de activación de procedimiento en, 348
verificación de tipos en, 260
Lenguaje Miranda, evaluación descuidada en, 386
Lenguaje ML, 140*e*
ambiente completamente dinámico para, 374
declaraciones colaterales en, 307
equivalencia de declaración en, 328-329
equivalencia de nombre en, 344*r*
tipos de datos recursivos en, 321-322
Lenguaje Modula-2, 141*e*, 489*r*
ámbitos de procedimientos y funciones en, 308
comentarios en, 45
Conversión de tipo en, 332
declaración antes del uso en, 301-302
declaraciones constantes en, 300
paso de procedimientos como parámetros en, 373, 374
tipo de datos función en, 319-320
Lenguaje natural, 3
Lenguaje objetivo, 1
Lenguaje Pascal,
ambiente de ejecución basado en pilas para, 365-370, 370-373, 389-391*e*
ambiente de ejecución para, 17, 345
ámbitos anidados en, 303*ilus.*
análisis semántico en, 257
arreglo con subíndices en, 418-419
asignación de memoria en, 347
comentarios en, 44-45

constructores tipo arreglo en, 315
declaración antes del uso en, 301-302
declaraciones al mismo nivel de anidación en, 306
declaraciones de constantes en, 300
declaraciones de tipo en, 299
declaraciones de tipo explícito e implícito en, 314
declaraciones forward (hacia adelante) en, 308
declaraciones y nombres de tipo en, 320
equivalencia de declaración en, 328
estructura de bloques de, 302
forma Backus-Naur para, 102
funciones y procedimientos anidados en, 303
generación de código P y, 404
información de tipo estático en, 313
lenguaje TINY y, 22, 133-134
mecanismo de paso por valor de, 382
mecanismos de paso de parámetros, 382
mecanismos de paso por referencia de, 383
operaciones de estructuras dinámicas (heap) en, 377
operadores de sobrecarga en, 332
registro de activación de procedimiento en, 348
tipo de datos apuntador en, 319
tipo de datos de función en, 319
tipo de datos de unión en, 318-319
tipos enumerado y de subrango en, 315
verificación de tipos en, 260
Lenguaje Scheme, declaraciones colaterales en, 307
Lenguaje Smalltalk, 17, 257, 375, 396*r*
Lenguaje TINY, 2, 22-26, 486-487*e*, 488-489*e*
ambiente de ejecución para, 386-388, 387*ilus.*
análisis semántico de, 334-338
analizador semántico para, 259, 336-338, 343*e*
compilador TINY y, 22
expresiones booleanas en, 22, 133-134, 192*e*
forma Backus-Naur extendida para, 180-181
generador de código ensamblador para, 398
generador de código para, 459-467
gramática libre de contexto para, 133-134
implementación de un analizador léxico para, 75-79
programa de muestra en, 78, 137-138*ilus.*
sintaxis de, 133-138
tipos en, 334-335
tabla de símbolos para, 335-336
Lenguaje TINY, analizador sintáctico ascendente para, 197-198
Lenguaje TINY, analizador sintáctico recursivo descendente para, 180-182
Lenguajes anfitriones, 18-21
Lenguajes de formato libre, 46
Lenguajes de programación. *Véase también* Lenguaje Ada; lenguajes tipo Algol; Lenguaje ensamblador; Lenguaje BASIC; Lenguaje C; Lenguaje C-Minus; Lenguaje FORTRAN; Lenguaje Haskell; Lenguaje de alto nivel; Código intermedio; Lenguajes; Lenguaje LISP; Lenguaje de máquina; Lenguaje Miranda; Lenguaje ML; Lenguaje Pascal; Código P; Lenguaje Smalltalk; Lenguaje fuente; Lenguaje objetivo; Código en tres direcciones; Lenguaje TINY; lenguaje ensamblador TM
estructura de bloques de, 302
estructura léxica de, 31
gramáticas para, 101-102
implementación de atributos y gramáticas con atributos para, 259-295
tablas de símbolos para, 295-313
tipos de datos y verificación de tipos para, 313-334
Lenguajes estructurados en bloques, 302
Lenguajes inherentemente ambiguos, 133
Lenguajes libres de contexto, 128-133
Lenguajes monomórficos, 333
Lenguajes orientados a objetos, 333, 375-376
Lenguajes polimórficos, 333
Lenguajes tipo Algol, ambientes de ejecución para, 17
Lenguajes. *Véase también* Lenguajes de programación
definidos por gramáticas, 100-106
generados por expresiones regulares, 34-35, 91*e*
generados por gramáticas libres de contexto, 129
para autómatas finitos, 49, 57-58
unión de, 35-36
Lesk, Mike, 4
Letras, 44, 47-48, 50. *Véase también* Caracteres; Metacaracteres; Cadenas
Lex, 4, 32, 41, 81-91, 89*t*, 94*r*
Lexemas, 32-33
LIFO (último en entrar-primero en salir), protocolo, 347
Ligadores, 5
Linealización, de árboles sintácticos, 398, 399-400
`LineList`, 336
LINGUIST, 344*r*
Listas lineales, 80, 296
Literales, 8, 43, 347
Literales de cadena. *Véase* Literales
Llamadas a procedimiento recursivo, ambiente de ejecución basado en pilas para, 352-353
Llamadas, 486*e*. *Véase también* Funciones; Llamadas de procedimiento; Procedimientos
código intermedio para, 437-439
código para compilador C de Sun, 452-453
código para el Compilador Borland C, 447-448
procedimiento para generación de código, 439-443
Llamadas de procedimiento, 486*e*. *Véase también* Argumentos; Llamadas; Funciones; Parámetros
código del compilador C de Borland para, 447-448
código del compilador C de Sun para, 452-453
con número variable de argumentos, 361, 391*e*, 486*e*
en bloques básicos de código, 475*n*
generación de código de, 436-443
optimización de código mediante eliminación de innecesarias, 470-471
organización de la pila de llamadas para, 352-353
sin parámetros, 215-216
Llaves ({ })
en Lex, 82-83
operación de repetición y, 123-124
para comentarios en el lenguaje TINY, 75 75
reglas de la forma Backus-Naur extendida y, 146-147

Macro `ECHO`, 86, 87
Macro `yyclearin`, 247
Macro `yyerrok`, 247, 248, 254*e*
Macros, 5
Manejadores, de formas de sentencias por la derecha, 201, 252*e*
Manejo de errores, 7*ilus.*, 8, 18, 183-189
Conjuntos Primero y Siguiente en, 152
en el analizador lexicográfico de TINY, 76-77
en verificadores de tipos, 331
en Yacc, 230, 234-235
para números octales, 267
por analizadores sintácticos, 96-97
Manuales de referencia de lenguaje, 16-17
Máquina P, 404-405
Máquina TINY (TM), 22, 486-487*e*, 488-489*e*. *Véase también* Generador de código ensamblador para simulador TM, 398, 453-459 22, 486-487*e*, 488-489*e*
arquitectura básica de, 454-457
compilador TINY y, 25-26
conjunto completo de instrucciones de, 455*t*
listado para, 545-557

Máquinas de estado finito. *Véase* Autómatas finitos
Marca DEBUG, código inaccesible y, 469-470
Marca EchoSource, para el compilador TINY, 25
Marca NO_ANALYZE, 24, 464
Marca NO_CÓDIGO, 24, 464
Marca NO_PARSE, 24, 464
Marca TraceAnalyze, 25, 335, 338
Marca TraceCódigo, 25, 461, 465
Marca TraceParse, para el compilador TINY, 25
Marca TraceScan, para el compilador TINY, 25
Marca YYPARSER, 243, 245
Marcas o banderas, para el compilador TINY, 24-25
Marcos de pila, 348
Mecanismo de evaluación retardada, 382, 385
Mecanismo de nombrado de tipo typedef, del lenguaje C, 320-321
Mecanismo de paso por nombre, 382, 385-386, 395*e*
Mecanismo de paso por referencia, 382, 383-384
Mecanismo de paso por texto, 395*e*
Mecanismo de paso por valor, 382-383, 391*e*
Mecanismo de paso por valor-resultado, 382, 384-385
Mecanismos de definición, en Yacc, 242*t*
Mecanismos de definición y nombres internos de Yacc, 242*t*
Mecanismos de paso de parámetros, en ambientes de ejecución, 381-386
Mecanismos para manejo de excepciones, 18
Memoria de acceso aleatorio (RAM, por sus siglas en inglés), organización de la, 346
Memoria de instrucción, para la máquina TM , 454
Memoria de registro (RM, por sus siglas en inglés), instrucciones de, para la máquina TM, 454, 455-456, 461
Memoria dinámica, 373-381, 396*r*
  administración automática de estructuras dinámicas para, 380-381
  administración de estructuras dinámicas para, 376-380
  en lenguajes orientados a objetos, 375-376
  para ambientes completamente dinámicos, 373-375
Memoria, para la máquina Tiny, 454-457
Memorización, 386, 395*e*
Menos unario, 140*e*, 193*e*
Mensajes de error, en Yacc, 254*e*
Metacarácter de signo de interrogación (?), 43
Metacarácter de signo numérico (#), en pilas por valor, 167
Metacarácter más (+), 42
Metacarácter punto (.), 42, 201-202
Metacaracteres, 35-37, 42-43, 81-82. *Véase también* Caracteres; Letras; Cadenas
Metacaracteres "Not". *Véase* Carácter Caret (^); Carácter tilde (~)
Metalenguajes, 268
Metasímbolos, 35, 98-99
Método basado en reglas, de construcción de gráficas de dependencia, 276
Método de árbol de análisis gramatical, de construcción de gráficas de dependencia, 276
Métodos, 320, 375
Métodos algorítmicos controlados por tabla, 63, 408-409, 409*t*
Modificadores del ámbito, 308
Modos de direccionamiento, 12
Módulo entero, 193*e*. *Véase también* Función mod; Función módulo
Movimiento de código, 475
Multiplicación, 117-118, 118-119, 470

Naur, Peter, 98
Niveles de anidación, 368-369, 399
No asociatividad, 117-118
no terminal *command*, en Yacc, 229, 234, 253*e*
No terminal *factor*
  en especificaciones de Yacc, 229-230
  en listados de Yacc, 234
No terminal *term*, 229, 234
No terminales
  apilamiento de, 152-153
  conjuntos Primero de, 168-169, 170-173
  conjuntos Siguiente de, 173-174
  eliminación de recursión por la izquierda y, 158-160
  en análisis sintáctico ascendente, 198-199
  en análisis sintáctico LR(1), 217-218
  en árboles de análisis gramatical, 107-108
  en diagramas de sintaxis, 125
  en especificaciones de Yacc, 229-230
  en gramáticas libres de contexto, 102, 128-129
  en listados de Yacc, 233-234*ilus.*, 234-235, 237*ilus.*
  en tablas de análisis sintáctico LL(1), 154-155, 178-180
  inútiles, 191*e*
  nulificables, 169
Nodo CallK, 440, 441, 441-442*ilus.*, 442
Nodo ConstK, 441, 441-442*ilus.*
Nodo FnK, 440, 441, 441-442*ilus.*, 442
Nodo IdK, 441, 441-442*ilus.*
Nodo ParamK, 440, 441, 441-442*ilus.*, 442
Nodo PlusK, 441, 442, 441-442*ilus.*
Nodo PrgK, 440, 441, 441-442*ilus.*
Nodos
  de árboles de activación, 356
  de árboles de análisis gramatical y árboles sintácticos, 9, 107-110, 129
  de árboles sintácticos para definiciones y llamadas de función, 440-441
  de estructura de árbol sintáctico de TINY, 134-138, 135*ilus.*
  de estructura sintáctica de TINY, 243-244, 462
  de gráficas de dependencia, 275-276, 275*ilus.*
  de gráficas de flujo, 475-477
  en tabla de símbolos de TINY, 337-338
Nodos de tipo, 330
Nodos hoja, 9, 107-108, 110, 129
Nodos interiores, de árboles de análisis gramatical, 107-108, 129
Nodos raíces
  de árboles de análisis gramatical, 107, 129
  de árboles sintácticos, 110
  de árboles sintácticos para llamadas y definiciones de función, 440
  de gráficas de dependencia, 275*ilus.*, 275*n*, 275-276
Nombres, 37-38
Nombres de tipos base, 328
Nombres internos, en Yacc, 242*t*
Notación punto, 317
Numeración, de nodos de árbol de análisis gramatical, 108-109
Numeración postorden, de nodos de árbol de análisis gramatical, 108-109
Numeración preorden, de nodos de árbol de análisis gramatical, 108
Números binarios, como cadenas, 41-42
Números con base. *Véase también* Números binarios; Números decimales; Números hexadecimales; Números; Números octales
  sin signo, 273-275, 282-283, 285-288
Números de línea, agregar al texto, 84-85
Números de punto flotante, 488*e*
  asignación de memoria para, 347
  autómatas finitos determinísticos para, 52*ilus.*
  en código Yacc, 241-242
  en expresiones aritméticas, 283-284
  gramática de, 339*e*
Números decimales
  conversión a números hexadecimales, 87-88
  gramática de, 266-268, 273-275, 339*e*
  notación para, 43
Números hexadecimales, conversión de números decimales a, 87-88
Números naturales, 51-52, 130

con signo, autómata finito determinístico para, 52
Números octales, gramática de los, 266-268, 273-275
Números primos, tamaños de tabla de cálculo de dirección y, 297
Números sin signo. *Véase también* Números
con base, 262-264, 266-268, 271-272, 276-277
gramáticas de, 276-277
Números. *Véase también* Expresiones aritméticas; Símbolos aritméticos; Números binarios; Números decimales; Dígitos; Números de punto flotante; Números hexadecimales; Expresiones aritméticas enteras; Enteros; Números de línea; Números naturales; Números octales
conversión de decimal a hexadecimal, 86
en expresiones regulares, 43-44, 131
en lenguaje TINY, 75
gramáticas basadas en, 273-275, 282-283, 285-286, 287-288
gramáticas de, sin signo, 262-264, 266-268, 271-272, 276-277

Objetos, 375, 376, 376*ilus.*
Opción `-d`, para Yacc, 230-231
Opción descriptiva ("verbose"), para Yacc, 231-232, 231*ilus.*
Opciones de compilador e interfaces, 17
Operación *allocate*, 377
Operación de asignación, 43, 75, 76-77
Operación de concatenación, 35, 36-37, 92*e*, 130
ambigüedad no esencial con, 123
con código P, 408
con gramáticas libres de contexto, 97
en expresiones regulares, 38-41, 41-43
implementación para autómatas finitos no determinísticos, 65
precedencia de, 37, 38
Operación de referenciación, 319
Operación de repetición, 35, 36-37
código para, 145-151
con gramáticas libres de contexto, 97
en análisis sintáctico LL(l), 157
en expresiones regulares, 38-41, 41-43
exclusión de la cadena vacía, 41-42
implementación para autómatas finitos no determinísticos, 66-67
notación de la forma Backus-Naur extendida para, 123-124
precedencia de, 37, 38
recursión y, 104-105
Operación `delete`, 377
Operación `dispose`, 377
Operación `div`, 283-284
Operación `free`, 377-380, 379*ilus.*, 396*r* 396*r*
recolección de basura y, 380
Operación *getBaseTypeName*, 328
Operación *getTypeExp*, 327
Operación `malloc`, 377-380
código C para, 379*ilus*
código para, 396*r*
recolección de basura y, 380-381, 395-396*e*
Operación `subs`, 422
Operaciones
en expresiones regulares, 35
optimización mediante la reducción de costos, 470
procedencia de, 37, 38-41, 117-118, 118-119
Operaciones asociativas, 123
Operaciones costosas, optimización mediante la reducción de las, 470
Operaciones de desbordamiento de registros, 469
Operaciones del compilador en la etapa final, 15
Operaciones del compilador en la etapa inicial, 15
Operaciones innecesarias, 469-470
Operador *(op)*
en árboles de análisis gramatical, 107-109
en gramáticas ambiguas, 114-115
en gramáticas libres de contexto, 98-100, 100-101
en sintaxis abstracta, 110-111
Operador `and`, 139*e*, 192*e*, 432-433
Operador `Assign`, 410-411
Operador de resolución de ámbito, en lenguaje C++, 305
Operador `else`, 120-122. *Véase también* Problema del else ambiguo, 120-122
Operador `not`, 139*e*, 192*e*
Operador `or`, 139*e*, 192*e*, 432-433
Operador `Plus`, 410-411
Operador porcentual (%), para módulo entero en lenguaje C, 297, 353-354
Operadores binarios, ambigüedad no esencial con, 123
Operadores booleanos, 255-256*e*
Operadores de comparación, 255*e*, 463. *Véase también* Expresiones booleanas
Operadores de dirección unitarios, 485*e*
Operadores, en lenguaje TINY, 22
Operadores sobrecargados, 332
Operation `new`, 377
Optimización, 489*e*, 489-490*r*
al mantener temporales en registros, 481-482, 483*ilus.*, 484*ilus.*
al mantener variables en registros, 482, 483*ilus.*, 484*ilus.*
al predecir el comportamiento del programa, 472
con gráficas de flujo, 475-477, 476*ilus.*
de código ejecutable, 397-398
de código fuente, 10-11, 345
efectos sobre el tiempo de compilación de la, 468
efectos sobre el tiempo de ejecución del programa de la, 468
en análisis semántico, 258
mediante el análisis estadístico de las ejecuciones del programa, 472
Optimización de salto, 470
Optimizaciones a nivel de fuente, 472
Optimizaciones a nivel de objetivo, 472-473
Optimizaciones de incorporación de constantes, 10, 28*e*, 260, 470, 489*e*
Optimizaciones "de mirilla", 472
Optimizaciones globales, 473, 474-475
Optimizaciones interprocedimientos, 473, 474
Optimizaciones locales, 473-474
Optimizador de código objeto, 12
Optimizadores de código fuente, 10-11
Orden de evaluación para atributos, 271, 276-277, 343*r*

Palabra reservada `do`, 43
Palabra reservada `if`, 31, 43
Palabra reservada `while`, 31, 43
Palabras clave, 31, 43, 122. *Véase también* Palabras reservadas
Palabras clave agrupadas, 122
Palabras reservadas, 32, 43, 45-46. *Véase también* Palabras clave
contra identificadores, en analizadores léxico, 80
expresiones regulares para, 44
para lenguaje TINY, 75*t*, 80
tablas de, 80
Palabras reservadas `end if`, 122
Paquete de compiladores Gnu, 81, 94*r*, 489*r*
para análisis semántico, 258-259
para análisis sintáctico, 95-96, 154-157, 155*ilus.*
para análisis sintáctico LR(0), 206-210
para análisis sintáctico LR(1), 220-223
para análisis sintáctico SLR(1), 210-211
para búsquedas generadas espontáneamente, 226
para construcción de subconjuntos, 69, 70-72
para el lenguaje TINY, 335-336
para factorizar por la izquierda una gramática, 163-166, 164*ilus.*
para generación de código práctica, 410-411
para gramática de sentencia "if", 176-177, 179
Parametrización de tipos, 333-334
Parámetro `label`, en el procedimiento genCode, 435*ilus.*, 436

Parámetro out, 384
parámetros in, 382
Parámetros in out, en lenguaje Ada, 384-385
Parámetros *synchset*, 186
Parámetros. *Véase también* Argumentos; Llamadas; Funciones; Llamadas de procedimiento
atributos como, 285-286
en ambientes de ejecución basados en pilas, 356-357
en llamadas de función y procedimientos, 381-386, 394-395*e*
paso de procedimientos como, 370-373, 371*ilus.*, 372*ilus.*, 373*ilus.*
Paréntesis, 102. *Véase también* Precedencia de operadores
analizador sintáctico ascendente para balance, 199*t*
analizador sintáctico descendente para balance, 152-154
analizador sintáctico LR(0) para balance, 208, 208*ilus.*, 209*t*
analizador sintáctico SLR(1) para balance, 212-213, 212-213*t*
autómata finito determinístico de elementos LR(1) para balance, 218-220, 220*ilus.*
autómata finito determinístico para la gramática de balance, 204-205, 205*ilus.*
autómata finito no determinístico para la gramática de balance, 204, 204*ilus.*
balanceados, 105, 154-155, 199
como metacaracteres, 37
como metasímbolos, 99
eliminación de ambigüedades y especificación de precedencia con, 118
gramática de balance, 202
tabla de análisis sintáctico LR(1) para la gramática de balance, 221-222, 222*t*
Pasadas, durante la compilación, 15-16
Pasos de derivación, en gramáticas libres de contexto, 128-129
Patrones, 34, 94*r*
de tipo, 333
Perfiladores, 6
Pila de máquina P, 414-416, 417-418
Pila de registros de activación, 352, 389-391*e*. *Véase también* Pilas de ejecución, 448, 352, 363*ilus.*, 364*ilus.*, 365*ilus.*, 366*ilus.*, 367*ilus.*, 369*ilus.*, 370*ilus.*, 372*ilus.*, 373*ilus.*, 392*e*. *Véase también* Pilas de llamada
Pilas de llamadas, 352, 353-354, 354*ilus.*, 355-356, 356*ilus.*, 358*ilus.*, 359*ilus.*, 360*ilus.*, 361*ilus.*, 362*ilus.*, 389-391*e*. *Véase también* Pilas de tiempo de ejecución; Registros de activación de pila; Pilas de valores
Pilas de valores, 167, 167*t*
en el cálculo de atributos heredados, 291-293
en el cálculo de atributos sintetizados, 290-291, 291*t*
para Yacc, 229-230, 240-241
Pilas del análisis sintáctico, 95, 252*e*
análisis sintáctico ascendente con, 198-199, 200-201
análisis sintáctico LL(1) con, 152-154, 166-167, 186-187
análisis sintáctico LR(0) con, 206-210
análisis sintáctico SLR(1) con, 210-213
con acciones con pila de valores, 167*t*
en el cálculo de atributos heredados, 291-293
en el cálculo de atributos sintetizados, 290-291, 291*t*
en recuperación de errores en modo de alarma, 246-247
producciones de errores en Yacc y, 247-250, 250*n*
recuperación de errores con, 186-187
Pilas. *Véase* Pilas de llamadas; Pilas de análisis sintáctico; Pila de máquina P; Pilas de valores
Portabilidad, de compiladores, 15
Pragmas, 17
Pragmática, 17
Precedencia, 253*e*
Precedencia de operaciones, 37, 38
en expresiones aritméticas, 117-118
en expresiones booleanas, 139*e*
en expresiones regulares, 38-41
en Yacc, 236-238, 238*ilus.*
resolución de ambigüedad con, 118-119
Precedencia en cascadas, 118-119
Prefijos viables, 201
Preludio estándar, para la máquina TM, 461-462
Preprocesadores, 5
Preprocesador Ratfor, 5, 30*r*
Primer principio de análisis sintáctico LALR(1), 224
Principio de la subcadena más larga, 46
Principio de subtipo, 333
Principio del "máximo bocado", 46
Problema de errores de cascada, 183, 188-189
Problema del análisis sintáctico, 3
Problema del else ambiguo, 120-122, 139*e*, 142*r*
análisis sintáctico LL(1) y, 156
definición, 121
para analizadores sintácticos SLR(1), 213-215
reglas de no ambigüedad para, en Yacc, 235-236
Problemas de fase, 473
Procedimiento buildSymTab, 337, 338
Procedimiento *checkinput*, 186, 186*n*
Procedimiento checkNode, 337, 338
Procedimiento closureEx, 371-373
Procedimiento *CombinedEval*, pseudocódigo para, 283
Procedimiento copyString, 80, 182, 244
Procedimiento *delete*, 288, 295, 295*n*
ámbito dinámico y, 306
ámbitos anidados y, 303, 304
declaraciones y, 298-299
estructura de tablas de símbolos y, 295-296
para tablas de símbolos de TINY, 33
Procedimiento emitCódigo, 411, 435*ilus.*, 436
Procedimiento *error*, 144-145
Procedimiento *EvalBasedNum*, pseudocódigo para, 286
Procedimiento *EvalNum*, pseudocódigo para, 286
Procedimiento *EvalType*, 280-282, 289
Procedimiento *EvalWithBase*, pseudocódigo para, 282-283, 285-286, 287, 342*e*
Procedimiento *exp*, 145, 150-151
Procedimiento *factor*, 144-145
Procedimiento fgets, 78
Procedimiento genCode, 424-425*ilus.*, 486*e*
para llamadas y definiciones de función, 441-442*ilus.*
para sentencias de control, 435*ilus.*
procedimiento *genCode*, 410-411, 412*ilus.*, 413*ilus.*
Procedimiento genLabel, 435*ilus.*, 43
Procedimiento getchar, 88
Procedimiento getToken, 33-34, 77, 78, 90-91, 96, 181, 244
procedimiento input, en Lex, 88
procedimiento *insert*, 288-289, 295, 310, 329
ámbitos anidados y, 303
declaraciones y, 298-299
estructura de tablas de símbolos y, 295-296
para tablas de símbolos de TINY, 33
procedimiento insertNode, 337, 337-338
Procedimiento *lookup*, 288, 295, 306, 310, 329, 332
ámbitos anidados y, 303, 304
estructura de tabla de símbolos y, 295-296
para la tabla de símbolos de TINY, 33
Procedimiento *makeTypeNode*, 330
Procedimiento *match*, 144-145
Procedimiento match, 181, 187-188, 194
Procedimiento NewExpNode, en el analizador sintáctico TINY, 182, 243
Procedimiento newStmtNode, en el analizador sintáctico TINY, 182, 24
Procedimiento nullProc, 337
Procedimiento *PostEval*, 277-278, 278*ilus.*
Procedimiento postproc, 337

Procedimiento *PreEval*, pseudocódigo para, 279
Procedimiento preproc, 337
Procedimiento printSymTab, 336, 338
Procedimiento printTree, en el analizador sintáctico de TINY, 182
Procedimiento putchar, 88, 89
Procedimiento QUADMEAN, 350-352
Procedimiento reservedLookup, para el analizador léxico de TINY, 78
Procedimiento *SetBase*, pseudocódigo para, 287
Procedimiento st_insert, 336, 338
Procedimiento stmt_sequence,
en el analizador sintáctico de TINY, 181, 243-244
recuperación de errores y, 189
Procedimiento sum, 362-363
Procedimiento swap, 333-334
Procedimiento syntaxError, en analizador sintáctico TINY, 181, 194*e*
Procedimiento *term*, 145
Procedimiento typeCheck, 337
Procedimiento *type-error*, 331
Procedimiento typeError, 338
Procedimiento ungetNextChar, 78
Procedimiento yyerror, 230, 244
Procedimiento yylex, 81, 85
en Yacc, 230, 244
Procedimiento yyparse, 230, 243, 244
Procedimientos locales
ambientes de ejecución basados en pilas sin, 353-365
ambientes de ejecución basados en pilas con, 365-370
Procedimientos. *Véase también* Argumentos; Secuencias de llamado; Llamadas; Funciones; Parámetros; Valores devueltos; Secuencia de retorno
ambiente de definición para, 367
anidados, 303
código objeto de, 259, 260
para generación de código con referencias de arreglos, 421-423
pasados como parámetros, 370-373, 371*ilus.*, 372*ilus.*, 373*ilus.*
secuencias de llamado de, 349, 359-361, 369-370
Procesadores 80 × 86, 398, 443-448
Procesadores Intel 80 × 86, generación de código para, 443-448
Producciones, 101, 128-129
Producciones de error, 247-250, 253-254*e*
Producciones ε, 105-106, 159
cálculo de atributos heredados, 292-293
conjuntos Primero y, 168-169, 170, 171-172
conjuntos Siguiente y, 151-152, 173-174
Producto cartesiano, 49, 317
Producto cruz, 49
Program gcd, 353-354, 485*e*, 486*e*
Programa de cálculo simple, definición de Yacc para, 228*ilus.*
Programa de máximo común divisor. *Véase también* Programa gcd
Programa Grep (global regular expression print, impresión de expresión regular global), 91*e*, 91*n*
Programa nonLocalRef, 366
Programas, 101-102
efectos de las técnicas de optimización sobre el tiempo de ejecución de los, 468
predicción del comportamiento de los, 472
Programas almacenados, 2
Programas de interfaz, 1
Programas de prueba estándar, para compiladores, 16
Programas TINY, 400-402, 401*ilus.*, 406*ilus.*, 485*e*
Propagación de búsquedas hacia adelante, 225-226
Propagación de constantes, 28*e*, 470
Pruebas "case" anidadas, 61
Pruebas de inducción, 40
Pseudorregistros, optimización de código con, 473
Pseudotoken de espacio en blanco, 45, 46, 47, 75
Pseudotoken error, en producciones de error, 247-250
Pseudotokens, 44
Pseudovariables, en código Yacc, 229
Punto de entrada, en una función o procedimiento, 437
Punto de retorno, desde una función o procedimiento, 437
Puntos de ruptura, 6
Puntos fijos, de funciones, 130

Rastreo, 31-94. *Véase también* Análisis léxico
autómatas finitos para, 47-80
convenciones de Lex para, 81-91
expresiones regulares en, 34-47, 64-80
proceso de, 32-34
Recolección de basura, 374, 395-396*e*, 396*r*
algoritmos para, 17
tipos de, en administración automática de estructuras dinámicas, 380-381
Recolección de basura con marcador y barrido, 380-381, 396*r*
Recolección de basura de interrupción y copia, 381
Recolección de basura en dos espacios, 381, 396*r*
Recolección de basura por generaciones, 381
Reconocedores, 183
Reconocimiento, 31, 43
autómata finito para, 48*ilus.*
contra palabras reservadas, en analizadores léxicos, 80
definición regular para, 47-48
en gramáticas LL(1), 165-166
en lenguaje TINY, 133-134
en tablas de símbolos, 13
espacio de asignación para, 80
expresiones regulares para, 44
Recorrido postorden, 108-109, 277-278, 278*ilus.*, 410-411
Recorrido Preorden/en orden, 279, 410-411
Recorrido recursivo, de árboles sintácticos en generación de código, 410-411
Recubrimiento de procedimiento, 470-471
Recuperación de errores, 254*e*, 256*r*. *Véase también* Recuperación de errores en modo de alarma, 256*r*
en análisis sintáctico descendente, 183-189
en análisis sintáctico LR(0), 209-210
en analizadores sintácticos ascendentes, 245-250
en analizadores sintácticos LL(1), 186-187
en analizadores sintácticos recursivos descendentes, 183-185
en el analizador sintáctico TINY, 187-189, 194*e*
en el compilador TINY, 250
en Yacc, 247-250
principios generales de, 183
Recuperación de errores de análisis sintáctico, 96-97
Recuperación de errores en modo de alarma, 192*e*, 196*r*, 256*r*
en analizadores sintácticos ascendentes, 245-247
en analizadores sintácticos LL(1), 186-187
en analizadores sintácticos recursivos descendentes, 183-185
en el analizador sintáctico de TINY, 187-189
Recursión de cola, optimización de código por eliminación de, 470-471
Recursión infinita, 102-103
Recursión por la derecha
eliminación de la recursión por la izquierda y, 158
Recursión por la izquierda
analizadores sintácticos LL(*k*) y, 180
eliminación de, en análisis sintáctico LL(1), 157-162, 167
en análisis sintáctico ascendente, 198
general, eliminación de, en análisis sintáctico LL(1), 159*t*, 159-160
Recursión por la izquierda inmediata, 104*n*, 158-159
simple, eliminación de, en análisis sintáctico LL(1), 158

Redirección de objetivo, 29*e*
Reducción en intensidad, optimización de código, 470
Referencias colgantes, 373-374
Referencias de apuntador
cálculo de; en generación de código, 423-428
código compilador C de Borland para, 445-446
código Compilador C de Sun para, 450-451
Referencias de arreglos
cálculo de, en la generación del código, 417, 418-423
código para el Compilador Borland C, 444-445
código para el Compilador Sun C, 449-450
Registro de apuntador base, en el código para expresiones aritméticas, 443-444
Registros, 481-482, 483*ilus.*, 484*ilus.*
Registros de activación
con arreglo de visualización, 392*e*
con datos de longitud variable, 362-363, 362*ilus.*, 391*e*
con declaraciones anidadas, 364-365, 364*ilus.*, 365*ilus.*
con referencias no globales y no locales, 365-366, 366*ilus.*
con temporales locales, 363, 363*ilus.*
con vínculos de acceso, 367*ilus.*, 369*ilus.*, 370*ilus.*, 372*ilus.*, 373*ilus.*
en ambientes de ejecución basados en pilas, 352-353, 354, 354*ilus.*, 355-356, 356*ilus.*, 358, 358*ilus.*
organización de, 348
para FORTRAN77, 350
para secuencias de llamadas a procedimientos, 359-361, 359*ilus.*, 360*ilus.*, 361*ilus.*
parámetros en, 381-382
Registros de activación de procedimiento. *Véase* Registros de activación
Registros de acumulador, 443-444, 460-461
Registros de entrada, en la SparcStation, 453, 453*n*
Registros de procesador, 449*n*, 449-450, 454-457. *Véase también* Área de registro
Registros de salida, en la SparcStation, 453, 453*n*
Registros de token, 33
Registros variantes, 318
Registros. *Véase también* Constructores del tipo de registros de activación para, 317
Regla de anidación más próxima, 121, 156, 157*t*
para estructuras de bloque, 302, 303
Regla de fuera de lugar, 46
Reglas de contexto, 131-132
Reglas de copia, 292-293
Reglas de no ambigüedad, 45-46, 116-118. *Véase también* Ambigüedad; Gramáticas ambiguas
definición del problema de else ambiguo, 116
para conflictos de análisis sintáctico SLR(1), 213-215
para conflictos de análisis sintáctico de Yacc, 235-238
para sentencias "if", 120-122, 156, 157*t*
Reglas de recursividad
en gramáticas libres de contexto, 95, 97, 129-130
gramáticas con, 102-103
operación de repetición y, 104-105
Reglas de semántica, 258, 261*n*. *Véase también* Ecuaciones de atributos
definición, 261-262
para expresiones aritméticas, 269*t*, 270*t*, 284*t*, 292-293, 294
para expresiones aritméticas enteras, 264*t*
para gramáticas de declaración de variable, 266*t*, 288*t*
para gramáticas de números con base, 288*t*
para gramáticas de números sin signo, 262-264, 263*t*, 266-268, 276*t*
Reglas gramaticales
atributos sintetizados y, 277
como ecuaciones, 129-130
contra derivaciones, 101
derivaciones de, 102-106
en archivos de especificación de Yacc, 227, 229
gráficas de dependencia asociadas para, 271
para expresiones aritméticas, 269*t*, 270*t*, 272-273, 284*t*, 292-293, 294
para expresiones aritméticas de enteros, 264*t*, 264-265
para gramáticas basadas en números, 273-275, 288*t*
para gramáticas de declaración de variable, 266*t*, 288*t*
para gramáticas de números sin signo, 262-264, 263*t*, 266-268, 271-272, 276*t*
para gramáticas libres de contexto, 95, 97-98, 98-100, 128-129, 129-130
sensibles al contexto, 132
traducción de, para la forma Backus-Naur extendida, 144-151
Reglas recursivas por la derecha, 104-105, 123-124, 125
Reglas recursivas por la izquierda, 104-105, 123-124
Rehospedaje, 29*e*
Reparación de errores, 96-97, 183
Representación de árbol con hermano a la derecha e hijo extremo izquierdo, 114
Representación intermedia (IR, por sus siglas en inglés), 11, 398
Resolución de colisión, en tablas de cálculo de dirección, 296, 297*ilus.*
Retroseguimiento (hacia atrás), 46-47, 53, 58, 63-64, 192*e*
Rutina `parse`, 181, 188, 244
Rutinas de usuario, en Lex, 83

Sección de definiciones, 83-84, 90, 227-230
Sección de reglas, 83-84, 227-230
Sección de rutinas auxiliares
de archivos de entrada en Lex, 83-84
de archivos de especificación en Yacc, 227-230
Sección de rutinas de usuario, de los archivos de entrada de Lex, 83-84
Secuencia de retorno, 349
para FORTRAN77, 350
Secuencias de llamada, 349, 359-361, 369-370, 437-439
Secuencias de sentencias
Cálculo de conjuntos Primero para, 173
Cálculo de conjuntos Siguiente para, 177
construcción de tablas de análisis sintáctico LL(1) para, 179-180
factorización izquierda de, 163, 164
gramáticas para, 106, 113-114
Segmentos de código en línea recta, optimización de, 473-474
Segundo principio de análisis sintáctico LALR(1), 224
Selección de instrucción, optimización de código a través de una apropiada, 471
Selecciones de producción, en tablas de análisis sintáctico LL(1), 154
Semántica, 16, 492-496
Semántica de sintaxis dirigida, 258
Semántica del menor punto fijo, 130
Semántica denotacional, 16, 142*r*
Semántica dinámica, 9-10,
Semántica estática, 9-10,
Sentencia `AssignK`, 338
Sentencia `break`, 434
Sentencia `IfK`, 338
Sentencia `ReadK`, 338
Sentencia `RepeatK`, 338
Sentencia vacía, en lenguaje TINY, 133-134
Sentencia Write, en lenguaje TINY, 133-134, 134-137, 243
Sentencia `WriteK`, 338
Sentencias, 101, 129
Sentencias Case, 60, 485*e*, 486*e*
Sentencias compuestas, 302, 364-365
Sentencias de asignación
en gramáticas LL(1), 165-166
en lenguaje TINY, 133-134, 134-136, 136*ilus.*, 243-244
incluyendo gramática simple, 215-216, 222-223
verificación de tipos de, 331

Sentencias de control. *Véase también* Sentencias de ruptura; Sentencias Case; Sentencias Goto; Sentencias If; Expresiones de prueba; Sentencias While
Compilador C de Borland código para, 446
Compilador C de Sun código para, 451-452
en lenguaje TINY, 22
generación de código para, 428-436
Sentencias de lectura, en el lenguaje TINY, 133-134, 134-137
Sentencias de ruptura, 428, 431-432
Sentencias `#define`, en código de salida Yacc, 229
Sentencias Do-while, 485*e*
Sentencias For, 485*e*
Sentencias goto, 428, 475
Sentencias "if" anidadas, 485*e*
Sentencias "If". *Véase también* Expresiones booleanas; Instrucciones de salto condicional
análisis sintáctico LL(1) de, 156, 156*t*, 157*t*
análisis sintáctico SLR(1) de, 213-215, 215*t*
anidadas, 485*e*
árboles de análisis gramatical para, 112-113
autómata finito determinístico de elementos LR(0) para, 214*ilus.*
cálculo de conjuntos Primero para, 171-172, 172*t*
cálculo de conjuntos Siguiente para, 176-177
código para, 145-151
Compilador C de Borland código para, 446
Compilador C de Sun código para, 451-452
construcción de tablas de análisis sintáctico LL(1) para, 179
diagramas de sintaxis para, 127-128, 128*ilus.*
en lenguaje C-Minus, 27
en lenguaje TINY, 22, 133-134, 134-136, 136*ilus.*
factorización por la izquierda de, 163, 164-165
generación de código para, 428-432, 429*ilus.*, 433-436, 463-464
gramáticas para, 103-104, 105-106, 111, 112
problema del else ambiguo con, 120-122
simplificación de código TINY para, 482-484
verificación de tipos de, 331
Sentencias Repeat, en lenguaje TINY, 22, 23, 133-134, 134-136, 136*ilus.*
Sentencias Repeat-until, 485*e*
Sentencias Switch, 485*e*, 486*e*
Sentencias While
código del compilador C de Borland para, 446
código del compilador C de Sun para, 451-452
en lenguaje C-Minus, 27
generación de código para, 428-432, 430-432, 431*ilus.*, 433-436, 463
simplificación de código TINY para, 482-484
Sentencias. *Véase también* Sentencias de asignación; Sentencias de ruptura; Sentencias Case; Sentencias de control; Sentencias Goto; Sentencias If; Sentencias Read; Sentencias Repeat; Sentencias While; Sentencias Write
en programas TINY, 133-134
gramáticas de, 103-104, 105-106, 215-216
verificación de tipos de, 330-331
Separadores, 106
Signos de comillas ("), 43-44, 81-82
Signos de punto y coma (;)
como terminadores y separadores de sentencias, 106, 113-114
en el lenguaje TINY, 133-134
en las especificaciones de reglas de Yacc, 227
errores generados por su olvido y por exceso de, 188-189
Símbolo de flecha (→), 97, 98, 99, 227
Símbolo epsilon (ε). *Véase* Conjunto vacío (ε)
Símbolo inicial (Start), 102
apilamiento de, 153
en gramáticas de expresión aritmética simple, 308
en gramáticas libres de contexto, 128-129
para analizadores sintácticos ascendentes, 199, 203
Símbolo Phi (Φ), 35, 64-65
Símbolo Sigma (Σ), 34
Símbolo `YYSTYPE`, 240-241
Símbolos, 34, 129
Símbolos aritméticos, 31, 32, 33, 43, 332
Símbolos de búsqueda hacia delante, 224*n*
Símbolos especiales, 31, 32, 43
para lenguaje TINY, 75*t*, 76
Simulador TM, 26, 457-459
Sincronización tokens, 184-185, 194*e*
Sintaxis, 110-111
análisis semántico y, 257, 258
dependencia del cálculo de atributo de la, 293-295
Sintaxis abstracta, 110-111, 258
Sintaxis concreta, 110-111
Sintaxis dirigida por la semántica, 261*n*
Síntesis, operaciones de compilador para, 15
Sistema de control de código fuente (scs, por sus siglas en inglés) para Unix, 6
Sistema de control de revisión (rcs, por sus siglas en inglés) para Unix, 6
Sistema P de Pascal, 29*e*, 30*r*
Sistemas operativos, interfases de los compiladores con, 17
Sobrecarga, 333
Sólo de registro (RO, por sus siglas en inglés), instrucciones, para la máquina TM, 454-455, 456, 461
SparcStations, 398
generación de código para, 443-448
registros en, 449*n*, 449-450
tamaños de tipo de datos para, 450
SparcStations de Sun. *Véase* Compilador C 2.0 SparcStations Sun
para SparcStations de Sun, 398, 443, 448-453, 485*e*
Subcadenas, 46
Subíndices, para arreglos, 316, 418-419, 421-423, 488*e*
Subtipos, 320*n*
Suspensión, 386
Sustracción o resta, asociatividad y precedencia de, 117-118, 118-119
Symbol `YYDEBUG`, 238-239
synchset, 186

Tabla de símbolos *errtab*, 312-313
Tabla de símbolos *intab*, 312-313
Tabla de símbolos *outtab*, 312-313
Tabla `reservedWords`, para el analizador lexicográfico de TINY, 78
Tablas de análisis sintáctico, 234-235, 235*t*, 245. *Véase también* Tablas de análisis sintáctico LL(1); Tablas de análisis sintáctico LR(0); Tablas de análisis sintáctico LR(1); Tablas de análisis sintáctico SLR(1)
Tablas de análisis sintáctico LL(1), 154-157
algoritmo para construcción de, 168, 177-180
con entradas de recuperación de errores, 186-187, 187*t*
conjuntos Primero y Siguiente para, 168-177
definición de, 154
entradas predeterminadas en, 191*e*
Tablas de análisis sintáctico LR(0), 209*t*, 209-210
Tablas de análisis sintáctico LR(1), 220-223, 222*t*
Tablas de análisis sintáctico SLR(1), 211*t*, 211-213, 212*t*, 213*t*
Tablas de dispersión o cálculo de dirección, 13, 80, 296-298, 343*r*
tamaños crecientes de, 297*n*
Tablas de función virtuales, 375-376, 376*ilus.*, 393-394*e*, 396*r*
Tablas de literales, 7*ilus.*, 8, 14
Tablas de método virtual, 320
Tablas de salto, 428, 486*e*

Tablas de símbolos, 7*ilus.*, 8, 13, 257, 288-289, 295-313, 304*ilus.*, 305*ilus.*, 343*r*. *Véase también* Estructuras de datos
como atributos heredados, 295
definición, 288
estructura de, 295-298
funciones de cálculo de dirección para, 296-298, 298*ilus.*
generación de código intermedio y, 398-399
gramáticas con atributos con, 308-313
declaraciones y, 298-301
operaciones principales para, 295
para palabras reservadas del compilador TINY, 80
reglas de ámbito y estructura de bloques, 301-306
Tablas de transición, 61-63
Técnicas de análisis sintáctico LL, vía análisis sintáctico durante cálculo de atributos, 289-290
Técnicas de mejoramiento de código, 4
Técnicas de optimización, 4, 468-481, 489-490*r*
clasificación de las, 472-475
fuentes de optimización del código en, 469-472
implementación de las, 475-481
naturaleza de las, 468
para el generador del código TINY, 481-484
Temporales locales, 363, 391*e*. *Véase también* Temporales
Temporales. *Véase también* Temporales locales
conservación en registros, 481-482, 483*ilus.*, 484*ilus.*
en código de tres direcciones, 400, 408-409
en código P, 407
prevención del uso innecesario de, 478-479
Teoría de autómatas, autómatas finitos determinísticos de estados mínimos o mínimos en, 73
Terminadores, 106
Terminales. *Véase también* Tokens
en análisis sintáctico LR(1), 217-218
en árboles de análisis gramatical, 107-108
en conjuntos Primero, 168-169
en conjuntos Siguiente, 173-174
en diagramas de sintaxis, 125
en gramáticas libres de contexto, 102, 128-129
en tablas de análisis sintáctico LL(1), 154-155
Texto, agregar números de línea a, 84-85
Tiempo de fijación, de un atributo, 259-260
Tiempo de vida, de declaraciones, 300
Tipo de datos `Align`, 378
Tipo de datos apuntador, constructores de tipo para, 319
Tipo de datos `array`, 314, 315-316
Tipo de datos `boolean`, 314
tipo de datos de unión en, 319
Tipo de datos de unión, constructores de tipos para, 317-319
Tipo de datos estructurados, constructores de tipo para, 317
Tipo de datos `Header`, 377-380, 378*ilus.*
Tipo de datos `int`, 314
Tipo de datos `integer`, 313-314
Tipo de datos `record`, 315
Tipo de datos recursivo, 321-322, 327
Tipo de datos `struct`, 315, 317
Tipo de datos `void`, en el lenguaje C, 315
Tipos de componente, para arreglos, 315-316
Tipos de datos, 313-334, 343-344*r*. *Véase también* Arreglos; Tipo de datos booleano; Funciones; Tipo de datos apuntador; Procedimientos; Campos de registro; Registros; Tipo de datos recursivo; Cadenas; Tipo de datos estructurado; Tipo de datos de unión
constructores y expresiones para, 314-320
declaraciones, nombres y recursión de, 320-322
definición, 313-314
equivalencia de, 322-329
predefinidos, 314-315
tamaños de, en la Sun SparcStation, 450
verificación e inferencia de, 329-334
Tipos de datos indirectamente recursivos, 322
Tipos de datos predefinidos, 314-315
Tipos de índices, para arreglos, 315-316
Tipos de valores booleanos, en lenguaje TINY, 334, 338
Tipos de valores de punto flotante, 240, 265-266, 272-273, 279-281, 292-293, 294-295
mezclados con tipos de valores enteros, 283-284
Tipos de valores enteros
en lenguaje TINY, 334, 338
mezclados con tipos de valores de punto flotante, 283-284
Tipos de valores, en Yacc, 239-241
Tipos de valores enteros, 239-240, 265-266, 292-293, 294-295
Tipos enumerados, 315
Tipos estructurados, 315
Tipos monomórficos, 334
Tipos ordinales, 315
Tipos subrango, 315
Tipos. *Véase* Tipo de datos
Token de fen de archivo, 77
Token `ELSE`, 32
Token `ENDFILE`, 90
Token `EOF`, 77, 90
Token `ERROR`, 77
Token `ID`, 32
Token `IF`, 32
Token `MINUS`, 32
Token `NUM`, 32, 33
Token `other`, 434
Token `PLUS`, 32, 33
Token `THEN`, 32
Construcción de Thompson, 64-69, 92*e*, 94*r*
Tokens, 8, 13, 32-34
acciones DFA y, 53-56
atributos de, 33
categorías de, 43
de lenguaje TINY, 75*t*
definición, 32*n*
en análisis sintáctico ascendente, 198-199
en análisis sintáctico LR(1), 217-218
en archivos de especificación de Yacc, 227, 229, 230
en gramáticas libres de contexto, 97-98, 100
en lenguajes de programación, 43-47
para fin de archivo, 77
sincronización de, 184-185
valores de cadena de, 32-33, 407
Véase también Terminales
Traducción, 7-12, 110
Traducción de sintaxis dirigida, principio de, 110
transformaciones para mejoramiento de código, 468
Transición `any`, 50*ilus.*, 51
Transiciones de error, 50
Transiciones ε, 56-57, 64-69, 217-218
Transiciones LR(1), 217-218, 222-223
Transiciones múltiples, eliminación, 69
Transiciones `other`, 50, 50*ilus.*, 52*ilus.*, 52-53
Transiciones que no consumen recursos de entrada, 62
Transiciones. *Véase también* Transiciones ε; Transiciones de error; Transiciones LR(1); transiciones `other`
en autómatas finitos, 48, 49-51
en autómatas finitos no determinísticos, 63-64
Transportabilidad, 18-21, 21*ilus.*
Triples, para código en tres direcciones, 402-404, 404*ilus.*

Unidades de compilación, 102
Unión, 130
disjunta, 317
de cadenas, 36
de conjunto infinito, 36
de conjuntos, 317-318
de lenguajes, 35-36
de un conjunto de estados, 69-70
Utilería `gperf`, 94*r*

Valor Nil, 321-322
Valores de cadena, de tokens, 32-33, 407
Valores devueltos, atributos como, 285-286
Variable currentToken para el analizador léxico de TINY, 78
Variable *expectedToken*, 145
Variable lineno, 78, 85, 90*n*, 135, 244
Variable listing, 78
Variable savedLineNo, 243, 244
Variable savedName, 243, 244
Variable savedTree, 243-244
Variable source, 78
Variable tmpOffset, 463, 481-482
Variable *token*, 144-145
Variable tokenString, 77, 80, 90
en análisis sintáctico Yacc para lenguaje TINY, 244
Variable yychar, 244
Variable yydebug, 238-239
Variable yylineno, 90*n*
Variable yylval, 230
Variable YYSTYPE, 231, 243
Variables booleanas, en código, 60
Variables de asignación simple, 300
Variables de estado, en código de autómatas finitos determinísticos, 60-61
Variables de inducción, 475
Variables de tipo, 333
Variables estáticas, en ambientes de ejecución basados en pila, 355-356, 356*ilus.*, 358-359
Variables globales, 349-350, 358-359
Variables locales, 349-350, 352, 355-356, 356*ilus.*, 356-357
Variables no globales, en ambientes de ejecución basados en pilas, 365-366, 366*ilus.*
Variables no locales, 358-359, 365-366, 366*ilus.*, 392*e*
Variables. *Véase también* Variables booleanas; Variables de inducción; Pseudovariables; Variables de estado
asignación de memoria para, 259, 260, 300-301
conservación en registros, 482, 483*ilus.*, 484*ilus.*
descriptores de registro y dirección para, 479*t*, 479-481, 480*t*, 481*t*
en lenguaje TINY, 22, 133-134
obtención de definiciones de, 476-477
Ventana de registro, en la SparcStation de Sun, 453*n*, 471*n*
Verificación de tipos, 10, 300, 301, 313, 329-334, 343-344*r*
Verificación de tipos de Hindley-Milner, 4, 30*r*
Verificadores de tipo, 260, 329-331
Vínculos de acceso, 367-370, 369*ilus.*, 370*ilus.*, 372*ilus.*, 373*ilus.*, 392*e*
Vínculos de control, 353, 354*ilus.*, 356*ilus.*, 357*ilus.*, 358*ilus.*, 359*ilus.*, 360*ilus.*, 361*ilus.*, 362*ilus.*, 363*ilus.*, 364*ilus.*, 365*ilus.*, 366*ilus.*, 367*ilus.*, 369*ilus.*, 370*ilus.*, 372*ilus.*, 373*ilus.*, 448, 448*ilus.*
Vínculos dinámicos, 353
Vínculos estáticos, 367. *Véase también* Vínculos de acceso
Vínculos hermanos, 114, 135-138, 278-279
Vínculos hijo (hijos)
atributos heredados por, 278-279
de nodos de árbol de análisis sintáctico, 107, 129
de nodos de árbol sintáctico, 110, 114
en árboles sintácticos de TINY, 135-138
von Neumann, John, 2

Yacc (yet another compiler-compiler, "otro compilador de compilador más"), 4, 196*r*, 226-242, 253-254*e*, 256*r*, 290, 292, 293, 342-343*e*, 488*e*
acciones integradas en, 241-242
descripción de la ejecución de analizadores sintácticos generados mediante, 238-239
ejercicios de programación con, 254-256*e*
generación de analizadores sintácticos ascendentes con, 197-198
generación de un analizador sintáctico TINY con, 243-245
nombres internos y mecanismos de definición de, 242*t*
opciones de, 230-235
organización básica de, 226-230
procedimiento *genCode* en, 411, 413*ilus.*
recuperación de errores en, 247-250
reglas de no ambigüedad y resolución de conflictos de análisis sintáctico con, 235-238
tablas de análisis sintáctico de, 234-235, 235*t*
tipos de valor arbitrario en, 239-241
Yuxtaposición, 35, 36. *Véase también* Operación de concatenación