

Diseño de Compiladores

Departamento de Lenguajes y Sistemas Informáticos

Universidad de Alicante

Alicia Garrido Alenda, José Manuel Iñesta Quereda,
Francisco Moreno Seco, Juan Antonio Pérez Ortiz

13 de junio de 2002

Índice General

Prólogo	1
1 Introducción	3
1.1 ¿Qué es un compilador? Tipos de compiladores	3
1.1.1 Tipos de compiladores	4
1.2 Historia de los primeros compiladores	5
1.2.1 El primer lenguaje y su compilador: FORTRAN	5
1.2.2 El lenguaje algorítmico ALGOL	6
1.2.3 Las primeras técnicas para desarrollar compiladores	7
1.2.4 Técnicas actuales para el desarrollo de compiladores	9
1.3 Estructura de un compilador	9
1.3.1 Las fases de un compilador	10
1.3.2 Implementación del compilador a partir de las fases	15
1.4 ¿Cómo se especifica un compilador?	16
1.5 Aplicaciones de estas técnicas	17
1.6 Referencias bibliográficas	18
2 Análisis léxico	19
2.1 El papel del analizador léxico	19
2.2 Errores léxicos	20
2.3 Funcionamiento del analizador léxico	21
2.4 Especificación de un analizador léxico	23
2.4.1 Definiciones de términos comunes en esta fase	23
2.4.2 Identificación de palabras reservadas	25
2.5 Implementación de analizadores léxicos	28
2.5.1 Prioridad de <i>tokens</i>	31
2.6 Referencias bibliográficas	33
2.7 Ejercicios	34
3 Análisis sintáctico	37
3.1 La función del analizador sintáctico	37
3.2 Diseño de gramáticas	38
3.2.1 Recursividad	38
3.2.2 Ambigüedad	40
3.2.3 Asociatividad y precedencia de los operadores	42

3.3	Tipos de análisis sintáctico	45
3.4	Referencias bibliográficas	47
3.5	Ejercicios	48
4	Análisis sintáctico descendente	51
4.1	Introducción	51
4.1.1	Análisis con retroceso	51
4.1.2	Análizadores sintácticos predictivos	52
4.1.3	Conjuntos de predicción	52
4.2	Cálculo de los conjuntos de predicción	54
4.2.1	Cálculo del conjunto de primeros	54
4.2.2	Cálculo del conjunto de siguientes	56
4.2.3	Cálculo del conjunto predicción	57
4.3	La condición LL(1)	57
4.4	Modificación de gramáticas no LL(1)	60
4.4.1	Eliminación de la ambigüedad	61
4.4.2	Factorización por la izquierda	61
4.4.3	Eliminación de la recursividad por la izquierda	62
4.5	Analizador descendente recursivo	66
4.6	ASDP dirigidos por tabla	69
4.7	Manipulación de errores	75
4.8	Referencias bibliográficas	76
4.9	Ejercicios	77
5	Análisis sintáctico ascendente	81
5.1	Introducción	81
5.2	Análisis sintáctico por desplazamiento y reducción	82
5.3	Método SLR de construcción de tablas de análisis	84
5.3.1	Construcción de la colección canónica de conjuntos de elementos	86
5.3.2	Construcción de autómatas reconocedores de prefijos viables	87
5.3.3	Construcción de tablas SLR	89
5.3.4	Construcción de tablas SLR a partir del autómata reconocedor de prefijos viables	90
5.4	Conflictos en las tablas SLR	91
5.5	Mensajes de error en un analizador SLR	93
5.6	Clasificación de gramáticas	94
5.7	Referencias bibliográficas	95
5.8	Ejercicios	96
6	Traducción dirigida por la sintaxis	99
6.1	Introducción	99
6.2	Gramáticas de atributos	101
6.3	Tipos de atributos	103
6.4	Grafos de dependencias	106
6.5	Especificación de un traductor	108
6.5.1	Definiciones dirigidas por sintaxis	108

6.5.2	Esquemas de traducción	111
6.6	Referencias bibliográficas	117
6.7	Ejercicios	118
7	Implementación de traductores	123
7.1	Introducción	123
7.2	Traductores descendentes recursivos	124
7.2.1	Eliminación de la recursividad izquierda	124
7.2.2	Factorización izquierda	126
7.2.3	Implementación de un traductor descendente recursivo	126
7.3	Traductores ascendentes	130
7.3.1	Equivalencia entre estados y símbolos	131
7.3.2	Implementación de ETDS sencillos	133
7.3.3	Implementación de ETDS más complejos	137
7.3.4	Notación de YACC	139
7.3.5	ETDS con atributos heredados	142
7.3.6	Principales usos de los marcadores	150
7.4	Referencias bibliográficas	150
7.5	Ejercicios	151
8	Tipos y generación de código	153
8.1	Introducción	153
8.2	Gestión de la tabla de símbolos	153
8.3	Representaciones intermedias	157
8.4	Comprobaciones y conversiones de tipos	161
8.5	Código intermedio para expresiones	163
8.6	Código intermedio para instrucciones	166
8.7	Tipos compuestos	173
8.7.1	Tabla de tipos	175
8.7.2	Ámbitos y tabla de tipos	177
8.7.3	Equivalencia de tipos	177
8.8	Código intermedio para <i>arrays</i>	178
8.9	Código intermedio para registros	181
8.10	Referencias bibliográficas	186
8.11	Ejercicios	187
9	Compilación de subprogramas	191
9.1	Introducción	191
9.1.1	El entorno de ejecución	191
9.1.2	Registro de activación	193
9.1.3	Secuencias de llamada y retorno	193
9.1.4	Comprobaciones semánticas	194
9.1.5	Implementación con m2r	194
9.1.6	Valor devuelto	196
9.1.7	Gestión de la tabla de tipos	197
9.1.8	Parámetros y argumentos	197

9.2	Funciones sin recursividad	198
9.2.1	Compilación del cuerpo de la función	199
9.2.2	Compilación de la llamada a la función	201
9.3	Funciones con recursividad	202
9.3.1	Compilación del cuerpo de la función	204
9.3.2	Compilación de instrucciones	205
9.3.3	Compilación de la llamada	205
9.3.4	Traducción a m2r de la función factorial	206
9.4	Funciones locales	206
9.4.1	Encadenamiento de accesos	207
9.4.2	Display	212
9.5	Paso de parámetros	214
9.6	Referencias bibliográficas	215
9.7	Ejercicios	216
A	Soluciones a los ejercicios	219
A.1	Soluciones a los ejercicios del capítulo 2	219
A.2	Soluciones a los ejercicios del capítulo 3	227
A.3	Soluciones a los ejercicios del capítulo 4	233
A.4	Soluciones a los ejercicios del capítulo 5	247
A.5	Soluciones a los ejercicios del capítulo 6	259
A.6	Soluciones a los ejercicios del capítulo 7	266
A.7	Soluciones a los ejercicios del capítulo 8	273
A.8	Soluciones a los ejercicios del capítulo 9	281
B	Ejercicios prácticos propuestos	289
B.1	Analizador léxico	289
B.2	Analizador sintáctico descendente recursivo	292
B.3	Analizador SLR	293
B.4	Traductor descendente recursivo	294
B.5	Traductor ascendente	295
B.6	Compilador para un lenguaje sencillo	297
B.6.1	Especificación semántica	300
B.7	Compilador con tipos complejos	302
B.7.1	Especificación semántica	303
B.8	Compilador con funciones	303
B.8.1	Especificación semántica	304
B.8.2	Ampliación opcional: funciones locales	305
C	El lenguaje intermedio m2r	307
	Bibliografía	313

Prólogo

Actualmente, la materia “Procesadores de Lenguaje” es troncal (con 9 créditos) en todas las titulaciones de ciclo largo de Ingeniería en Informática en España. Esta materia se suele denominar en la bibliografía en inglés *Compiler Design*, de ahí el título de este libro. Debido a que el diseño de compiladores es un tema muy complejo y muy amplio para los 9 créditos que suele tener asignados en los planes de estudios, normalmente no se profundiza mucho en los aspectos más complicados del proceso de compilación, como pueden ser la optimización o la generación de código objeto. En la mayoría de las asignaturas dedicadas a esta materia no se llega mucho más allá de la implementación de un compilador “de juguete” para un lenguaje fuente reducido y un lenguaje objeto parecido al ensamblador para una máquina virtual. Este libro pretende contener las técnicas y fundamentos básicos para enseñar a construir compiladores en las titulaciones de Ingeniería en Informática, teniendo en cuenta que para diseñar un compilador de envergadura sería necesario profundizar mucho más en prácticamente todos los aspectos de su diseño.

Aunque existen muchos libros sobre este tema en inglés, muy pocos de ellos se han traducido al castellano. Entre estos últimos destaca el libro clásico de Aho, Sethi y Ullman (1990), un excelente libro que trata en profundidad todos los aspectos del proceso de la compilación, aunque su traducción al castellano no es todo lo correcta que podría ser y en algunos aspectos se ha quedado un poco obsoleto (se publicó en 1986). Recientemente, se ha publicado un excelente libro en inglés [Louden, 1997] del que no existe hasta el momento traducción al castellano. Nuestro libro no pretende ser tan completo como éstos pero sí pretende recoger aquéllos conocimientos mínimos que deberían conocer los ingenieros en Informática en nuestro país. Por este motivo, algunos temas como la optimización o la generación de código objeto no se tratan o se tratan muy superficialmente. En este libro se asume que el lector tiene unos conocimientos sólidos de programación en C y que conoce razonablemente al menos los conceptos básicos de la teoría de lenguajes, gramáticas y autómatas. También es conveniente tener conocimientos de Pascal y de algún lenguaje ensamblador para entender mejor algunos pasajes del libro, aunque no se asume que el lector tenga conocimientos profundos en estos aspectos.

Como sucede en muchos otros aspectos de la informática, la mejor manera de aprender a diseñar compiladores es implementando un compilador con la ayuda y guía de un buen libro. Por este motivo hemos incluido una serie de ejercicios prácticos que permiten aprender mediante la práctica los conocimientos y las técnicas que se desarrollan a lo largo del libro¹. Estos ejercicios prácticos provienen de prácticas reales propuestas a los

¹La mayoría de los libros sobre compiladores incluyen la construcción de un compilador “de juguete”

alumnos de esta materia en la Universidad de Alicante.

Organización del libro

Después de un primer capítulo introductorio en el que se trata de definir qué es un compilador y cómo se estructura, se dedica un capítulo al análisis léxico y tres al análisis sintáctico. Posteriormente, los capítulos 6 y 7 tratan sobre la traducción dirigida por la sintaxis, las técnicas que se suelen utilizar y cómo se pueden implementar en la práctica. Finalmente, los capítulos 8 y 9 tratan de la compilación de las construcciones más comunes en los lenguajes de programación de alto nivel como C o Pascal. Al final de cada capítulo se proponen unos ejercicios cuyas soluciones pueden encontrarse en el apéndice A. Además, se mencionan los epígrafes de los libros más importantes (siempre según nuestra opinión) en los que se tratan los conceptos estudiados en el capítulo.

Además del apéndice con las soluciones de los ejercicios, se incluyen dos apéndices más. El primero contiene un conjunto de ejercicios prácticos que sirven para reforzar el aprendizaje mediante la práctica de los conceptos explicados en el texto. El segundo y último apéndice incluye una descripción del lenguaje intermedio que se utiliza en los ejercicios prácticos y en los capítulos 8 y 9. Finalmente, el libro incluye una bibliografía en la que se comentan algunos aspectos de los libros más interesantes.

Agradecimientos

Este libro es el fruto de más de ocho años de experiencia impartiendo esta materia en las universidades de Alicante y Jaume I de Castellón. Durante estos años muchos alumnos y alumnas han colaborado en nuestro aprendizaje acerca de esta materia y de su enseñanza con sugerencias muy interesantes que han servido para mejorar este libro, por lo que no podemos dejar de agradecer su entusiasmo por aprender y su colaboración.

También debemos agradecer a nuestros compañeros de los departamentos de Lenguajes y Sistemas Informáticos de la Universidad de Alicante y de la Jaume I su colaboración y sus sugerencias, especialmente a aquellos que han colaborado de manera más directa con nosotros: Pedro García e Isabel Gracia de la Universidad Jaume I, y Manuel Marco, Mikel L. Forcada, Jorge Calera, Ramón Neco y Javier Traver de la Universidad de Alicante. Finalmente, debemos agradecer muy especialmente a Mikel L. Forcada todo lo que hemos aprendido junto a él no sólo sobre esta materia², sino también sobre los aspectos relacionados con su docencia y con la docencia en general; además, ha conseguido transmitirnos su entusiasmo por la docencia como un aspecto esencial de este servicio público que es la universidad.

Los autores

que sirve para ilustrar las técnicas expuestas en el libro.

²Muchos de los planteamientos y ejercicios que aparecen en este libro provienen directamente de él.

Capítulo 1

Introducción

1.1 ¿Qué es un compilador? Tipos de compiladores

Un *traductor* es cualquier programa que toma como entrada un texto escrito en un lenguaje, llamado fuente, y da como salida otro texto en un lenguaje denominado objeto.



En el caso de que el lenguaje fuente sea un lenguaje de programación de alto nivel y el objeto sea un lenguaje de bajo nivel (ensamblador o código de máquina), a dicho traductor se le denomina *compilador*. Un *ensamblador* es un compilador cuyo lenguaje fuente es el lenguaje ensamblador. Por otro lado, un *intérprete* no genera un programa equivalente, sino que toma una sentencia del programa fuente en un lenguaje de alto nivel, la traduce al código equivalente y al mismo tiempo la ejecuta.

Históricamente, debido a la escasez de memoria de los primeros ordenadores, se puso de moda el uso de intérpretes frente a los compiladores, pues el programa fuente sin traducir y el intérprete juntos requerían una cantidad de memoria menor que la del compilador. Por ello, los primeros ordenadores personales (Spectrum, Commodore VIC-20, PC XT de IBM, etc.) iban siempre acompañados de un intérprete de BASIC. La mejor información sobre los errores por parte del compilador así como una mayor velocidad de ejecución del código resultante hizo que poco a poco se impusieran los compiladores. Hoy en día, y con el problema de la memoria prácticamente resuelto, se puede hablar de un gran predominio de los compiladores frente a los intérpretes, aunque intérpretes como los incluidos en los navegadores de Internet para Java son la gran excepción.

Algunas de las ventajas de compilar frente a interpretar son:

- Se compila una vez; se ejecuta muchas veces.
- La ejecución del programa objeto es mucho más rápida que si se interpreta el programa fuente.
- El compilador tiene una visión global del programa, por lo que la información de mensajes de error es más detallada.

Por otro lado, algunas de las ventajas de interpretar frente a compilar son:

- Un intérprete necesita menos memoria que un compilador. En las primeras etapas de la informática eran más abundantes dado que los ordenadores tenían poca memoria.
- Permiten una mayor interactividad con el código en tiempo de desarrollo.
- En algunos lenguajes (Smalltalk, Prolog, LISP) está permitido y es frecuente añadir código según se ejecuta otro código, y esta característica solamente es posible implementarla en un intérprete.

Un compilador no es un programa que funciona de manera aislada, sino que normalmente se apoya en otros programas para conseguir su objetivo: obtener un programa ejecutable a partir de un programa fuente en un lenguaje de alto nivel. Algunos de esos programas son el preprocesador, el enlazador (*linker*), el depurador y el ensamblador. El preprocesador se ocupa (dependiendo del lenguaje) de incluir ficheros, expandir macros, eliminar comentarios, y otras tareas similares. El enlazador se encarga de construir el fichero ejecutable añadiendo al fichero objeto generado por el compilador las cabeceras necesarias y las funciones de librería utilizadas por el programa fuente. El depurador permite, si el compilador ha generado adecuadamente el programa objeto, seguir paso a paso la ejecución de un programa. Finalmente, muchos compiladores en vez de generar código objeto, generan un programa en lenguaje ensamblador que debe después convertirse en un ejecutable mediante un programa ensamblador.

1.1.1 Tipos de compiladores

Ensamblador: el lenguaje fuente es lenguaje ensamblador y posee una estructura sencilla.

Compilador cruzado: se genera código en lenguaje objeto para una máquina diferente de la que se está utilizando para compilar. Es perfectamente normal construir un compilador de Pascal que genere código para MS-DOS y que el compilador funcione en Linux y se haya escrito en C++. Además, es la única manera de construir un compilador para un nuevo procesador, para el que no exista ningún otro tipo de compilador (excepto quizá un ensamblador).

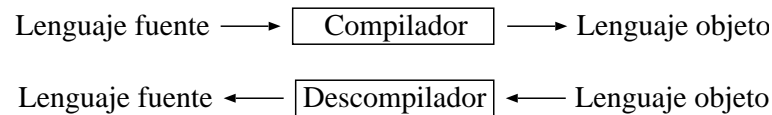
Compilador con montador: compilador que compila distintos módulos de forma independiente y después es capaz de enlazarlos.

Autocompilador: compilador que está escrito en el mismo lenguaje que va a compilar. Evidentemente, no se puede ejecutar la primera vez. Sirve para hacer ampliaciones al lenguaje, mejorar el código generado, etc.

Metacompilador: es sinónimo de compilador de compiladores y se refiere a un programa que recibe como entrada las especificaciones del lenguaje para el que se desea obtener un compilador y genera como salida el compilador para ese lenguaje. Aunque en la construcción de los compiladores actuales se utilizan múltiples herramientas

para ayudar en la generación de cada parte del compilador, no existen herramientas ampliamente aceptadas excepto para las partes más sencillas y formales del compilador.

Descompilador: es un programa que acepta como entrada código máquina y lo traduce a un lenguaje de alto nivel, realizando el proceso inverso a la compilación.



Si hay optimización de código es imposible descompilar. Hasta ahora no se han obtenido buenos descompiladores, sólo desensambladores. Para el lenguaje Java existen descompiladores que obtienen resultados bastante aceptables, aunque el Java no es un lenguaje compilado en el sentido estricto de la palabra, el código que se genera no se corresponde con ninguna máquina real y es de muy alto nivel, lo cual facilita su descompilación.

1.2 Historia de los primeros compiladores

1.2.1 El primer lenguaje y su compilador: FORTRAN

En 1946 se desarrolló el primer ordenador digital. En un principio, estas máquinas ejecutaban instrucciones consistentes en códigos numéricos que señalaban a los circuitos de la máquina los estados correspondientes a cada operación. Esta expresión mediante códigos numéricos se llamó *lenguaje máquina*, interpretado por un secuenciador cableado o por un microprograma.

Pero los códigos numéricos de las máquinas son engorrosos. Pronto los primeros usuarios de estos ordenadores descubrieron la ventaja de escribir sus programas mediante claves más fáciles de recordar que esos códigos numéricos; al final, todas esas claves juntas se traducían manualmente a lenguaje máquina. Estas claves constituyen los llamados *lenguajes ensambladores*, que se generalizaron en cuanto se dio el paso decisivo de hacer que las propias máquinas realizaran el proceso mecánico de la traducción. A este trabajo se le llama *ensamblar* el programa.

Dada su correspondencia estrecha con las operaciones elementales de las máquinas, las instrucciones de los lenguajes ensambladores obligan a programar cualquier función de una manera minuciosa e iterativa. De hecho, normalmente, cuanto menor es el nivel de expresión de un lenguaje de programación, mayor rendimiento se obtiene en el uso de los recursos físicos (*hardware*).

A pesar de todo, el lenguaje ensamblador seguía siendo el de una máquina, pero más fácil de manejar. Los trabajos de investigación se orientaron entonces hacia la creación de un lenguaje que expresara las distintas acciones a realizar de una manera lo más sencilla posible para el hombre. Así, en 1950 John Backus dirigió una investigación en I.B.M. sobre un lenguaje algebraico. En 1954 se empezó a desarrollar un lenguaje que permitía escribir fórmulas matemáticas de manera traducible por un ordenador; le llamaron FORTRAN (FORmulae TRANslator). Fue el primer lenguaje de alto nivel

y se introdujo en 1957 para el uso de la computadora IBM modelo 704. Permitía una programación más cómoda y breve que la existente hasta ese momento, lo que suponía un considerable ahorro de trabajo. Surgió así por primera vez el concepto de un traductor como un programa que traducía un lenguaje a otro lenguaje. En el caso particular de que el lenguaje a traducir es un lenguaje de alto nivel y el lenguaje traducido de bajo nivel, se emplea el término *compilador*.

La tarea de realizar un compilador no fue fácil. El primer compilador de FORTRAN tardó 18 años-persona en realizarse y era muy sencillo. Este desarrollo de FORTRAN estaba muy influenciado por la máquina objeto en la que iba a ser implementado. Como un ejemplo de ello tenemos el hecho de que los espacios en blanco fuesen ignorados, debido a que el periférico que se utilizaba como entrada de programas (una lectora de tarjetas perforadas) no contaba correctamente los espacios en blanco.

1.2.2 El lenguaje algorítmico ALGOL

Paralelamente al desarrollo de FORTRAN en América, en Europa surgió una corriente más universitaria, que pretendía que la definición de un lenguaje fuese independiente de la máquina y que los algoritmos se pudieran expresar de forma más simple. Esta corriente estuvo muy influida por los trabajos sobre gramáticas de contexto libre publicados por Chomsky dentro de su estudio de lenguajes naturales.

Con estas ideas surgió un grupo europeo encabezado por el profesor F.L. Bauer (de la Universidad de Munich). Este grupo definió un lenguaje de usos múltiples independiente de una realización concreta sobre una máquina. Pidieron colaboración a la asociación americana A.C.M. (Association for Computing Machinery) y se formó un comité en el que participó J. Backus que colaboraba en esta investigación. De esa unión surgió un informe que definía el International Algebraic Language (I.A.L.), publicado en Zurich en 1958. Posteriormente este lenguaje se llamó ALGOL 58 (ALGORitmic Language). En 1969, el lenguaje fue revisado y llevó a una nueva versión que se llamó ALGOL 60. La versión actual es ALGOL 68, un lenguaje modular estructurado en bloques, que es el precursor de muchos lenguajes posteriores, como Pascal, Modula, Ada, etc.

En el ALGOL aparecen por primera vez muchos de los conceptos de los nuevos lenguajes algorítmicos:

- Definición de la sintaxis en notación BNF (Backus-Naur Form).
- Formato libre.
- Declaración explícita de tipo para todos los identificadores.
- Estructuras iterativas más generales.
- Recursividad.
- Paso de parámetros por valor y por nombre.
- Estructura de bloques, lo que determina la visibilidad de los identificadores.

1.2.3 Las primeras técnicas para desarrollar compiladores

Junto a este desarrollo en los lenguajes, también se iba avanzando en la técnica de compilación. En 1958 Strong y otros proponían una solución al problema de que un compilador fuera utilizable por varias máquinas objeto. Este problema tiene una formulación simple: dados M lenguajes fuente y N máquinas (lenguajes) objeto, construir de la forma más eficiente posible los $M \times N$ compiladores. La solución consistió en dividir el compilador en dos partes, designadas como el “*front end*” y el “*back end*”.

A grandes rasgos, la primera fase (*front end*) es la encargada de analizar el programa fuente, mientras que la segunda fase (*back end*) se ocupa de generar código para la máquina objeto. La solución al problema de los $M \times N$ compiladores consistió en diseñar M *front ends* y N *back ends*, utilizando como puente de unión entre las dos partes un lenguaje intermedio que se designó con el nombre de UNCOL (UNiversal Computer Oriented Language). Aunque se hicieron varios intentos para definir UNCOL, el proyecto se ha quedado simplemente en un ejercicio teórico. De todas formas, la división de un compilador en *front end* y *back end* fue un adelanto muy importante y perdura en los compiladores actuales.

Como estudiaremos más adelante, se puede hacer una división de un compilador a nivel lógico en una serie de fases, que en el código real del compilador están muy entrelazadas. Las fases más importantes son: análisis léxico, análisis sintáctico, análisis semántico, generación de código y optimización.

Análisis léxico

En 1959 Rabin y Scott proponen el empleo de autómatas deterministas y no deterministas para el reconocimiento lexicográfico de los lenguajes. Rápidamente se aprecia que la construcción de analizadores léxicos a partir de expresiones regulares es muy útil en la implementación de los compiladores. En 1968 Johnson apunta diversas soluciones. En 1975, con la aparición de LEX, surge el concepto de un generador automático de analizadores léxicos a partir de expresiones regulares, basado en el sistema operativo UNIX.

Análisis sintáctico

A partir de los trabajos de Chomsky ya citados, se produce una sistematización de la sintaxis de los lenguajes de programación, y con ello un desarrollo de diversos métodos de análisis sintáctico. Con la aparición de la notación BNF – desarrollada en primer lugar por Backus en 1960 cuando trabajaba en un borrador del ALGOL 60, modificada en 1963 por Naur y formalizada por Knuth en 1964 – se tiene una guía para el desarrollo del análisis sintáctico.

Los diversos métodos de análisis sintáctico (*parsing*) ascendente y descendente se desarrollan durante la década de los 60. En 1959 Sheridan describe un método de análisis sintáctico de FORTRAN que introducía paréntesis adicionales alrededor de los operandos para ser capaz de analizar las expresiones. Más adelante, Floyd introduce la técnica de la precedencia de operador y el uso de las funciones de precedencia. A mitad

de la década de los 60, Knuth define las gramáticas LR y describe la construcción de una tabla canónica de análisis sintáctico LR.

Por otra parte, el uso por primera vez de un analizador sintáctico descendente recursivo tuvo lugar en el año 1961. En el año 1968 se estudian y definen las gramáticas LL así como los analizadores sintácticos predictivos. También se estudia la eliminación de la recursividad por la izquierda de producciones que contienen acciones semánticas sin afectar a los valores de los atributos.

En los primeros años de la década de los 70, se describen los métodos SLR y LALR de análisis LR. Debido a su sencillez y a su capacidad de análisis para una gran variedad de lenguajes, la técnica de análisis LR va a ser la elegida para los generadores automáticos de analizadores sintácticos. A mediados de los 70, Johnson crea el generador de analizadores sintácticos YACC para funcionar bajo un entorno UNIX.

Análisis semántico

Junto al análisis sintáctico, también se fue desarrollando el análisis semántico. En los primeros lenguajes (FORTRAN y ALGOL 60) los tipos posibles de los datos eran muy simples y la comprobación de tipos muy sencilla. No se permitía la restricción de tipos (*type coercion*), pues ésta era una cuestión difícil y era más fácil no permitirla. Con la aparición de ALGOL 68 se permitía que las expresiones de tipo fueran construidas sistemáticamente. Más tarde, de ahí surgió la equivalencia de tipos por nombre y la equivalencia estructural.

Generación de código y gestión de la memoria

En la generación de código uno de los aspectos más importantes es la gestión que va a hacer el compilador de la memoria de la máquina objeto. En los primeros compiladores la memoria se gestionaba de forma estática. Sin embargo, la aparición de los subprogramas (y la recursividad) hizo necesario utilizar otro planteamiento para el manejo de la memoria: la pila.

El manejo de la memoria con una implementación tipo pila se usó por primera vez en 1958 en el primer proyecto de LISP. La inclusión en el ALGOL 60 de procedimientos recursivos potenció el uso de la pila como una forma cómoda de manejo de la memoria. Dijkstra introdujo posteriormente el uso del *display* para acceso a variables no locales en un lenguaje de bloques.

También se desarrollaron estrategias para mejorar las rutinas de entrada y de salida de un procedimiento. Asimismo, y ya desde los años 60, se estudió el paso de parámetros a un procedimiento por nombre, valor y por referencia.

Con la aparición de lenguajes que permiten la localización dinámica de datos, se desarrolla otra forma de manejo de la memoria, conocida por el nombre de *heap* (montículo). Se han desarrollado varias técnicas para el manejo del *heap* y los problemas que con él se presentan, como son las referencias perdidas y la recogida de basura.

Optimización

La necesidad de la optimización apareció desde el desarrollo del primer compilador de FORTRAN. Backus comenta cómo durante el desarrollo del FORTRAN se tenía el miedo de que el programa resultante de la compilación fuera más lento que si se hubiera escrito a mano. Para evitar esto, se introdujeron algunas optimizaciones en el cálculo de los índices dentro de un bucle.

Pronto se sistematizan y se recoge la división de optimizaciones independientes de la máquina y dependientes de la máquina. Entre las primeras están la propagación de valores, la eliminación de redundancias, etc. Entre las segundas se podría encontrar la localización de registros, el uso de instrucciones propias de la máquina y el reordenamiento de código.

A partir de 1970 comienza el estudio sistemático de las técnicas del análisis de flujo de datos. Su repercusión ha sido enorme en las técnicas de optimización global de un programa.

1.2.4 Técnicas actuales para el desarrollo de compiladores

En la actualidad, el proceso de la compilación está muy asentado. Un compilador es una herramienta bien conocida, dividida en diversas fases. Algunas de estas fases se pueden generar automáticamente (analizador léxico y sintáctico) y otras requieren una mayor atención por parte del diseñador de compiladores (las partes de traducción y generación de código).

De todas formas, y en contra de lo que quizá pueda pensarse, todavía se están llevando a cabo varias vías de investigación en este fascinante campo de la compilación. Por una parte, se están mejorando las diversas herramientas disponibles y por otra, la aparición de nuevas generaciones de lenguajes ha provocado la revisión y optimización de cada una de las fases del compilador.

Uno de los últimos lenguajes de programación de amplia aceptación que se ha diseñado, el lenguaje Java, establece que el compilador no genera código para una máquina determinada sino para una virtual, la *Java Virtual Machine* (JVM), que posteriormente será ejecutado por un intérprete, normalmente incluido en un navegador de Internet. El gran objetivo de esta exigencia es conseguir la máxima portabilidad de los programas escritos y compilados en Java, pues es únicamente la segunda fase del proceso la que depende de la máquina concreta en la que se ejecuta el intérprete.

1.3 Estructura de un compilador

Un compilador es un programa muy complejo en el que es difícil distinguir claramente unas partes de otras. Sin embargo, se ha conseguido establecer una división lógica del compilador en fases, lo cual ha permitido formalizar y estudiar por separado cada fase. En la práctica estas fases no funcionan secuencialmente, sino que actúan muchas veces simultáneamente o como subrutinas de otras fases.

1.3.1 Las fases de un compilador

La división en fases de un compilador nos ayuda a entender mejor las diversas tareas que debe realizar y nos permite un estudio independiente y en profundidad de cada una de ellas (véase la figura 1.1).

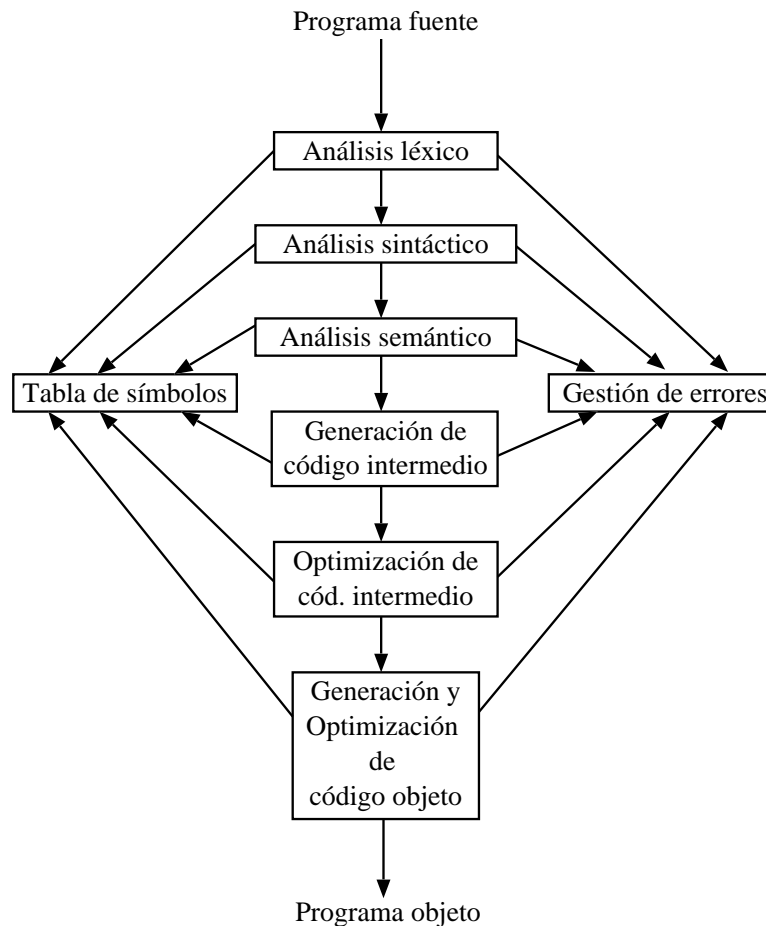


Figura 1.1: Etapas que constituyen el proceso de la compilación.

Estudiemos ahora cada una de las fases con un poco más de detenimiento.

Análisis léxico

El *analizador léxico*, también conocido como *scanner*, lee los caracteres uno a uno desde la entrada y va formando grupos de caracteres con alguna relación entre sí (*tokens*), que constituirán la entrada para la siguiente etapa del compilador. Cada *token* representa una secuencia de caracteres que son tratados como una única entidad. Por ejemplo, en Pascal un *token* es la palabra reservada **BEGIN**, en C **while**, etc.

Hay dos tipos de *tokens*: cadenas específicas, tales como palabras reservadas (**if**, **while**, **begin**, etc.), el punto y coma, los operadores aritméticos o lógicos, etc.; y cadenas

no específicas, como identificadores, constantes o etiquetas.

Se considera que un *token* tiene dos partes componentes: el tipo de *token* y su valor o *lexema*. Algunas cadenas específicas sólo tienen tipo (lo que representan), mientras que las cadenas no específicas siempre tienen tipo y valor. Por ejemplo, si “Contador” es un identificador, el tipo de *token* será *identificador* y su valor será la cadena “Contador”.

El analizador léxico es la etapa del compilador que permite saber si es un lenguaje de formato libre o no. Frecuentemente va unido al analizador sintáctico en la misma pasada, y funciona como una subrutina de este último. Ya que es el que va leyendo los caracteres del programa, ignorará aquellos elementos innecesarios para la siguiente fase, como los tabuladores, comentarios, espacios en blanco, etc.

Análisis sintáctico

El *analizador sintáctico*, también llamado *parser*, recibe como entrada los *tokens* que le pasa el analizador léxico (el analizador sintáctico no maneja directamente caracteres) y comprueba si esos *tokens* van llegando en el orden correcto (orden permitido por el lenguaje). La salida “teórica” de la fase de análisis sintáctico sería un árbol sintáctico.

Así pues, sus funciones son:

- Guiar el proceso de traducción (*traducción dirigida por la sintaxis*) mientras se analiza el programa fuente. Si al final del análisis no se ha encontrado ningún error, se habrá obtenido el código intermedio y podrá ser procesado por las siguientes fases.
- Cuando el programa fuente es incorrecto, producir un mensaje de error adecuado.

Un aspecto importante en el diseño del analizador sintáctico es que se debe tratar de hacer explícito el orden jerárquico que tienen los operadores en el lenguaje de que se trate. Por ejemplo, la cadena $A/B*C$ es interpretada como $(A/B)*C$ en FORTRAN y como $A/(B*C)$ en APL.

Análisis semántico

El análisis semántico es mucho más difícil de formalizar. Se trata de determinar el tipo de los resultados intermedios, comprobar que los argumentos que tiene un operador pertenecen al conjunto de los operandos posibles, y si son compatibles entre sí, etc. En definitiva, comprobará que el significado de lo que se va leyendo es válido.

La salida “teórica” de la fase de análisis semántico sería un árbol semántico. Consiste en un árbol sintáctico en el que cada una de sus ramas ha adquirido el significado que debe tener.

En el caso de los operadores polimórficos (un único símbolo con varios significados), el análisis semántico determina cuál es el aplicable. Por ejemplo, consideremos la siguiente sentencia de asignación:

$A := B + C$

En Pascal, el signo “+” sirve para sumar enteros y reales, concatenar cadenas de caracteres y unir conjuntos. El análisis semántico debe comprobar que B y C sean de un

tipo común o compatible y que se les pueda aplicar dicho operador. Si B y C son enteros o reales los sumará, si son cadenas las concatenará y si son conjuntos calculará su unión.

Un resumen de las tareas de análisis que hace un compilador con un programa dado se puede ver en el ejemplo siguiente:

Generación de código intermedio

Cuando una empresa desarrolla un compilador para un lenguaje fuente y un lenguaje objeto determinados, normalmente no es el único compilador que la empresa piensa desarrollar; es más, muchos fabricantes de microprocesadores tienen una división dedicada a desarrollar compiladores para los nuevos chips que construyen.

Cuando el número de lenguajes fuente crece hasta un valor grande M , o cuando el número de lenguajes objeto también crece hasta un valor grande N , es necesario encontrar una técnica para evitar tener que diseñar $M \times N$ compiladores. Como hemos comentado anteriormente, la solución consiste en utilizar un lenguaje intermedio o una representación intermedia; de esta forma sólo hay que construir M programas que traduzcan de cada lenguaje fuente al lenguaje intermedio (los *front ends*), y N programas que traduzcan del lenguaje intermedio a cada lenguaje objeto (los *back ends*). Desgraciadamente, no existe un único lenguaje intermedio en todos los compiladores, sino que cada empresa que diseña compiladores suele tener su propio lenguaje intermedio. La utilización de un lenguaje intermedio permite construir en mucho menos tiempo un compilador para otra máquina y también permite construir compiladores para otros lenguajes fuente generando código para la misma máquina.

Por ejemplo, el compilador de C de GNU que se distribuye con Linux es una versión de una familia de compiladores de C para diferentes máquinas o sistemas operativos: Alpha, AIX, Sun, HP, MS-DOS, etc. Además, GNU ha desarrollado un compilador de FORTRAN y otro de Pascal que, al utilizar el mismo lenguaje intermedio, pueden ser portados a todos los sistemas y máquinas en las que ya exista un compilador de C de GNU con relativamente poco esfuerzo.

La generación de código intermedio transforma un árbol de análisis sintáctico (semántico) en una representación en un lenguaje intermedio, que suele ser código suficientemente sencillo para poder luego generar código máquina.

Una forma de hacer esto es mediante el llamado *código de tres direcciones*. Una sentencia en código de tres direcciones es $A := B \text{ op } C$, donde A , B y C son operandos y op es un operador binario. También se permiten condicionales simples y saltos. Por ejemplo, para la siguiente sentencia:

```
WHILE (A>B) AND (A<=2*B-5) DO A:=A+B
```

el código intermedio generado (código en tres direcciones) será:

```
L1:      IF A>B GOTO L2
          GOTO L3
L2:      T1 := 2*B          (* nivel más alto que ensamblador *)
          T2 := T1-5        (* pero más sencillo que Pascal *)
          IF A<=T2 GOTO L4
```

```

        GOTO L3
L4:      A := A+B
        GOTO L1
L3:      .....

```

Optimización de código

En un compilador es posible realizar mejoras en el código intermedio¹ y en el código objeto. La mayoría de los compiladores suelen tener una fase de optimización de código intermedio (que es independiente del lenguaje objeto y del lenguaje fuente), y una fase de optimización del código objeto, que suele estar muy relacionada con la fase de generación de código objeto y suele incluir optimizaciones dependientes de la máquina objeto y, por tanto, no aplicables a otras máquinas.

Estas fases se añaden al compilador para conseguir que el programa objeto sea más rápido en la ejecución y necesite menos memoria a la hora de ejecutarse. El término optimización no es correcto, ya que nunca podemos asegurar que conseguimos un programa óptimo. Con una buena optimización, el tiempo de ejecución puede llegar a reducirse a la mitad, aunque hay compiladores que pueden no llevar esta etapa y generar directamente código máquina. Ejemplos de posibles optimizaciones locales:

- cuando hay dos saltos seguidos, se puede quedar uno solo. El fragmento de programa del ejemplo anterior (ver sección 1.3.1) quedaría así:

```

L1:      IF A<=B GOTO L3      (*)
        T1 := 2 * B
        T2 := T1 - 5
        IF A>T2 GOTO L3      (*)
        A := A + B
        GOTO L1
L3:      .....

```

De esta forma, se eliminan algunas instrucciones, lo cual supone un ahorro en tiempo y espacio.

- Eliminar expresiones comunes en favor de una sola expresión. Por ejemplo:

```

A := B+C+D
E := B+C+F

```

se convierte en:

```

T1 := B+C
A  := T1+D
E  := T1+F

```

¹El código generado por el generador de código intermedio no suele (ni debe) ser muy eficiente.

- Optimización de bucles. Se trata de sacar de los bucles las expresiones que sean invariantes dentro de ellos. Por ejemplo, dado el siguiente código:

```
REPEAT
  B := 1
  A := A-B
UNTIL A=0
```

la asignación $B:=1$ se puede realizar fuera del bucle.

Generación de código objeto

En esta parte el código intermedio optimizado es traducido a una secuencia de instrucciones en ensamblador o en el código de máquina del procesador que nos interese. Por ejemplo, la sentencia $A:=B+C$ se convertirá en:

```
LOAD  B
ADD   C
STORE A
```

suponiendo que estas instrucciones existan de esta forma en el ordenador de que se trate.

Una conversión tan directa produce generalmente un programa objeto que contiene muchas cargas (*loads*) y almacenamientos (*stores*) redundantes, y que utiliza los recursos de la máquina de forma ineficiente. Existen técnicas para mejorar esto, pero son complejas. Una, por ejemplo, es tratar de utilizar al máximo los registros de acceso rápido que tenga la máquina. Así, en el procesador 8086 tenemos los registros internos AX, BX, CX, DX, etc. y podemos utilizarlos en vez de direcciones de memoria.

Tabla de símbolos

Un compilador necesita guardar y usar la información de los objetos que se va encontrando en el texto fuente, como variables, etiquetas, declaraciones de tipos, etc. Esta información se almacena en una estructura de datos interna conocida como tabla de símbolos.

El compilador debe desarrollar una serie de funciones relativas a la manipulación de esta tabla como insertar un nuevo elemento en ella, consultar la información relacionada con un símbolo, borrar un elemento, etc. Como se tiene que acceder mucho a la tabla de símbolos, los accesos deben ser lo más rápidos posibles para que la compilación sea eficiente.

Manejo de errores

Es una de las misiones más importantes de un compilador, aunque, al mismo tiempo, es la que más dificulta su realización. Donde más se utiliza es en las etapas de análisis sintáctico y semántico, aunque los errores se pueden descubrir en cualquier fase de un compilador. Es una tarea difícil, por dos motivos:

- A veces unos errores ocultan otros.
- A veces un error provoca una avalancha de muchos errores que se solucionan con el primero.

Es conveniente un buen manejo de errores, y que el compilador detecte todos los errores que tiene el programa y no se pare en el primero que encuentre. Hay, pues, dos criterios a seguir a la hora de manejar errores:

1. Pararse al detectar el primer error.
2. Una vez detectado el primer error, *recuperar* el proceso normal de análisis y detectar todos los errores de una pasada.

En el caso de un compilador interactivo (dentro de un entorno de desarrollo integrado, como Turbo-Pascal o Borland C++) no importa que se pare en el primer error detectado, debido a la rapidez y facilidad para la corrección de errores.

1.3.2 Implementación del compilador a partir de las fases

Al principio de la historia de los compiladores, el tamaño del programa ejecutable era un recurso crítico, así como la memoria que utilizaba el compilador para sus datos, por lo que era frecuente que cada fase leyerá un fichero escrito por la fase anterior y produjera un nuevo fichero con el resultado de las transformaciones realizadas en dicha fase. Esta técnica (inevitable en aquellos tiempos) hacía que el compilador realizara muchas pasadas sobre el programa fuente.

En los últimos años el tamaño del fichero ejecutable de un compilador es relativamente pequeño comparado con el de otros programas del sistema, y además (gracias a los sistemas de memoria virtual) normalmente no se tienen problemas de memoria para compilar un programa medio. Por estos motivos, y dado que escribir y leer un fichero de tamaño similar o mayor que el del programa fuente en cada fase es una pérdida considerable de tiempo (incluso en los sistemas modernos), la tendencia actual es la de reducir el número de ficheros que se leen o escriben y por tanto reducir el número de pasadas, incluso el de aquellas que se realizan en memoria, sin escribir ni leer nada del disco. En las primeras épocas de la historia de los compiladores había compiladores que realizaban todas las fases en una única pasada, pero producían un código bastante ineficiente.

Como hemos estudiado anteriormente, las fases se agrupan en dos partes o etapas: *front end* (las fases de análisis y generación de código intermedio) y *back end* (las fases de generación y optimización de código). Estas dos etapas se comunican mediante una representación intermedia (generada por el *front end*), que puede ser una representación de la sintaxis del programa (un árbol sintáctico abstracto) o bien puede ser un programa en un lenguaje intermedio. El *front end* depende del lenguaje fuente y casi siempre es independiente (o debe serlo) de la máquina objeto para la que se va a generar código; el *back end* depende del lenguaje objeto y debe ser independiente del lenguaje fuente, excepto quizá para algún tipo de optimización (véase la figura 1.2).

En los compiladores actuales, el *front end* suele realizar una pasada o como mucho dos pasadas: en la primera se construye un árbol sintáctico abstracto y en la siguiente

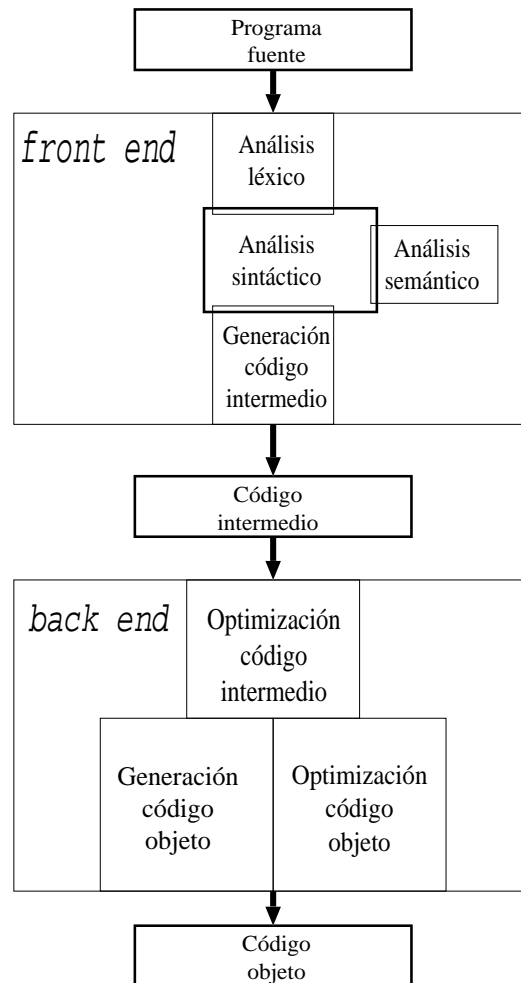


Figura 1.2: Organización de las fases en *front end* y *back end*.

se recorre el árbol y se genera la representación intermedia. El *back end* suele llevar al menos una primera pasada de optimización del código fuente, y una o más pasadas para generar código objeto y optimizarlo.

1.4 ¿Cómo se especifica un compilador?

Cuando se va a diseñar un compilador, puesto que se trata de un traductor entre dos lenguajes, es necesario especificar el lenguaje fuente y el lenguaje objeto, así como el sistema operativo sobre el que va a funcionar el compilador y el lenguaje utilizado para construirlo. Existen algunas herramientas formales para especificar los lenguajes objeto, pero no existe ninguna herramienta cuyo uso sea extendido.

La especificación del lenguaje fuente se divide en tres partes:

Especificación léxica: en esta parte se especifican los componentes léxicos (*tokens*) o

palabras del lenguaje; para ello se utilizan expresiones regulares. A partir de estas expresiones regulares se construye el analizador léxico del compilador.

Especificación sintáctica: en esta parte se detalla la forma o estructura (la sintaxis) que tendrán los programas en este lenguaje fuente. Para esta tarea se utiliza una gramática independiente del contexto o un diagrama sintáctico que posteriormente se convierte en una gramática; a partir de la gramática se construye el analizador sintáctico del compilador.

Especificación semántica: en esta parte se describe el significado de cada construcción sintáctica y las reglas semánticas que deben cumplirse; aunque existen notaciones formales para especificar la semántica de un lenguaje, normalmente se especifica con palabras (lenguaje natural). Algunas veces es posible recoger en la especificación sintáctica algunas partes (restricciones, asociatividad y precedencia, etc.) de la especificación semántica. El analizador semántico y el generador de código intermedio se construyen a partir de la especificación semántica.

1.5 Aplicaciones de estas técnicas

La importancia práctica de los traductores de lenguajes en informática se manifiesta principalmente en el uso cotidiano que hace el profesional informático de compiladores e intérpretes, consustancial a la gestión y programación de los sistemas informáticos. Así pues, un conocimiento acerca del funcionamiento interno de estas herramientas básicas resulta fundamental. Pero los conocimientos adquiridos en su estudio encuentran aplicación fuera del campo de la compilación.

Ya Nicklaus Wirth apuntó la importancia del estudio del desarrollo de compiladores para el ingeniero en informática, puesto que son programas muy complejos y que, por tanto, requieren de una técnica de programación disciplinada y estructurada, y porque además desarrolla la visión recursiva del programador, inherente a la compilación. Sólo por esos motivos ya es interesante el estudio de estas técnicas. Por otro lado, es probable que pocas personas con esta formación tengan que realizar o mantener un compilador para un lenguaje de programación, pero muchos pueden obtener provecho del uso de un gran número de sus técnicas para el diseño de *software* en general.

En efecto, entre los campos de la informática en los que encuentran aplicación las técnicas desarrolladas en este libro se pueden citar los siguientes:

- Desarrollo de interfaces textuales. Cualquier programa cuya interacción con el usuario sea algo más que pulsar teclas de opciones o pinchar aquí o allá con el ratón necesitará de estas técnicas para interpretar comandos o cualquier tipo de diálogo hombre-máquina.
- Tratamiento de ficheros de texto con información estructurada. Lenguajes como Perl y Tcl, o comandos como el sed o egrep de UNIX, incorporan tratamiento de expresiones regulares para la detección y/o modificación de patrones en textos.
- Procesadores de texto. Procesadores como vi o Emacs incorporan también la posibilidad de efectuar búsquedas y sustituciones mediante expresiones regulares.

Existen también procesadores (entre ellos el Emacs) capaces de analizar y tratar ficheros de textos de organización compleja.

- Diseño e interpretación de lenguajes para el formateo de texto y descripción de gráficos. Sistemas de formateo de texto (como HTML o \TeX) o para la especificación de tablas (tbl), ecuaciones (eqn), gráficos (Postscript), etc. requieren sofisticados macroprocesadores.
- Gestión de bases de datos. Las técnicas que estamos considerando pueden explotarse tanto en la exploración y proceso de ficheros de información como en la realización de la interfaz de usuario.
- Procesamiento del lenguaje natural. Las primeras fases de cualquier manipulación de textos escritos en lenguaje natural son las de análisis léxico y sintáctico. Más aún nos acercamos a las técnicas propias de la compilación cuando hablamos de traducción automática.
- Traducción de formatos de ficheros. Si conocemos la estructura de los datos de ficheros con registros de programas obsoletos podremos, utilizando técnicas de traducción propias de la compilación, actualizarlos a formatos actualizados.
- Cálculo simbólico. Manejo simbólico de fórmulas.
- Reconocimiento de formas. Las técnicas de análisis sintáctico son ampliamente utilizadas en la detección de patrones en textos, el reconocimiento automático del habla o la visión por computador.

1.6 Referencias bibliográficas

REFERENCIAS	EPÍGRAFES
[Louden, 1997]	1.1, 1.2, 1.3 y 1.5
[Aho, Sethi y Ullman, 1990]	1.1, 1.2, 1.3, 1.4, 1.5 y 1.6
[Bennett, 1990]	1.1 y 1.2
[Fischer y LeBlanc, 1991]	1.1, 1.2, 1.3 y 7.1.2

Capítulo 2

Análisis léxico

2.1 El papel del analizador léxico

El analizador léxico (AL) es la primera fase de un programa traductor. Es, por otra parte, el único que gestiona el fichero de entrada. Es la parte del compilador que lee los caracteres del programa fuente y que construye unos símbolos intermedios (*elementos léxicos*, que llamaremos “*tokens*”) que serán posteriormente utilizados por el analizador sintáctico como entrada.

El analizador sintáctico debe obtener una representación de la estructura (sintaxis) del programa fuente. Para realizar esta tarea debería concentrarse solamente en la estructura y no en otros aspectos menos importantes, como los espacios en blanco o tabuladores, los cambios de línea, los comentarios, etc. Además, los árboles sintácticos contruidos con una gramática que genere los programas carácter a carácter no son útiles para construir una traducción.

¿Por qué separar el análisis léxico del sintáctico?

- El diseño de las partes posteriores dedicadas al análisis queda simplificado.
- Con fases separadas, se pueden aplicar técnicas específicas y diferenciadas para cada fase, que son más eficientes en sus respectivos dominios.
- Se facilita la adaptación del compilador a sistemas con distintos juegos de caracteres, o bien a distintas formas de la entrada.

Si tomamos por ejemplo las expresiones “ $6-2*30/7$ ” y “ $6 - 2* 30/7$ ”, podemos comprobar que la estructura de ambas expresiones es equivalente; sin embargo, los caracteres que componen ambas cadenas no son los mismos. Si tuviéramos que trabajar directamente con los caracteres estaríamos dificultando la tarea de obtener la misma representación para ambas cadenas. Si consideramos además la cadena “ $8-2*3/5$ ”, la estructura de esta cadena es de nuevo la misma que la de las cadenas anteriores, lo único que cambia son los valores concretos de los números. Por estos motivos (y también por eficiencia), el procesamiento de los caracteres se deja en manos del analizador léxico que entregará a las sucesivas etapas del compilador los componentes léxicos (*tokens*) significativos.

Ejemplo 2.1

Usando la cadena mencionada anteriormente, “6-2*30/7” (o la otra igual salvo el número de espacios), ambas serían representadas por el analizador léxico como la siguiente cadena de elementos léxicos:

```
entero,6  resta,-  entero,2  por,*  entero,30  div,/  entero,7
```

donde cada *token* ha sido representado por un par en el que la primera componente de cada par es el tipo de *token* y la segunda componente es el *lexema* (el valor concreto de ese *token*). La tercera cadena del ejemplo anterior tendría la misma estructura que las otras dos, pero con distintos valores de los *lexemas*.

<

En definitiva, el análisis léxico agrupará los caracteres de la entrada por categorías léxicas, establecidas por la especificación léxica del lenguaje fuente como veremos más adelante. Esta especificación también establecerá el alfabeto con el que se escriben los programas válidos en el lenguaje fuente y, por tanto, el analizador léxico también deberá rechazar cualquier texto en el que aparezcan caracteres ilegales (no recogidos en ese alfabeto) o combinaciones ilegales (no permitidas por las especificaciones léxicas) de caracteres del alfabeto.

Veremos que los componentes léxicos se especifican mediante *expresiones regulares* que generan lenguajes regulares, más sencillos de reconocer que los lenguajes independientes del contexto, y permiten hacer un análisis más rápido. Además, una gramática que represente la sintaxis de un lenguaje de alto nivel carácter a carácter sería mucho más compleja (para implementar un proceso de traducción a partir de ella) que otra que representase la misma sintaxis en función de sus componentes léxicos.

2.2 Errores léxicos

Pocos son los errores característicos de esta etapa, pues el compilador tiene todavía una visión muy local del programa. Por ejemplo, si el analizador léxico encuentra y aísla la cadena “wihle” creará que es un identificador, cuando posiblemente se tratara de un **while** mal escrito y no será él el que informe del error, sino que lo harán sucesivas etapas del análisis del texto.

Los errores que típicamente detecta el analizador léxico son:

- Utilizar caracteres que no pertenecen al alfabeto del lenguaje (p. ej.: ‘ñ’ o ‘a’).
- Se encuentra una cadena que no coincide con ninguno de los patrones de los *tokens* posibles (p. ej.: en un lenguaje “:=” puede ser la asignación pero el símbolo “:=” no puede aparecer solo).

Cuando el analizador léxico encuentra un error, lo habitual es parar su ejecución e informar, pero hay una serie de posibles acciones por su parte para anotar los errores, recuperarse de ellos y seguir trabajando:

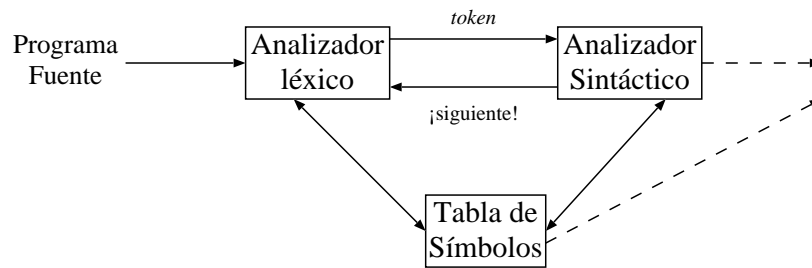


Figura 2.1: Analizador Léxico como subrutina del A.Sintáctico

- Ignorar los caracteres no válidos hasta formar un *token* según los patrones dados;
- Borrar los caracteres extraños;
- Insertar un carácter que pudiera faltar;
- Reemplazar un carácter presuntamente incorrecto por uno correcto;
- Conmutar las posiciones de dos caracteres adyacentes.

Estas transformaciones se realizan sobre el prefijo de entrada que no concuerda con el patrón de ningún *token*, intentando conseguir un lexema válido. No obstante, todas son complicadas de llevar a cabo y peligrosas por lo equivocadas que pueden resultar para el resto del análisis.

2.3 Funcionamiento del analizador léxico

La principal función del analizador léxico es procesar la cadena de caracteres y devolver pares (*token*, lexema). Debe funcionar como una subrutina del analizador sintáctico (véase la figura 2.1).

Las operaciones que realiza el analizador léxico son:

- Procesado léxico del programa fuente: identificación de *tokens* y de sus lexemas que deberá entregar al analizador sintáctico así como interaccionar con la tabla de símbolos.
- Manejo del fichero del programa fuente; es decir: abrirlo, leer sus caracteres y cerrarlo. También debería ser capaz de gestionar posibles errores de lectura.
- Ignora los comentarios y, en los lenguajes de formato libre, ignora los separadores (espacios en blanco, tabuladores, retornos de carro, etc.).
- Cuando se produzca una situación de error será el analizador léxico el que sitúe el error en el programa fuente. Lleva, por lo tanto, la cuenta de las líneas y columnas procesadas.

- Preproceso de macros, definiciones, constantes y órdenes de inclusión de otros ficheros.

Cada vez que el analizador sintáctico llame al léxico éste debe leer caracteres desde donde se quedó en la anterior llamada hasta conseguir completar un nuevo *token*, y en ese momento debe devolver el par (*token* , lexema). Cuando el analizador léxico intenta reconocer algunos tipos de *tokens* como los identificadores o los números se produce una circunstancia especial: el analizador léxico debe leer caracteres hasta que lea uno que no pertenece a la categoría del *token* que está leyendo; ese último carácter (que no tiene por qué ser un espacio en blanco) no puede perderse, y debe devolverse a la entrada para ser leído en primer lugar la próxima vez que se llame al analizador léxico.

Ejemplo 2.2

En la cadena “Grande / 307>=” marcamos las posiciones a las que tiene que llegar el analizador léxico para decidir qué *tokens* ha reconocido:

```
Grande /307>=
  ↑↑   ↑↑
```

Como se puede observar, para el caso del identificador “Grande”, ha tenido que ir una posición más allá del final del mismo, pues sólo allí, al encontrar el espacio en blanco puede saber que el nombre del identificador ha concluido. Para encontrar el símbolo “/” basta con ponerse sobre él y *verlo* si consideramos que no es prefijo de ningún otro (para este lenguaje). Para el número “307” sucede lo mismo que con el identificador: hay que llegar hasta un carácter que no sea un número para saber que el número ha terminado. Ese símbolo en este caso es el signo “>” de “>=” (mayor o igual que). Después, para reconocer este nuevo *token* el léxico avanzará para ver si es el “=” lo que sigue al “>”. Como sí que lo es y suponemos que “>=” no es prefijo de ningún otro, el analizador devolverá el *token* “mayor o igual”; si no hubiera aparecido el “=” al avanzar, hubiera tenido que retroceder una posición y devolver el *token* “mayor”.

◁

El analizador léxico debe intentar leer siempre el *token* más largo posible. Puede ocurrir que haya leído ya un *token* correcto y al intentar leer un *token* más largo no sea posible; en este caso no se debe producir un error, sino que el analizador léxico debe devolver el *token* correcto y debe retroceder en la entrada hasta el final de ese *token*.

Ejemplo 2.3

Si el operador “!=” (*distinto*) pertenece al lenguaje pero el carácter “!” no, cuando aparezca en la entrada este carácter, el analizador debe leer el siguiente carácter; si es un “=”, devolverá el *token* correspondiente al operador *distinto*, pero si no es un “=”, debe producir un error léxico si el carácter “!” por sí solo no perteneciera al lenguaje.

◁

2.4 Especificación de un analizador léxico

2.4.1 Definiciones de términos comunes en esta fase

- **Tokens:** desde el punto de vista léxico son los elementos léxicos del lenguaje mientras que para el resto de las fases de un compilador son los símbolos terminales de la gramática (por ejemplo: palabras reservadas, identificadores, signos de puntuación, constantes numéricas, operadores, cadenas de caracteres, etc.). Es posible, dependiendo del lenguaje, que varios signos formen un solo *token* (":=", "==", "+=", "|", etc.).
- **Patrón:** expresión regular que define el conjunto de cadenas que cada uno de los *tokens* representa.
- **Lexema:** secuencia de caracteres del código fuente que concuerda con el patrón de un *token*. Es decir, cuando analizamos el texto fuente y encontramos una cadena de caracteres que representa un *token* determinado diremos que esa cadena es su lexema.
- **Atributos:** El análisis léxico debe proporcionar información adicional sobre los *tokens* en sus atributos asociados. El número de atributos depende de cada *token*. En la práctica, se puede considerar que los *tokens* tienen un único atributo, un registro que contiene toda la información propia de cada caso (por ejemplo, lexema, tipo de *token* y línea y columna en la que fue encontrado). Lo normal es que toda esa información se entregue a los analizadores sintáctico y semántico para que la usen como convenga.

Ejemplo 2.4

Token	Lexemas	Patrón
Identificador	Pepe, cons1, ...	Letra·(Letra Dígito)*
Num.Entero	10, -105, +24, ...	(+ ϵ)·Dígito ⁺
PR_IF	If, if, IF, iF	(i I)·(f F)

Cuando el analizador léxico encuentra un lexema devuelve como información a qué *token* pertenece y todo lo que sabe de él, incluido el propio lexema. En el último caso, se supone que esa palabra pertenece a un lenguaje en el que mayúsculas y minúsculas son equivalentes.

◀

Para especificar correctamente el funcionamiento de un analizador léxico se debe utilizar una máquina de estados, llamada *diagrama de transiciones* (DT), muy parecida a un autómata finito determinista, con las siguientes diferencias:

- Un AFD sólo dice si la cadena de caracteres pertenece al lenguaje o no; un DT debe funcionar como un analizador léxico; es decir, debe leer caracteres hasta que complete un *token*, y en ese momento debe retornar (en los estados de aceptación) el *token* que ha leído y dejar la entrada preparada para la siguiente llamada.

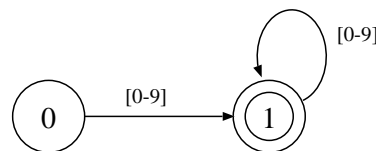
- Un DT no puede tener estados de absorción (para cadenas incorrectas en AFDs) ni de error (se considerará que las entradas para las que no hay una transición desde cada estado son error).
- De los estados de aceptación de un DT no deben salir transiciones.
- En el caso de las cadenas no específicas, necesitamos otro estado al que ir cuando se lea un carácter que no pueda formar parte del patrón. En este último estado (al que se llega con la transición especial **otro**) se debe devolver a la entrada el carácter leído (que puede ser parte del siguiente *token*), lo cual se indica marcando el estado con un asterisco, y se debe retornar el *token* correspondiente a ese estado de aceptación. Por ejemplo, para reconocer números enteros, con un AFD son necesarios solamente dos estados; con un DT necesitamos ese otro estado al que ir cuando se lea un carácter que no pueda formar parte del número.

En el caso más general, se suelen utilizar estos diagramas de transiciones para reconocer los *tokens* de entrada, contruidos a partir de sus patrones correspondientes, expresados mediante las respectivas expresiones regulares. Estos autómatas se combinan en una máquina única que, partiendo de un único estado inicial, sigue un recorrido u otro por los estados hasta llegar a alguno de los estados de aceptación. En función de en cuál se detenga devolverá un *token* u otro. Si no llega a un estado de aceptación o recibe una entrada que no le permite hacer una transición a otro estado, entonces dará error.

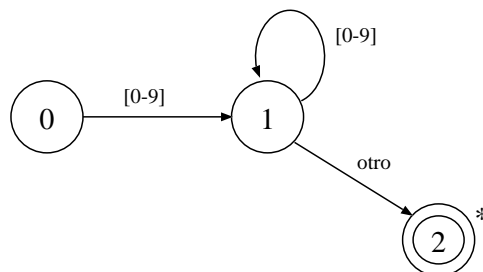
Ejemplo 2.5

A continuación se muestra un ejemplo de reconocedor de números enteros sin signo mediante la expresión regular $[0-9]^+$.

El AFD sería¹:



El DT sería²:



¹En el AFD, el estado 1 reconoce números enteros.

²En el DT, el estado 2 devuelve el *token* Num_entero y devuelve un carácter a la entrada.

Como se observa, en el DT surge un nuevo estado, que es realmente el de aceptación y que está marcado con un asterisco que indica que se llega a él leyendo un carácter más de los necesarios para reconocer ese *token*, y por tanto hay que devolver ese carácter a la entrada. La transición a ese estado se hace mediante la entrada **otro** que significa cualquier otro carácter del alfabeto del lenguaje que no esté en el rango [0-9].

<

Si durante el recorrido del autómata se produce una transición no autorizada o la cadena de entrada finaliza en un estado que no es de aceptación, el analizador informará del error. Este tipo de máquinas es útil para lenguajes con grandes conjuntos de elementos léxicos distintos y las matrices de transición resultantes tienen grandes zonas vacías que conviene comprimir y resumir mediante algoritmos adecuados. Cuando los lenguajes son poco extensos es mejor redactar los analizadores “a mano”, tratando de tomar decisiones adecuadas en función de los caracteres que van apareciendo en la entrada (ver apartado de implementación más adelante).

El analizador suele tener unos subprogramas auxiliares encargados de gestionar la entrada (técnicas de doble *buffer*, saltos de línea, <EOF>, etc.) y de ir devolviendo caracteres a la entrada cada vez que el procedimiento de reconocimiento y aislamiento de *tokens* lo requiera.

2.4.2 Identificación de palabras reservadas

Las palabras reservadas son aquellas que los lenguajes de programación reservan para usos particulares. El problema que surge es: ¿cómo reconocer las palabras reservadas si responden al mismo patrón que los identificadores, pero son *tokens* diferentes al *token* “*identificador*” ?

Existen dos enfoques para resolver este problema:

1. **Resolución implícita:** considerar que todas son identificadores y buscarlas en una tabla. Implica saltarse el formalismo para buscar una solución práctica (factible si se implementa el analizador léxico “a mano” y preferible si el lenguaje tiene muchas palabras reservadas);
2. **Resolución explícita:** se indican todas las expresiones regulares de todas las palabras reservadas y se integran los DT resultantes de sus especificaciones léxicas en la máquina reconocedora (los analizadores resultan mucho más complejos, pero es necesario si usamos programas de generación automática de analizadores a partir de especificaciones).

La primera solución citada consiste en considerar que las palabras reservadas son en principio identificadores, y entonces el analizador leerá letras y dígitos hasta completar un identificador, e inmediatamente antes de retornar el *token* “*identificador*”, comparará el lexema leído con una lista de las palabras reservadas para ver si coincide con alguna de ellas.

En definitiva, se procede normalmente tratando las palabras reservadas como lexemas particulares del patrón del identificador y cuando se encuentra una cadena que responde a dicho patrón, se analiza si es una palabra reservada o un identificador.

Una posible solución para ello es:

- Primero inicializar la tabla de símbolos con todas las palabras reservadas (lo normal es hacerlo por orden alfabético para facilitar la posterior búsqueda y acceso).
- Cuando encuentre un identificador se irá a mirar la tabla de símbolos:
 - Si lo encuentra en la zona reservada para ellas entonces es una palabra reservada
 - Si no, será un identificador, que, como tal, será añadido a la tabla de símbolos.

Ejemplo 2.6

Si se encuentra el identificador “Cont” en la entrada, antes de que el analizador léxico devuelva el *token* “*identificador*” deberá comprobar si se trata de una palabra reservada. Si el número de palabras reservadas es muy grande lo mejor es tenerlas almacenadas desde el principio de la compilación en la tabla de símbolos, para ver si allí ya se encuentra definida esa cadena como tal. Aquí hemos supuesto que no es así y “Cont” queda registrado como un identificador:

Tabla de Símbolos

do	Pal.Reservada	Zona de palabras reservadas
end	Pal.Reservada	
for	Pal.Reservada	
while	Pal.Reservada	
...	...	
Cont	Identificador	Zona de identificadores
...	...	

◁

La disposición de una tabla ordenada con las palabras reservadas es útil cuando el número de éstas es grande. Cuando el lenguaje tiene sólo unas pocas puede ser más práctico realizar la identificación “directamente” mediante una serie de sentencias “if” que comparen con las cadenas correspondientes a esas palabras.

Cuando la detección de palabras reservadas se hace, en cambio, explícitamente, entonces los patrones de la especificación léxica del lenguaje tendrán su correspondencia en el diagrama de transiciones global a partir del cual se implementará el analizador léxico. Las especificaciones léxicas de las palabras reservadas, como cadenas específicas que son, constarán de concatenaciones de caracteres y pueden ser siempre prefijos de identificadores (como por ejemplo “do” -palabra reservada- y “dos” -identificador-). Aparecerán en estos casos los problemas de prefijos y cadenas no específicas que se describen en los ejemplos que se ofrecen más adelante.

Ejemplo 2.7

Si el lenguaje se define como *sensible al tamaño* (*case sensitive* en inglés):

d·o
e·n·d
f·o·r
w·h·i·l·e

Si no lo es (mayúsculas y minúsculas equivalentes):

(d|D)·(o|O)
(e|E)·(n|N)·(d|D)
(f|F)·(o|O)·(r|R)
(w|W)·(h|H)·(i|I)·(l|L)·(e|E)

<

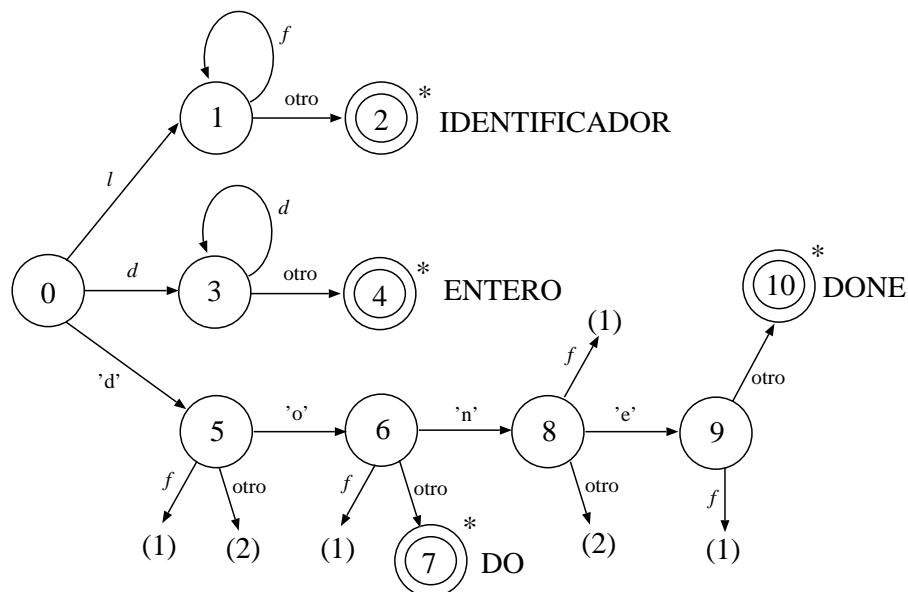
Cuando en la especificación léxica del lenguaje coexisten expresiones regulares de cadenas no específicas como los identificadores con las de específicas como las palabras reservadas, hay que llevar más cuidado porque cualquiera de las palabras reservadas puede ser un prefijo de un identificador válido. Esto motiva que los subautómatas que reconocen las palabras deben estar comunicados con el de los identificadores (véase el ejemplo 2.8).

Por otra parte, cuando un elemento léxico es prefijo de otro y ambos son cadenas específicas, aparecerán estados de aceptación que partirán de estados intermedios (véase el ejemplo 2.9).

Ejemplo 2.8

Constrúyase un diagrama de transiciones para el reconocimiento de identificadores, números enteros sin signo y las palabras reservadas “do” y “done”.

Notación: d = dígito; l = letra; f = otro alfanumérico (dígito o letra); (*número*) = ir al estado *número*.

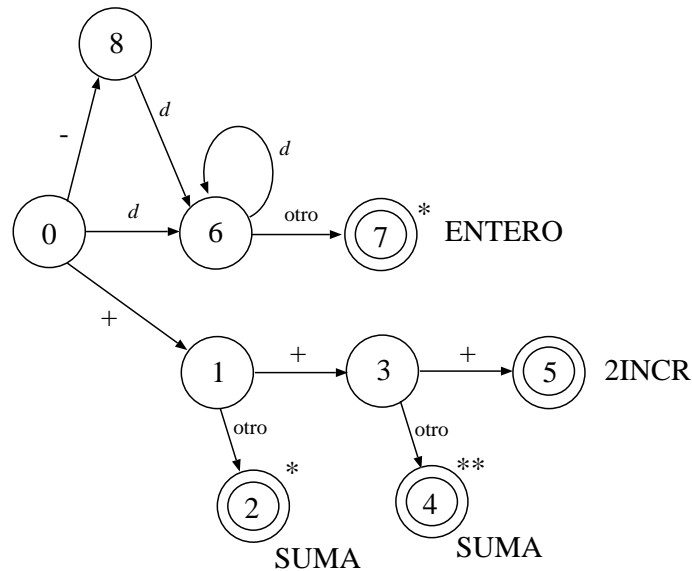


Como se observa en este diagrama de transiciones, todos los estados de aceptación están marcados con asterisco, por lo que siempre habrá que devolver el último carácter leído a la entrada. Esto es debido, en este ejemplo, a que todos los *tokens* son bien unos prefijos de otros (*do* → *done* → *identificador*) o bien son cadenas no específicas (*entero* e *identificador*).

<

Ejemplo 2.9

Constrúyase un diagrama de transiciones para el reconocimiento de números enteros con signo negativo o sin signo ($(-|\epsilon) \cdot d^+$) y los operadores suma (“+”) y doble incremento (“+++”).



Obsérvese que el estado de aceptación del *token* “doble incremento” no lleva asterisco por ser cadena específica y no ser prefijo de ninguna otra, y por tanto no necesita leer el siguiente carácter y retroceder. Sí que lo llevan los estados de aceptación del *token* “suma” a pesar de ser específicas, por ser prefijos del “doble incremento”. Además, uno de ellos lleva dos asteriscos, indicando que si se llega a ese estado hay que retornar el *token* “suma” y devolver *dos* caracteres a la entrada.

<

2.5 Implementación de analizadores léxicos

Existen distintas posibilidades para crear un analizador léxico; las tres más generales son:

1. Usar un generador automático de analizadores léxicos, como LEX: su entrada es un código fuente con la especificación de las expresiones regulares de los patrones que representan a los *tokens* del lenguaje y las acciones a tomar cuando los detecte.
 - Ventaja: comodidad y rapidez en el desarrollo.
 - Inconveniente: ineficiencia del analizador resultante y complicado mantenimiento del código generado.
2. Escribir el AL en un lenguaje de alto nivel de uso general utilizando sus funciones de entrada/salida.
 - Ventaja: más eficiente y compacto.
 - Inconveniente: hay que hacerlo todo a mano.
3. Hacerlo en lenguaje ensamblador.
 - Ventaja: máxima eficiencia y compacidad.
 - Inconveniente: muy complicado de desarrollar y mantener.

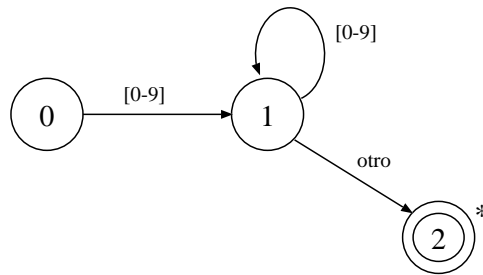
Como se ha indicado, la forma más cómoda de implementar un analizador léxico es con un generador automático de analizadores léxicos como LEX, si bien no es la forma más eficiente. Si se opta por hacerlo “a mano”, se puede hacer de varias maneras: implementando el diagrama de transiciones simulando las transiciones entre estados o bien se pueden implementar “directamente”, usando estructuras de selección múltiple (`switch` en C, `case` en Pascal, etc.) para, según cual sea el primer carácter del *token*, leer caracteres hasta completar el *token*. Por supuesto, con esta técnica también es necesario devolver caracteres a la entrada.

La opción intermedia es utilizar un enfoque mixto en el que se mezcle el análisis manual con el análisis mediante máquinas reconocedoras. Se optará así por analizar mediante estructuras de selección múltiple los elementos léxicos de estructura más sencilla (usualmente los operadores) y dejar para el análisis mediante diagramas de transiciones los elementos léxicos definidos como cadenas no específicas, prefijos comunes, etc. Luego todo ello se empaquetará dentro de una única función que se encargará del análisis léxico.

La forma de implementar el diagrama de transiciones es mediante la construcción de su *tabla de transiciones*. Para ello se etiquetan las filas como los estados del DT y las columnas como las distintas posibles entradas a las que hay que añadir el *token* que se reconoce y el número de caracteres que hay que devolver a la entrada después del reconocimiento.

Ejemplo 2.10

A partir del DT construido antes para los números enteros:



La tabla de transiciones correspondiente será:

	Entradas			
estado	0-9	otro	token	Retroceso
0	1	Error	-	-
1	1	2	-	-
2	-	-	Num_entero	1

◁

Esta es la forma de trabajo de cualquier construcción de un analizador léxico: a partir de las especificaciones léxicas en forma de expresiones regulares se construye la máquina reconocedora (DT) y se representa mediante la tabla de transiciones. Una vez que se tiene ésta, el analizador léxico la recorrerá cada vez mediante un bucle con la sentencia:

```
Estado := TablaTransiciones [ Estado , Entrada ];
```

que intentará llegar a un estado de aceptación en el que restaurará la entrada según lo que diga el campo “Retroceso” para ese número de estado, y devolverá el lexema y *token* encontrados.

El recorrido se inicializa con la variable Estado en el valor del estado inicial (0 en el ejemplo anterior) e itera hasta llegar a un estado de aceptación (2 en el ejemplo anterior).

Ejemplo 2.11

Vamos a construir la tabla de transiciones del ejemplo 2.9, en el que se reconocían números enteros con signo negativo o sin signo $((-|\epsilon) \cdot d^+)$ y los operadores suma (“+”) y doble incremento (“+++”). La tabla de transiciones correspondiente al DT dibujado allí sería en este caso la siguiente:

	Entradas				
estado	+	-	d	token	Retroceso
0	1	8	6	-	-
1	3	2	2	-	-
2	-	-	-	SUMA	1
3	5	4	4	-	-
4	-	-	-	SUMA	2
5	-	-	-	2INCR	0
6	7	7	6	-	-
7	-	-	-	ENTERO	1
8	Error	Error	6	-	-

Obsérvese que las casillas de las filas correspondientes a estados de aceptación nunca tienen valores porque, por definición de los DT, de esos estados no se puede ir a ningún otro. El analizador léxico debe detenerse al llegar a cualquiera de ellos. En cambio las casillas vacías de filas que corresponden a estados que no son de aceptación están etiquetadas como error.

<

2.5.1 Prioridad de *tokens*

Por otro lado, lo normal cuando se construye un AL es establecer criterios para dar más prioridad a unos *tokens* que a otros:

- Dar prioridad al *token* para el que encontramos el lexema más largo. P. ej.: “DO” / “DOT”, el generador se quedaría con el más largo (“DOT”) como identificador (otro ejemplo: “>” y “>=” se debe quedar con el segundo).
- Si el mismo lexema se puede asociar a dos *tokens* (patrones), estos patrones estarán definidos en un orden, y el analizador debe elegir uno de ellos. El generador de analizadores léxicos LEX elige el que aparezca primero en la especificación.

Ejemplo 2.12

Si en la especificación léxica (para un generador como LEX) aparecen entre otras las expresiones regulares

$$\begin{aligned} \mathbf{w \cdot h \cdot i \cdot l \cdot e} &\rightarrow \text{Palabra reservada } \mathbf{while} \\ l \cdot (l|d)^* &\rightarrow \text{Identificador} \end{aligned}$$

si en la entrada aparece el lexema “while”, éste puede ser generado por ambas expresiones regulares, pero como está primero la de la palabra reservada, el AL debe devolver dicho *token*, no el de identificador. Si estas especificaciones aparecieran en el orden inverso:

$$\begin{aligned} l \cdot (l|d)^* &\rightarrow \text{Identificador} \\ \mathbf{w \cdot h \cdot i \cdot l \cdot e} &\rightarrow \text{Palabra reservada } \mathbf{while} \end{aligned}$$

Entonces, el AL debería devolver siempre el *token* identificador y no devolver nunca la palabra reservada “while” (lo cual no suele ser práctico).

<

En el siguiente ejemplo vamos a ver cómo se suelen estructurar los analizadores léxicos contruidos con lenguajes de programación de alto nivel (C en este caso), utilizando la técnica de la diferenciación manual de los distintos *tokens* mediante estructuras de selección múltiple.

Ejemplo 2.13

El lexema puede ser una variable global, aunque si se organizan los *tokens* como estructuras que almacenan todos los atributos relativos a cada *token* devuelto, puede ser un campo de ellas.

```
int analex(void)
{
    c = obtenercaracter();
    switch (c)
    {
        case ' ' :
        case '\t':
        case '\n': /* y para los demás separadores, no hacer nada */
            break;

        case '+':
        case '-': return(ADDOP);

        case '*':
        case '/': return(MULOP);

        /* ... resto de operadores y elementos de puntuación ... */

        default:
            if (ESNUMERO(c))
            { /* leer caracteres mientras sean números */
                /* devolver a la entrada el último carácter leído */
                /* almacenar el lexema leído */
                return (NUMINT);
            }
            else if (ESLETRA(c))
            { /* leer caracteres mientras sean letras o números */
                /* devolver al buffer de entrada ultimo carácter leído */
                /* comprobar si es una palabra reservada */
                /* almacenar el lexema leído */
                return(token); /* que será palab.reservada o ident. */
            }
    } /* del switch */
} /* de analex */
```

<

2.6 Referencias bibliográficas

REFERENCIAS	EPÍGRAFES
[Louden, 1997]	2.1, 2.2.3, 2.3.2 y 2.3.3
[Aho, Sethi y Ullman, 1990]	3.1 y 3.4
[Bennett, 1990]	5.1
[Fischer y LeBlanc, 1991]	3.1 y 3.5

2.7 Ejercicios

Ejercicio 2.1

Diseñar un analizador léxico que utilice un DT construido a partir de las expresiones regulares de los patrones de los tokens involucrados en expresiones algebraicas (los operadores son “*”, “+”, “-”, “/”, “(” y “)”) en las que intervengan números enteros y reales (en notación no exponencial) sin signo y variables expresadas mediante identificadores.

Las expresiones regulares no triviales son:

Números enteros: dígito^+
 Números reales: $\text{dígito}^+(\text{. dígito}^+)?$
 Identificadores: $\text{letra}(\text{letra} \mid \text{dígito})^*$

Ejercicio 2.2

Hacer lo mismo pero implementándolo “a mano” en C.

Ejercicio 2.3

Diseñar un diagrama de transiciones determinista para reconocer los siguientes componentes léxicos:

while	la palabra reservada “while” (en minúsculas).
when	la palabra reservada “when” (en minúsculas).
ident	cualquier secuencia de letras (mayúsculas y minúsculas) y dígitos que empiece por una letra, y que no coincida con ninguna de las palabras reservadas.
opersum	el símbolo ‘+’.
opermul	el símbolo ‘*’.
operinc	el símbolo ‘++’.

Ejercicio 2.4

Diseñar un diagrama de transiciones determinista para reconocer los siguientes componentes léxicos:

letras	cualquier secuencia de una o más letras (mayúsculas y minúsculas);
entero	cualquier secuencia de uno o más dígitos;
explos1	la palabra reservada “bang” (en minúsculas)
explos2	la palabra reservada “boom” (en minúsculas)
true	la secuencia “:-)”
false	la secuencia “:- (“
asignar	la secuencia “:=”

Ejercicio 2.5

Diseñar un diagrama de transiciones determinista para reconocer los siguientes componentes léxicos:

read	la palabra reservada 'read'.
print	la palabra reservada 'print'.
pradir	la palabra reservada 'pradir'.
redir	la palabra reservada 'redir'.
ident	cualquier secuencia de letras y dígitos que empiece por una letra y no coincida con ninguna de las palabras reservadas.
raya	el símbolo '-'.
punto	el símbolo '.'.
uno	el símbolo '-----'.
dos	el símbolo '..---'.

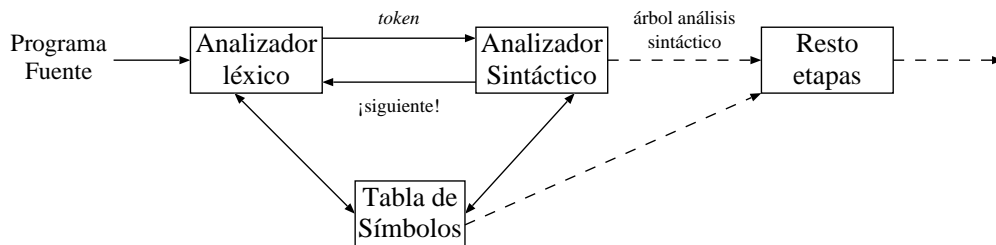
E indíquese cómo separa este analizador la secuencia de entrada
"pradir6dire..--.-".

Capítulo 3

Análisis sintáctico

3.1 La función del analizador sintáctico

Como ya se indicó en la introducción, la principal tarea del analizador sintáctico (o *parser*) no es comprobar que la sintaxis del programa fuente sea correcta, sino construir una representación intermedia de ese programa (necesaria para la traducción) y, en el caso en que sea un programa incorrecto, dar un mensaje de error. Para ello, el analizador sintáctico (AS) comprueba que el orden en que el analizador léxico le va entregando los *tokens* es válido. Si esto es así, significará que la sucesión de símbolos que representan dichos *tokens* puede ser generada por la gramática correspondiente al lenguaje fuente.

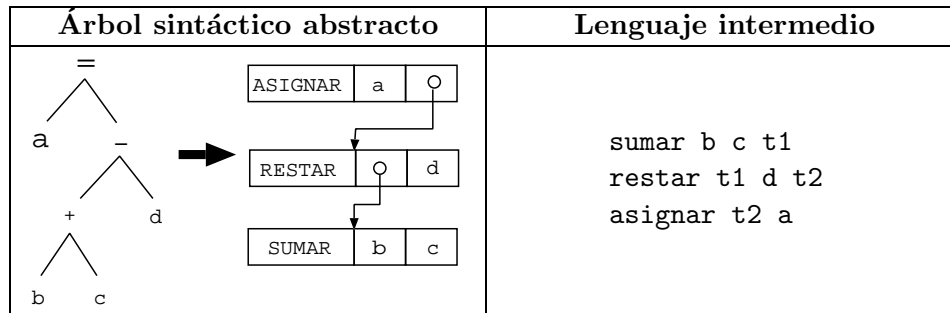


La forma más habitual de representar la sintaxis de un programa es el árbol de análisis sintáctico, y lo que hacen los analizadores sintácticos es construir una derivación por la izquierda o por la derecha del programa fuente, que en realidad son dos recorridos determinados del árbol de análisis sintáctico. A partir de ese recorrido el analizador sintáctico debe construir una representación intermedia de ese programa fuente: un árbol sintáctico abstracto o bien un programa en un lenguaje intermedio; por este motivo, es muy importante que la gramática esté bien diseñada, e incluso es frecuente rediseñar la gramática original para facilitar la tarea de obtener la representación intermedia mediante un analizador sintáctico concreto.

Ejemplo 3.1

Los *árboles sintácticos abstractos* son materializaciones de los árboles de análisis sintáctico en los que se implementan los nodos de éstos, siendo el nodo padre el operador involucrado en cada instrucción y los hijos sus operandos. Por otra

parte, las representaciones intermedias son lenguajes en los que se han eliminado los conceptos de mayor abstracción de los lenguajes de programación de alto nivel. Sea la instrucción “ $a=b+c-d$ ”. A continuación se muestra su representación mediante ambos esquemas citados.



<

El AS constituye el esqueleto principal del compilador. Habitualmente el analizador léxico se implementa como una rutina dentro del sintáctico, al que devuelve el siguiente *token* que encuentre en la entrada cada vez que éste se lo pide. Asimismo, gran parte del resto de etapas de un programa traductor están integradas de una u otra forma en el analizador sintáctico.

Principalmente hay dos opciones para implementar un AS:

1. *a mano*, utilizando una serie de técnicas que se describirán en los siguientes capítulos;
2. utilizando un generador de analizadores sintácticos (p. ej. YACC).

Como siempre, ambos enfoques tienen ventajas e inconvenientes, muy similares al caso de los analizadores léxicos (para el segundo caso el inconveniente de la ineficiencia y la ventaja de la sencillez, y viceversa para el primero).

3.2 Diseño de gramáticas

El diseño de gramáticas para lenguajes de programación es una materia que difícilmente puede enseñarse en su totalidad, sino que debe ser aprendida en la mayor medida posible. Sin embargo, la forma de recoger parte de la semántica de los operadores en la gramática es bastante sencilla de explicar. A continuación vamos a ver cómo se plasma en el aspecto de las reglas sintácticas algunas propiedades de los operadores y operandos en los lenguajes de programación.

3.2.1 Recursividad

Una de las principales dificultades a la hora de diseñar un compilador es que debe procesar correctamente un número en principio infinito de programas distintos. Por otro lado, es evidente que la especificación sintáctica de un lenguaje debe ser finita. El concepto que hace compatible las dos afirmaciones anteriores es el de recursividad. Ésta nos permite definir sentencias complicadas con un número pequeño de sencillas reglas de producción.

Estructura de la recursividad

1. Una o más reglas no recursivas que se definen como caso base.
2. Una o más reglas recursivas que permiten el crecimiento a partir del caso base.

Ejemplo 3.2

Supongamos que queremos expresar la estructura de un tren formado por una locomotora y un número cualquiera de vagones detrás. Si lo hicieramos de esta forma:

```
tren  →  locomotora
tren  →  locomotora vagón
tren  →  locomotora vagón vagón
...
```

necesitaríamos infinitas reglas de derivación (una por cada número de vagones posibles en el tren). Para expresar lo mismo con un par de sentencias podemos utilizar la recursividad de la siguiente manera:

1. definimos la regla base (no recursiva), que define el concepto elemental de partida y que en este caso sería:
tren → locomotora
2. definimos una o más reglas recursivas que permitan el crecimiento ilimitado de la estructura partiendo del concepto elemental anterior. En este caso:
tren → tren vagón

<

Ejemplo 3.3

Un elemento común en muchos lenguajes de programación son las declaraciones de variables. La siguiente gramática independiente del contexto genera una secuencia de uno o más identificadores separados por comas:

N	=	{ Lista }	Notación: N = cjto. de no terminales T = cjto. de terminales P = cjto. de producciones S = símbolo inicial o axioma
T	=	{ ident coma }	
P	=	{ Lista → ident Lista → Lista coma ident }	
S	=	Lista	

<

Definición: Una gramática se dice que es *recursiva* si podemos hacer una derivación (sucesión de una o más producciones) de un símbolo no terminal tras la cual nos vuelve a aparecer dicho símbolo entre los símbolos de la parte derecha de la derivación $A \xRightarrow{*} \alpha A \beta$.

Unos casos especiales de recursividad son aquellos en los que aparecen derivaciones como $A \xRightarrow{*} A \beta$ o $A \xRightarrow{*} \alpha A$ y se denominan *recursividad por la izquierda* y *por la derecha*, respectivamente.

Un no terminal A se dice que es recursivo si a partir de A se puede derivar una forma sentencial en que aparece él mismo en la parte derecha.

3.2.2 Ambigüedad

Una gramática es *ambigua* si el lenguaje que define contiene alguna cadena que tenga más de un árbol de análisis sintáctico para esa gramática. Es decir, si se puede construir más de un árbol de análisis sintáctico, quiere decir que esa sentencia puede “significar” cosas diferentes (tiene más de una interpretación), por lo que tiene varias traducciones. Una gramática es *no ambigua* cuando cualquier cadena del lenguaje tiene un único árbol sintáctico.

Como veremos más adelante, no es posible construir analizadores sintácticos eficientes para gramáticas ambiguas y, lo que es peor, al poderse obtener más de un árbol sintáctico para la misma cadena de entrada, es complicado conseguir en todos los casos la misma representación intermedia. Por estos motivos debemos evitar diseñar gramáticas ambiguas para los lenguajes de programación; aunque no disponemos de técnicas para saber a priori si una gramática es ambigua o no, si ponemos ciertas restricciones a la gramática estaremos en condiciones de afirmar que no es ambigua.

La única forma de saber que una gramática es ambigua es encontrando una cadena con dos o más árboles sintácticos distintos (o dos derivaciones por la izquierda). Las gramáticas que vamos a utilizar normalmente generan lenguajes infinitos, por tanto no es posible encontrar en todos los casos en un tiempo finito dicha cadena. Sin embargo, si la gramática tiene alguna de las siguientes características, es sencillo encontrar una cadena con dos o más árboles:

- Gramáticas con ciclos simples o menos simples:

$$\begin{array}{lcl} S & \longrightarrow & A \\ S & \longrightarrow & a \\ A & \longrightarrow & S \end{array}$$

- Alguna regla con una forma

$$E \longrightarrow E \dots E$$

con cualquier cadena de terminales y no terminales entre las dos E . Es posible que con algún terminal antes de la primera E o algún terminal después de la última E pueda producirse también ambigüedad; por ejemplo, el `if-then-else` de Pascal y C es fácil que se exprese con una construcción ambigua.

- Un conjunto de reglas que ofrezcan caminos alternativos entre dos puntos, como en:

$$\begin{array}{lcl} S & \longrightarrow & A \\ S & \longrightarrow & B \\ A & \longrightarrow & B \end{array}$$

- Producciones recursivas en las que las variables no recursivas de la producción puedan derivar a la cadena vacía:

$$\begin{array}{lcl} S & \longrightarrow & H R S \\ S & \longrightarrow & \mathbf{s} \\ H & \longrightarrow & \mathbf{h} \mid \epsilon \\ R & \longrightarrow & \mathbf{r} \mid \epsilon \end{array}$$

- Símbolos no terminales que puedan derivar a la cadena vacía y a la misma cadena de terminales, y que aparezcan juntas en la parte derecha de una regla o en alguna forma sentencial:

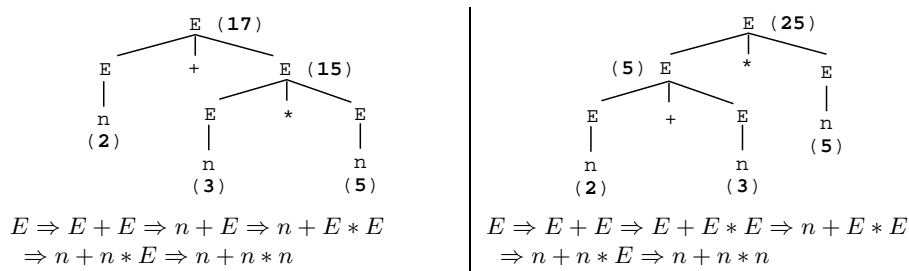
$$\begin{aligned} S &\longrightarrow H R \\ H &\longrightarrow \mathbf{h} \mid \epsilon \\ R &\longrightarrow \mathbf{r} \mid \mathbf{h} \mid \epsilon \end{aligned}$$

Ejemplo 3.4

Sea una gramática cuyas reglas de producción son:

$$\begin{aligned} E &\longrightarrow E + E \\ E &\longrightarrow E * E \\ E &\longrightarrow \mathbf{n} \\ E &\longrightarrow (E) \end{aligned}$$

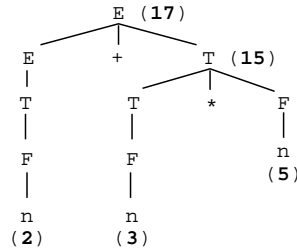
Con estas reglas se puede generar la cadena “2+3*5” que tiene dos posibles árboles sintácticos. En función de que se escoja uno u otro, el resultado de evaluar dicha expresión matemática es distinto, como se ve a continuación:



Para solucionar esta ambigüedad se debe modificar las reglas de producción de la gramática. En este caso, se trata de distinguir en estas expresiones matemáticas lo que es un factor y lo que es un término (producto de dos factores — monomio). Así se establece la jerarquía de precedencias de los operadores:

$$\begin{aligned} E &\longrightarrow E + T \mid T \\ T &\longrightarrow T * F \mid F \\ F &\longrightarrow (E) \mid \mathbf{n} \end{aligned}$$

De esta forma sólo hay un posible árbol sintáctico para “2+3*5”:



3.2.3 Asociatividad y precedencia de los operadores

Asociatividad

La *asociatividad* de un operador binario define cómo se operan tres o más operandos; cuando se dice que la asociatividad de un operador es *por la izquierda* se quiere decir que si aparecen tres o más operandos se evalúan de izquierda a derecha: primero se evalúan los dos operandos de más a la izquierda, y el resultado de esa operación se opera con el siguiente operando por la izquierda, y así sucesivamente.

Si la asociatividad del operador es *por la derecha*, los operandos se evalúan de derecha a izquierda. En los lenguajes de programación imperativos más utilizados (Pascal, C, C++, Java, etc.) la asociatividad de la mayoría de los operadores y en particular la de los operadores aritméticos es por la izquierda. Por el contrario, en el lenguaje APL, que es un lenguaje orientado al cálculo numérico, la asociatividad de todos los operadores es por la derecha.

Ejemplo 3.5

Si la asociatividad del operador “#” es por la izquierda, la expresión “2#a#7.5” se evalúa operando primero el “2” con la variable “a”, y operando después el resultado con “7.5”. Si la asociatividad fuera por la derecha, primero se operarían la variable “a” con el “7.5”, y después se operaría el “2” con el resultado de esa operación. La posición de los operandos con respecto al operador suele ser importante, ya que aunque algunos operadores son conmutativos, la mayoría no lo son.

<

Ejemplo 3.6

Existen lenguajes que combinan operadores asociativos por la izquierda con otros asociativos por la derecha; en FORTRAN existen cinco operadores aritméticos: suma (“+”), resta (“-”), multiplicación (“*”), división (“/”) y exponenciación (“**”). Los cuatro primeros son asociativos por la izquierda, mientras que el último lo es por la derecha. Así, tendremos las siguientes equivalencias:

- “A/B/C” se evalúa como “A/B” y el resultado se divide por “C”.
- “X**Y**Z” se evalúa como “Y**Z” y “X” se eleva al resultado.

<

La forma de reflejar la asociatividad de un operador en la gramática es la siguiente: cuando la asociatividad del operador es por la izquierda, la regla sintáctica en la que interviene dicho operador debe ser recursiva por la izquierda, y cuando es por la derecha, la regla en la que interviene debe tener recursión por la derecha. Para comprender estas reglas basta con pensar cómo se desarrollarán los árboles sintácticos con ambos tipos de recursividad y cómo se operará en los nodos del árbol a la subida de un recorrido en profundidad por la izquierda.

Precedencia

La precedencia de un operador especifica el orden relativo de cada operador con respecto a los demás operadores; de esta manera, si un operador “#” tiene mayor precedencia que otro operador “%”, cuando en una expresión aparezcan los dos operadores, se debe evaluar primero el operador con mayor precedencia.

Ejemplo 3.7

Con los operadores definidos más arriba, si aparece una expresión como “2%3#4”, al tener el operador “#” mayor precedencia, primero se operarían el “3” y el “4”, y después se operaría el “2” con el resultado de esa operación.

<

Siguiendo los criterios aritméticos habituales, en la mayoría de los lenguajes de programación los operadores multiplicativos tienen mayor precedencia que los aditivos, por lo que cuando se mezclan ambos tipos de operaciones en una misma sentencia, se evalúan las multiplicaciones y divisiones de izquierda a derecha antes que las sumas y restas. Existen excepciones: en el lenguaje *Smalltalk* no existe precedencia ni asociatividad, todos los operandos se evalúan de izquierda a derecha sin importar lo que aparezca más a la izquierda de ellos; los conceptos de precedencia y asociatividad se establecen indirectamente por medio de los paréntesis.

La forma de reflejar la precedencia de los operadores aritméticos en una gramática es bastante sencilla. Es necesario utilizar una variable en la gramática por cada operador de distinta precedencia. Cuanto más “cerca” esté la producción de la del símbolo inicial, menor será la precedencia del operador involucrado. La noción de cercanía tiene que ver con el número de producciones que hay que llevar a cabo para llegar hasta esa regla desde el símbolo inicial.

Parentización

En la mayoría de los lenguajes de programación se utilizan los paréntesis (que son operadores especiales que siempre tienen la máxima precedencia) para agrupar los operadores según la conveniencia del programador y sortear las precedencias y asociatividades definidas en el lenguaje.

Para incluirlos en la gramática, se añade una variable que produzca expresiones entre paréntesis y los operandos (números, variables, etc.) a la mayor distancia posible del símbolo inicial. En esta producción también se pondrían los operadores unarios a no ser que tengan una precedencia menor (véase más adelante).

Ejemplo 3.8

Supónganse los operadores “+”, “-”, con asociatividad por la izquierda (en “6-3-2”, se calcula primero “6-3” y después se le resta el “2”) y los operadores “*”, “/” con asociatividad por la derecha (al contrario de lo habitual en los lenguajes C y Pascal, por ejemplo). Sean para los dos tipos de operadores la precedencia habitual en los lenguajes de programación (“*” y “/” tienen mayor precedencia y, por tanto,

se evalúan antes que las sumas y las restas) y los paréntesis tienen la máxima precedencia. La gramática que genera las expresiones con estos operadores y además recoge la asociatividad y la precedencia es la siguiente:

$$\begin{aligned}
 E &\longrightarrow E + T \\
 E &\longrightarrow E - T \\
 E &\longrightarrow T \\
 T &\longrightarrow F * T \\
 T &\longrightarrow F / T \\
 T &\longrightarrow F \\
 F &\longrightarrow (E) \\
 F &\longrightarrow \text{número}
 \end{aligned}$$

Se dice que esta gramática refleja la precedencia y asociatividad de los operadores porque si construimos el árbol sintáctico para una cadena cualquiera, p. ej. “12-4-6/2/2”, veremos que los operandos están agrupados según su asociatividad y precedencia en las ramas del árbol, lo cual facilitará su traducción en un recorrido del árbol.

<

Es importante que la gramática refleje la precedencia y asociatividad de los operadores puesto que en la mayoría de los lenguajes objeto los operadores no tienen precedencia o asociatividad (normalmente porque no pueden aparecer expresiones aritméticas complejas), y por tanto si el árbol sintáctico mantiene correctamente agrupados los operandos de dos en dos será más sencillo traducir la expresión a un lenguaje objeto típico, como puede ser el código máquina de un procesador cualquiera.

La forma de recoger la semántica de los operadores unarios y ternarios depende de cada caso, por lo que es necesario estudiar bien el comportamiento de estos operadores para diseñar una gramática para ellos.

Ejemplo 3.9

En el caso del operador “!” de C (el “not” de Pascal) la semántica permite que aparezcan varios operadores seguidos (como por ejemplo “!!!!0”). En este caso, habría que añadir una regla como la siguiente a la gramática del ejemplo anterior (dependiendo de la precedencia del operador):

$$F \longrightarrow ! F$$

Sin embargo, el caso del operador de signo, la semántica de muchos lenguajes como C, C++, Pascal, etc. no permite que aparezca más de un signo delante de un término, y además se especifica que el signo afecta a todo el término. En el ejemplo anterior, habría que añadir las siguientes reglas:

$$\begin{aligned}
 E &\longrightarrow - T \\
 E &\longrightarrow + T
 \end{aligned}$$

<

3.3 Tipos de análisis sintáctico

Existen algoritmos de análisis sintáctico para cualquier gramática independiente del contexto (incluidas las ambiguas); los más conocidos son el de Earley y el de Cocke, Younger y Kasami (CYK). Ambos tienen un coste temporal del orden de $O(n^3)$; este es un coste demasiado elevado para un compilador, por lo que es necesario buscar subclases de gramáticas que permitan un análisis sintáctico en tiempo lineal. Además, estos algoritmos no son adecuados para guiar la traducción del programa fuente, que es la principal misión del analizador sintáctico en un compilador.

Existen dos estrategias para el análisis sintáctico lineal, que construyen el árbol sintáctico (o lo recorren) de diferente manera:

- **Análisis descendente:** partimos de la raíz del árbol (donde estará situado el axioma o símbolo inicial de la gramática) y se van aplicando reglas por la izquierda de forma que se obtiene una derivación por la izquierda del símbolo inicial. Para decidir qué regla aplicar, se lee uno o más *tokens* de la entrada. Recorriendo el árbol de análisis sintáctico resultante, en profundidad de izquierda a derecha, encontraremos en las hojas del árbol los *tokens* que nos devuelve el analizador léxico en ese mismo orden.
- **Análisis ascendente:** partiendo de la cadena de entrada, se construye el árbol de análisis sintáctico empezando por las hojas (donde están los *tokens*) y se van creando nodos intermedios hasta llegar a la raíz (hasta el símbolo inicial), construyendo así el árbol de abajo a arriba. En un recorrido en profundidad del árbol de izquierda a derecha también se encontrarán en las hojas los *tokens* en el orden entregado por el analizador léxico. El orden en el que el analizador va entregando las producciones corresponde a la inversa de una derivación por la derecha de la cadena de entrada.

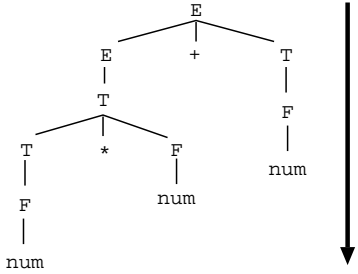
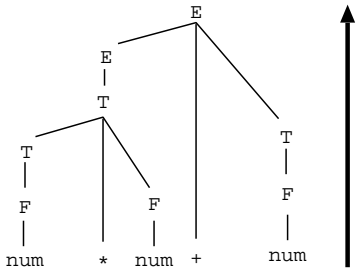
Las dos estrategias recorren la cadena de entrada de izquierda a derecha una sola vez y necesitan que la gramática no sea ambigua.

Ejemplo 3.10

Dada la entrada “num*num+num” y dada la gramática:

$$\begin{aligned} E &\longrightarrow E + T \mid T \\ T &\longrightarrow T * F \mid F \\ F &\longrightarrow \text{num} \end{aligned}$$

Las derivaciones que se obtendrían según el método de análisis serían:

Árbol descendente	Lista de producciones
	$ \begin{aligned} E &\longrightarrow E + T \\ E &\longrightarrow T \\ T &\longrightarrow T * F \\ T &\longrightarrow F \\ F &\longrightarrow \text{num} \\ F &\longrightarrow \text{num} \\ T &\longrightarrow F \\ F &\longrightarrow \text{num} \end{aligned} $ <p>(derivación por la izquierda)</p>
Árbol ascendente	Lista de producciones
	$ \begin{aligned} F &\longrightarrow \text{num} \\ T &\longrightarrow F \\ F &\longrightarrow \text{num} \\ T &\longrightarrow T * F \\ E &\longrightarrow T \\ F &\longrightarrow \text{num} \\ T &\longrightarrow F \\ E &\longrightarrow E + T \end{aligned} $ <p>(inversa de derivación por la derecha)</p>

Obsérvese que en el análisis descendente, partiendo del símbolo inicial hasta alcanzar las hojas, obtenemos una derivación por la izquierda. En el ascendente, partiendo de las hojas hasta llegar al axioma obtenemos la inversa de una derivación por la derecha.

◁

Ambos tipos de análisis son eficientes (coste lineal $O(n)$) pero no son capaces de trabajar con todo tipo de gramáticas (el análisis de tipo general sí que es capaz de tratar cualquier gramática, pero no es adecuado para el diseño de compiladores). Existen dos grandes familias de gramáticas que permiten un análisis en tiempo lineal, cada una de las cuales es adecuada para un tipo de análisis:

- Análisis descendente: gramáticas $LL(k)$
- Análisis ascendente: gramáticas $LR(k)$

donde:

$L \Rightarrow$ *left to right*: la secuencia de *tokens* de entrada se analiza de izquierda a derecha.

$L/R \Rightarrow$ *left-most/right-most*: obtiene la derivación por la izquierda/derecha.

$k \Rightarrow$ es el número de símbolos de entrada que es necesario conocer en cada momento para poder hacer el análisis.

Por ejemplo, una gramática LL(2) es aquella cuyas cadenas son analizadas de izquierda a derecha, derivando el no terminal que quede por derivar más a la izquierda y para las que es necesario mirar un máximo de dos *tokens* en la entrada para saber qué producción de la gramática tomar en cada instante.

En los siguientes capítulos estudiaremos cómo construir analizadores sintácticos para estas dos clases de gramáticas.

3.4 Referencias bibliográficas

REFERENCIAS	EPÍGRAFES
[Louden, 1997]	3.1, 3.2, 3.3, 3.4 y 3.5.1
[Aho, Sethi y Ullman, 1990]	2.2, 2.4, 4.1, 4.2 y 4.3
[Bennett, 1990]	3.1, 3.2 y 6.1
[Fischer y LeBlanc, 1991]	4.1, 4.4 y 6.12.3

3.5 Ejercicios

Ejercicio 3.1

Diseñar una gramática *no ambigua* para el lenguaje de las expresiones que se pueden construir en Zaskal usando únicamente “**true**”, “**false**” y operadores booleanos. Los operadores de Zaskal son: “**or**” (binario, infijo), “**and**” (binario, infijo), “**not**” (unario, *postfijo*), “(” y “)”. Sin contar los paréntesis, la precedencia relativa de los operadores es

not > **and** > **or**;

además, “**and**” y **or**” son asociativos por la derecha. Por ejemplo, la expresión

(true and false not) or false and true not not

sería una expresión correcta en Zaskal (que se evalúa como **true**, por cierto).

Ejercicio 3.2

Diseñar una gramática *no ambigua* para el lenguaje de las expresiones regulares que se pueden construir con el alfabeto $\{0,1\}$. Los operadores que se usan para construir expresiones regulares son los siguientes (ordenados de menor a mayor precedencia):

$a b$	unión	binario	asociatividad por la izquierda
ab	concatenación	binario	asociatividad por la derecha
a^+, a^*	clausura	unarios	

Algunas expresiones regulares que deben poder ser generadas por la gramática son:

010
 $(01^*|(0|1^+)^*|1)1$
 $(0(1^+)0|1(0^+)1)^*0011$
 $(1100^{*+*})^{+*}$

Ejercicio 3.3

Diseñar una gramática *no ambigua* para los lenguajes que permiten escribir cualquier número de declaraciones de variables enteras, caracteres o reales en Pascal y C.

Ejercicio 3.4

Las reglas siguientes definen el lenguaje **LogPro**. Escribanse las expresiones regulares que definen los *tokens* de este lenguaje y después diseñese una gramática para él.

- Un programa en el lenguaje **LogPro** consta de una secuencia de cero o más hechos o reglas y *una* única pregunta.
- Un hecho es un predicado seguido de un punto.
- Una regla es un predicado seguido del símbolo \leftarrow , la parte derecha de la regla y un punto.
- Una pregunta empieza con el símbolo \leftarrow seguido de la parte derecha de una regla y termina en un punto.
- Un predicado tiene un nombre (que es una secuencia de letras, dígitos y el símbolo `_` (el caracter de subrayado) que empieza por una letra minúscula) y cero o más argumentos separados por comas y encerrados entre paréntesis (al contrario que en C, si no tiene argumentos, no se ponen los paréntesis).
- Un argumento puede ser un número (una secuencia de uno más dígitos sin signo), una variable (una secuencia de letras, dígitos y el símbolo `_`, que empieza por una letra mayúscula o por `_`) o un predicado.
- La parte derecha de una regla es una expresión booleana, y está formada por una secuencia de términos booleanos combinados con los operadores `,` (una coma, representa la operación lógica *and*) y `;` (un punto y coma, representa al *or* lógico). Los dos operandos tienen la asociatividad por la izquierda, y el `;` tiene menor precedencia (se evalúa después) que el `,`. Se pueden utilizar paréntesis para agrupar expresiones, de la misma manera que se utilizan en expresiones de C o Pascal.
- Un término booleano puede ser un predicado o una expresión relacional. Una expresión relacional está formada por dos expresiones aritméticas separadas por un operador relacional, que puede ser uno de estos símbolos: `==`, `!=`, `<`, `<=`, `>`, `>=`.
- Una expresión aritmética está formada por números, variables, paréntesis y los operadores `+`, `-`, `*` y `/`, con la misma precedencia y asociatividad que tienen en lenguajes como C o Pascal (una expresión aritmética en **LogPro** es sintácticamente correcta en C o Pascal).
- De igual forma que en C o Pascal, se pueden utilizar espacios en blanco, tabuladores y cambios de línea para mejorar la legibilidad de un programa en **LogPro**, pero no son elementos del lenguaje.
- Un argumento de un predicado puede ser también una lista, que está formada por un corchete izquierdo, una secuencia de elementos separados por comas, una cola opcional y un corchete derecho. Cualquier tipo de argumento de un predicado puede ser un elemento de una lista (se puede construir una lista de listas), y la cola (que puede aparecer o no) está formada por el símbolo `|` seguido de una lista. La cola, si aparece, va situada inmediatamente antes del corchete derecho (aunque puede haber blancos entre estos elementos).

Ejemplo:

```

preD1      .
preD2(23,_23,f(a)) <- (eurt,eslaf;cierto), 2+3*4<= 3, X == 5.
preD3(Predicado, [l,i,s,t,a
               |[c,o,l,a]]).
<-preD4(preD2(Y,2,f(X)),preD1)
.

```

Ejercicio 3.5

Diseñar una gramática *no ambigua* que genere el lenguaje G definido por las siguientes frases:

1. Un programa en G es una secuencia de uno o más métodos.
2. Un método está formado por una declaración de variables y una secuencia de cero o más mensajes acabada en una sentencia de retorno.
3. Una declaración de variables está formada por una barra vertical ('|'), una secuencia de cero o más identificadores y otra barra vertical.
4. Un mensaje puede ser una asignación, una expresión o un mensaje de método.
5. Un mensaje de asignación está formado por una variable, el simbolo ':= ' y un mensaje.
6. Las expresiones pueden contener números enteros, variables, los operadores '+', '-', '*', '/', con la misma asociatividad que en C o Pascal, pero *sin precedencia*. Una expresión puede ser un mensaje entre paréntesis.
7. Un mensaje de método está formado por un identificador, un corchete izquierdo, una secuencia de cero o más expresiones separadas por un punto y coma entre cada dos expresiones, y un corchete derecho.
8. Una sentencia de retorno está formada por el símbolo '^' y una expresión.

Ejercicio 3.6

Diseñar una gramática *no ambigua* que genere (carácter a carácter) el lenguaje de todos los números enteros sin signo que sean pares, considerando que el 0 es par también. Algunos números que tendría que generar esta gramática son:

```

0
234
11112
2350
00078

```

Capítulo 4

Análisis sintáctico descendente

4.1 Introducción

4.1.1 Análisis con retroceso

El análisis sintáctico descendente (ASD) intenta encontrar entre las producciones de la gramática la derivación por la izquierda del símbolo inicial para una cadena de entrada. Una forma intuitiva de realizar un análisis descendente sería utilizando *backtracking* o análisis con retroceso, que puede implicar examinar varias veces la entrada. Sin embargo, en la práctica no se suele utilizar esta técnica ya que su complejidad es exponencial en el peor caso y, como hemos comentado en el capítulo anterior, es necesario que el analizador sintáctico tenga (como todo o casi todo el compilador) una complejidad lineal con respecto al tamaño del programa fuente.

Ejemplo 4.1

$$\begin{array}{lcl} S & \longrightarrow & \mathbf{c} \ A \ \mathbf{d} \\ A & \longrightarrow & \mathbf{a} \ \mathbf{b} \mid \mathbf{a} \end{array}$$

Analicemos la cadena de entrada: “cad”

1. En la situación en la que el símbolo analizado es el primero, “cad”, la única producción que se puede escoger es la primera; luego $S \Rightarrow cAd$, y el primer símbolo de la entrada “c” queda emparejado con la primera hoja izquierda del árbol que también es “c”, con lo que se puede seguir adelante con el análisis.
2. Se avanza la marca de entrada al segundo símbolo, “cad”, y se considera la siguiente hoja del árbol (siempre de izquierda a derecha), etiquetada con A . Entonces se expande el árbol con la primera producción de A , $cAd \Rightarrow cabd$, y como el segundo símbolo de la entrada “a” queda emparejado con la segunda hoja por la izquierda, podemos avanzar la marca de análisis.
3. Se avanza la marca de entrada al símbolo siguiente, “cad”, y se compara con la siguiente hoja etiquetada con “b”. Como no concuerda con “d”, se indica el error y se vuelve a A para ver si hay otra alternativa no intentada, y se reestablece la marca de entrada a la posición que ocupaba entonces.

4. Con la marca en “cad” se expande la producción no intentada $A \rightarrow a$, que hace que $cAd \Rightarrow cad$. Ahora el símbolo de entrada “a” coincide con la nueva hoja “a” y se puede proseguir el análisis.
5. Se avanza de nuevo la marca a “cad” y coincide con la hoja de la derecha que quedaba por visitar en el árbol, por lo que se da por finalizado el análisis con éxito.

<

4.1.2 Analizadores sintácticos predictivos

Para que el algoritmo tenga una complejidad lineal, siempre debe saber qué regla se debe aplicar, por lo que es necesario que el analizador realice una *predicción* de la regla a aplicar. Para ello, se debe conocer, dado el *token* de la entrada, a , que esté siendo analizado y el no terminal a expandir A , cuál de las alternativas de producción $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ es la única posible que da lugar a que el resto de la cadena que se está analizando empiece por a . Dicho de otra manera, la alternativa apropiada debe poderse predecir sólo con ver el primer símbolo que produce (como así sucede en la mayoría de lenguajes de programación). Veremos qué forma deben tener las gramáticas a las que se puede aplicar esta metodología.

Ejemplo 4.2

$$\begin{aligned} Sent &\longrightarrow \text{if } \textit{Expres} \text{ then } Sent \\ Sent &\longrightarrow \text{while } \textit{Expres} \text{ do } Sent \\ Sent &\longrightarrow \text{begin } Sent \text{ end} \end{aligned}$$

En esta gramática siempre existe sólo una posibilidad de derivación, según que el primer símbolo que haya en la entrada en el momento de tomar esa decisión sea **if**, **while** o **begin**.

<

Según la nomenclatura que ya hemos introducido, las gramáticas que son susceptibles de ser analizadas sintácticamente de forma descendente mediante un análisis predictivo y consultando únicamente un símbolo de entrada pertenecen al conjunto de gramáticas denominado LL(1). En un análisis de izquierda a derecha de la cadena de entrada y haciendo derivaciones por la izquierda, debe bastar con ver un solo símbolo (terminal) en la cadena para saber en cada caso qué producción escoger. A partir de las gramáticas de tipo LL(1) se pueden construir automáticamente *analizadores sintácticos descendentes predictivos* (ASDP), que no son más que ASD sin retroceso.

4.1.3 Conjuntos de predicción

Los conjuntos de predicción son conjuntos de símbolos terminales que ayudan a predecir qué regla se debe aplicar para el no terminal que hay que derivar. Se construyen,

como veremos a continuación, a partir de los símbolos de las partes derechas de las producciones de la gramática.

Para saber qué regla se debe aplicar en cada caso, el analizador consulta el siguiente símbolo en la entrada y si pertenece al conjunto de predicción de una regla (correspondiente al no terminal que hay que derivar), aplica esa regla. Si no puede aplicar ninguna regla, se produce un mensaje de error.

Ejemplo 4.3

Supóngase la siguiente gramática:

$$\begin{array}{lcl} A & \longrightarrow & a B c \mid x C \mid B \\ B & \longrightarrow & b A \\ C & \longrightarrow & c \end{array}$$

y la entrada “babxcc”. Supongamos que el análisis ha progresado a lo largo de los símbolos subrayados: “babxcc”. En esta situación, la cadena de derivaciones habrá sido esta:

$$A \Rightarrow B \Rightarrow bA \Rightarrow baBc \Rightarrow babAc$$

La cuestión en este momento es: ¿qué producción tomar para seguir el análisis? Para seguir analizando, hay que desarrollar la variable A para lo que hay tres posibles opciones. Es fácil, observando las producciones de A en la gramática, darse cuenta que para escoger la primera opción el resto de la cadena debería empezar por “a”; para escoger la segunda, por “x” y para la tercera, por “b”. Como el resto de la cadena es “xcc”, no hay duda de que hay que tomar la segunda opción. Hemos hecho uso de los conjuntos de predicción.

◁

Ejemplo 4.4

La siguiente gramática (de la que se muestra sólo las producciones del no terminal A , pues el resto no influye en este caso) no cumple los requisitos para ser LL(1):

$$\begin{array}{lcl} \dots & & \\ A & \longrightarrow & a B c \mid a C \mid B \\ \dots & & \end{array}$$

Si tenemos que derivar el no terminal A , no podemos saber, viendo un único símbolo en la entrada, cuál es la opción a escoger, pues si aparece “a” en la entrada tenemos dos posibles opciones: la primera y la segunda. Luego el análisis no puede ser predictivo y la gramática no es LL(1).

◁

4.2 Cálculo de los conjuntos de predicción

Los conjuntos de predicción de una regla se calculan en función de los primeros símbolos que puede generar la parte derecha de esa regla y, a veces (cuando esa parte derecha puede generar la cadena vacía), en función de los símbolos que pueden aparecer a continuación de la parte izquierda de la regla en una forma sentencial derivable del símbolo inicial. Para especificar formalmente cómo se calculan los conjuntos de predicción es necesario estudiar antes cómo se calculan los primeros símbolos que genera una cadena de terminales y no terminales (conjunto de *primeros*) y cómo obtener los símbolos que pueden seguir a un no terminal en una forma sentencial (conjunto de *siguientes*).

4.2.1 Cálculo del conjunto de primeros

Dada una gramática $G = (N, T, S, P)$, la función PRIM se aplica a cadenas de símbolos de la gramática ($\alpha \in (T \cup N)^*$) y devuelve un conjunto que puede contener cualesquiera terminales de la gramática y la cadena vacía, ϵ . A partir de ahora, cuando aparezca $\text{PRIM}(\alpha)$, siendo α una cadena de $(T \cup N)^*$, nos referiremos al conjunto resultado de aplicar la función PRIM a la cadena α .

Definición

Si α es una forma sentencial compuesta por una concatenación de símbolos, $\text{PRIM}(\alpha)$ es el conjunto de terminales (o ϵ) que pueden aparecer iniciando las cadenas que pueden derivar (en cero o más pasos) de α .

Definición formal

$$a \in \text{PRIM}(\alpha) \text{ si } a \in (T \cup \{\epsilon\}) \text{ y } \alpha \xRightarrow{*} a\beta$$

para alguna cadena β .

Reglas para el cálculo de $\text{Prim}(\alpha)$

1. Si $\alpha \equiv \epsilon$, $\text{PRIM}(\epsilon) = \{ \epsilon \}$
2. Si $\alpha \in (T \cup N)^+$, $\alpha = a_1 a_2 \dots a_n$, puede darse dos casos:
 - (a) Si $a_1 \equiv a \in T$, $\text{PRIM}(\alpha) = \{ a \}$.
 - (b) Si $a_1 \equiv A \in N$, es necesario calcular $\text{PRIM}(A)$, para lo cual es necesario obtener los primeros de todas las partes derechas de las producciones de A en la gramática:

$$\forall A \rightarrow \alpha_i \in P, \quad 1 \leq i \leq m, \quad \text{PRIM}(A) = \cup_{i=1}^m \text{PRIM}(\alpha_i)$$

Si, después de calcular $\text{PRIM}(A)$, $\epsilon \in \text{PRIM}(A)$ y A no es el último símbolo de α ($A \neq a_n$), entonces

$$\text{PRIM}(\alpha) = (\text{PRIM}(A) - \{ \epsilon \}) \cup \text{PRIM}(a_2 \dots a_n)$$

Tanto si A es el último símbolo de α como si resulta que $\epsilon \notin \text{PRIM}(A)$,

$$\text{PRIM}(\alpha) = \text{PRIM}(A)$$

Como se puede observar, esta última regla es una regla recursiva en la que los casos base de la recursión son los terminales de la gramática y ϵ .

Ejemplo 4.5

Sea la siguiente gramática para expresiones aritméticas con sumas y multiplicaciones:

$$\begin{aligned} E &\longrightarrow T E' \\ E' &\longrightarrow + T E' \mid \epsilon \\ T &\longrightarrow F T' \\ T' &\longrightarrow * F T' \mid \epsilon \\ F &\longrightarrow (E) \mid \text{ident} \end{aligned}$$

Cálculo de los primeros de todos los no terminales de esta gramática:

$$\begin{aligned} \text{PRIM}(E') &= \{ +, \epsilon \} \\ \text{PRIM}(T') &= \{ *, \epsilon \} \\ \text{PRIM}(F) &= \{ (, \text{ident} \} \\ \text{PRIM}(E) &=^{E \rightarrow T E', T \in N} \text{PRIM}(T) =^{T \rightarrow F T', F \in N} \text{PRIM}(F) = \{ (, \text{ident} \} \end{aligned}$$

◁

Ejemplo 4.6

Sea la gramática siguiente:

$$\begin{aligned} A &\longrightarrow A \mathbf{a} \mid B C D \\ B &\longrightarrow \mathbf{b} \mid \epsilon \\ C &\longrightarrow \mathbf{c} \mid \epsilon \\ D &\longrightarrow \mathbf{d} \mid C \mathbf{e} \end{aligned}$$

Calculamos los conjuntos de primeros de todas las variables de esta gramática:

$$\begin{aligned} \text{PRIM}(C) &= \{ \mathbf{c}, \epsilon \} \\ \text{PRIM}(B) &= \{ \mathbf{b}, \epsilon \} \\ \text{PRIM}(D) &= \text{PRIM}(\mathbf{d}) \cup \text{PRIM}(C\mathbf{e}) = \{ \mathbf{d} \} \cup (\text{PRIM}(C) - \{\epsilon\}) \cup \text{PRIM}(\mathbf{e}) = \\ &\quad \{ \mathbf{d}, \mathbf{c}, \mathbf{e} \} \\ \text{PRIM}(A) &= \text{PRIM}(A\mathbf{a}) \cup \text{PRIM}(BCD) = \text{PRIM}(BCD) = \{ \mathbf{b} \} \cup \text{PRIM}(CD) = \\ &\quad \{ \mathbf{b}, \mathbf{c} \} \cup \text{PRIM}(D) = \{ \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e} \} \end{aligned}$$

Si añadimos la regla $D \longrightarrow \epsilon$, hay que cambiar los cálculos, pues habría que añadir ϵ a $\text{PRIM}(D)$ y eso cambiaría el cálculo de $\text{PRIM}(A)$:

$$\text{PRIM}(BCD) = \{ \mathbf{b} \} \cup \text{PRIM}(CD) = \{ \mathbf{b}, \mathbf{c} \} \cup \text{PRIM}(D) = \{ \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \epsilon \}$$

Entonces $\text{PRIM}(A) = \{ \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \epsilon \}$, pero puesto que ahora la regla $A \longrightarrow BCD$ puede derivar a ϵ , eso implica que A puede desaparecer de la primera posición de la regla $A \longrightarrow A\mathbf{a}$ y, por tanto, también hay que añadir “a” al conjunto de $\text{PRIM}(A)$:

$$\text{PRIM}(A) = \{ \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \epsilon, \mathbf{a} \}$$

◁

4.2.2 Cálculo del conjunto de siguientes

La función SIG se aplica a no terminales de la gramática ($A \in N$) y devuelve un conjunto que puede contener cualesquiera terminales de la gramática y un símbolo especial, “\$”, que representa el final de la cadena de entrada (el final del fichero en un compilador).

Definición

Si A es un símbolo no terminal de la gramática, $\text{SIG}(A)$ es el conjunto de terminales (y \$) que pueden aparecer a continuación de A en alguna forma sentencial derivada del símbolo inicial.

Definición formal

$a \in \text{SIG}(A)$ si $a \in (T \cup \{\$\})$ y $\exists \alpha, \beta / S \xRightarrow{*} \alpha A a \beta$ para algún par de cadenas α, β .

Reglas para su cálculo

1. Inicialmente,

$$\text{SIG}(A) = \emptyset$$
2. Si A es el símbolo inicial, entonces

$$\text{SIG}(A) = \text{SIG}(A) \cup \{\$\}$$
3. **(S1)** Para cada regla de la forma $B \longrightarrow \alpha A \beta$

$$\text{SIG}(A) = \text{SIG}(A) \cup (\text{PRIM}(\beta) - \{\epsilon\})$$
4. **(S2)** Para cada regla de la forma $B \longrightarrow \alpha A$ o bien de la forma $B \longrightarrow \alpha A \beta$ en la que $\epsilon \in \text{PRIM}(\beta)$

$$\text{SIG}(A) = \text{SIG}(A) \cup \text{SIG}(B)$$
5. Repetir los pasos 3 y 4 hasta que no se puedan añadir más símbolos a $\text{SIG}(A)$.

Nota: Las reglas **(S1)** y **(S2)** no son excluyentes. Primero habrá que intentar aplicar **(S1)** y luego **(S2)**. Sólo en el caso de producciones del tipo $B \longrightarrow \alpha A$ no tendrá sentido intentar aplicar **(S1)**.

Ejemplo 4.7

En la gramática de las expresiones aritméticas del ejemplo 4.5 calcularemos los conjuntos siguientes de todos los símbolos no terminales:

$$\begin{aligned}
 \text{SIG}(E) &= \{ \$ \} \text{ (porque } E \text{ es el símbolo inicial)} \cup ((\text{S1}) F \rightarrow (E)) \{) \} = \{ \$,) \} \\
 \text{SIG}(E') &= ((\text{S2}) E \rightarrow TE' \text{ y } E' \rightarrow +TE') \text{SIG}(E') (= \emptyset) \cup \text{SIG}(E) = \{ \$,) \} \\
 \text{SIG}(T) &= ((\text{S1}) E \rightarrow TE' \text{ y } E' \rightarrow +TE') (\text{PRIM}(E') - \{\epsilon\}) \cup ((\text{S2}) E' \rightarrow \epsilon) (\text{SIG}(E) \cup \text{SIG}(E')) \\
 &= \{ + \} \cup \{ \$,) \} = \{ +, \$,) \} \\
 \text{SIG}(T') &= ((\text{S2}) T \rightarrow FT' \text{ y } T' \rightarrow *FT') \text{SIG}(T') (= \emptyset) \cup \text{SIG}(T) = \{ +, \$,) \} \\
 \text{SIG}(F) &= ((\text{S1}) T \rightarrow FT' \text{ y } T' \rightarrow *FT') (\text{PRIM}(T') - \{\epsilon\}) \cup ((\text{S2}) T' \rightarrow \epsilon) (\text{SIG}(T) \cup \text{SIG}(T')) \\
 &= \{ * \} \cup \{ +, \$,) \} = \{ *, +, \$,) \}
 \end{aligned}$$

◁

4.2.3 Cálculo del conjunto predicción

La función PRED se aplica a producciones de la gramática ($A \rightarrow \alpha$) y devuelve un conjunto, llamado *conjunto de predicción*, que puede contener cualesquiera terminales de la gramática y el símbolo “\$”, pero nunca puede contener ϵ . Cuando el ASDP tiene que derivar un no terminal, consulta el símbolo de la entrada que espera a ser analizado y lo busca en los conjuntos de predicción de cada regla de ese no terminal (véase el ejemplo 4.3). De esta forma y siempre que los conjuntos de predicción de todas las reglas de cada no terminal por separado sean disjuntos entre sí (aunque puede suceder que dos conjuntos de predicción de no terminales distintos tengan símbolos comunes), el analizador sintáctico puede construir una derivación por la izquierda de la cadena de entrada.

Regla para su cálculo

$$\begin{aligned} \text{PRED}(A \rightarrow \alpha) = \\ \text{si } \epsilon \in \text{PRIM}(\alpha) \text{ entonces } &= (\text{PRIM}(\alpha) - \{ \epsilon \}) \cup \text{SIG}(A) \\ \text{si no } &= \text{PRIM}(\alpha) \end{aligned}$$

Ejemplo 4.8

Supóngase la siguiente gramática:

$$\begin{aligned} S &\rightarrow A B \mid s \\ A &\rightarrow a S c \mid e B f \mid \epsilon \\ B &\rightarrow b A d \mid \epsilon \end{aligned}$$

Calculamos los conjuntos de predicción utilizando la regla adecuada en cada caso:

$$\begin{aligned} \text{PRED}(S \rightarrow AB) &= (\text{PRIM}(AB) - \{\epsilon\}) \cup \text{SIG}(S) \\ &= \{ a, e, b, \$, c \} \\ \text{PRED}(S \rightarrow s) &= \text{PRIM}(s) = \{ s \} \\ \text{PRED}(A \rightarrow aSc) &= \text{PRIM}(aSc) = \{ a \} \\ \text{PRED}(A \rightarrow eBf) &= \text{PRIM}(eBf) = \{ e \} \\ \text{PRED}(A \rightarrow \epsilon) &= (\text{PRIM}(\epsilon) - \{\epsilon\}) \cup \text{SIG}(A) = \{ d, b, \$, c \} \\ \text{PRED}(B \rightarrow bAd) &= \text{PRIM}(bAd) = \{ b \} \\ \text{PRED}(B \rightarrow \epsilon) &= (\text{PRIM}(\epsilon) - \{\epsilon\}) \cup \text{SIG}(B) = \{ f, \$, c \} \end{aligned}$$

¿Podía construirse un ASDP a la vista de estos conjuntos?

◁

4.3 La condición LL(1)

Para que una gramática pertenezca al conjunto de gramáticas LL(1) ha de cumplir la condición LL(1). Esta condición no “salta a la vista” a partir del aspecto de las producciones de la gramática, sino que tiene que ver con el contenido de los conjuntos de predicción de las reglas que derivan de un mismo no terminal.

Para que la regla a aplicar sea siempre única, se debe exigir que los conjuntos de predicción de las reglas de cada no terminal sean disjuntos entre sí; es decir, no puede

haber ningún símbolo terminal que pertenezca a dos o más conjuntos de predicción de las reglas de un mismo no terminal. Si se cumple esta condición, la gramática es LL(1) y se puede realizar su análisis sintáctico en tiempo lineal.

La condición LL(1) es necesaria y suficiente para poder construir un ASDP para una gramática. Una gramática que cumple la condición LL(1) se dice que es una gramática LL(1). Estas gramáticas tienen las siguientes propiedades o características para su análisis:

- La secuencia de *tokens* se analiza de izquierda a derecha.
- Siempre se deriva el no terminal que aparezca más a la izquierda.
- Sólo será necesario ver un *token* de la secuencia de entrada para averiguar qué producción seguir.

Condición LL(1)

Dadas todas las producciones de la gramática para un mismo terminal:

$$A \longrightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n \quad \forall A \in N$$

se debe cumplir la siguiente condición:

$$\forall i, j \ (i \neq j) \quad \text{PRED}(A \rightarrow \alpha_i) \cap \text{PRED}(A \rightarrow \alpha_j) = \emptyset$$

Ejemplo 4.9

Sea la siguiente gramática con sus conjuntos de predicción ya calculados para cada regla:

$$\begin{array}{lll} A & \longrightarrow & \mathbf{a} \mathbf{b} B \quad \{\mathbf{a}\} \\ A & \longrightarrow & B \mathbf{b} \quad \{\mathbf{b}, \mathbf{c}\} \\ B & \longrightarrow & \mathbf{b} \quad \{\mathbf{b}\} \\ B & \longrightarrow & \mathbf{c} \quad \{\mathbf{c}\} \end{array}$$

Se puede afirmar que es LL(1) porque los conjuntos de predicción de las dos reglas del no terminal A son disjuntos entre sí, y los conjuntos de predicción de las reglas de B también lo son. Como se puede comprobar, los símbolos \mathbf{b} y \mathbf{c} pertenecen a varios conjuntos de predicción de reglas de diferentes no terminales y, sin embargo, la gramática sigue siendo LL(1). Si añadimos la regla

$$B \longrightarrow \mathbf{a}$$

los conjuntos de predicción quedarían de la siguiente manera:

$$\begin{array}{lll} A & \longrightarrow & \mathbf{a} \mathbf{b} B \quad \{\mathbf{a}\} \\ A & \longrightarrow & B \mathbf{b} \quad \{\mathbf{a}, \mathbf{b}, \mathbf{c}\} \\ B & \longrightarrow & \mathbf{b} \quad \{\mathbf{b}\} \\ B & \longrightarrow & \mathbf{c} \quad \{\mathbf{c}\} \\ B & \longrightarrow & \mathbf{a} \quad \{\mathbf{a}\} \end{array}$$

Con esta nueva regla, los conjuntos de predicción de las reglas del no terminal A ya no son disjuntos (el símbolo **a** pertenece a ambos) y por tanto la gramática no es LL(1), independientemente de si los conjuntos de predicción de las reglas de B son o no disjuntos.

<

Ejemplo 4.10

Sea la siguiente gramática:

$$\begin{aligned} E &\longrightarrow E + T \mid T \\ T &\longrightarrow T * F \mid F \\ F &\longrightarrow \mathbf{num} \mid (E) \end{aligned}$$

Se trata de estudiar si cumple la condición LL(1). Para ello se calculan los conjuntos de predicción:

$$\begin{aligned} \text{PRED}(E \longrightarrow E + T) &= \text{PRIM}(E + T) = \{ \mathbf{num}, (\} \\ \text{PRED}(E \longrightarrow T) &= \text{PRIM}(T) = \{ \mathbf{num}, (\} \\ \text{PRED}(T \longrightarrow T * F) &= \text{PRIM}(T * F) = \{ \mathbf{num}, (\} \\ \text{PRED}(T \longrightarrow F) &= \text{PRIM}(F) = \{ \mathbf{num}, (\} \\ \text{PRED}(F \longrightarrow \mathbf{num}) &= \text{PRIM}(\mathbf{num}) = \{ \mathbf{num} \} \\ \text{PRED}(F \longrightarrow (E)) &= \text{PRIM}((E)) = \{ (\} \end{aligned}$$

Para el símbolo F , la intersección de los conjuntos de predicción de todas las reglas en las que se desarrolla es:

$$\text{PRED}(F \longrightarrow \mathbf{num}) \cap \text{PRED}(F \longrightarrow (E)) = \{ \mathbf{num} \} \cap \{ (\} = \emptyset$$

pero no sucede lo mismo con los conjuntos de predicción de las producciones de T , que son iguales (por lo tanto, no disjuntos), por lo que la gramática no cumple la condición LL(1). Con el no terminal E ocurre lo mismo que con T .

<

Ejemplo 4.11

¿Cumple la condición LL(1) la siguiente gramática?

$$\begin{aligned} E &\longrightarrow T E' \\ E' &\longrightarrow + T E' \mid \epsilon \\ T &\longrightarrow F T' \\ T' &\longrightarrow * F T' \mid \epsilon \\ F &\longrightarrow \mathbf{num} \mid (E) \end{aligned}$$

No hace falta calcular los conjuntos de predicción de aquellas variables que no tienen más que una opción para su desarrollo. Si sólo hay una opción, no se planteará nunca dudas sobre qué opción elegir. Sólo aquellas variables con dos o más alternativas son las que hay que estudiar para ver si sus conjuntos de predicción son disjuntos. Por lo tanto, en este ejemplo no hace falta calcular $\text{PRED}(E \rightarrow TE')$ ni $\text{PRED}(T \rightarrow FT')$. Vamos a ver qué ocurre con los restantes.

$$\begin{aligned}\text{PRED}(E' \longrightarrow + TE') &= \{ + \} \\ \text{PRED}(E' \longrightarrow \epsilon) &= \text{SIG}(E') = \{ \$,) \}\end{aligned}$$

$$\begin{aligned}\text{PRED}(T' \longrightarrow * FT') &= \{ * \} \\ \text{PRED}(T' \longrightarrow \epsilon) &= \text{SIG}(T') = \{ +, \$,) \}\end{aligned}$$

$$\begin{aligned}\text{PRED}(F \longrightarrow \text{num}) &= \{ \text{num} \} \\ \text{PRED}(F \longrightarrow (E)) &= \{ (\}\end{aligned}$$

Ahora, para cada uno de estos no terminales con alternativas, comprobamos si los conjuntos de predicción son disjuntos entre sí:

$$\begin{aligned}\text{PRED}(E' \longrightarrow + TE') \cap \text{PRED}(E' \longrightarrow \epsilon) &= \{ + \} \cap \{), \$ \} = \emptyset \\ \text{PRED}(T' \longrightarrow * FT') \cap \text{PRED}(T' \longrightarrow \epsilon) &= \{ * \} \cap \{ +,), \$ \} = \emptyset \\ \text{PRED}(F \longrightarrow \text{num}) \cap \text{PRED}(F \longrightarrow (E)) &= \{ \text{num} \} \cap \{ (\} = \emptyset\end{aligned}$$

Luego esta gramática cumple la condición LL(1).

◁

4.4 Modificación de gramáticas no LL(1)

Existen algunas características que, en el caso de ser observadas en una gramática, garantizan que no es LL(1) (sin necesidad de calcular los conjuntos de predicción); sin embargo, si ninguna de estas características aparece, la gramática puede que sea LL(1) o puede que no lo sea (en este caso sí que hay que calcular los conjuntos de predicción para comprobarlo). También, si la gramática no es LL(1) no necesariamente debe tener alguna de estas características; puede que tenga alguna, puede que las tenga todas o puede que no tenga ninguna de ellas. Estas características son la ambigüedad, los factores comunes por la izquierda y la recursividad izquierda.

Cualquier gramática recursiva por la izquierda o con símbolos comunes por la izquierda en algunas producciones no será LL(1). Si en una gramática se elimina (como veremos más adelante) su recursividad por la izquierda, se factoriza por la izquierda (si tuviera factores comunes por ese lado) y no presenta ambigüedad¹, la gramática modificada resultante podría ser LL(1) (analizable por un ASDP), pero también podría resultar que no lo fuera, como la siguiente gramática:

$$\begin{aligned}S &\longrightarrow A \mid B \\ A &\longrightarrow \mathbf{a} \mathbf{a} \\ B &\longrightarrow C \mathbf{b} \\ C &\longrightarrow \mathbf{a} A\end{aligned}$$

Por tanto, que una gramática no sea recursiva por la izquierda, no tenga factores comunes por la izquierda y no sea ambigua es una condición necesaria para que sea LL(1), pero no suficiente.

Por otra parte, veremos que si nos encontramos una gramática que no sea LL(1) existen métodos para modificarla y convertirla en una gramática que pudiera ser LL(1).

¹Si tiene ambigüedad, en algún momento el analizador no sabrá qué producción seleccionar, puesto que hay al menos dos posibles análisis.

4.4.1 Eliminación de la ambigüedad

Cualquier gramática ambigua no cumple la condición LL(1) ya que, por la propia definición de esta propiedad, en algún caso no sabremos qué producción tomar a la vista de un único símbolo, pues habrá más de un posible árbol de análisis sintáctico. Esto no quiere decir que si la gramática no es ambigua entonces será LL(1), pues puede presentar otros problemas, como veremos a continuación.

El problema de la ambigüedad es el más difícil de resolver pues no existe una metodología para eliminarla y tampoco hay otra fórmula para saber que una gramática es ambigua más que la de encontrar alguna sentencia que tenga dos o más árboles de análisis sintáctico distintos.

La mejor opción cuando se presenta una gramática ambigua es replantearse el diseño de la misma para encontrar una gramática no ambigua equivalente (que genere el mismo lenguaje).

4.4.2 Factorización por la izquierda

Si dos producciones alternativas de un símbolo A empiezan igual, no se sabrá por cuál de ellas seguir. Se trata de reescribir las producciones de A para retrasar la decisión hasta haber visto lo suficiente de la entrada como para elegir la opción correcta.

Ejemplo 4.12

Sean las producciones:

$$\begin{aligned} Sent &\longrightarrow \text{if Expr then Sent else Sent} \\ Sent &\longrightarrow \text{if Expr then Sent} \\ Sent &\longrightarrow Otras \end{aligned}$$

Al ver “if” no se sabe cuál de las dos producciones hay que tomar para expandir *Sent*. De hecho, un analizador sintáctico descendente no sabría qué hacer hasta superar el cuarto símbolo de esta producción. Si entonces llega “else” ya sabe que se trataba de la primera y si entra cualquier otro *token* entonces se trataba de la segunda. Pero esto ocurre demasiado *tarde* para el análisis sintáctico predictivo.

◁

Nos enfrentamos, pues, al problema de producciones que tienen símbolos comunes por la izquierda; es decir, si son del tipo $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$. En estos casos, ante una entrada del prefijo α , no sabemos si derivar por $\alpha\beta_1$ o por $\alpha\beta_2$. La solución pasa por modificar las producciones afectadas cambiando

$$A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2$$

por dos producciones:

$$\begin{aligned} A &\longrightarrow \alpha A' \\ A' &\longrightarrow \beta_1 \mid \beta_2 \end{aligned}$$

Con esto se retrasa el momento de la decisión hasta después de analizar los símbolos comunes y se soluciona uno de los problemas que tenía esa gramática para no cumplir la condición LL(1).

Veamos la regla general para factorizar por la izquierda una gramática. Para todos los no terminales, A , de la gramática, se debe encontrar el prefijo α más largo común a dos o más producciones de A , pero siempre aquel que sea común a más producciones. Es posible que dos producciones presenten un prefijo común γ pero exista otra producción que tenga un prefijo común con ambas δ que sea más corto ($\gamma = \delta\beta$ y $|\delta| < |\gamma|$). En este caso, primero es necesario eliminar el factor común δ y después habrá que eliminar el resto de γ , es decir, β .

Si existe dicho prefijo común $\alpha \neq \epsilon$, entonces hay que sustituir las producciones

$$A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma_i$$

donde γ_i representa a todas las alternativas que no empiezan por α , por:

$$\begin{aligned} A &\longrightarrow \alpha A' \mid \gamma_i \\ A' &\longrightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{aligned}$$

Este paso debe repetirse para todos los prefijos comunes por la izquierda (de un mismo no terminal, por supuesto) que queden en la gramática.

Ejemplo 4.13

$$\begin{aligned} Sent &\longrightarrow \text{if Expr then Sent else Sent endif} \\ Sent &\longrightarrow \text{if Expr then Sent endif} \\ Sent &\longrightarrow Otras \end{aligned}$$

Fijándonos en la regla para factorizar, podemos identificar:

$$\begin{aligned} \alpha &\equiv \text{if Expr then Sent} \\ \beta_1 &\equiv \text{else Sent endif} \\ \beta_2 &\equiv \text{endif} \\ \gamma_i &\equiv Otras \end{aligned}$$

Con lo que la gramática factorizada queda

$$\begin{aligned} Sent &\longrightarrow \text{if Expr then Sent Sent'} \\ Sent &\longrightarrow Otras \\ Sent' &\longrightarrow \text{else Sent endif} \\ Sent' &\longrightarrow \text{endif} \end{aligned}$$

Adicionalmente, supóngase que el no terminal *Otras* deriva en un terminal cualquiera “x”. Compruébese que esta gramática es ahora LL(1).

◁

4.4.3 Eliminación de la recursividad por la izquierda

Una gramática es recursiva por la izquierda si tiene alguna producción que sea recursiva por la izquierda (*recursividad directa por la izquierda*) o bien si a partir de una forma sentencial como $A\gamma$ se obtiene (después de dos o más derivaciones) una forma sentencial

$A\beta\gamma$ en la que el no terminal A vuelve a ser el primero por la izquierda (*recursividad indirecta por la izquierda*). En general, una gramática es recursiva por la izquierda si

$$\exists A \in N \ / \ A \xRightarrow{*} A\alpha \text{ para alguna cadena } \alpha$$

Los analizadores sintácticos descendentes no pueden manejar estas gramáticas pues entrarían en ciclos de recursividad infinita al ejecutarse. Cualquier gramática recursiva por la izquierda no cumple la condición LL(1).

Para gramáticas de la forma:

$$A \rightarrow A\alpha \mid \beta \longrightarrow \text{genera el lenguaje: } \beta, \beta\alpha, \beta\alpha\alpha, \dots$$

existe una regla para modificar este tipo de producciones para que dejen de ser recursivas por la izquierda:

$$\begin{aligned} A &\longrightarrow \beta A' \quad (\text{nuevo no terminal auxiliar}) \\ A' &\longrightarrow \alpha A' \quad (\text{recursiva por la derecha}) \\ A' &\longrightarrow \epsilon \end{aligned}$$

Esta gramática modificada ya no es recursiva por la izquierda, por lo que ya no presenta el problema que le impedía ser LL(1).

Ejemplo 4.14

Eliminar la recursividad izquierda de la gramática de las expresiones aritméticas.

$$\begin{aligned} E &\longrightarrow E + T \mid E - T \mid T \\ T &\longrightarrow T * F \mid T / F \mid F \\ F &\longrightarrow \text{num} \mid (E) \end{aligned}$$

En el caso del no terminal E ,

$$\begin{aligned} \alpha_1 &\equiv + T \\ \alpha_2 &\equiv - T \\ \beta &\equiv T \end{aligned} \left\| \Rightarrow \begin{aligned} E &\longrightarrow T E' \\ E' &\longrightarrow + T E' \mid - T E' \mid \epsilon \end{aligned}$$

Lo mismo sucede con el no terminal T . Por tanto nos queda la siguiente gramática:

$$\begin{aligned} E &\longrightarrow T E' \\ E' &\longrightarrow + T E' \mid - T E' \mid \epsilon \\ T &\longrightarrow F T' \\ T' &\longrightarrow * F T' \mid / F T' \mid \epsilon \\ F &\longrightarrow \text{num} \mid (E) \end{aligned}$$

◁

En el caso general, independientemente de cuántas producciones alternativas haya de A , se puede eliminar la recursividad por la izquierda utilizando las siguientes reglas:

$$A \longrightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

Se sustituye por:

$$\begin{aligned} A &\longrightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A' \\ A' &\longrightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

Estas reglas no sirven para eliminar la recursividad indirecta por la izquierda, como la que se presenta en el siguiente ejemplo.

Ejemplo 4.15

Sea la siguiente gramática:

$$\begin{aligned} S &\longrightarrow A \mathbf{a} \mid \mathbf{b} \\ A &\longrightarrow A \mathbf{c} \mid S \mathbf{d} \mid \epsilon \end{aligned}$$

$S \Rightarrow Aa \Rightarrow Sda$, luego $S \xRightarrow{2} Sda$ (recursividad indirecta).

◁

Para resolver este problema se proporciona el siguiente algoritmo capaz de eliminar la recursividad indirecta sistemáticamente. Funciona si la gramática no tiene ciclos ($A \xRightarrow{+} A$) o producciones vacías ($A \rightarrow \epsilon$).

Pasos:

1. Ordenar los no terminales según A_1, A_2, \dots, A_n
2. DESDE $i \leftarrow 1$ HASTA n HACER
 DESDE $j \leftarrow 1$ HASTA $i - 1$ HACER
 Sustituir cada $A_i \rightarrow A_j \gamma$ por $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$
 donde $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ son las producciones
 actuales de A_j
 Eliminar la recursividad izquierda directa de A_i
 FIN_DESDE
 FIN_DESDE

Ejemplo 4.16

Vamos a deshacer la recursividad por la izquierda que presentaba el ejemplo 4.15:

$$\begin{aligned} S &\longrightarrow A \mathbf{a} \mid \mathbf{b} \\ A &\longrightarrow A \mathbf{c} \mid S \mathbf{d} \mid \epsilon \end{aligned}$$

1. $A_1 = S, A_2 = A$
2. Para $i = 1$: desde $j \leftarrow 1$ hasta 0 hacer: *nada*
 Para $i = 2$: desde $j \leftarrow 1$ hasta 1 hacer:
 tomamos el segundo y lo sustituimos por el primero en todos los
 lugares donde éste aparezca a la derecha:

$$\begin{aligned} S &\longrightarrow A \mathbf{a} \mid \mathbf{b} \\ A &\longrightarrow A \mathbf{c} \mid S \mathbf{d} \mid \epsilon \end{aligned} \quad \parallel \quad \begin{aligned} S &\longrightarrow A \mathbf{a} \mid \mathbf{b} \\ A &\longrightarrow A \mathbf{c} \mid A \mathbf{a} \mathbf{d} \mid \mathbf{b} \mathbf{d} \mid \epsilon \end{aligned}$$

Si hubiera otra regla: $X \rightarrow Sa \mid Ab \mid \dots$ sustituir S y A por su definición en las producciones de X .

A esto le quitamos la recursividad directa. Sólo hay que modificar A y queda

$$\begin{aligned} S &\longrightarrow A \mathbf{a} \mid \mathbf{b} \\ A &\longrightarrow \mathbf{b} \mathbf{d} A' \mid A' \\ A' &\longrightarrow \mathbf{c} A' \mid \mathbf{a} \mathbf{d} A' \mid \epsilon \end{aligned}$$

◁

Ejemplo 4.17

Sea la siguiente gramática:

$$\begin{aligned} S &\longrightarrow S \mathbf{inst} \mid T R V \\ T &\longrightarrow \mathbf{tipo} \mid \epsilon \\ R &\longrightarrow \mathbf{blq} V \mathbf{fblq} \mid \epsilon \\ V &\longrightarrow \mathbf{id} S \mathbf{fin} \mid \mathbf{id} ; \mid \epsilon \end{aligned}$$

Es evidente que esta gramática no es LL(1) pues la primera producción es recursiva por la izquierda y las dos primeras de V tienen factores comunes por la izquierda. A continuación se presenta la gramática a la cual se han eliminado estos problemas mediante las técnicas descritas:

$$\begin{aligned} S &\longrightarrow T R V S' \\ S' &\longrightarrow \mathbf{inst} S' \mid \epsilon \\ T &\longrightarrow \mathbf{tipo} \mid \epsilon \\ R &\longrightarrow \mathbf{blq} V \mathbf{fblq} \mid \epsilon \\ V &\longrightarrow \mathbf{id} V' \mid \epsilon \\ V' &\longrightarrow S \mathbf{fin} \mid ; \end{aligned}$$

Ahora ya la gramática no presenta los problemas anteriores, pero no se puede asegurar que sea LL(1) mientras no se aplique la condición que verifica dicha propiedad. Nótese que todos los no terminales excepto el símbolo inicial, S , tienen más de una alternativa.

$$\begin{aligned} \text{PRED}(S' \rightarrow \mathbf{inst} S') &= \{ \mathbf{inst} \} \\ \text{PRED}(S' \rightarrow \epsilon) &= \{ \$, \mathbf{fin} \} \end{aligned}$$

$$\begin{aligned} \text{PRED}(T \rightarrow \mathbf{tipo}) &= \{ \mathbf{tipo} \} \\ \text{PRED}(T \rightarrow \epsilon) &= \{ \mathbf{blq}, \mathbf{id}, \mathbf{inst}, \$, \mathbf{fin} \} \end{aligned}$$

$$\begin{aligned} \text{PRED}(R \rightarrow \mathbf{blq} V \mathbf{fblq}) &= \{ \mathbf{blq} \} \\ \text{PRED}(R \rightarrow \epsilon) &= \{ \mathbf{id}, \mathbf{inst}, \$, \mathbf{fin} \} \end{aligned}$$

$$\begin{aligned} \text{PRED}(V \rightarrow \mathbf{id} V') &= \{ \mathbf{id} \} \\ \text{PRED}(V \rightarrow \epsilon) &= \{ \mathbf{inst}, \$, \mathbf{fin} \} \end{aligned}$$

$$\begin{aligned} \text{PRED}(V' \rightarrow S \mathbf{fin}) &= \{ \mathbf{tipo}, \mathbf{blq}, \mathbf{id}, \mathbf{inst}, \$, \mathbf{fin} \} \\ \text{PRED}(V' \rightarrow ;) &= \{ ; \} \end{aligned}$$

Como los conjuntos de cada no terminal son disjuntos, la gramática es LL(1).

◁

4.5 Analizador descendente recursivo

Es un tipo de analizador sintáctico descendente que sólo se puede utilizar con gramáticas LL(1) y por tanto es un analizador sintáctico descendente predictivo (ASDP). Consiste en un conjunto de funciones recursivas (una por cada no terminal de la gramática) que son diseñadas a partir de los elementos que definen cada una de las producciones de la gramática. La secuencia de llamadas al procesar la cadena de entrada define implícitamente un recorrido del árbol de análisis sintáctico de la cadena de entrada.

Veremos a continuación cómo implementar un analizador sintáctico descendente recursivo (ASDR) a partir de las producciones de la gramática que genera las cadenas del lenguaje que se pretende analizar. Vamos a describir la implementación de un ASDR en un lenguaje de alto nivel (en C) y vamos a definir los elementos necesarios para ello.

Símbolo de preanálisis

Si esta metodología es aplicable a las gramáticas LL(1) es porque es suficiente con ver un único *token* para saber qué hacer. Ese *token* se llama símbolo de *preanálisis* (*lookahead* en inglés) y se pedirá al analizador léxico cada vez que se necesite. Será este *token* el que se busque en los conjuntos de predicción de las diferentes reglas para escoger aquella en la que aparezca.

Función de emparejamiento

La función de emparejamiento (*match* en inglés) es la encargada de comprobar si el símbolo de preanálisis coincide con el terminal de la gramática que, de acuerdo con los elementos de la producción escogida, debería aparecer en esa posición. Esta función también se encarga de otra misión fundamental como es la petición del siguiente *token* al analizador léxico si se ha producido la coincidencia o invocar la función de error en caso contrario.

Su estructura en lenguaje algorítmico sería la siguiente:

```

FUNCIÓN Emparejar ( Parámetro: token; Usa variable global preanálisis )
  SI preanálisis coincide con token ENTONCES
    Pedir el siguiente preanálisis al analizador léxico
  SI NO
    Error Sintáctico(Encontrado preanálisis, esperaba token)

```

Su implementación en C, si “t” es el *token* que el ASDR espera encontrar en la entrada y, por tanto, el que la función *Emparejar* recibe como parámetro, sería:

```

void Emparejar (int t )
{
  if ( t == preanalisis )
    preanalisis = analex(); /* 'analex' es el nombre que hemos
                             dado al analizador léxico */
  else
    ErrorSintactico(preanalisis,t);
}

```

Para construir un ASDR, además de utilizar estos elementos auxiliares, hay que hacer lo siguiente:

1. Escribir una función por cada símbolo no terminal de la gramática. Cada una de estas funciones llevará a cabo el análisis de las producciones de dicho no terminal, como se indicará más adelante.
2. Cuando este no terminal tenga distintas alternativas en la gramática, para decidir durante su ejecución cuál de las producciones utilizar, se optará por aquella alternativa a cuyo conjunto de predicción pertenezca el *token de preanálisis*.

EN CASO DE QUE *preanálisis* PERTENEZCA A

PRED(α_1): ... proceder según alternativa α_1

PRED(α_2): ... proceder según alternativa α_2

...

FIN

Si el token de preanálisis no pertenece a ninguno de los conjuntos de predicción, entonces se producirá un error sintáctico.

3. Si una de las alternativas para el no terminal que se está analizando es la cadena vacía ($A \rightarrow \epsilon$), en ese caso no se hará nada.
4. Para analizar cada alternativa, se aplican distintas metodologías a los símbolos de la parte derecha de la producción, en el mismo orden en el que aparecen, según si son terminales o no:
 - Para cada $A \in N$ hacemos una llamada a su función correspondiente.
 - Para cada $a \in T$ hacemos una llamada a la función *Emparejar* con a como parámetro.
5. La invocación o puesta en marcha del ASDR se realiza mediante una llamada a la función del símbolo inicial de la gramática. Para hacer esa llamada se supone que el *token* de preanálisis habrá sido inicializado por una llamada previa al analizador léxico.

Ejemplo 4.18

Sea la siguiente gramática LL(1) con sus conjuntos de predicción:

$$\begin{array}{lll}
 S & \longrightarrow & A \quad \{\mathbf{a}, \$, \mathbf{c}\} \\
 S & \longrightarrow & \mathbf{s} \quad \{\mathbf{s}\} \\
 A & \longrightarrow & \mathbf{a} S \mathbf{c} \quad \{\mathbf{a}\} \\
 A & \longrightarrow & \epsilon \quad \{\mathbf{c}, \$\}
 \end{array}$$

Vamos a ver la implementación de un ASDR para ella. Supondremos siempre en este tipo de problemas que tenemos definidos los *tokens* (\mathbf{a} , \mathbf{c} y \mathbf{s}) y además una variable *lexema* como un *array* de caracteres. Supondremos también en estos ejemplos que ya tenemos implementada previamente la función *Emparejar* que acabamos de definir.


```

void S(void)
{
    if ( preanalisis == a || preanalisis == FINDEFICHERO ||
        preanalisis == c )    A();
    else if ( preanalisis == s )    emparejar(s);
    else ErrorSintactico(lexema,a,s,FINDEFICHERO);
    /* encontrado 'lexema', esperaba 'a', 's' o fin de fichero */
}

void A(void)
{
    if ( preanalisis == a ) { emparejar(a); S(); emparejar(c); }
    else if ( preanalisis == c || preanalisis == FINDEFICHERO )
        ; /* producción vacía */
    else ErrorSintactico(lexema,a,c,FINDEFICHERO);
}

```

A la función de error sintáctico se le suele pasar como argumentos el lexema que ha producido el error (el del *token* de preanálisis) y los *tokens* que esperaba en su lugar (los terminales que aparezcan en la unión de todos los conjuntos de predicción de las reglas del no terminal al que corresponde la función).

<

Ejemplo 4.19

Implementación de un ASDR para la gramática LL(1) del ejemplo 4.14, usando la siguiente definición de *tokens*: MAS, MENOS, POR, DIV, LPAR, RPAR, NUM, FINDEFICHERO. En los no terminales con *prima* sustituiremos ésta por un 2.

```

void E()
{
    T(); E2();    /* sólo hay una producción, no es necesario
                   consultar el conjunto de preanálisis */
}

void E2()
{
    switch ( preanalisis ) {
        case MAS      : Emparejar(MAS); T(); E2(); break;
        case MENOS    : Emparejar(MENOS); T(); E2(); break;
        case RPAR      : case FINDEFICHERO : /* E' deriva epsilon */ break;
        default        : ErrorSintactico(lexema,MAS,MENOS,RPAR,FINDEFICHERO);
    }
}

void T()
{
    F(); T2();
}

```

```

void T2()
{
    switch ( preanalisis ) {
        case POR : Emparejar(POR); F(); T2(); break;
        case DIV : Emparejar(DIV); F(); T2(); break;
        case MAS : case MENOS :
        case RPAR : case FINDEFICHERO : /* T' deriva epsilon */ break;
        default: ErrorSintactico(lexema,POR,DIV,MAS,MENOS,RPAR,FINDEFICHERO);
    }
}

void F()
{
    switch ( preanalisis ) {
        case NUM : Emparejar(NUM); break;
        case LPAR : Emparejar(LPAR); E(); Emparejar(RPAR); break;
        default : ErrorSintactico(lexema,NUM,LPAR);
    }
}

```

Nótese que se han permitido algunas licencias en el lenguaje C utilizado, que en una implementación real habría que realizar con más cuidado.

<

Comentarios adicionales para la implementación

Tras la devolución del control a la función main que inició el análisis con una llamada al símbolo inicial de la gramática, se debe comprobar que la cadena de entrada se ha analizado en su totalidad para dar como correcto el análisis:

```

token = analex();
S();
if (token != FINDEFICHERO) error(token,lexema,{FINDEFICHERO});
/* encontrado 'lexema', esperaba fin de fichero */

```

4.6 ASDP dirigidos por tabla

Se puede construir un analizador sintáctico descendente predictivo mediante una técnica completamente distinta a la descrita en el apartado anterior, utilizando una pila de símbolos (terminales y no terminales) en vez de haciendo llamadas recursivas. Para determinar qué producción debe aplicarse a la vista de un terminal (*token* de preanálisis), se buscará en una tabla llamada *tabla de análisis* (véase la figura 4.1).

El proceso de análisis consiste en aplicar el algoritmo que describiremos más adelante, pero previamente se tendrá que construir la tabla, que es lo que diferencia un analizador de otro y que depende de la gramática a ser analizada. El procedimiento para construir tablas de análisis LL(1) es el siguiente:

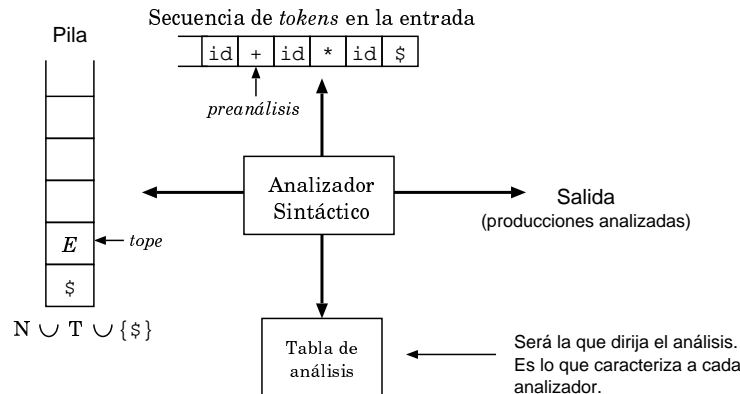


Figura 4.1: Analizador descendente predictivo dirigido por tabla

1. Las filas de la tabla, que llamaremos *Tabla*, se etiquetan con los no terminales de la gramática, y las columnas se etiquetan con los terminales y el símbolo de fin de fichero, $\$$. En cada celda de la tabla se va a colocar la regla que se debe aplicar para analizar la variable de esa fila cuando el terminal de esa columna (o $\$$) aparezca en la entrada; ese terminal o $\$$ debe pertenecer al conjunto de predicción de dicha regla.
2. Se calculan los conjuntos de predicción para cada regla.
3. Para cada producción $A \rightarrow \alpha$ de la gramática, hacer:
Para cada terminal $a \in \text{PRED}(A \rightarrow \alpha)$, añádase $A \rightarrow \alpha$ a $\text{Tabla}[A, a]$.
4. Cada entrada no definida de *Tabla* será *error sintáctico*.

Aunque hemos dicho que la tabla de análisis contiene las reglas, al implementar el algoritmo no es necesario almacenar la regla completa en la tabla sino sólo los símbolos de su parte derecha (que es conveniente guardar al revés por motivos que se explicarán más adelante) o incluso es más eficiente almacenar un número de regla que haga referencia a otra tabla con los símbolos de las partes derechas de las reglas.

Mensajes de error

En este tipo de analizador existen dos tipos de errores sintácticos: los provocados al intentar emparejar dos terminales (uno en la entrada y otro en el tope de la pila), y los producidos al acceder a una casilla marcada como *error sintáctico* en la tabla de análisis.

Existen varias estrategias para proporcionar mensajes de error con este tipo de analizador:

1. Cuando en el tope de la pila hay un terminal, el mensaje de error debe decir que se esperaba ese terminal. En cambio, si hay un no terminal, el error se habrá producido al consultar la tabla; en este caso, es posible proporcionar un mensaje

de error específico para cada casilla de la tabla. Si la tabla es muy grande, es poco práctico producir un mensaje de error específico para cada caso.

2. Producir un error genérico, basándose únicamente en el tope de la pila y en la tabla de análisis. Como en la alternativa anterior, pueden darse dos situaciones:
 - En el tope de la pila hay una variable; en este caso, el conjunto de símbolos que se esperaban en lugar del que se ha leído se calcula recorriendo la fila correspondiente a la variable en la tabla de análisis, anotando todos los símbolos para los que la entrada correspondiente en la tabla contiene un número de regla. Se puede comprobar fácilmente que ese conjunto es la unión de los conjuntos de predicción de las reglas de esa variable.
 - En el tope de la pila hay un terminal; en este caso, el mensaje de error debe decir que se esperaba ese terminal en lugar del lexema leído.
3. La alternativa anterior (mensajes de error genéricos) puede producir mensajes imprecisos en algunos casos, diciendo que se esperaba algún símbolo que realmente no se puede esperar. Sin embargo, si se toma la pila como una cadena de símbolos α en la que el símbolo en el tope de la pila es el primer símbolo de la cadena y el \$ que hay en el fondo es el último, los símbolos que realmente se esperan cuando se produce un error son exactamente $\text{PRIM}(\alpha)$. Aunque esta alternativa produce mensajes de error exactos, exige calcular los primeros de una cadena en tiempo de compilación (que no es muy complicado si se almacenan los primeros de los no terminales).

Ejemplo 4.20

Sea de nuevo la gramática de las expresiones aritméticas (ya modificada para que sea LL(1)):

$$\begin{aligned}
 E &\longrightarrow TE' \\
 E' &\longrightarrow +TE' \mid -TE' \mid \epsilon \\
 T &\longrightarrow FT' \\
 T' &\longrightarrow *FT' \mid /FT' \mid \epsilon \\
 F &\longrightarrow \text{num} \mid (E)
 \end{aligned}$$

Veamos cada una de las producciones (recuérdese que las producciones con varias alternativas son realmente una producción de la variable por cada una de las alternativas):

1. $\text{PRED}(E \rightarrow TE') = \{ (, \text{num} \} \implies$
en las celdas $[E, (]$ y $[E, \text{num}]$ añadir $E \rightarrow TE'$
2. $\text{PRED}(E' \rightarrow +TE') = \{ + \} \implies$
en la celda $[E', +]$ añadir $E' \rightarrow +TE'$
3. $\text{PRED}(E' \rightarrow -TE') = \{ - \} \implies$
en la celda $[E', -]$ añadir $E' \rightarrow -TE'$
4. $\text{PRED}(E' \rightarrow \epsilon) = \{), \$ \} \implies$
en las celdas $[E',)]$ y $[E', \$]$ añadir $E' \rightarrow \epsilon$

5. $\text{PRED}(T \rightarrow FT') = \{ (, \text{num} \} \implies$
en las celdas $[T, (]$ y $[T, \text{num}]$ añadir $T \rightarrow FT'$
6. $\text{PRED}(T' \rightarrow * FT') = \{ * \} \implies$
en la celda $[T', *]$ añadir $T' \rightarrow * FT'$
7. $\text{PRED}(T' \rightarrow / FT') = \{ / \} \implies$
en la celda $[T', /]$ añadir $T' \rightarrow / FT'$
8. $\text{PRED}(T' \rightarrow \epsilon) = \{ +, -,), \$ \} \implies$
en las celdas $[T', +]$, $[T', -]$, $[T',)]$ y $[T', \$]$ añadir $T' \rightarrow \epsilon$
9. $\text{PRED}(F \rightarrow (E)) = \{ (\} \implies$
en la celda $[F, (]$ añadir $F \rightarrow (E)$
10. $\text{PRED}(F \rightarrow \text{num}) = \{ \text{num} \} \implies$
en la celda $[F, \text{num}]$ añadir $F \rightarrow \text{num}$

- Todas las celdas vacías se marcan como *error sintáctico*.

Con estos cálculos, la tabla de análisis para esta gramática resulta así:

	num	+	-	*	/	()	\$
E	$E \rightarrow TE'$	e	e	e	e	$E \rightarrow TE'$	e	e
E'	e	$E' \rightarrow + TE'$	$E' \rightarrow - TE'$	e	e	e	$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	e	e	e	e	$T \rightarrow FT'$	e	e
T'	e	$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$	$T' \rightarrow * FT'$	$T' \rightarrow / FT'$	e	$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{num}$	e	e	e	e	$F \rightarrow (E)$	e	e

◁

Una importante observación a tener en cuenta es que este tipo de analizador, al ser predictivo, sólo podrá construirse si la gramática a analizar es LL(1). Esto se refleja en la tabla en el hecho de que no aparezcan dos (o más) producciones en la misma casilla. Si en alguna de ellas aparecieran dos producciones significaría que, llegando a esa situación, el analizador no sabría qué decisión tomar, lo cual está en contra del concepto de gramática LL(1).

Así pues, la no aparición de casillas con dos o más producciones puede considerarse como una demostración de que la gramática analizada es LL(1).

Veámoslo con un ejemplo.

Ejemplo 4.21

Sea una gramática para las sentencias IF en Pascal:

$$\begin{aligned}
 S &\longrightarrow \text{if } E \text{ then } S S' \mid \text{otras} \\
 S' &\longrightarrow \text{else } S \mid \epsilon \\
 E &\longrightarrow \text{lógico}
 \end{aligned}$$

Se puede comprobar que la tabla resultante es la siguiente:

	otras	lógico	else	if	then	\$
S	$S \rightarrow \text{otras}$	e	e	$S \rightarrow \text{if } \dots$	e	e
S'	e	e	$S' \rightarrow \text{else } S'$ $S' \rightarrow \epsilon$	e	e	$S' \rightarrow \epsilon$
E	e	$E \rightarrow \text{lógico}$	e	e	e	e

Luego la gramática no es LL(1) debido a la existencia de dos producciones en la casilla señalada. Este problema (debido a la ambigüedad de esta gramática para las sentencias IF en Pascal y C) se podría resolver durante el análisis aplicando sistemáticamente una de las dos. La producción $S' \rightarrow \text{else } S$ equivale a asociar el ELSE al IF más cercano.

◁

Una vez que se tiene la tabla de análisis de una gramática ya se puede construir un analizador sintáctico descendente dirigido por esa tabla. En la figura 4.2 se presenta el algoritmo de análisis que utiliza este tipo de tablas.

```

push(Pila,$)
push(Pila,S) /* S es el símbolo inicial */
REPETIR
    A :=Tope(Pila) /* el símbolo en el tope de la pila */
    a :=analex() /* a es el siguiente símbolo de preanálisis */
    SI A es terminal o $ ENTONCES
        SI A = a ENTONCES
            pop(Pila,A)
            a :=analex()
        SI NO
            ErrorSintáctico(encontrado [lexema de a], esperaba A)
        FINSI
    SI NO /* A es no terminal */
        SI Tabla[A, a] = A → c1c2...ck ENTONCES
            pop(Pila,A)
            para i := k hasta 1 hacer push(Pila,ci) fpara
        SI NO
            ErrorSintáctico(encontrado [lexema de a],
                           esperaba  $\bigcup_{i=1}^n \text{PRED}(A \rightarrow \alpha_i)$ )
        FINSI
    FINSI
HASTA A = $

```

Figura 4.2: Algoritmo de análisis descendente dirigido por tabla.

Ejemplo 4.22

Para hacer una traza de este analizador sintáctico sobre la gramática LL(1) de las expresiones aritméticas tomamos la tabla de análisis del ejemplo 4.20 y aplicaremos el algoritmo de análisis a la siguiente entrada: “num + num * num \$”.

PILA	ENTRADA	SALIDA
\$ E	num + num * num \$	$E \rightarrow T E'$
\$ $E' T$	num + num * num \$	$T \rightarrow F T'$
\$ $E' T' F$	num + num * num \$	$F \rightarrow \mathbf{num}$
\$ $E' T' \mathbf{num}$	num + num * num \$	emparejar num
\$ $E' T'$	+ num * num \$	$T' \rightarrow \epsilon$
\$ E'	+ num * num \$	$E' \rightarrow + T E'$
\$ $E' T +$	+ num * num \$	emparejar +
\$ $E' T$	num * num \$	$T \rightarrow F T'$
\$ $E' T' F$	num * num \$	$F \rightarrow \mathbf{num}$
\$ $E' T' \mathbf{num}$	num * num \$	emparejar num
\$ $E' T'$	* num \$	$T' \rightarrow * F T'$
\$ $E' T' F *$	* num \$	emparejar *
\$ $E' T' F$	num \$	$F \rightarrow \mathbf{num}$
\$ $E' T' \mathbf{num}$	num \$	emparejar num
\$ $E' T'$	\$	$T' \rightarrow \epsilon$
\$ E'	\$	$E' \rightarrow \epsilon$
\$	\$	aceptar

La columna de la izquierda muestra el contenido de la pila en cada momento, quedando el fondo a su izquierda y el tope a su derecha, donde se van colocando los símbolos de las partes derechas de las producciones en orden inverso a como aparecen en dichas producciones.

La columna central representa lo que en cada paso queda por analizar de la cadena de entrada. El primer símbolo de la izquierda representa el símbolo de preanálisis. Cuando este símbolo coincide con el terminal de la pila se eliminan ambos (el tope y el preanálisis) y avanza el análisis al siguiente símbolo de la entrada.

La columna de salida muestra los emparejamientos de *tokens* que se realizan y las producciones que se van aplicando. Al aplicar una producción se desapila su parte izquierda y se apilan los símbolos de la parte derecha en orden inverso.

◁

Ejemplo 4.23

Sea la gramática siguiente con sus conjuntos de predicción ya calculados:

$$\begin{aligned}
 S &\rightarrow C A B & \{c\} \\
 S &\rightarrow \mathbf{a} C \mathbf{b} & \{a\} \\
 A &\rightarrow \mathbf{a} S \mathbf{d} & \{a\} \\
 A &\rightarrow \epsilon & \{b, d, \$\} \\
 B &\rightarrow \mathbf{b} & \{b\} \\
 B &\rightarrow \epsilon & \{d, \$\} \\
 C &\rightarrow \mathbf{c} & \{c\}
 \end{aligned}$$

La tabla de análisis resulta

	a	b	c	d	\$
S	$S \rightarrow \mathbf{a} C \mathbf{b}$	e	$S \rightarrow C A B$	e	e
A	$A \rightarrow \mathbf{a} S \mathbf{d}$	$A \rightarrow \epsilon$	e	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$
B	e	$B \rightarrow \mathbf{b}$	e	$B \rightarrow \epsilon$	$B \rightarrow \epsilon$
C	e	e	$C \rightarrow \mathbf{c}$	e	e

Y la traza del análisis para la cadena “cacdb” sería:

PILA	ENTRADA	SALIDA
\$ S	c a c d b \$	$S \rightarrow C A B$
\$ B A C	c a c d b \$	$C \rightarrow c$
\$ B A c	c a c d b \$	emparejar c
\$ B A	a c d b \$	$A \rightarrow a S d$
\$ B d S a	a c d b \$	emparejar a
\$ B d S	c d b \$	$S \rightarrow C A B$
\$ B d B A C	c d b \$	$C \rightarrow c$
\$ B d B A c	c d b \$	emparejar c
\$ B d B A	d b \$	$A \rightarrow \epsilon$
\$ B d B	d b \$	$B \rightarrow \epsilon$
\$ B d	d b \$	emparejar d
\$ B	b \$	$B \rightarrow b$
\$ b	b \$	emparejar b
\$	\$	aceptar

◁

4.7 Manipulación de errores

Durante la ejecución de un ASDR, se detecta un error cuando se llega a una llamada al procedimiento de *Error* y en el analizador dirigido por tabla, cuando el terminal del tope de la pila no coincide con el símbolo de preanálisis o si se accede a una celda vacía (por tanto, etiquetada como *error*).

Ya se ha indicado que los mensajes de error constan típicamente del último lexema que entregó el analizador léxico (el del *token* de preanálisis), como causante del error, y una indicación de los *tokens* que esperaba en su lugar (los terminales que aparezcan en la unión de todos los conjuntos de predicción de las reglas del no terminal al que pertenece la función). Por supuesto, al imprimir el mensaje de error se deben describir los *tokens* en un lenguaje comprensible por el usuario, es decir, si por ejemplo se esperaban los *tokens* “**num**” e “**id**”, se debe decir que se espera “un número o un identificador”, no que se espera “num o id”.

La posibilidad más sencilla en el diseño de todo compilador es detener el proceso de compilación al detectar el primer error, pero también existen muchas técnicas para recuperarse de un error y seguir analizando, lo cual permite detectar la mayor parte de los errores que hay en el programa fuente en una única compilación. A veces, sin embargo, un error provoca muchos otros errores (lo que se conoce como *errores en cascada*), por lo que la estrategia de recuperación de errores de un compilador debe estudiarse con cuidado para detectar el mayor número de errores posible, pero no producir muchos errores en cascada.

Conjuntos de sincronización

Una posible estrategia para recuperarse de un error es seguir avanzando por la entrada hasta encontrar algún símbolo que pertenezca a un determinado conjunto de símbolos de sincronización. La construcción de este conjunto se basa en diversas técnicas empíricas:

- Para el no terminal A que se define en la producción en la que se ha producido el error, poner en el conjunto todos los símbolos de $\text{SIG}(A)$;
- Poner también los de $\text{PRIM}(A)$;
- Si se puede generar la cadena vacía, tomarla por omisión; etc.

En general, los sistemas de recuperación de errores están llenos de reglas empíricas y soluciones *ad hoc*, por lo que hay mucho escrito sobre estas técnicas, pero nada que sea definitivo en cuanto a robustez y fiabilidad.

4.8 Referencias bibliográficas

REFERENCIAS	EPÍGRAFES
[Louden, 1997]	4.1, 4.2, 4.3 y 4.5
[Aho, Sethi y Ullman, 1990]	4.4
[Bennett, 1990]	6.2.1 y 6.2.3
[Fischer y LeBlanc, 1991]	4.5, 5.1, 5.2, 5.3, 5.4, 5.6 y 5.9

4.9 Ejercicios

Ejercicio 4.1

Compruébese que la siguiente gramática es LL(1) sin modificarla (en esta y en el resto de gramáticas de estos ejercicios se supondrá que el símbolo inicial es el primero).

$$\begin{aligned}
 A &\longrightarrow B C D \\
 B &\longrightarrow \mathbf{a} C \mathbf{b} \\
 B &\longrightarrow \epsilon \\
 C &\longrightarrow \mathbf{c} A \mathbf{d} \\
 C &\longrightarrow \mathbf{e} B \mathbf{f} \\
 C &\longrightarrow \mathbf{g} D \mathbf{h} \\
 C &\longrightarrow \epsilon \\
 D &\longrightarrow \mathbf{i}
 \end{aligned}$$

Ejercicio 4.2

¿Es LL(1) la siguiente gramática?

$$\begin{aligned}
 A &\longrightarrow B C D \\
 B &\longrightarrow \mathbf{b} \mid \epsilon \\
 C &\longrightarrow \mathbf{c} \mid \epsilon \\
 D &\longrightarrow \mathbf{d} \mid \epsilon
 \end{aligned}$$

Ejercicio 4.3

De un simple vistazo se puede comprobar que la siguiente gramática no es LL(1):

$$\begin{aligned}
 S &\longrightarrow S \mathbf{inst} \\
 S &\longrightarrow S \mathbf{var} D \\
 S &\longrightarrow \epsilon \\
 D &\longrightarrow D \mathbf{ident} E \\
 D &\longrightarrow D \mathbf{ident} \mathbf{sep} \\
 D &\longrightarrow \mathbf{int} \\
 D &\longrightarrow \mathbf{float} \\
 E &\longrightarrow S \mathbf{fproc}
 \end{aligned}$$

Elimínese la recursividad por la izquierda y los factores comunes por la izquierda y compruébese si la gramática equivalente resultante cumple la condición LL(1).

Ejercicio 4.4

Dada la siguiente gramática:

$$\begin{aligned}
 S &\longrightarrow A B \\
 A &\longrightarrow \text{begin } S \text{ end } B \text{ theend} \\
 A &\longrightarrow \epsilon \\
 B &\longrightarrow \text{var } L : \text{tipo} \\
 B &\longrightarrow B \text{ fvar} \\
 B &\longrightarrow \epsilon \\
 L &\longrightarrow L , id \\
 L &\longrightarrow \text{id}
 \end{aligned}$$

1. Háganse las transformaciones necesarias para eliminar la recursividad por la izquierda.
2. Calcúlense los conjuntos de primeros y siguientes de cada no terminal.
3. Compruébese que la gramática modificada cumple la condición LL(1).
4. Constrúyase la tabla de análisis sintáctico LL(1) para esa nueva gramática.
5. Finalmente hágase la traza del análisis de la cadena:

```

begin
  var a,b:tipo
  var c:tipo
  fvar
end
  var d:tipo
theend

```

comprobando que las derivaciones son correctas mediante la construcción del árbol de análisis sintáctico.

Ejercicio 4.5

Dada la siguiente gramática:

$$\begin{aligned}
 S &\longrightarrow S \textbf{ inst} \\
 S &\longrightarrow T R V \\
 T &\longrightarrow \textbf{ tipo} \\
 T &\longrightarrow \epsilon \\
 R &\longrightarrow \textbf{ blq } V \textbf{ fblq} \\
 R &\longrightarrow \epsilon \\
 V &\longrightarrow \textbf{ id } S \textbf{ fin} \\
 V &\longrightarrow \textbf{ id } ; \\
 V &\longrightarrow \epsilon
 \end{aligned}$$

Constrúyase un analizador sintáctico descendente recursivo (ASDR) para la gramática LL(1) equivalente a esta gramática. Supóngase que ya existen las funciones **analex** (analizador léxico), **emparejar** (función de emparejamiento) y **error** (emisión de error sintáctico). La función **error** se debe llamar con una cadena de caracteres que indique exactamente qué lexema ha provocado el error y qué símbolos se esperaban en lugar del encontrado en la entrada.

Ejercicio 4.6

Háganse las mismas operaciones que en el ejercicio anterior para la gramática:

$$\begin{aligned}
 E &\longrightarrow [L \\
 E &\longrightarrow \textbf{ a} \\
 L &\longrightarrow E Q \\
 Q &\longrightarrow , L \\
 Q &\longrightarrow]
 \end{aligned}$$

Con los mismos supuestos y condiciones que en aquel caso. Escribese la secuencia de llamadas recursivas a las funciones que haría el ASDR durante su ejecución, incluidas las llamadas a la función de emparejamiento, para la cadena de entrada “[a,a]”.

Ejercicio 4.7

Dada la siguiente gramática:

$$\begin{aligned}
 P &\longrightarrow D S \\
 D &\longrightarrow D V \\
 D &\longrightarrow \epsilon \\
 S &\longrightarrow S I \\
 S &\longrightarrow \epsilon \\
 V &\longrightarrow \text{decl id ;} \\
 V &\longrightarrow \text{decl id (} P \text{) ;} \\
 V &\longrightarrow \text{decl [} D \text{] id ;} \\
 I &\longrightarrow \text{id ;} \\
 I &\longrightarrow \text{begin } P \text{ end}
 \end{aligned}$$

1. Háganse las transformaciones necesarias para que cumpla la condición LL(1).
2. Constrúyase la tabla de análisis sintáctico LL(1) para esa nueva gramática.
3. Finalmente hágase la traza del análisis de las cadenas:

`decl id (begin id ;)`

`decl id (decl [decl id ;] id ;) ; id ;`

Si durante el análisis se produjera algún error sintáctico indíquese qué símbolos podrían esperarse en lugar del *token* de preanálisis que entró.

Capítulo 5

Análisis sintáctico ascendente

5.1 Introducción

Recordemos ahora del capítulo de análisis sintáctico que el análisis ascendente, parte de la cadena de entrada y construye la inversa de la derivación por la derecha para ella, que representa un recorrido del árbol de análisis empezando por las hojas (donde están los *tokens*), visitando nodos intermedios hasta llegar a la raíz (el símbolo inicial), y recorriendo así el árbol de abajo a arriba.

El problema clave del análisis ascendente consiste en decidir cuándo lo que parece ser la parte derecha de una producción puede ser reemplazada por su parte izquierda. Esto no es trivial pues puede haber partes derechas comunes a varias producciones o producciones que derivan a ϵ .

La ventaja que presenta el análisis ascendente es que es aplicable a un mayor número de gramáticas que el descendente, pues para aplicar el análisis ascendente lineal la gramática debe ser LR(1) y este conjunto es mucho más amplio que el de las gramáticas LL(1). Además, las gramáticas LR(1) son más adecuadas para la traducción de los operadores binarios (asociativos por la izquierda o por la derecha) que las gramáticas LL(1), que en muchos casos necesitan procesos de traducción bastante más complejos que aquéllas, como veremos en los siguientes capítulos.

Existen varios algoritmos de análisis ascendente lineal. El más comúnmente utilizado es el algoritmo de análisis por desplazamiento-reducción. Este algoritmo está basado en una pila y una tabla de análisis, como un analizador descendente dirigido por tabla, pero tiene un funcionamiento completamente diferente, como veremos más adelante. Existen distintos métodos para construir tablas para el algoritmo de análisis por desplazamiento-reducción: SLR, LALR(1), LR(1), etc. El más sencillo de todos es el método SLR, que es el que vamos a estudiar en este capítulo. Las tablas construidas según el método SLR (*Simple LR*) se caracterizan por ser mucho compactas que las LR(1), pero el conjunto de gramáticas SLR (analizables con tablas SLR) es más reducido que el conjunto de gramáticas LR(1). Existe un conjunto intermedio de gramáticas, LALR(1), que es un subconjunto muy amplio de las LR(1) para el cual se pueden construir tablas casi tan compactas como las SLR.

Así como existe una condición LL(1) que distingue a las gramáticas susceptibles de ser analizadas mediante un ASDP, la condición equivalente para las gramáticas LR(1)

se basa simplemente en la posibilidad de construir sin problemas las tablas de análisis sintáctico para analizadores ascendentes.

5.2 Análisis sintáctico por desplazamiento y reducción

De forma similar a como sucedía en los analizadores descendentes lineales dirigidos por tabla, veremos que el algoritmo de análisis sintáctico ascendente por desplazamiento y reducción es siempre el mismo con independencia de la gramática utilizada y lo que cambia de un analizador a otro es la tabla de análisis. En este algoritmo (que veremos con más detalle más adelante) existen dos acciones básicas:

Desplazar: consiste en llevar de alguna manera el símbolo de la entrada a la pila y pedir el siguiente al analizador léxico (equivalente a lo que hacía la función de emparejamiento en los ASDP con tabla).

Reducir: consiste en sustituir en la pila los símbolos de la parte derecha de una producción por su parte izquierda (como si fuera la “inversa” de derivar por esa regla).

En este capítulo se describe este algoritmo con alguna simplificación para facilitar su implementación. El algoritmo utiliza una pila de *estados* y una *tabla de análisis* para decidir qué acción tomar según el estado del tope de la pila y el símbolo de la entrada; la tabla tiene dos partes, aunque también se suele hablar de dos tablas: *Acción* e *Ir-a*. Cada estado representa que se ha reconocido un prefijo de la parte derecha de alguna regla de la gramática. A veces el estado también representa el *contexto* en que se ha reconocido ese prefijo. El algoritmo va leyendo terminales de la entrada y deshaciendo reglas hasta llegar a obtener la inversa de una derivación por la derecha de la cadena de entrada.

Las acciones que pueden aparecer en la tabla *Acción* para un estado s y un símbolo de la entrada a son:

- dj** desplazar el puntero de la entrada (leer el siguiente *token* de la entrada) y apilar el estado j .
- rk** reducir por la regla k (deshacer la regla k). Para ello hay que desapilar tantos estados como símbolos tenga la parte derecha de la regla (si es una producción vacía no hay que desapilar ningún estado) y, siendo p el estado que queda en el tope de la pila y A la parte izquierda de la regla, apilar el estado indicado por $Ir-a[p, A]$.
- aceptar** terminar el análisis con éxito.
- error** producir un error. El conjunto de símbolos que podrían aparecer en lugar del *token* de la entrada son todos aquellos para los que exista una acción de desplazar o reducir en el estado s .

La figura 5.1 muestra este algoritmo en notación pseudoalgorítmica.

```

push(0) /* estado inicial */
a := analex() /* a siempre será el siguiente token */
REPETIR
    sea s el estado en el tope de la pila
    SI Accion[s, a] = dj ENTONCES
        push(j)
        a := analex()
    SI NO SI Accion[s, a] = rk ENTONCES
        PARA i := 1 HASTA Longitud.Parte_Derecha(k) HACER pop()
        sea p el estado en el tope de la pila
        sea A la parte izquierda de la regla k
        push(Ir_a[p, A])
    SI NO SI Accion[s, a] = aceptar ENTONCES
        fin del analisis
    SI NO
        error()
    FIN_SI
HASTA fin del analisis

```

Figura 5.1: Algoritmo de análisis por desplazamiento y reducción.

Ejemplo 5.1

Dada la tabla de análisis de la figura 5.2, que está construida con el método SLR, vamos a hacer la traza del análisis de la entrada “**id tipo begin codigo end**”.

De la misma forma que para el análisis descendente, compondremos una tabla en cuya primera columna situaremos el estado de la pila en cada momento, con la base en la izquierda y el tope en la derecha. En la columna central situaremos la entrada que se leerá de izquierda a derecha y de la que se irán retirando los símbolos que ya hayan sido analizados. En la columna de la derecha se indicará la acción que se ha llevado a cabo en cada momento. La secuencia de reglas por las que se ha reducido constituyen la inversa de la derivación por la derecha de la cadena de entrada.

PILA	ENTRADA	ACCIÓN
0	id tipo begin codigo end \$	d4
0 4	tipo begin codigo end \$	d3
0 4 3	begin codigo end \$	r4 ($B \rightarrow \text{tipo}$)
0 4 7	begin codigo end \$	r5 ($B \rightarrow \text{id } B$)
0 2	begin codigo end \$	d6
0 2 6	codigo end \$	d10
0 2 6 10	end \$	r3 ($C \rightarrow \text{codigo}$)
0 2 6 9	end \$	r2 ($A \rightarrow \text{begin } C$)
0 2 5	end \$	d8
0 2 5 8	\$	r1 ($S \rightarrow B A \text{ end}$)
0 1	\$	aceptar

Gramática: $S \longrightarrow B A \text{ end}$
 $A \longrightarrow \text{begin } C$
 $C \longrightarrow \text{codigo}$
 $B \longrightarrow \text{tipo}$
 $B \longrightarrow \text{id } B$

Estado	Acción						Ir_a			
	end	begin	codigo	tipo	id	\$	S	A	B	C
0	-	-	-	d3	d4	-	1		2	
1	-	-	-	-	-	a!				
2	-	d6	-	-	-	-		5		
3	-	r4	-	-	-	-				
4	-	-	-	d3	d4	-			7	
5	d8	-	-	-	-	-				
6	-	-	d10	-	-	-				9
7	-	r5	-	-	-	-				
8	-	-	-	-	-	r1				
9	r2	-	-	-	-	-				
10	r3	-	-	-	-	-				

Las casillas vacías de la tabla **Acción** son *error*, y están marcadas con un guión. En la tabla **Ir_a** no se puede dar esta situación si está bien construida, y por eso no se han marcado las casillas vacías como error.

Figura 5.2: Tabla de análisis por desplazamiento y reducción

◁

5.3 Método SLR de construcción de tablas de análisis por desplazamiento y reducción

Existen diferentes métodos para construir tablas de análisis por desplazamiento y reducción. Uno de los más sencillos (aunque no el método más general posible con un símbolo de preanálisis) es el método SLR¹.

Para explicar el método SLR para la construcción de tablas de análisis es necesario definir previamente una serie de conceptos.

Elemento

Un elemento (*item* en la bibliografía en inglés) se obtiene situando un punto “•” en cualquier posición de la parte derecha de una regla.

$$A \longrightarrow X_1 X_2 \dots X_i \bullet X_{i+1} \dots X_n$$

Esta marca indica qué símbolos de una producción han sido ya reconocidos (los que quedan a su izquierda, X_k , $k = 1, \dots, i$, en la expresión anterior).

¹Ver [Aho, Sethi y Ullman, 1990, pp. 227 y ss.], también en [Fischer y LeBlanc, 1991, pp. 161 y ss.].

A partir de las producciones vacías ($A \rightarrow \epsilon$) sólo se obtiene un elemento: $A \rightarrow \bullet$.

Ejemplo 5.2

Sea la gramática del ejemplo 5.1:

$$\begin{aligned} S &\longrightarrow B A \text{ end} \\ A &\longrightarrow \text{begin } C \\ C &\longrightarrow \text{codigo} \\ B &\longrightarrow \text{tipo} \\ B &\longrightarrow \text{id } B \end{aligned}$$

Los posibles *elementos* que se pueden obtener de esa gramática son:

$$\begin{array}{ll} S \longrightarrow \bullet B A \text{ end} & B \longrightarrow \text{id } \bullet B \\ S \longrightarrow B \bullet A \text{ end} & B \longrightarrow \text{id } B \bullet \\ S \longrightarrow B A \bullet \text{ end} & A \longrightarrow \bullet \text{begin } C \\ S \longrightarrow B A \text{ end } \bullet & A \longrightarrow \text{begin } \bullet C \\ B \longrightarrow \bullet \text{tipo} & A \longrightarrow \text{begin } C \bullet \\ B \longrightarrow \text{tipo } \bullet & C \longrightarrow \bullet \text{codigo} \\ B \longrightarrow \bullet \text{id } B & C \longrightarrow \text{codigo } \bullet \end{array}$$

◁

Clausura

La operación *clausura* se aplica a un conjunto de *elementos*, I , y devuelve otro conjunto de *elementos*, $\text{clausura}(I)$, que incluye I , según las siguientes reglas:

1. Todo *elemento* del conjunto I se añade al conjunto $\text{clausura}(I)$.
2. Si $A \longrightarrow \alpha \bullet B\beta$ es un *elemento* de $\text{clausura}(I)$ siendo $B \in N$ y $B \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$, entonces se añaden los *elementos* $B \rightarrow \bullet \alpha_1$, $B \rightarrow \bullet \alpha_2$, ..., $B \rightarrow \bullet \alpha_n$ a $\text{clausura}(I)$ si no estaban. Esta regla se sigue aplicando hasta que no se puedan añadir más *elementos* a $\text{clausura}(I)$.

Ejemplo 5.3

Para la gramática del ejemplo anterior, si tenemos el conjunto de *elementos* formado únicamente por el *elemento* $S \longrightarrow \bullet B A \text{ end}$, el conjunto $\text{clausura}(\{S \longrightarrow \bullet B A \text{ end}\})$ sería:

$$\begin{aligned} S &\longrightarrow \bullet B A \text{ end} \\ B &\longrightarrow \bullet \text{tipo} \\ B &\longrightarrow \bullet \text{id } B \end{aligned}$$

◁

Ir_a

La operación *ir_a* tiene dos argumentos: un conjunto de elementos I y un símbolo gramatical A (terminal o no terminal), y da como resultado otro conjunto de elementos. Se define como sigue: para todos los elementos de la forma $B \rightarrow \alpha \bullet A\beta$ que haya en I , añadir los elementos de $clausura(B \rightarrow \alpha A \bullet \beta)$ al conjunto $ir_a(I, A)$.

Ejemplo 5.4

Si el conjunto I fuese

$$\begin{array}{lcl} S & \longrightarrow & \bullet B A \text{ end} \\ B & \longrightarrow & \bullet \text{ tipo} \\ B & \longrightarrow & \text{id } \bullet B \end{array}$$

el conjunto $ir_a(I, B)$ sería:

$$\begin{array}{lcl} S & \longrightarrow & B \bullet A \text{ end} \\ A & \longrightarrow & \bullet \text{ begin } C \\ B & \longrightarrow & \text{id } B \bullet \end{array}$$

<

5.3.1 Construcción de la colección canónica de conjuntos de elementos

El primer paso para construir una tabla de análisis con el método SLR es obtener la *colección canónica de conjuntos de elementos*, C . Para ello deben darse los siguientes pasos:

1. Ampliar la gramática añadiéndole la regla $X \rightarrow S$, donde S es el símbolo inicial.
2. Añadir el conjunto $I_0 = clausura(\{X \rightarrow \bullet S\})$ a la colección C .
3. Para cada conjunto de elementos I_i de C , y para cada símbolo gramatical A (terminal y no terminal) para el que exista en I_i un elemento del tipo $B \rightarrow \alpha \bullet A\beta$, $\beta \in (N \cup T)^+$ (es decir, que la marca no esté al final del elemento), añadir $ir_a(I_i, A)$ a C si no se había añadido antes. Esta operación se repite hasta que no se añadan más conjuntos nuevos a la colección de conjuntos de elementos $C = \{I_0, I_1, \dots\}$.

Es conveniente seguir un orden predeterminado para recorrer los símbolos gramaticales; por ejemplo, se puede empezar por los no terminales en orden de aparición en la gramática y seguir después por los terminales, también en orden de aparición en la gramática.

Ejemplo 5.5

En la gramática de los ejemplos anteriores, la colección canónica de conjuntos de elementos, C , se calcularía así:

1. Se calcula $I_0 = clausura(\{X \rightarrow \bullet S\})$:
 $I_0 = \{X \rightarrow \bullet S, S \rightarrow \bullet B A \text{ end}, B \rightarrow \bullet \text{ tipo}, B \rightarrow \bullet \text{id } B\}$

2. Se obtienen ahora los conjuntos de elementos correspondientes a $ir_a(I_0, X)$, donde X es cualquiera de los símbolos que en algún elemento de I_0 esté a continuación de un “•”:

$$\begin{aligned} I_1 &= ir_a(I_0, S) = clausura(\{X \longrightarrow S \bullet\}) = \{X \longrightarrow S \bullet\} \\ I_2 &= ir_a(I_0, B) = clausura(\{S \longrightarrow B \bullet A \text{ end}\}) = \{S \longrightarrow B \bullet A \text{ end}, \\ &\quad A \longrightarrow \bullet \text{ begin } C\} \\ I_3 &= ir_a(I_0, \text{tipo}) = clausura(\{B \longrightarrow \text{tipo} \bullet\}) = \{B \longrightarrow \text{tipo} \bullet\} \\ I_4 &= ir_a(I_0, \text{id}) = clausura(\{B \longrightarrow \text{id} \bullet B\}) = \{B \longrightarrow \text{id} \bullet B, \\ &\quad B \longrightarrow \bullet \text{ tipo}, B \longrightarrow \bullet \text{ id } B\} \end{aligned}$$

3. Desde los conjuntos anteriores que sólo tienen el “•” al final no se puede seguir la construcción (I_1, I_3). Calculamos, por tanto, los nuevos conjuntos de ir_a con los elementos que no tienen el “•” al final.

$$\begin{aligned} I_5 &= ir_a(I_2, A) = clausura(\{S \longrightarrow B A \bullet \text{ end}\}) = \{S \longrightarrow B A \bullet \text{ end}\} \\ I_6 &= ir_a(I_2, \text{begin}) = clausura(\{A \longrightarrow \text{begin} \bullet C\}) = \\ &\quad \{A \longrightarrow \text{begin} \bullet C, C \longrightarrow \bullet \text{ codigo}\} \end{aligned}$$

4. Es evidente que si se calcula $ir_a(I_4, \text{tipo})$ se vuelve a obtener I_3 , que ya existe. También se puede comprobar que $ir_a(I_4, \text{id}) = I_4$. El único conjunto nuevo que se obtiene es

$$I_7 = ir_a(I_4, B) = clausura(\{B \longrightarrow \text{id } B \bullet\}) = \{B \longrightarrow \text{id } B \bullet\}$$

5. De I_5 se obtiene:

$$\begin{aligned} I_8 &= ir_a(I_5, \text{end}) = clausura(\{S \longrightarrow B A \text{ end} \bullet\}) = \\ &\quad \{S \longrightarrow B A \text{ end} \bullet\} \end{aligned}$$

6. De I_6 :

$$\begin{aligned} I_9 &= ir_a(I_6, C) = clausura(\{A \longrightarrow \text{begin } C \bullet\}) = \{A \longrightarrow \text{begin } C \bullet\} \\ I_{10} &= ir_a(I_6, \text{codigo}) = clausura(\{C \longrightarrow \text{codigo} \bullet\}) = \\ &\quad \{C \longrightarrow \text{codigo} \bullet\} \end{aligned}$$

7. Todos los nuevos conjuntos de elementos llevan el “•” al final, luego ya no se pueden construir nuevos conjuntos y la colección $C = \{I_0, I_1, \dots, I_{10}\}$ ya está completa.

◁

5.3.2 Construcción de autómatas reconocedores de prefijos viables

Una forma más intuitiva de construir la colección canónica de conjuntos de elementos (y, como veremos más adelante, la tabla de análisis) es construir el autómata reconocedor de prefijos viables, utilizando el siguiente algoritmo:

$$S_0 = clausura(\{X \longrightarrow \bullet S\}) \equiv S_i$$

REPETIR

PARA cada $A \in N \cup T / \exists B \rightarrow \alpha \bullet A\beta \in S_i$ HACER

Crear un nuevo estado $S_n = ir_a(S_i, A)$ (si no existe)

Crear una transición $S_i \xrightarrow{A} S_n$

FIN_PARA

En la siguiente iteración considerar como S_i cada S_n nuevo.

HASTA no poder crear más S_n

Los estados S_i del autómata son los conjuntos de elementos I_0, I_1, \dots de la colección canónica de elementos, C .

Ejemplo 5.6

El autómata para la gramática

$$\begin{aligned} E &\longrightarrow E + a \\ E &\longrightarrow a \end{aligned}$$

sería el de la figura 5.3.

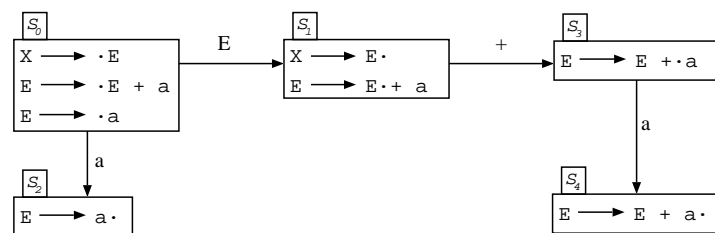


Figura 5.3: Autómata reconocedor de prefijos viables.

◁

Ejemplo 5.7

En la figura 5.4 se muestra el autómata reconocedor de prefijos viables para la gramática que estamos estudiando en los ejemplos de este capítulo.

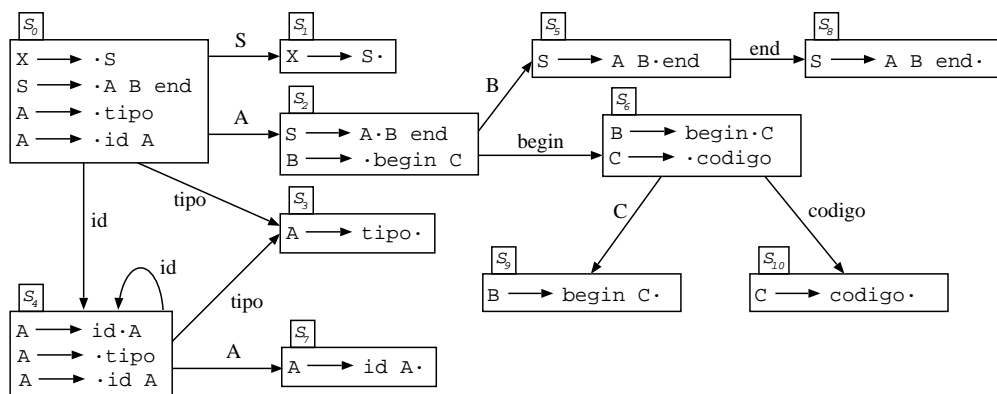


Figura 5.4: Autómata para la gramática de ejemplo.

◁

5.3.3 Construcción de tablas SLR

Como se indicó al principio de este capítulo, las tablas de análisis sintáctico SLR están divididas en dos partes con la siguiente estructura y valores para su indexación:

	$T \cup \{\$ \}$	N
	Acción	Ir_a
Estados	$\mathbf{d}j$ $\mathbf{r}k$ aceptar error	Estados

El método de construcción de tablas SLR se puede resumir en los siguientes pasos:

1. Obtener la colección canónica de conjuntos de *elementos* $C = \{I_0, I_1, \dots, I_n\}$.
2. Cada conjunto I_i de C se corresponde con el estado i del analizador.
3. Construcción de la tabla **Ir_a**:

Se buscan no terminales que aparezcan antes del punto en el conjunto I_i , es decir, *elementos* de la forma $A \rightarrow \alpha \bullet B \beta$ donde $B \in N$. Según el método de construcción de la colección C , existirá un I_j tal que $ir_a(I_i, B) = I_j$. En este caso, en la entrada **Ir_a** $[i, B]$ habrá que poner el estado j .

4. Construcción de la tabla **Acción**:
 - (a) Para todos los *elementos* de la forma $A \rightarrow \alpha \bullet a \beta$ (con $a \in T$) en el conjunto I_i , existirá un I_j tal que $ir_a(I_i, a) = I_j$. En este caso, la entrada **Acción** $[i, a]$ será **dj**, es decir, desplazar e ir al estado j .
 - (b) Para los *elementos* de I_i con el punto al final de la regla, es decir, *elementos* de la forma $A \rightarrow \beta \bullet$ o de la forma $A \rightarrow \bullet$, hay que calcular los $SIG(A)$, y para todo s que aparezca en $SIG(A)$ hay que poner en la entrada **Acción** $[i, s]$ la acción **rk**, es decir, reducir por la regla número k . La regla $X \rightarrow S$ se añade sólomente para construir la colección C , nunca se reduce por ella y, por tanto, no se numera.
 - (c) Si el *elemento* $X \rightarrow S \bullet$ está en el conjunto I_i , poner **aceptar** en la entrada **Acción** $[i, \$]$.
 - (d) Todas las entradas de la tabla **Acción** que queden vacías son entradas erróneas y por tanto el analizador debe producir un error cuando acceda a ellas.

Ejemplo 5.8

A partir de la colección canónica de conjuntos de elementos calculada en el ejemplo 5.5, vamos a construir la tabla de análisis para la gramática que estamos utilizando como ejemplo desde el principio de este capítulo. La tabla de análisis

SLR que va a resultar es la que aparece en la figura 5.2. Consideraremos que cada producción está numerada según el orden en el que están escritas al presentar esta gramática al principio del capítulo. A partir de ella, se obtienen las siguientes casillas de la tabla:

Estado 0: Puesto que $ir_a(I_0, S) = I_1$ y también $ir_a(I_0, B) = I_2$, se llenan $\mathbf{Ir_a}[0, S] = 1$ e $\mathbf{Ir_a}[0, B] = 2$. Además, desde el estado 0 se va al estado 3 con **tipo** ($ir_a(I_0, \mathbf{tipo}) = I_3$), y al estado 4 con **id**; así, $\mathbf{Acción}[0, \mathbf{tipo}] = \mathbf{d3}$ y $\mathbf{Acción}[0, \mathbf{id}] = \mathbf{d4}$. Como no hay ningún *elemento* en el conjunto I_0 con el punto al final de la regla, no hay reducciones en el estado 0.

Estado 1: La única acción que hay que poner es $\mathbf{Acción}[1, \$] = \mathbf{aceptar}$.

Estado 2: En este estado hay que poner un 5 en $\mathbf{Ir_a}[2, A]$ porque $ir_a(I_2, A) = I_5$, y también hay que poner en $\mathbf{Acción}[2, \mathbf{begin}]$ la acción **d6** ya que $ir_a(I_2, \mathbf{begin}) = I_6$.

Estado 3: El único *elemento* del conjunto I_3 tiene el punto al final de la regla, luego en este estado sólo habrá que poner acciones de reducción por la regla $B \rightarrow \mathbf{tipo}$ en todos los $\mathbf{SIG}(B)$, que en este caso sólo es **begin**. Por tanto, la única acción en este estado será $\mathbf{Acción}[3, \mathbf{begin}] = \mathbf{r4}$.

Estado 4: Desde este estado hay tres transiciones: con **tipo** se va a I_3 (por tanto, $\mathbf{Acción}[4, \mathbf{tipo}] = \mathbf{d3}$), con **id** vuelve al propio I_4 ($\mathbf{Acción}[4, \mathbf{id}] = \mathbf{d4}$), y con una B va al estado I_7 ($\mathbf{Ir_a}[4, B] = 7$).

Estado 5: En este estado sólo hay una transición con **end**, luego hay que hacer $\mathbf{Acción}[5, \mathbf{end}] = \mathbf{d8}$.

Estado 6: En este estado hay una transición a I_9 con C ($\mathbf{Ir_a}[6, C] = 9$), y a I_{10} con **codigo** ($\mathbf{Acción}[6, \mathbf{codigo}] = \mathbf{d10}$).

Estado 7: En este estado hay que reducir por la regla $B \rightarrow \mathbf{id} B$ en todos los $\mathbf{SIG}(B)$, luego hay que hacer $\mathbf{Acción}[7, \mathbf{begin}] = \mathbf{r5}$.

Estado 8: En este estado hay reducir por la primera regla en todos los $\mathbf{SIG}(S)$, luego hay que hacer $\mathbf{Acción}[8, \$] = \mathbf{r1}$.

Estado 9: De igual forma que en los dos estados anteriores, en este estado solamente hay que reducir por la regla $A \rightarrow \mathbf{begin} C$. El único siguiente de A es **end**, luego habrá que hacer $\mathbf{Acción}[9, \mathbf{end}] = \mathbf{r2}$.

Estado 10: Finalmente, en este estado hay que reducir por la regla de C en los $\mathbf{SIG}(C)$ (que son los $\mathbf{SIG}(A)$, es decir, solamente **end**), luego habrá que hacer $\mathbf{Acción}[10, \mathbf{end}] = \mathbf{r3}$.

◁

5.3.4 Construcción de tablas SLR a partir del autómata reconocedor de prefijos viables

En el algoritmo descrito para la construcción de la tabla SLR a partir de la colección canónica de elementos basta con sustituir los conjuntos de elementos I_i por los estados

del autómata S_i (que tendrán idéntica numeración si el orden de construcción de los estados ha sido el mismo que el de aquellos conjuntos).

La función de transición se corresponde con las entradas de la tabla de la siguiente manera:

1. Las transiciones etiquetadas con un no terminal A con los valores de la tabla **Ir_a** $[S_i, A]$;
2. Las transiciones etiquetadas con un terminal a con los desplazamientos desde el estado S_i a S_j (**dj**) para ese terminal en la celda **Acción** $[S_i, a]$.
3. La metodología para poner las reducciones en la tabla es la misma que en el caso de las colecciones canónicas, pero usando los estados del autómata. En cada estado, es necesario calcular los siguientes de las partes izquierdas de aquellas producciones con el punto al final del elemento, y poner la reducción correspondiente en la casilla de la tabla asociada a ese estado y a cada símbolo siguiente.

Se puede comprobar esta técnica aplicándola al autómata del ejemplo 5.7 para comprobar que sale la tabla de análisis SLR de la figura 5.2.

5.4 Conflictos en las tablas SLR

En las tablas SLR (y en las de la familia LR en general) pueden aparecer dos clases de conflictos que provocan entradas múltiples en la tabla. La aparición de un conflicto indica que la gramática no es SLR, aunque podría ser LALR(1), LR(1), etc., o podría ser una gramática de otro tipo o bien una gramática ambigua. Es importante destacar que no todos los conflictos aparecen por culpa de que la gramática sea ambigua; también existen gramáticas no ambiguas que producen conflictos porque no son SLR.

Desplazamiento-reducción

Este conflicto se da cuando en una misma entrada de la tabla **Acción** es posible desplazar y reducir con el mismo símbolo de la entrada. Aunque la gramática no sea SLR, se podría construir el analizador resolviendo el conflicto, es decir, eligiendo una de las dos posibilidades (desplazar o reducir); la elección depende del lenguaje concreto que se quiera analizar y debe hacerse con mucho cuidado para conseguir un analizador que reconozca *exactamente* el lenguaje descrito por la gramática y con la semántica deseada.

Ejemplo 5.9

Sea la siguiente gramática²:

$$\begin{array}{ll}
 S & \longrightarrow L = R \\
 S & \longrightarrow R \\
 L & \longrightarrow * R \\
 L & \longrightarrow \mathbf{id} \\
 R & \longrightarrow L
 \end{array}$$

²[Aho, Sethi y Ullman, 1990, p. 235]

Compruébese que esta gramática tiene un conflicto desplazamiento-reducción en el estado 2 (o en el que contenga los elementos $S \rightarrow L \bullet =R$ y $R \rightarrow L \bullet$), ya que con el símbolo “=” es posible desplazar o reducir por la regla $R \rightarrow L$.

◁

Ejemplo 5.10

La siguiente gramática es una simplificación de una gramática ambigua que define la construcción IF-THEN-ELSE de la mayoría de lenguajes de programación de alto nivel. Aunque la gramática sea ambigua, muchos compiladores utilizan analizadores ascendentes que resuelven este conflicto escogiendo la opción de desplazar en lugar de la de reducir; de esta forma, se asocia el ELSE al IF más cercano.

$$\begin{array}{lcl} S & \longrightarrow & \mathbf{i} S \\ S & \longrightarrow & \mathbf{i} S \mathbf{e} S \\ S & \longrightarrow & \mathbf{o} \end{array}$$

Al construir la colección canónica de conjuntos encontraremos el conjunto $I_j = \{ S \rightarrow \mathbf{i} S \bullet, S \rightarrow \mathbf{i} S \bullet \mathbf{e} S \}$. Si calculamos los $\text{SIG}(S)$ veremos que son $\{\mathbf{e}, \mathbf{o}\}$. Por tanto, en el estado j , en la entrada **Acción** $[j, \mathbf{e}]$ podremos poner la acción **r1** o bien la acción **dk**, siendo k el estado correspondiente a $I_k = \text{ir}_a(I_j, \mathbf{e})$.

En este caso concreto, si elegimos la acción de desplazar, **dk**, el analizador reconocerá perfectamente el lenguaje generado por la gramática; si hubieramos elegido reducir, el analizador no podría reconocer las cadenas con una “e”, como por ejemplo “**i o e o**”.

◁

Reducción-reducción

Este conflicto aparece cuando en una misma entrada de la tabla **Acción** es posible reducir por dos o más reglas distintas ante el mismo símbolo en la entrada. La solución más razonable para resolver este conflicto suele ser cambiar la gramática, aunque es posible elegir una de las reducciones y descartar las demás, teniendo en cuenta siempre que el analizador resultante debe reconocer *exactamente* el lenguaje definido por la gramática.

Ejemplo 5.11

La siguiente gramática tiene un conflicto reducción-reducción:

$$\begin{array}{lcl} S & \longrightarrow & \mathbf{id} A \mid \mathbf{id} B \mathbf{fin} \\ B & \longrightarrow & \mathbf{prin} A \mathbf{fin} \mid \epsilon \\ A & \longrightarrow & \mathbf{otro} \mid \epsilon \end{array}$$

Al hacer el autómata, encontramos que el estado 2 presenta ese conflicto, puesto que en ese estado se debe reducir por la regla $A \rightarrow \epsilon$ con los $\text{SIG}(A)$, y por la regla $B \rightarrow \epsilon$ con los $\text{SIG}(B)$. El conflicto aparece porque “**fin**” pertenece a ambos conjuntos de siguientes.

◁

5.5 Mensajes de error en un analizador SLR

Un analizador SLR se puede implementar a partir de una tabla de análisis construida *a mano*, es decir, con lápiz y papel. En cuanto a los errores, existen dos técnicas alternativas para producir mensajes de error:

- Es posible producir mensajes de error específicos rellenando aquellas casillas de la tabla **Acción** que queden vacías con un número que indique un mensaje de error particular para cada caso. Aunque la tabla **Ir_a** tenga entradas vacías, nunca se van a llegar a utilizar. Algunas entradas de la tabla **Acción** tampoco se utilizarán nunca, pero la mayoría pueden ser utilizadas. En este caso, para reducir el número de mensajes de error específicos que hay que implementar se puede, en aquellos estados en los que existan reducciones y sean todas por la misma regla, extender esas reducciones también a las casillas vacías; de esta forma, los errores se detectarán al desplazar en lugar de al reducir.
- Cuando el lenguaje a analizar tiene un tamaño medio, el número de estados crece considerablemente y una técnica de mensajes de error específicos es muy costosa y laboriosa. Por este motivo, se pueden producir mensajes de error generales de la siguiente manera: cuando se produce el error, en el tope de la pila hay un estado s determinado; los símbolos esperados en lugar del *token* de la entrada son aquellos para los que la tabla **Acción** en la fila del estado s tenga alguna acción, ya sea una reducción o un desplazamiento.

Los mensajes de error producidos por un analizador SLR suelen ser diferentes (en el número de símbolos esperados) que los producidos por analizadores descendentes recursivos o con tabla para la misma gramática.

De igual forma que con los mensajes de error, es posible implementar una estrategia de recuperación de errores específica, que trate cada error o tipo de error de forma diferente. Sin embargo, lo normal es que el número de estados de una tabla SLR no permita emplear esta estrategia y se tenga que implementar una técnica de recuperación de errores general (no dependiente del lenguaje).

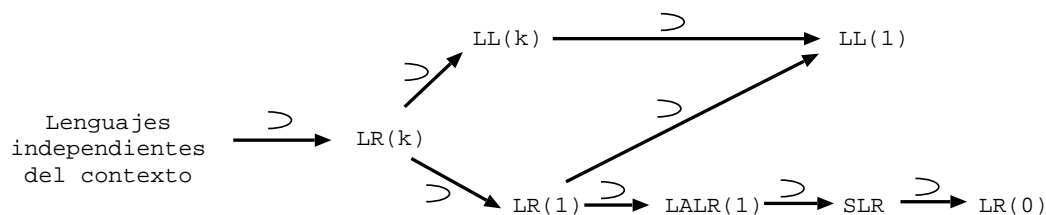
Una buena técnica de recuperación de errores debe eliminar el menor número de *tokens* de la entrada antes de reanudar el análisis, y a la vez debe dejar al analizador en un estado en el que sea posible analizar el resto de la entrada y detectar más errores si los hubiera. Conseguir que se cumplan ambas condiciones a la vez es difícil y se debe intentar conseguir un compromiso entre ambas opciones, es decir, eliminar pocos *tokens* de la entrada y a la vez dejar el analizador en un estado en el que pueda continuar el análisis.

5.6 Clasificación de gramáticas

Hemos visto que no todos los analizadores sirven para todas las gramáticas, ya que a la hora de construir las tablas de análisis hay que imponer determinadas restricciones a las gramáticas. Vamos a ver una comparación entre las gramáticas que pueden ser analizadas según cada uno de los métodos, estableciendo así una comparación entre distintos tipos de análisis.

Entre todos los métodos existentes en la práctica, el más amplio es el LR. Se sabe que, en lo que respecta a lenguajes de programación, todos los que pueden ser analizados con analizadores $LR(k)$, pueden serlo con $LALR(1)$ o SLR, puesto que puede encontrarse una gramática equivalente que verifique esta propiedad. Estos lenguajes se llaman deterministas y son un subconjunto propio de los de contexto libre.

Por otra parte, puede demostrarse que los lenguajes $LL(k)$ son un subconjunto propio de los $LR(k)$. A continuación se muestra un diagrama que muestra estas relaciones, en el que cada flecha representa la inclusión del conjunto que está al final de la flecha en el conjunto origen.



La relación entre las gramáticas $LL(1)$, SLR y las gramáticas ambiguas (todas ellas independientes del contexto) es compleja y se podría definir con las siguientes afirmaciones:

- No existe ningún algoritmo para saber (en tiempo finito) si una gramática G es ambigua o no. La única forma de asegurar que una gramática es ambigua es encontrando una cadena con dos o más árboles de derivación para esa gramática.
- Una gramática G es $LL(1)$ si y sólo si los conjuntos de predicción de las reglas de cada variable son disjuntos entre sí.
- Una gramática G es SLR si es posible construir una tabla de análisis SLR (sin entradas múltiples, obviamente) para ella.
- Si una gramática tiene recursividad por la izquierda, entonces no es $LL(1)$. Sin embargo, si una gramática no es $LL(1)$ no necesariamente tiene que tener recursividad por la izquierda.
- Si una gramática tiene factores comunes por la izquierda, entonces no es $LL(1)$.
- Que una gramática G tenga recursividad por la izquierda o factores comunes por la izquierda no influye en que la gramática sea o no SLR.

- Las transformaciones para eliminar la recursividad por la izquierda (o los factores comunes por la izquierda) lo que hacen es construir una gramática equivalente (que genera exactamente el mismo lenguaje). Si transformamos una gramática G eliminando la recursividad por la izquierda y los factores comunes, obtendremos una gramática equivalente G' que genera el mismo lenguaje y es posible que sea $LL(1)$, pero no se puede asegurar siempre que G' sea $LL(1)$.
- Si una gramática G es ambigua, entonces no es $LL(1)$, ni SLR , ni $LR(1)$. Sin embargo, si G no es ambigua es posible que sea $LL(1)$ y también es posible que no lo sea. De igual forma, una gramática G no ambigua puede ser o no SLR .
- Si una gramática G es $LL(1)$, entonces no es ambigua (y puede que sea SLR). En cambio, si G' no es $LL(1)$, es posible que sea ambigua y es posible que no lo sea. Por ejemplo, la siguiente gramática no es $LL(1)$ y no es ambigua:

$$\begin{array}{lcl} E & \longrightarrow & E + a \\ E & \longrightarrow & a \end{array}$$

- Si una gramática G es SLR , entonces no es ambigua (y puede que sea $LL(1)$). En cambio, si G' no es SLR , puede que sea ambigua o puede que no. Por ejemplo, la siguiente gramática no es SLR (ni tampoco $LL(1)$) y no es ambigua:

$$\begin{array}{lcl} S & \longrightarrow & Qq \\ S & \longrightarrow & Rr \\ Q & \longrightarrow & (A) \\ R & \longrightarrow & (E) \\ A & \longrightarrow & a \\ E & \longrightarrow & a \end{array}$$

- Se dice que un lenguaje L es $LL(1)$ (o es SLR) si existe una gramática $LL(1)$ (o SLR) que genera ese lenguaje. Puede haber otras gramáticas de otras categorías que generen ese lenguaje, e incluso puede haber gramáticas ambiguas que generen un lenguaje $LL(1)$ (o SLR).

Nota: En la página 68 de [Appel, 1998] se puede observar, en la figura 3.29, un diagrama que muestra gráficamente la relación entre todos estos tipos de gramáticas.

5.7 Referencias bibliográficas

REFERENCIAS	EPÍGRAFES
[Louden, 1997]	5.1, 5.2 y 5.3
[Aho, Sethi y Ullman, 1990]	4.5 y 4.7
[Bennett, 1990]	6.3.1, 6.3.2 y 6.3.3
[Fischer y LeBlanc, 1991]	6.1, 6.2, 6.4 y 6.11

5.8 Ejercicios

Ejercicio 5.1

A partir de la siguiente gramática (de expresiones aritméticas), construir la tabla de análisis SLR a partir del autómata reconocedor de prefijos viables y, a continuación, hacer la traza del análisis ascendente de la cadena “**id * id + id**”:

$$\begin{aligned} E &\longrightarrow E + T \\ E &\longrightarrow T \\ T &\longrightarrow T * F \\ T &\longrightarrow F \\ F &\longrightarrow (E) \\ F &\longrightarrow \text{id} \end{aligned}$$

Ejercicio 5.2

Dada la siguiente gramática:

$$\begin{aligned} S &\longrightarrow S \text{ inst} \\ S &\longrightarrow T R V \\ T &\longrightarrow \text{tipo} \\ T &\longrightarrow \epsilon \\ R &\longrightarrow \text{blq } V \text{ fblq} \\ R &\longrightarrow \epsilon \\ V &\longrightarrow \text{id } S \text{ fin} \\ V &\longrightarrow \text{id ;} \\ V &\longrightarrow \epsilon \end{aligned}$$

Escribid la tabla de análisis sintáctico SLR para esta gramática y haced la traza del funcionamiento del analizador SLR para las cadenas “**tipo id blq id ; fblq fin inst**” e “**id tipo id fin ;**”

Ejercicio 5.3

Dada la siguiente gramática:

$$\begin{aligned} E &\longrightarrow [L] \\ E &\longrightarrow \mathbf{a} \\ L &\longrightarrow E Q \\ Q &\longrightarrow , L \\ Q &\longrightarrow \epsilon \end{aligned}$$

Escribid la tabla de análisis sintáctico SLR para esta gramática y haced la traza del funcionamiento del analizador SLR para la cadena “[**a**, [**a**, **a**],]”.

Ejercicio 5.4

Dada la siguiente gramática:

$$\begin{aligned}
 S &\longrightarrow S \text{ inst} \\
 S &\longrightarrow S \text{ var } D \\
 S &\longrightarrow \epsilon \\
 D &\longrightarrow D \text{ ident } E \\
 D &\longrightarrow D \text{ ident sep} \\
 D &\longrightarrow \text{int} \\
 D &\longrightarrow \text{float} \\
 E &\longrightarrow S \text{ fproc}
 \end{aligned}$$

Construye una tabla de análisis SLR para ella y haz la traza del funcionamiento del analizador para las cadenas “var int ident inst fproc inst” e “inst fproc”.

Ejercicio 5.5

Escribe la tabla de análisis sintáctico SLR para la gramática siguiente:

$$\begin{aligned}
 S &\longrightarrow \text{id } (L) \\
 S &\longrightarrow \text{id} \\
 L &\longrightarrow \epsilon \\
 L &\longrightarrow S Q \\
 Q &\longrightarrow \epsilon \\
 Q &\longrightarrow , S Q
 \end{aligned}$$

Haz también la traza de las cadenas “a(b())” y “c(d,e,())”.

Ejercicio 5.6

Dada la siguiente gramática:

$$\begin{aligned}
 S &\longrightarrow A B C \\
 A &\longrightarrow \text{begin } C \text{ end} \\
 A &\longrightarrow \epsilon \\
 B &\longrightarrow \text{var tipo} \\
 C &\longrightarrow B C \\
 C &\longrightarrow \text{fvar}
 \end{aligned}$$

Sin modificar en absoluto la gramática (excepto la modificación que prescribe el algoritmo, la de añadir una regla al principio), diseña un analizador SLR para esa gramática y haz la traza de la cadena “begin var tipo fvar end var tipo fvar”.

Capítulo 6

Traducción dirigida por la sintaxis

6.1 Introducción

Las fases de análisis semántico y generación de código intermedio están fuertemente entrelazadas, y normalmente son coordinadas por el analizador sintáctico. El proceso de análisis semántico y el de traducción se dice que están dirigidos por la sintaxis, ya que es el analizador sintáctico el que va invocando rutinas que realizan el análisis semántico y la traducción a la vez que va analizando sintácticamente el programa fuente.

En el lenguaje humano, mediante el análisis del significado (de la semántica) de las frases las comprendemos y obramos en consecuencia. Los compiladores analizan semánticamente los programas para ver si son compatibles con las especificaciones semánticas del lenguaje al cual pertenecen y así poder *comprenderlos*, lo cual permite su traducción a otro lenguaje. Así pues, el compilador verifica la coherencia semántica de un programa y a la vez lo traduce al lenguaje de la máquina o a una representación intermedia. Es importante resaltar que, salvo casos excepcionales, el análisis semántico y la traducción se realizan de forma simultánea. Por todo ello, en este capítulo hablaremos del significado (semántica) de los símbolos gramaticales pero también de cómo traducirlos a código intermedio.

Para que los símbolos de una gramática puedan adquirir significado, se les asocia información (*atributos*) y a las producciones gramaticales se les asocian *acciones semánticas*, que serán código en un lenguaje de programación (o pseudo-código algorítmico) y cuya misión es evaluar los atributos y manipular dicha información para llevar a cabo las tareas de traducción.

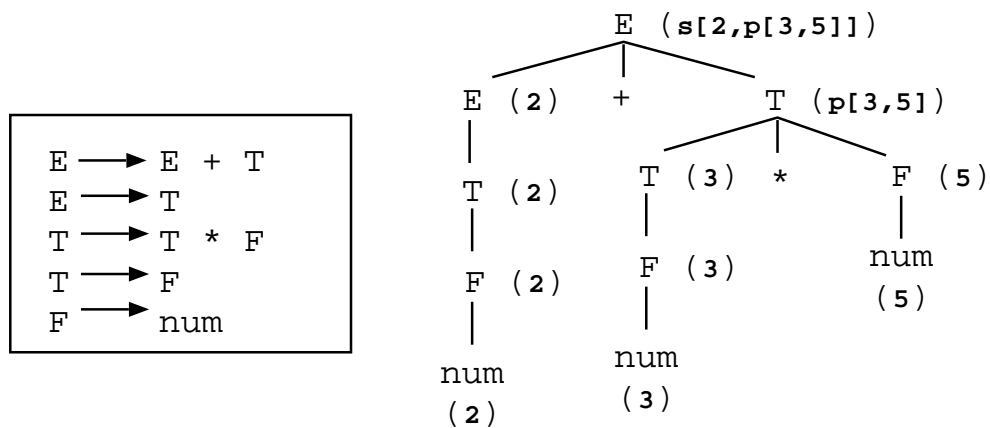
Desde un punto de vista teórico las distintas fases de análisis son independientes entre sí, pero en la práctica se hace todo de manera simultánea. Veremos que el concepto de *gramática con atributos* y la inclusión de las acciones semánticas en las producciones lleva a que, conforme se va haciendo el análisis sintáctico (que es el encargado de ir pidiendo *tokens* conforme los va necesitando), se vayan evaluando las acciones semánticas. De esta forma, los analizadores sintáctico y semántico y el generador de código intermedio se implementarán como un único subprograma, imposible de disociar por un programador no experto en el diseño de compiladores.

La razón de este hecho es el *principio de la traducción dirigida por la sintaxis* que dice que el significado (la semántica) de una frase está directamente relacionado por su

estructura sintáctica, según se representa en su árbol de análisis sintáctico. Por tanto, la traducción de un programa (una frase en un lenguaje de programación) está íntimamente ligada a la sintaxis concreta de ese programa, como lo está su semántica.

Ejemplo 6.1

Vamos a utilizar la gramática de las expresiones aritméticas para ver cómo se van haciendo traducciones parciales en cada nodo del árbol de análisis sintáctico para llegar a obtener la traducción a una notación prefija de la frase de entrada “2+3*5” en el nodo raíz. En la figura siguiente se muestra el árbol con el valor de la traducción parcial correspondiente a la parte del árbol que queda por debajo de cada nodo.



<

Ejemplo 6.2

Vamos a ver ahora otro ejemplo orientado hacia la traducción entre lenguajes de programación en el que declaraciones de variables de tipo real o entero en C van a ir traduciéndose a las equivalentes declaraciones en Pascal conforme se va recorriendo el árbol de análisis sintáctico en un recorrido en profundidad por la izquierda. La gramática de las declaraciones en C vamos a suponer que es la siguiente:

$$\begin{array}{lcl} S & \longrightarrow & T \text{ id } L ; \\ L & \longrightarrow & , \text{ id } L \mid \epsilon \\ T & \longrightarrow & \text{float} \mid \text{int} \end{array}$$

De manera que, por ejemplo, la cadena “float a,b,c;” será traducirá como “var a,b,c:real;”. La figura 6.1 muestra el árbol con las traducciones parciales que se obtienen en cada nodo. También se indica con flechas el flujo de información (cómo “viajan” esas traducciones parciales por el árbol) desde las hojas hasta el nodo raíz, donde se obtiene la traducción de la entrada analizada.

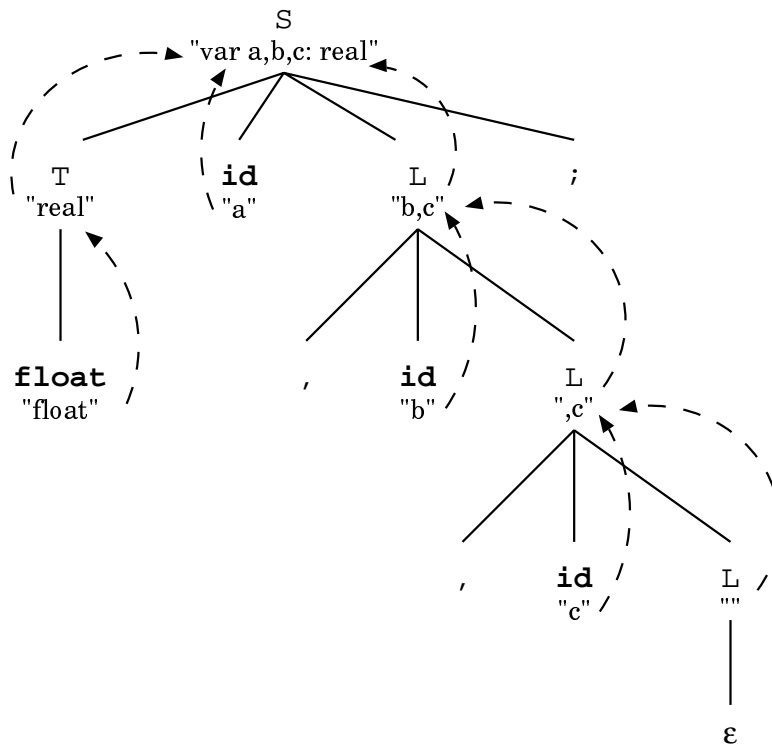


Figura 6.1: Flujo de la información en un proceso de traducción.

Veremos que estas operaciones de traducción son propias de cada nodo del árbol (de cada regla). El comportamiento de la información siempre es el mismo en cada regla.

<

6.2 Gramáticas de atributos

En el apartado anterior hemos visto que cuando la semántica y la traducción entran en juego, a los árboles de análisis sintáctico se les añade información en cada nodo. Estos árboles reciben el nombre por parte de algunos autores de “árboles con adornos”. Para que esa información pueda viajar a través del árbol durante el análisis hace falta un “almacén” para ella. Estos almacenes se llaman *atributos*.

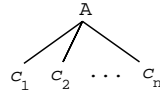
Los atributos son propiedades de los símbolos del lenguaje que almacenan contenidos relacionados con el significado de los símbolos a los que pertenecen; por ejemplo: tipos de datos, direcciones de memoria, traducciones parciales, etc. Así pues, cada símbolo de la gramática tiene sus propios atributos en los que almacena la información semántica.

La existencia de atributos para los símbolos de la gramática conduce al concepto de *gramática de atributos*, en donde a la gramática se le añaden atributos para sus símbolos y *reglas semánticas* para calcular sus valores. Veremos más adelante la importancia

de que estas reglas semánticas se limiten a manipular y calcular atributos, evitando la aparición de efectos laterales. En una gramática de atributos, cada símbolo (terminal o no terminal) puede contemplarse como un registro, cada uno de cuyos campos es un atributo distinto:

$$A = (A.tipo, A.traduccion, A.direccion, \dots)$$

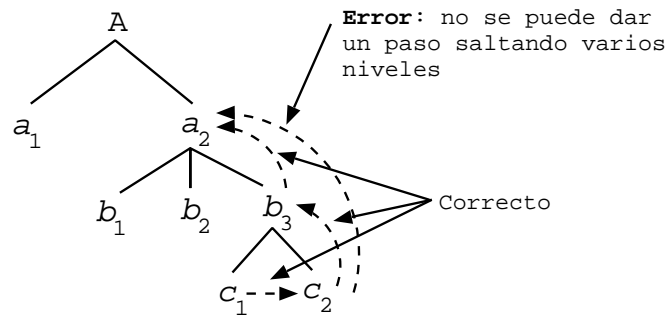
Cualquier producción de la gramática tiene asociado un subárbol sintáctico. Si $A \rightarrow c_1 c_2 \dots c_n$ su árbol será:



La evaluación de los atributos de los símbolos que aparecen en este subárbol se realiza mediante operaciones que reciben el nombre de *reglas o acciones semánticas*¹. Estas operaciones se realizan sobre los valores de los atributos de los símbolos de esa misma regla (subárbol). Así pues, el valor de cualquiera de los atributos de un símbolo de una regla será función de los valores de (como máximo) todos los atributos de todos los símbolos de la regla (subárbol):

$$A.a_i = f(c_1.a_{11}, \dots, c_1.a_{k1}, c_2.a_{12}, \dots, c_2.a_{k2}, \dots, c_n.a_{1n}, \dots, c_n.a_{kn}, A.a_1, \dots, A.a_k)$$

Es decir, pueden aparecer uno o más atributos de uno o más símbolos, pero no tienen porqué aparecer todos los atributos de todos los símbolos, y *no pueden aparecer* atributos de símbolos que no aparezcan en la producción. Así pues, los atributos de un árbol sólo se pueden calcular de un nivel al adyacente o en el mismo nivel.



Las gramáticas de atributos suelen escribirse como tablas de dos columnas: la columna izquierda contiene las producciones de la gramática y la de la derecha las reglas semánticas asociadas a cada producción, expresando así la relación entre sintaxis (producciones) y semántica asociada (reglas).

¹También reciben el nombre de *ecuaciones de atributos*.

Ejemplo 6.3

En el ejemplo anterior, vamos a asociar a cada producción una regla semántica de traducción que va a calcular la traducción parcial que corresponde a cada nodo del árbol en función de los valores de los nodos hijos (según el flujo de información que se dibujó en aquel árbol) y de acuerdo con la notación que acabamos de definir para representar los atributos de los símbolos como campos de un registro. En el metalenguaje de las reglas semánticas “:=” es la asignación y “||” representa un operador de concatenación de cadenas.

Producción	Regla semántica asociada
$S \rightarrow T \text{ id } L ;$	$S.trad := \text{"var"} \text{id.lexema} L.trad \text{":"} T.trad \text{";"}$
$L \rightarrow , \text{ id } L_1$	$L.trad := \text{","} \text{id.lexema} L_1.trad$
$L \rightarrow \epsilon$	$L.trad := \text{" "}$
$T \rightarrow \text{float}$	$T.trad := \text{"real"}$
$T \rightarrow \text{int}$	$T.trad := \text{"integer"}$

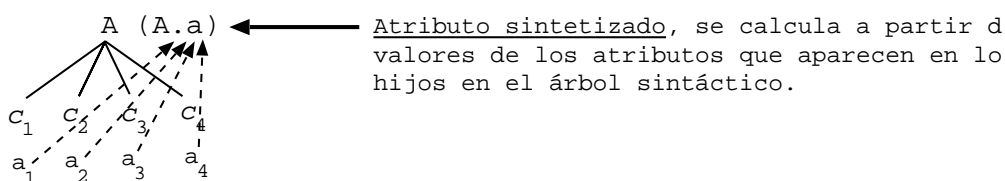
Nota: El símbolo L_1 en realidad es L , pero se le pone un subíndice para distinguirlo de la parte izquierda de la regla en la acción semántica.

◁

6.3 Tipos de atributos

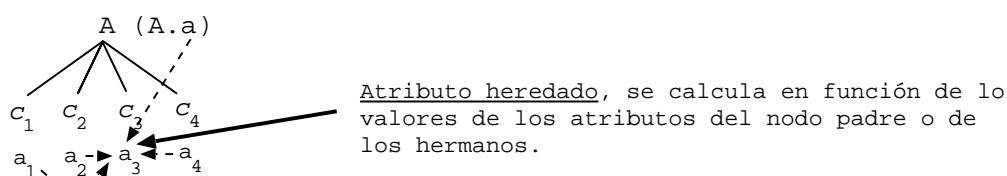
Los atributos que se asignan a terminales y no terminales pueden ser de cualquier tipo de datos y dependen exclusivamente de lo que se pretenda hacer en el análisis semántico y en la traducción a código intermedio. Los atributos asociados pueden ser de dos clases dependiendo de quién les suministra sus valores:

Atributos sintetizados: se calculan a partir de los valores de los nodos hijos en cualquier subárbol en el que aparezcan. Por tanto, un atributo sintetizado (cuando recibe valor) siempre será del no terminal de la izquierda en una producción. En $A \rightarrow c_1 c_2 \dots c_n$, el atributo $A.a = f(c_1.a_{i1}, c_2.a_{i2}, \dots, c_n.a_{in})$ será sintetizado. Se trata de información que asciende por el árbol durante el recorrido del mismo. Por ejemplo, $A \rightarrow c_1 c_2 \dots c_n$



Atributos heredados: son los atributos que no son sintetizados; su valor se calcula a partir de los atributos del nodo padre o de los hermanos del nodo del símbolo

al que pertenece. Por tanto, un atributo heredado (cuando recibe valor) siempre será el de un símbolo de la derecha en una producción. En la producción $A \rightarrow c_1 c_2 \dots c_n$, el atributo $c_i.a = g(A.a, c_1.a_{i1}, c_2.a_{i2}, \dots, c_n.a_{in})$ será heredado. Se trata de información descendente o “en tránsito horizontal” de un lado a otro del subárbol.



Los atributos mantienen este carácter a lo largo de toda la gramática. Si un atributo es heredado (o sintetizado) debe ser siempre heredado (o sintetizado) en todas las producciones. Así pues, si en una regla se le asigna un valor a un atributo heredado de un símbolo, en todas las demás reglas en las que aparezca ese símbolo en la parte derecha se le debe asignar un valor a ese atributo.

Los atributos de los terminales (*tokens*) son siempre sintetizados y sus valores vienen siempre suministrados por el analizador léxico. Los atributos heredados suelen ser “herramientas para la comunicación” entre diferentes partes del árbol. Sirven a menudo para expresar la dependencia, con respecto al contexto en el que aparece (que viene de arriba por el árbol), de una construcción de un lenguaje de programación.

Ejemplo 6.4

Vamos a ver de nuevo el ejemplo las expresiones aritméticas con la traducción a notación prefija de la cadena de entrada expresada como una gramática de atributos. Todos sus atributos son sintetizados.

$E \rightarrow E + T$	E, T y F tienen un atributo al que llamamos <i>trad</i>
$E \rightarrow T$	(cadena)
$T \rightarrow T * F$	num tiene un atributo llamado <i>valex</i> (valor léxico o
$T \rightarrow F$	lexema)
$F \rightarrow \text{num}$	

En la tabla que se ofrece a continuación, la columna de la izquierda corresponde a las distintas producciones que la forman y la de la derecha describe la regla semántica que hay que llevar a cabo para cada producción. En la figura 6.2 aparece el árbol de análisis sintáctico de una frase, con los correspondientes valores calculados para los atributos.

Producción	Regla semántica asociada
$E \rightarrow E_1 + T$	$E.trad := \text{"s["} E_1.trad \text{" , " } T.trad \text{"]"}$
$E \rightarrow T$	$E.trad := T.trad$
$T \rightarrow T_1 * F$	$T.trad := \text{"p["} T_1.trad \text{" , " } F.trad \text{"]"}$
$T \rightarrow F$	$T.trad := F.trad$
$F \rightarrow \text{num}$	$F.trad := \text{num.valex}$

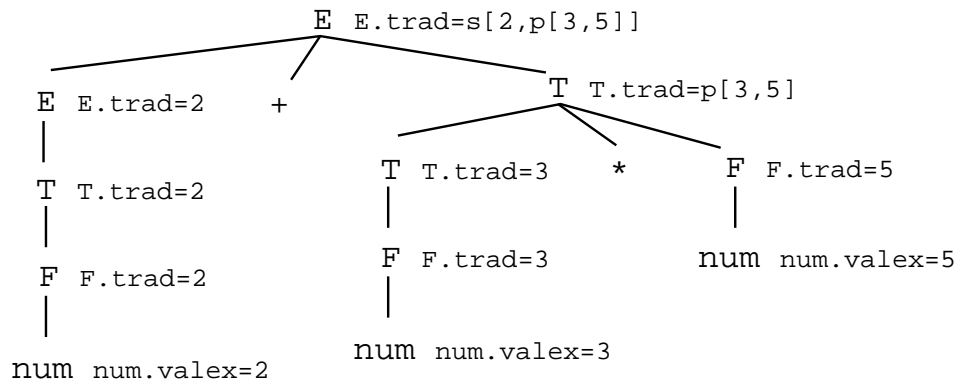


Figura 6.2: Árbol con los atributos calculados para “2+3*5”.

Notación de los subíndices semánticos: E y E_1 son el mismo símbolo (una expresión), como también lo son T y T_1 (un término), pero como los valores de los atributos de una ocurrencia y de la otra pueden ser distintos (compruébese fácilmente en el árbol) hay que distinguirlos. En estos casos los símbolos iguales se numeran en orden creciente de izquierda a derecha, pero sin numerar el símbolo en el lado izquierdo de la producción y asignando 1, 2, ... para las ocurrencias de la parte derecha. Es únicamente una estrategia para distinguir los valores de los atributos, no los símbolos sintácticos.

<

Ejemplo 6.5

Vamos a ver ahora una gramática de atributos que utiliza atributos sintetizados y heredados para procesar las declaraciones sencillas de variables de tipos real o entero en C. El atributo tipo del no terminal T es sintetizado pero el atributo inherente a la lista de identificadores L es heredado. D no tiene atributos pues no los usa. En el árbol de la figura 6.3 se indica el flujo de la información:

Producción	Regla semántica asociada
$D \rightarrow T L ;$	$L.hered := T.tipo$
$T \rightarrow \text{float}$	$T.tipo := real$
$T \rightarrow \text{int}$	$T.tipo := entero$
$L \rightarrow L_1 , \text{id}$	$L_1.hered := L.hered$
	$AsignaTipo(\text{id.lexema}, L.hered)$
$L \rightarrow \text{id}$	$AsignaTipo(\text{id.lexema}, L.hered)$

Nota: El procedimiento *AsignaTipo* se supone (aunque su implementación no importa ahora) que inserta en la tabla de símbolos del compilador la información relativa al tipo que acarrea el atributo $L.tipo$ para el identificador que se le pasa como parámetro.

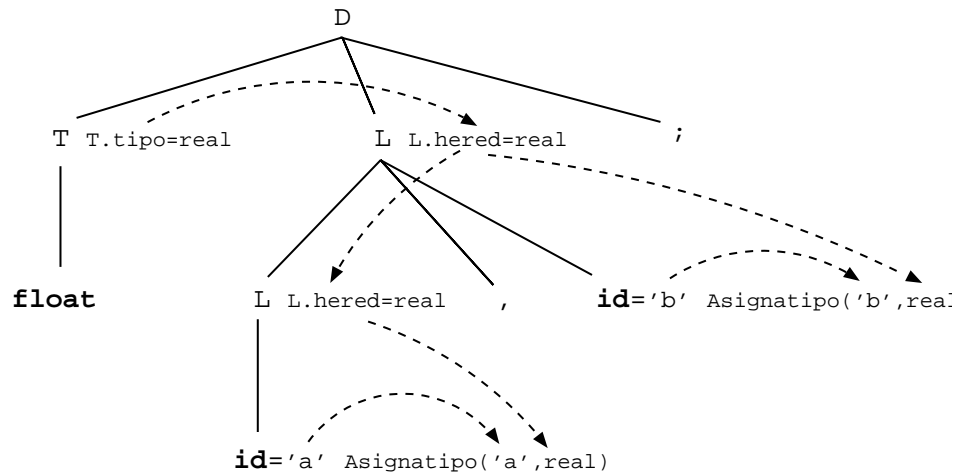


Figura 6.3: Árbol para “float a,b;”.

Obsérvese que, al tratarse de atributos heredados, las flechas que indican el flujo de la información siempre van de padres a hijos o entre hermanos. Los atributos *lexema* de los identificadores y *T.tipo* son sintetizados en esta gramática de atributos.

<

6.4 Grafos de dependencias

Hasta ahora no hemos hecho ninguna consideración acerca del orden en el que deben ejecutarse las reglas semánticas y retrasaremos esta discusión hasta el apartado siguiente, pero es evidente que los atributos no se pueden calcular en cualquier orden. Para que un atributo pueda recibir valor es necesario que todos los atributos de los que depende hayan sido previamente calculados. Para verificar que este requisito se cumple se recurre al grafo de dependencias, que además es útil para analizar como fluye la información cuando se dibuja sobre el árbol de análisis sintáctico.

El grafo de dependencias se construye para cada producción $A \rightarrow X_1X_2 \dots X_n$ y luego se unen todos para el análisis de una cadena en concreto. Se suelen dibujar sobre el árbol de análisis sintáctico de cada producción. Para ello se siguen los siguientes pasos:

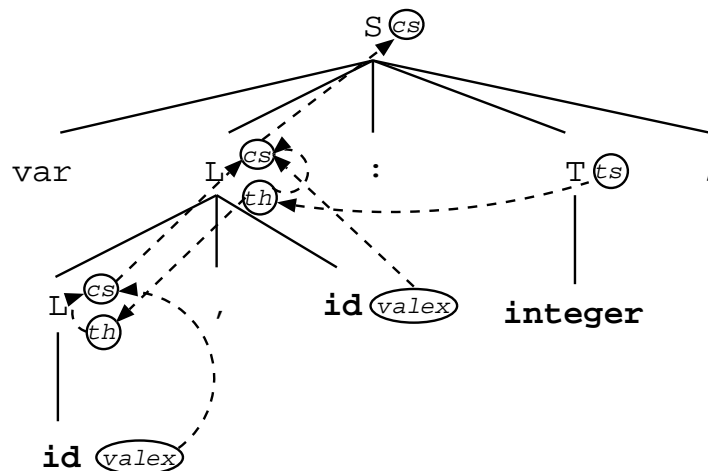
1. Se crea un nodo por cada atributo $X_i.a_j$ de cada símbolo de la producción.
2. Para cada regla semántica de la forma $X_i.a_j := f(\dots, X_k.a_l, \dots)$ asociada a esa producción se crea un arco desde cada nodo correspondiente a los distintos $X_k.a_l$ que intervienen en la regla hasta el $X_i.a_j$ expresando así la dependencia que tiene el valor que tomará $X_i.a_j$ del valor suministrado por $X_k.a_l$, repitiendo esto para cada k y cada l .

Ejemplo 6.6

Sea la siguiente gramática de atributos para traducir declaraciones de variables de tipo entero o real de Pascal a C situando cada variable en una declaración distinta. A la derecha de cada regla semántica dibujamos el grafo de dependencias que le corresponde, superpuesto al árbol de análisis sintáctico de esa producción:

Producción	Regla semántica	Grafo de dependencias
$S \rightarrow \text{var } L : T ;$	$L.th := T.ts$ $S.cs := L.cs$	
$L \rightarrow L_1 , \text{id}$	$L.cs := L_1.cs L.th$ $ \text{id.valex} \text{";"}$ $L_1.th := L.th$	
$L \rightarrow \text{id}$	$L.cs := L.th $ $\text{id.valex} \text{";"}$	
$T \rightarrow \text{real}$ $T \rightarrow \text{integer}$	$T.ts := \text{"float"}$ $T.ts := \text{"int"}$	Estas dos reglas asignan constantes a los atributos y por tanto no existen dependencias.

Vamos ahora a ver el grafo de dependencias completo superpuesto sobre el árbol de análisis sintáctico de la sentencia “var a,b:integer;”:



Obsérvese que a la hora de dibujar el grafo hemos indicado sólo la parte del nombre del atributo que viene después del punto, puesto que el símbolo al que pertenece es evidente al figurar a su lado, así ponemos *th* o *valex* para indicar *L.th* o *id.valex*.

<

A la hora de calcular los atributos, el grafo de dependencias impone una serie de restricciones sobre el orden en el cual se pueden evaluar. Si encontramos un orden topológico en los nodos del grafo, ése será el orden en que se deben evaluar los atributos². Si el recorrido de árbol sobre el cual se superpone el grafo es, como hemos asumido desde el principio, en profundidad por la izquierda, la condición que necesita el analizador semántico para poder evaluar todos sus atributos en una sola pasada es que no existan arcos desde nodos que se visiten después de los nodos a los cuales se dirigen (es decir, arcos de derecha a izquierda en el árbol).

Como podemos observar en el ejemplo anterior, esta restricción no se cumple por culpa de la asignación $L.th := T.ts$ y, por tanto, dicha gramática de atributos no se podría implementar en una sola pasada. Cuando en la producción $S \rightarrow \mathbf{var} L : T ;$ se quiere dar valor a *L.th*, todavía no se ha dado valor a *T.ts* pues todavía no se ha visitado su nodo y, por tanto, no se puede ejecutar.

6.5 Especificación de un traductor

Existen dos tipos de notaciones para especificar el diseño de un traductor:

1. **Definiciones Dirigidas por Sintaxis (DDS):** son un formalismo de alto nivel para describir traducciones, pero en el que se ocultan los detalles de la implementación. No se impone un orden en la ejecución de las reglas semánticas, sólo que se calculen los atributos de acuerdo con la producción en la que aparecen.
2. **Esquemas de Traducción (ETDS):** son una notación de bajo nivel para especificar un *traductor de una sola pasada* (traduce a la vez que analiza), explicitando, por tanto, todos los detalles de la traducción, incluido el orden en el que se deben ejecutar las reglas (ahora acciones) semánticas.

Los ETDS son *front end* o traductores de una sola pasada, mientras que las definiciones dirigidas por la sintaxis (y las gramáticas de atributos) se utilizan cuando no es posible diseñar un ETDS, y por tanto se necesita que el *front end* realice más de una pasada. Normalmente, suele ser más rápido un *front end* de una pasada que uno de varias pasadas, pero a veces el lenguaje fuente es demasiado complejo y no es posible diseñarlo; en estos casos es necesario emplear una DDS.

6.5.1 Definiciones dirigidas por sintaxis

Las DDS son una generalización del concepto de gramática de atributos que hemos visto al principio de este tema. La diferencia entre ambos conceptos es que una gramática

²Es posible que existan ciclos en el grafo, lo cual impide evaluar la gramática de atributos.

de atributos es un tipo concreto de DDS en el que las reglas semánticas se limitan exclusivamente al cálculo y manipulación de atributos, no teniendo, por tanto, efectos laterales (entendiendo por éstos, toda instrucción que no sea cálculo de atributos, como escritura de valores o modificación de variables globales). Se procurará que las reglas semánticas calculen o manipulen atributos o, en el peor de los casos, que actúen sobre estructuras globales del traductor.

En una DDS, si una producción $A \rightarrow X_1 X_2 \dots X_n$ lleva asociada una regla $b = f(a_1, \dots, a_k)$, hay dos posibilidades:

1. que b sea atributo sintetizado de A y los atributos a_1, \dots, a_k pertenezcan a cualesquiera símbolos de la producción;
2. que b sea atributo heredado de un símbolo de la parte derecha de la producción y a_1, \dots, a_k pertenezcan a cualesquiera símbolos de la producción.

Cualquiera de los ejemplos vistos en este capítulo es una DDS (pues las gramáticas de atributos que hemos visto son un caso concreto de DDS). En la columna de la izquierda se indica cada producción y en la columna de la derecha las reglas semánticas que cada una lleva asociadas.

Producción	Regla semántica
$E \rightarrow E_1 \text{ addop } T$	if addop . <i>valex</i> = "+" then $E.\text{valor} := E_1.\text{valor} + T.\text{valor}$ else $E.\text{valor} := E_1.\text{valor} - T.\text{valor}$ escribir $E.\text{valor}$

La DDS especifica lo que hay que hacer para cada producción, pero no el orden de ejecución de las reglas o acciones semánticas. Como se puede observar, cada producción puede llevar asociada una o varias acciones semánticas.

Gramáticas con atributos por la izquierda

Existe un subconjunto de las gramáticas de atributos que son las que pueden ser implementadas mediante un traductor de una sola pasada. En este tipo de traductores, toda la información manipulada por las reglas semánticas está disponible en el momento de la ejecución de cada una de ellas. De esta forma, es posible construir un programa que, con una única pasada mientras realiza el análisis sintáctico, sea capaz de llevar a cabo la traducción.

Este tipo de gramáticas de atributos recibe el nombre de *gramáticas con atributos por la izquierda* (GAI)³. En ellas, la información de los atributos siempre fluye de abajo a arriba, de arriba a abajo o de izquierda a derecha, pero *nunca de derecha a izquierda*. Este último sentido del flujo de la información semántica implicaría, en un recorrido en profundidad por la izquierda del árbol de análisis sintáctico, que para evaluar algún atributo es necesario recibir información desde zonas del árbol que todavía no han sido recorridas por el análisis (no hay valores calculados allí todavía).

Una DDS es una GAI si es una gramática de atributos en la que para cada atributo heredado que pueda aparecer $X_i.h$, $1 \leq i \leq n$, en una producción cualquiera $A \rightarrow X_1 X_2 \dots X_n$, $X_i.h$ depende sólo de:

³En la bibliografía en inglés se las denomina *L-attributed grammars*.

1. los atributos de $X_1 X_2 \dots X_{i-1}$ (los de los símbolos a su izquierda en la producción);
2. los atributos heredados de A .

Evidentemente, si todos los atributos son sintetizados, la DDS será siempre una GAI.

Existen, por tanto una serie de limitaciones en el diseño de un traductor para que tengamos una GAI. Estas limitaciones tienen que ver con el hecho estudiado anteriormente de que el orden de recorrido del grafo de dependencias sea compatible con el orden de recorrido en profundidad por la izquierda del árbol de análisis sintáctico (que es el que realizan los analizadores sintácticos LL y LR). Esto se traducía en que no existieran arcos dibujados de derecha a izquierda. En el siguiente apartado describiremos con detalle estas restricciones.

Ejemplo 6.7

Consideremos la siguiente definición dirigida por sintaxis para contar el número de aes en una cadena y vamos a construir los grafos de dependencias para las tres producciones de dicha gramática. Se muestran en la tabla siguiente, en la columna derecha.

Producción	Regla semántica	Grafo de dependencias
$L \rightarrow A L_1$	$L.s := L_1.s$ $A.h := L_1.s$	
$L \rightarrow A$	$A.h := 0$ $L.s := A.s$	
$A \rightarrow a$	$A.s := A.h + 1$	

Como se observa en el primero de los tres grafos la flecha de derecha a izquierda impide la implementación de esta DDS como un traductor en una sola pasada y no es una GAI, lo cual se ve mucho más claramente así que analizando las reglas semánticas, además de ser útil para comprender como circula la información por el árbol.

Ejemplo 6.8

Consideremos de nuevo el ejemplo de traducción de declaraciones de Pascal a C con el que ilustramos la construcción de grafos de dependencias (ejemplo 6.6). Aquella DDS no era una GAI como se demostró allí, pero la gramática puede modificarse para que sí lo sea. Se puede comprobar muy fácilmente que la siguiente DDS cumple los requisitos para ser una GAI:

Producción	Regla semántica
$S \rightarrow \text{var } D ;$	$S.cs := D.cs$
$D \rightarrow \text{id} , D_1$	$D.cs := D.ts \text{id.valex} ";" D_1.cs$ $D.ts := D_1.ts$
$D \rightarrow \text{id} : T$	$D.cs := T.ts \text{id.valex} ":"$ $D.ts := T.ts$
$T \rightarrow \text{integer}$	$T.ts := \text{"int"}$
$T \rightarrow \text{real}$	$T.ts := \text{"float"}$

<

6.5.2 Esquemas de traducción

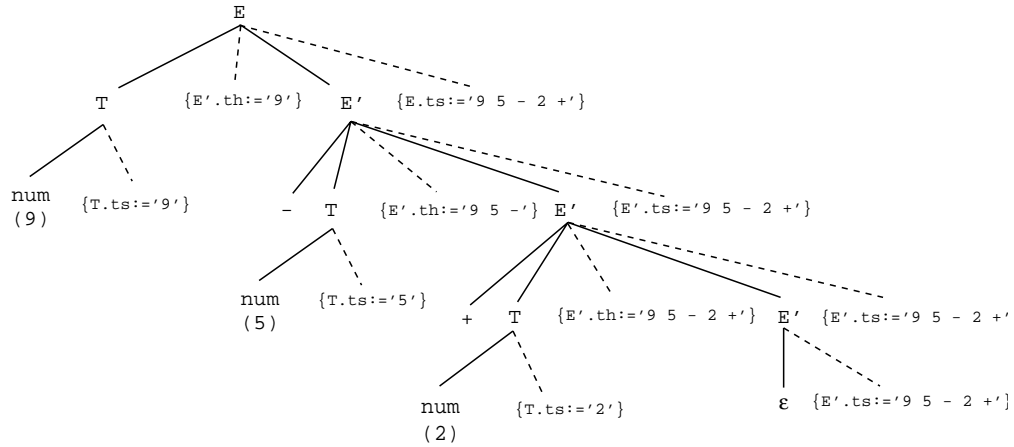
Un ETDS es una notación para construir traductores de una sola pasada. Es un analizador semántico escrito en papel, una gramática de atributos en la que se insertan acciones semánticas encerradas entre llaves (“{” y “}”) en las partes derechas de las producciones (con lo que se especifica explícitamente el orden de evaluación de las acciones semánticas) que son fragmentos de código en un lenguaje de programación o en un lenguaje algorítmico.

Ejemplo 6.9

El siguiente ETDS transforma expresiones con sumas y restas escritas en notación infija en las equivalentes en postfija. La gramática está basada en la que resulta de eliminar la recursividad por la izquierda de las producciones de las expresiones aritméticas con sumas y restas.

$$\begin{aligned}
E &\longrightarrow T \{ E'.th := T.ts \} E' \{ E.ts := E'.ts \} \\
E' &\longrightarrow \text{op } T \{ E'_1.th := E'.th || T.ts || \text{op.lexema} \} \\
&\quad E'_1 \{ E'.ts := E'_1.ts \} \\
E' &\longrightarrow \epsilon \{ E'.ts := E'.th \} \\
T &\longrightarrow \text{num} \{ T.ts := \text{num.lexema} \}
\end{aligned}$$

Para ver su funcionamiento, supondremos el ejemplo “9-5+2” y construiremos el árbol de análisis sintáctico con adornos. Cuando se plantea el dibujo de estos árboles con las acciones definidas en un ETDS, se dibujan las acciones semánticas consideradas como si fueran símbolos terminales de la gramática (aunque se les suele distinguir por el trazo de la rama).



Al recorrer este árbol por la izquierda en profundidad, llevando a cabo las acciones cuando son encontradas, se puede comprobar que se obtiene el resultado “9 5 - 2 +”, que es lo que se pretendía.

◁

Diseño de un esquema de traducción

Como se ha dicho anteriormente, para especificar traductores *de una sola pasada*, en las acciones semánticas no se deben utilizar atributos de símbolos que estén más a la derecha en la producción que dicha acción, pues aún no habrán recibido valor. Este tipo de propiedades se observará a través de unas reglas a seguir durante el diseño de un ETDS y que vamos a ver a continuación.

Pasos para el diseño de un ETDS a partir de una gramática

1. Decidir qué atributos son necesarios, en función del problema a resolver y de la información que para ello debe acarrear cada símbolo de la gramática. Conviene hacer un árbol de análisis de una cadena representativa para analizar el comportamiento de la información.
2. Añadir a la gramática las acciones semánticas que calculen los valores de los atributos.
3. Estudiar de qué tipo es cada atributo (heredado o sintetizado).
4. ¿Se cumplen las restricciones de diseño para que el ETDS se pueda implementar como un traductor de una sola pasada? Si todos los atributos son sintetizados seguro que sí y se pueden colocar las acciones semánticas al final del lado derecho de las producciones, pero si hay también heredados, hay que tenerlas en cuenta (ver el grafo de dependencias).

Ejemplo 6.10

Si sólo hay atributos sintetizados se pueden colocar las acciones al final del lado derecho de la posición asociada. Como la siguiente DDS:

Producción	Regla semántica asociada
$E \rightarrow E_1 + T$	$E.trad := \text{"s["} E_1.trad \text{"}, " T.trad \text{"}]"}\}$
$E \rightarrow T$	$E.trad := T.trad$
$T \rightarrow T_1 * F$	$T.trad := \text{"p["} T_1.trad \text{"}, " F.trad \text{"}]"}\}$
$T \rightarrow F$	$T.trad := F.trad$
$F \rightarrow \mathbf{num}$	$F.trad := \mathbf{num.valex}$

que en la notación de ETDS se puede escribir como:

$$\begin{array}{llll}
E & \longrightarrow & E_1 + T & \{ E.trad := \text{"s["} || E_1.trad || \text{"}, " || T.trad || \text{"}]"} \} \\
E & \longrightarrow & T & \{ E.trad = T.trad \} \\
T & \longrightarrow & T_1 * F & \{ T.trad := \text{"p["} || T_1.trad || \text{"}, " || F.trad || \text{"}]"} \} \\
T & \longrightarrow & F & \{ T.trad = F.trad \} \\
F & \longrightarrow & \mathbf{num} & \{ F.trad := \mathbf{num.valex} \}
\end{array}$$

◀

Surge ahora la siguiente cuestión: no siempre las acciones semánticas se han de situar al final de las reglas porque no siempre todos los atributos son sintetizados. Cuando hay herencia vimos que hay que tener en cuenta unas normas para que una DDS se pueda implementar como un compilador en una sola pasada. Pero un ETDS realmente ya es el diseño de un compilador por lo tanto para que una DDS se pueda convertir en un ETDS deben darse dos condiciones:

1. En el caso de que la DDS no sea una gramática de atributos, el orden de ejecución de las acciones semánticas tiene que estar claramente definido, y debe ser compatible con el recorrido del árbol sintáctico que se hace al ejecutar las acciones del ETDS, ya que el ETDS solamente realiza una pasada.
2. Sea o no una gramática de atributos, debe cumplirse la misma restricción de las GAI, es decir, que no exista herencia desde la derecha en el árbol hacia la izquierda. Si no se cumple esta condición no es posible evaluar la DDS en una única pasada, y por tanto no se puede convertir en ETDS.

Ejemplo 6.11

La DDS del ejemplo 6.6 es una gramática de atributos, pero no es una GAI; la siguiente DDS es otro ejemplo de una DDS que no puede transformarse en ETDS sin información adicional:

Producción	Regla semántica asociada
$E \rightarrow E + T$	Escribir("+")
$E \rightarrow T$	
$T \rightarrow T * F$	Escribir("*")
$T \rightarrow F$	
$F \rightarrow \mathbf{num}$	Escribir(num.valex)

Tal como está planteada, y sin más información, esta DDS puede servir para traducir expresiones aritméticas en notación infija a notación prefija, pero también para traducirlas a notación postfija (e incluso para dejarlas en notación infija). Por tanto, esta DDS no tiene información suficiente para ser evaluada, lo cual impide además transformarla en ETDS. En este caso habría que rediseñar la DDS para explicitar el tipo de traducción que se desea (transformándola en una gramática de atributos, por ejemplo) y, una vez hecho esto, estudiar si se puede o no transformar en ETDS.

<

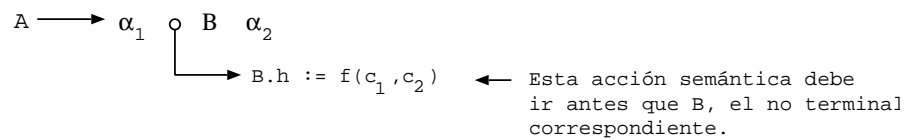
Vamos a ver las precauciones que hay que tener en cuenta al diseñar un ETDS para que estas condiciones se verifiquen.

Restricciones en el diseño de un ETDS

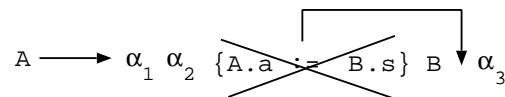
1. *Si sólo existen atributos sintetizados:* hay que poner las acciones semánticas después de todos los símbolos implicados. Su valor sólo se debe calcular tras haber calculado todos los atributos a los que hace referencia. Lo más sencillo es situar la acción o acciones que los calcula al final del lado derecho de la producción.

2. *Si existen atributos heredados:*

- (a) Un atributo heredado $X_i.h$ debe calcularse antes de que aparezca el símbolo X_i .



- (b) Un atributo sintetizado $X_i.s$ no debe usarse antes de que aparezca el símbolo X_i .



- (c) Un atributo sintetizado $A.s$ sólo puede calcularse después de que hayan tomado valor todos los atributos que intervienen en su cálculo. Mejor poner este cálculo al final de la producción.

3. *Si existen acciones con efectos laterales:* deben situarse en el punto exacto de la parte derecha de la regla en el que deberían evaluarse, y además habría que verificar que no utilizan atributos de símbolos situados a la derecha de dicho punto.

Ejemplo 6.12

A partir de la gramática

$$\begin{aligned} E &\longrightarrow E + T \mid T \\ T &\longrightarrow T * F \mid F \\ F &\longrightarrow \mathbf{num} \mid (E) \end{aligned}$$

vamos a ver cómo se ejecutan los pasos indicados anteriormente para diseñar un ETDS que traduzca a notación prefija expresiones como “2+3*5”.

1. ¿Qué atributos son necesarios?
 - Para los terminales, sólo necesitamos saber cuáles son, y en particular para **num** nos interesa conocer qué número ha aparecido en la expresión, y eso nos lo devolverá el AL en **num.lexema**.
 - Para los no terminales necesitaremos un atributo para almacenar la traducción parcial asociada a la subexpresión correspondiente: $F.trad$, $T.trad$, $E.trad$.

Así ya están estudiados todos los símbolos para los que son necesarios atributos.

2. Añadir las acciones semánticas:
 - En una producción sólo se pueden utilizar atributos de los símbolos que aparecen en ella.
 - Para $F \rightarrow \mathbf{num}$, al analizar **num** se conoce su lexema (es decir, la cadena encontrada en la entrada que contiene su valor numérico), por lo que simplemente tendremos que transmitir ese lexema al factor:

$$\{ F.trad := \mathbf{num.lexema} \}$$

- Una vez se haya analizado una expresión, el atributo $E.trad$ contendrá la traducción de dicha expresión, por lo tanto en la producción $F \rightarrow (E)$ lo único que hay que hacer es transmitírsela a $F.trad$; entonces:

$$\{ F.trad := E.trad \}$$

- Una vez analizado el factor, en $T \rightarrow F$ debe transmitir su traducción al término, por tanto sería:

$$\{ T.trad := F.trad \}$$

- Lo mismo sucede en $E \rightarrow T$, luego sería:

$$\{ E.trad := T.trad \}$$

- El significado de la producción $T \rightarrow T_1 * F$ lleva a diseñar una acción semántica en la que el atributo del término T sea la traducción a prefija de la multiplicación del término T_1 y del factor F . Por tanto, la acción será:

$$\{ T.trad := \text{"p["} || T_1.trad || \text{"}, " || F.trad || \text{"}]" \}$$

- Y lo equivalente para $E \rightarrow E_1 + T$:

$$\{ E.trad := \text{"s["} || E_1.trad || \text{"}, " || T.trad || \text{"}]" \}$$

3. Ahora hay que estudiar si los atributos son heredados o sintetizados, en función de los otros atributos a partir de los cuales se calculan.

$F.trad$: Se le da valor en las dos siguientes producciones:

$$F \rightarrow \text{num} \{ F.trad := \text{num.lexema} \}$$

$F.trad$ es sintetizado en esta producción, por tanto deberá aparecer siempre como sintetizado.

$$F \rightarrow (E) \{ F.trad := E.trad \}$$

Igualmente, en esta producción $F.trad$ también es sintetizado (como debería ser).

$T.trad$: Se le da valor en las dos siguientes producciones:

$$T \rightarrow T_1 * F \{ T.trad := \text{"p["} || T_1.trad || \text{"}, " || F.trad || \text{"}]" \}$$

Como $T.trad$ depende de $T_1.trad$ y $F.trad$, se puede decir que $T.trad$ es sintetizado.

$$T \rightarrow F \{ T.trad := F.trad \}$$

De igual manera que en la producción anterior, $T.trad$ es sintetizado.

$E.trad$: Los razonamientos para $T.trad$ son equivalentes para el caso del atributo de E .

Luego todos los atributos son sintetizados en este caso.

4. Ver si se cumplen las condiciones que tienen que cumplir las acciones semánticas para que el esquema de traducción lo podamos implementar sin problemas como traductor en una sola pasada. Pero en este caso es sencillo: como todos los atributos son sintetizados, basta con poner todas las acciones semánticas en el extremo derecho de las producciones, para que se cumplan todas las condiciones sin problemas.

Una vez llevados a cabo estos pasos, que nos aseguran la correcta construcción del esquema de traducción, podemos utilizarlo para evaluar la cadena propuesta. Lo haremos mediante la aplicación de las acciones semánticas en el recorrido del árbol, como muestra la figura 6.4.

Cada vez que pasamos por una acción semántica la aplicamos (este análisis se hará empezando por las hojas). De esta forma conseguimos traducir la expresión, quedando la traducción en $E.trad$ al llegar a la raíz.

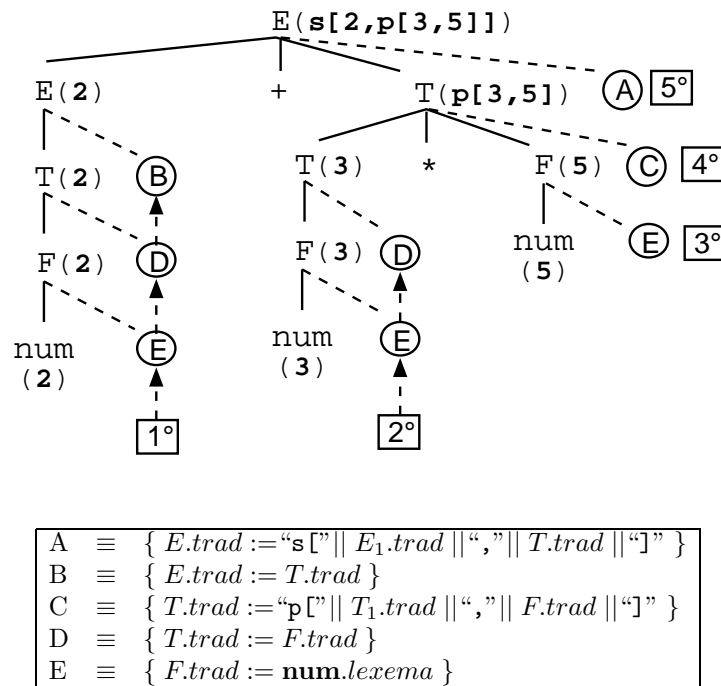


Figura 6.4: Árbol sintáctico de “2+3*5” con las acciones semánticas (indicadas como referencias a la tabla que aparece debajo del árbol). Se indican las traducciones parciales en cada nodo y el orden de ejecución de las acciones.

6.6 Referencias bibliográficas

REFERENCIAS

[Louden, 1997]
 [Aho, Sethi y Ullman, 1990]
 [Bennett, 1990]
 [Fischer y LeBlanc, 1991]

EPÍGRAFES

6.1, 6.2.1, 6.2.2 y 6.2.6
 5.1 y 5.4
 3.3
 7.1.1, 7.1.3 y 14.1

6.7 Ejercicios

Ejercicio 6.1

Diseñar un esquema de traducción para asignar tipo a una lista de variables declaradas en Pascal, siendo la sintaxis la siguiente.

<i>Variables</i>	\longrightarrow	var <i>Declaración</i>
<i>Declaración</i>	\longrightarrow	identificador <i>Lista</i>
<i>Lista</i>	\longrightarrow	coma identificador <i>Lista</i>
<i>Lista</i>	\longrightarrow	dosptos <i>Tipo</i> ptocomas
<i>Tipo</i>	\longrightarrow	integer real

Ejercicio 6.2

Diseñar un esquema de traducción para asignar tipo a una lista de variables declaradas en C, siendo la sintaxis la siguiente.

<i>Declaración</i>	\longrightarrow	<i>Tipo</i> identificador <i>Lista</i> ptocomas
<i>Lista</i>	\longrightarrow	coma identificador <i>Lista</i>
<i>Lista</i>	\longrightarrow	ϵ
<i>Tipo</i>	\longrightarrow	int float

Ejercicio 6.3

Diseñar un ETDS para traducir declaraciones de arrays de tipos simples de C a Pascal, como en los siguientes ejemplos:

<code>int c[10];</code>	<code>var c: array [0..9] of integer;</code>
<code>float d[5][6];</code>	<code>var d: array [0..4,0..5] of real;</code>
<code>char e;</code>	<code>var e: char;</code>
<code>char f[3][5][7];</code>	<code>var f: array [0..2,0..4,0..6] of char;</code>

El lenguaje fuente está generado por la siguiente gramática:

<i>S</i>	\longrightarrow	<i>T V</i> ;
<i>V</i>	\longrightarrow	id <i>V'</i>
<i>V'</i>	\longrightarrow	[nint] <i>V'</i>
<i>V'</i>	\longrightarrow	ϵ
<i>T</i>	\longrightarrow	int float char

Ejercicio 6.4

Diseñar un ETDS para traducir secuencias de declaraciones de variables en C a otro lenguaje de alto nivel, como se especifica en los siguientes ejemplos de traducciones:

```

int a,b[10],c[5][4];      variables
                           enteras : a;
                           tabla[10] : enteras : b;
                           tabla[5] : tabla[4] : enteras : c
                           fvar

int **a[10][5];           variables
                           tabla[10] : tabla[5] : puntero :
                           puntero : enteras : a
                           fvar

float **a;                variables
                           puntero : puntero : reales : a
                           fvar

```

El lenguaje fuente está generado por la siguiente gramática:

$$\begin{aligned}
S &\longrightarrow T D ; \\
D &\longrightarrow D , V \\
D &\longrightarrow V \\
V &\longrightarrow E \\
V &\longrightarrow V [\text{ nint }] \\
E &\longrightarrow * E \\
E &\longrightarrow \text{ id } \\
T &\longrightarrow \text{ int } \mid \text{ float } \mid \text{ char }
\end{aligned}$$

Ejercicio 6.5

Diseñar e implementar un ETDS para la traducción de los siguientes ejemplos:

```

int a,b;                  a,b : integer;

struct s {                d,e : registro [
    int a,b;              a(s),b(s):integer;
    float c;              c(s):real;
} d,e;                    ];

struct s1 {               h : registro [
    int a,b,b2;           a(s1),b(s1),b2(s1):integer;
    struct s2 {           e(s1),f(s1):registro [
        float c,d;        c(s2),d(s2):real;
    } e,f;                ];
} h;                      ];

```

El lenguaje fuente está generado por la siguiente gramática:

$$\begin{aligned}
 S &\longrightarrow D \\
 D &\longrightarrow T \text{ id } L ; \\
 T &\longrightarrow \text{int} \mid \text{float} \mid \text{char} \\
 T &\longrightarrow \text{struct id } \{ D M \} \\
 M &\longrightarrow D M \\
 M &\longrightarrow \epsilon \\
 L &\longrightarrow , \text{ id } L \\
 L &\longrightarrow \epsilon
 \end{aligned}$$

Ejercicio 6.6

¿Por qué no se puede escribir un ETDS directamente a partir de la siguiente DDS (sin modificarla, solamente introduciendo las acciones semánticas en la parte derecha de las reglas)?

PRODUCCIÓN	REGLA SEMÁNTICA
$A \longrightarrow B C$	$B.h := C.s$
$B \longrightarrow B_1 \text{ b}$	if $B.h \geq 0$ then $B_1.h := B.h - 1;$ $mark(\text{b}.dir);$ endif
$B \longrightarrow \epsilon$	
$C \longrightarrow \text{c } C_1$	$C.s := C_1.s + 1$
$C \longrightarrow \epsilon$	$C.s := 0$

Ejercicio 6.7

Transformar la siguiente DDS en un ETDS, situando las acciones semánticas en los lugares apropiados en las partes derechas de las reglas.

PRODUCCIÓN	REGLA SEMÁNTICA
$S \longrightarrow \{ B \}$	$B.bh := \text{NULL}; S.cs := B.cs;$ $B.ch := \text{NULL};$
$B \longrightarrow I; B_1$	$B_1.bh := B.bh; B.cs := I.cs \parallel B_1.cs;$ $I.bh := B.bh; B_1.ch := B.ch;$ $I.ch := B.ch;$
$B \longrightarrow \epsilon$	$B.cs := \text{NULL};$
$I \longrightarrow \{ B \}$	$B.bh := I.bh; I.cs := B.cs;$ $B.ch := I.ch;$
$I \longrightarrow \text{while } (E) I_1$	$I_1.bh := e1; e1 := \text{etiquueva}();$ $I_1.ch := e2; e2 := \text{etiquueva}();$ $I.cs := \text{gen}(e2 \text{ ":"}) \parallel E.cs$ $\parallel \text{gen}(\text{"jumpz " } e1) \parallel I_1.cs$ $\parallel \text{gen}(e1 \text{ ":"});$
$I \longrightarrow \text{print } E$	$I.cs := \text{gen}(\text{"print "}) \parallel E.cs;$
$I \longrightarrow \text{break}$	if $I.bh \neq \text{NULL}$ then $I.cs := \text{gen}(\text{"jump " } I.bh)$ else $\text{error}()$ endif
$I \longrightarrow \text{continue}$	if $I.ch \neq \text{NULL}$ then $I.cs := \text{gen}(\text{"jump " } I.ch)$ else $\text{error}()$ endif
$E \longrightarrow \mathbf{a}$	$E.cs := \text{gen}(\text{" a "});$

Ejercicio 6.8

Explicar si la siguiente DDS se puede convertir en un ETDS (situando las acciones semánticas donde corresponda sin modificar la gramática en absoluto).

PRODUCCIÓN	REGLAS SEMÁNTICAS
$D \rightarrow T L$	$L.tip := T.tip$
$T \rightarrow \text{int}$	$T.tip := \text{integer}$
$T \rightarrow \text{real}$	$T.tip := \text{real}$
$L \rightarrow L_1 , I$	$L_1.tip := L.tip ; I.tip := L.tip$
$L \rightarrow I$	$I.tip := L.tip$
$I \rightarrow I_1 [\text{num}]$	$I_1.tip := \text{array}(\text{num.val}, I.tip)$
$I \rightarrow \text{id}$	$\text{introtip}(\text{id.vallex}, I.tip)$

Capítulo 7

Implementación de traductores

7.1 Introducción

Como se ha explicado anteriormente, los compiladores suelen realizar en la primera pasada las fases de análisis (léxico, sintáctico y semántico) y la generación de la representación intermedia o código intermedio correspondiente al programa fuente, lo que normalmente constituye el *front end* del compilador, y que se suele especificar utilizando un ETDS.

En este capítulo vamos a explicar las técnicas para construir traductores de una sola pasada (traductores dirigidos por la sintaxis, basados en ETDS), que se pueden dividir en dos grupos, según la clase de análisis sintáctico que se realice:

Traductores descendentes: son traductores basados en analizadores descendentes predictivos, y por tanto requieren que la gramática sea $LL(1)$ (aunque existen traductores basados en gramáticas $LL(k)$).

En general, implementar un ETDS basado en una gramática $LL(1)$ suele conllevar la utilización de atributos heredados para las expresiones aritméticas, que suele ser la parte más sencilla del proceso de traducción. Sin embargo, para lenguajes sencillos puede ser una alternativa muy interesante, ya que, como se estudia a continuación, la implementación de ETDS sobre ASDR (los traductores descendentes recursivos) suele ser muy sencilla y rápida, una vez construido el analizador sintáctico. Aunque es posible implementar traductores a partir de analizadores descendentes dirigidos por tabla, suele ser mucho más complicado que utilizar un ASDR, por lo que casi nunca se utilizan este tipo de analizadores.

Traductores ascendentes: son traductores basados en analizadores ascendentes de la familia LR (SLR, LALR(1), LR(1), ...). Como veremos más adelante, es relativamente sencillo transformar un analizador SLR en un traductor, aunque es un poco más complicado implementar los atributos heredados.

Aunque los analizadores SLR son sencillos de construir, el conjunto de lenguajes que pueden analizar está muy limitado, por lo que se suelen utilizar analizadores LALR(1), que permiten analizar un conjunto más amplio de lenguajes con un coste espacial mucho menor que el de un analizador LR(1) y poco mayor que el de un analizador SLR.

Desde hace bastantes años se ha ido desarrollando muchas herramientas para facilitar la construcción de traductores de una o más pasadas, normalmente utilizando analizadores ascendentes. La más conocida de todas estas herramientas es YACC, que genera traductores basados en analizadores LALR(1) a partir de una notación muy similar a la de los ETDS. La popularidad de esta herramienta se debe a su disponibilidad en prácticamente todos los sistemas UNIX¹, aunque existen también versiones para otros sistemas operativos.

7.2 Traductores descendentes recursivos

La implementación de un esquema de traducción se hace incorporando las acciones semánticas a un analizador sintáctico descendente recursivo (ASDR) y eso exige que la gramática sea LL(1). Así pues, antes de implementar el esquema de traducción, debemos asegurarnos de que la gramática es LL(1).

Aunque lo normal sería partir de la gramática LL(1) para diseñar el ETDS, si tenemos un ETDS basado en una gramática que sea recursiva por la izquierda o que tenga factores comunes por la izquierda es posible aplicar algunas reglas para eliminar estas características (que impiden que la gramática sea LL(1)) manteniendo a la vez las acciones semánticas especificadas en el ETDS original. Aún así, una vez eliminadas estas características es posible que la gramática resultante no sea LL(1), por lo que es necesario comprobarlo, como ya vimos en el capítulo 4.

7.2.1 Eliminación de la recursividad izquierda

En el capítulo 4 se estudia el problema de la eliminación de la recursividad izquierda en una producción, utilizando la siguiente regla (simplificada):

Dos producciones recursivas, $A \rightarrow A Y \mid X$ se transforman en tres producciones que ya no presentan recursividad por la izquierda:

$$\begin{aligned} A &\longrightarrow X A' \\ A' &\longrightarrow Y A' \mid \epsilon \end{aligned}$$

Ahora se trata de ampliar esta regla para incluir el tratamiento que hay que dar a las acciones semánticas que aparezcan en esas producciones. Dadas las siguientes producciones en un ETDS,

$$\begin{aligned} A &\longrightarrow A_1 Y \{ A.a := g(A_1.a, Y.y) \} \\ A &\longrightarrow X \{ A.a := f(X.x) \} \end{aligned}$$

habría que transformarlas en:

$$\begin{aligned} A &\longrightarrow X \{ A'.h := f(X.x) \} A' \{ A.a := A'.s \} \\ A' &\longrightarrow Y \{ A'_1.h := g(A'.h, Y.y) \} A'_1 \{ A'.s := A'_1.s \} \\ A' &\longrightarrow \epsilon \{ A'.s := A'.h \} \end{aligned}$$

¹En Linux y otros sistemas UNIX existe una versión más avanzada desarrollada bajo licencia GNU llamada **bison**.

El símbolo A' es introducido de la misma forma que se hizo para el caso de las gramáticas. Este nuevo símbolo auxiliar tiene siempre, por definición, dos atributos: “ s ”, que es sintetizado y “ h ” que es heredado². Las funciones f y g son la forma de representar cuáles son las relaciones entre los atributos. Es decir:

- En la producción recursiva aparece $A.a := g(A_1.a, Y.y)$, lo que indica que, en ese caso, el atributo $A.a$ se calcula a partir de los valores de los atributos $A_1.a$ e $Y.y$ mediante la función g .
- En la parte no recursiva, $A.a := f(X.x)$ indica que, en este caso, el valor de $A.a$ se calcula a partir de $X.x$ por medio de esa función f . Estas funciones f y g en las producciones que resultan de aplicar este procedimiento se aplican a otros atributos y sirven para calcular otros distintos, como se deriva de que $A'.h := f(X.x)$ y $A'_1.h := g(A'.h, Y.y)$.

Se puede comprobar muy fácilmente que ambos esquemas de traducción (el original y el modificado) llevan a cabo la misma tarea. Para ello basta con hacer el análisis sintáctico de una misma cadena con ambas gramáticas y evaluar los atributos con ambos ETDS hasta ver qué traducción se obtiene en el atributo del nodo raíz ($A.a$ en este caso), comprobando que es idéntica.

Ejemplo 7.1

Sea la primera producción del ETDS del ejemplo 6.10, con sus acciones semánticas:

$$\begin{aligned} E &\longrightarrow E_1 + T \{ E.trad := \text{“s[”} || E_1.trad || \text{“,”} || T.trad || \text{”]} \} \\ E &\longrightarrow T \{ E.trad := T.trad \} \end{aligned}$$

Antes de transformar estas producciones debemos identificar los siguientes elementos:

A	Y	X	$f(X.x)$	$g(A_1.a, Y.y)$
E	$+ T$	T	$T.trad$	$\text{“s[”} E_1.trad \text{“,”} T.trad \text{”]} \}$

por tanto la aplicación de la regla transforma las dos producciones anteriores en las tres siguientes:

$$\begin{aligned} E &\longrightarrow T \{ E'.h := T.trad \} E' \{ E.trad := E'.s \} \\ E' &\longrightarrow + T \{ E'_1.h := \text{“s[”} || E'.h || \text{“,”} || T.trad || \text{”]} \} E' \{ E'.s := E'_1.s \} \\ E' &\longrightarrow \epsilon \{ E'.s := E'.h \} \end{aligned}$$

Como se observa, los atributos de los símbolos originales mantienen sus nombres, mientras que los del símbolo auxiliar E' introducido por la regla son siempre “ s ” para el sintetizado y “ h ” para el heredado.

◀

²En realidad, habría que añadir una pareja de atributos (uno heredado y otro sintetizado) a ese nuevo símbolo por cada atributo distinto que manipule el ETDS original. Es decir, si A tiene un atributo de tipo y otro de código traducido, serán necesarios dos pares de atributos para A' .

7.2.2 Factorización izquierda

¿Qué sucede con las acciones semánticas cuando el ETDS está construido sobre producciones que tienen factores comunes por la izquierda? Volvemos a estar ante una situación que no permite la implementación descendente del ETDS. Para poder hacerlo hay que factorizar dichas producciones teniendo cuidado con lo que pasa con la evaluación de atributos. La factorización introducirá un nuevo símbolo (véase la factorización izquierda en el capítulo 4) y ese nuevo símbolo aportará una traducción o evaluación parcial que luego habrá que encajar con el resto de la traducción hecha por las partes diferentes de las producciones. Básicamente podríamos considerar que las acciones semánticas también quedan factorizadas.

Ejemplo 7.2

Sea el siguiente problema de traducción basado en el típico problema de las sentencias *if*:

```
if ( e ) s else s endif  →  si e entonces s si_no s
if ( e ) s endif         →  si e entonces s
```

El ETDS para hacer esta traducción resulta:

$$\begin{aligned} S &\longrightarrow \text{if } (E) S_1 \text{ else } S_2 \text{ endif } \{ S.trad := \text{"si"} || \\ &\quad E.trad || \text{"entonces"} || S_1.trad || \text{"si_no"} || S_2.trad \} \\ S &\longrightarrow \text{if } (E) S_1 \text{ endif } \{ S.trad := \text{"si"} || E.trad || \\ &\quad \text{"entonces"} || S_1.trad \} \end{aligned}$$

Y la modificación resultante de factorizar es:

$$\begin{aligned} S &\longrightarrow \text{if } (E) S_1 S' \{ S.trad := \text{"si"} || \\ &\quad E.trad || \text{"entonces"} || S_1.trad || S'.trad \} \\ S' &\longrightarrow \text{endif } \{ S'.trad := \text{""} \} \\ S' &\longrightarrow \text{else } S \text{ endif } \{ S'.trad := \text{"sino"} || S.trad \} \end{aligned}$$

Se puede comprobar fácilmente que las traducciones parciales de S' luego encajan en la traducción de S de forma que la traducción queda igual que antes de factorizar.

◁

7.2.3 Implementación de un traductor descendente recursivo

Para poder construirlo hay que asegurarse de que la gramática cumpla la condición LL(1). La implementación de un ETDS parte de un ASDR (analizador sintáctico descendente recursivo), en el que se incluyen las acciones semánticas casi literalmente y se realizan algunos cambios en las cabeceras de las funciones que analizan los no terminales y en sus partes declarativas. No obstante, siguiendo las siguientes reglas se puede implementar un ETDS partiendo de cero.

1. Aspecto de las funciones asociadas a los no terminales:
Para cada no terminal A , se construye una función que tenga un parámetro por cada atributo heredado de A , y que devolverá como resultado el valor del atributo sintetizado de A .
 - Si hubiera varios atributos sintetizados de A , habría que cambiar la función para que devuelva los valores de todos ellos. Si A no tuviera atributos sintetizados, la función no devolvería ningún valor.
 - Dentro de la función se declara *una variable local* por cada atributo que aparezca en los símbolos de las partes derechas de las distintas producciones alternativas de A , salvo los que coincidan con atributos heredados de A (que son parámetros de la función).
2. Elección de la producción a aplicar:
En el código de la función para A se decide la alternativa a aplicar igual que en el análisis sintáctico (según los terminales que hay en los conjuntos de predicción de cada alternativa).
3. Análisis sintáctico:
Se procesa de izquierda a derecha cada elemento de cada alternativa (que puede ser un terminal, un no terminal o una acción semántica) de la siguiente forma:
 - Si es un terminal (*token*) x : almacenar el valor del atributo del terminal en la variable local correspondiente para ese atributo (*sólo* si existe un atributo asociado a ese terminal) y después llamar a la función “emparejar” para ese terminal: “`emparejar(x)`”.
 - Si es un no terminal B : incluir la sentencia $c = B(b_1, b_2, \dots, b_k)$, donde b_1, b_2, \dots, b_k son las variables que corresponden a los atributos heredados del símbolo B y “ c ” es la variable que corresponde al atributo sintetizado de B .
 - Si es una acción semántica: copiarla en ese mismo sitio, adaptándola si es necesario para que utilice los atributos correctos.
4. Retorno de valores:
La última sentencia de una función de análisis debe ser devolver el valor sintetizado del símbolo no terminal que analiza esa función, aunque es posible hacerlo al terminar de analizar cada producción en lugar de al final de la función.
5. Ejecución:
El traductor se pone en marcha con una llamada a la función del símbolo inicial.

Ejemplo 7.3

Un ETDS para traducir a notación prefija expresiones aritméticas con sumas y

multiplicaciones (como el que aparece en el ejemplo 6.10) es el siguiente:

$$\begin{array}{lll}
E & \longrightarrow & E_1 + T \quad \{ E.trad := \text{"s["} \parallel E_1.trad \parallel \text{"}, " \parallel T.trad \parallel \text{"}]"} \} \\
E & \longrightarrow & T \quad \{ E.trad = T.trad \} \\
T & \longrightarrow & T_1 * F \quad \{ T.trad := \text{"p["} \parallel T_1.trad \parallel \text{"}, " \parallel F.trad \parallel \text{"}]"} \} \\
T & \longrightarrow & F \quad \{ T.trad = F.trad \} \\
F & \longrightarrow & \mathbf{num} \quad \{ F.trad := \mathbf{num.valex} \} \\
F & \longrightarrow & (E) \quad \{ F.trad := E.trad \}
\end{array}$$

Como las producciones de E y T son recursivas por la izquierda, hay que deshacer esa recursividad aplicando la metodología descrita en el apartado 7.2.1, quedando como sigue el ETDS modificado:

$$\begin{array}{lll}
E & \longrightarrow & T \{ E'.h := T.trad \} E' \{ E.trad := E'.s \} \\
E' & \longrightarrow & + T \{ E'_1.h := \text{"s["} \parallel E'.h \parallel \text{"}, " \parallel T.trad \parallel \text{"}]"} \} E'_1 \{ E'.s := E'_1.s \} \\
E' & \longrightarrow & \epsilon \{ E'.s := E'.h \} \\
T & \longrightarrow & F \{ T'.h := F.trad \} T' \{ T.trad := T'.s \} \\
T' & \longrightarrow & * F \{ T'_1.h := \text{"p["} \parallel T'.h \parallel \text{"}, " \parallel F.trad \parallel \text{"}]"} \} T'_1 \{ T'.s := T'_1.s \} \\
T' & \longrightarrow & \epsilon \{ T'.s := T'.h \} \\
F & \longrightarrow & \mathbf{num} \{ F.trad := \mathbf{num.lexema} \} \\
F & \longrightarrow & (E) \{ F.trad := E.trad \}
\end{array}$$

A continuación se procede a la implementación en C de los subprogramas del ETDS, siguiendo los pasos descritos anteriormente:

- E : tiene un único atributo sintetizado (una cadena de caracteres, como todos en este ejemplo) y ninguno heredado, luego será una función sin parámetros formales.

```

char *E()
{
    char *E_trad,*T_trad,*Ep_h,*Ep_s;

    T_trad=T();
    Ep_h=T_trad; Ep_s=Ep(Ep_h); E_trad=Ep_s;
    return(E_trad);  (* devuelve el atributo sintetizado de E *)
}

```

- E' : como todos los símbolos auxiliares introducidos en la eliminación de recursividad tiene un atributo sintetizado y uno heredado, luego será una función con un parámetro formal para el atributo heredado. Como hay dos alternativas hay que distinguir (como se hace en el ASDR) qué alternativa tomar en función del *token* que lleve el símbolo de preanálisis.

```

char *Ep ( char *Ep_h )
{
    char *T_trad,*Ep1_h,*Ep1_s,*Ep_s;

    if ( preanalisis==MAS )
    { /* E' -> + T E' */
        emparejar(MAS); T_trad=T();
        Ep1_h=concat("s[",Ep_h,"",T_trad,"");
        Ep1_s=Ep(Ep1_h); Ep_s=Ep1_s;
    }
}

```

```

    }
    else if ( preanalisis==RPAR || preanalisis==FDF )
        Ep_s=Ep_h; /*E' -> epsilon */
    else ErrorSintactico(lexema,MAS,RPAR,FDF);

    return(Ep_s);
}

```

- T : tiene un único atributo sintetizado y sin heredados, como E .

```

char *T()
{
    char *T_trad,*F_trad,*Tp_h,*Tp_s;

    F_trad=F();
    Tp_h=F_trad; Tp_s=Tp(Tp_h); T_trad=Tp_s();
    return(T_trad); (* devuelve el atributo sintetizado de T *)
}

```

- T' : tiene el mismo comportamiento que E' .

```

char *Tp ( char *Tp_h )
{
    char *F_trad,*Tp1_h,*Tp1_s,*Tp_s;

    if ( preanalisis==POR )
        { /* T' -> * F T' */
            emparejar(POR); F_trad=F();
            Tp1_h=concat("p[",Tp_h,",",F_trad,"");
            Tp1_s=Tp(Tp1_h); Tp_s=Tp1_s;
        }
    else if ( preanalisis==MAS || preanalisis==RPAR
              || preanalisis==FDF )
        Tp_s=Tp_h; /*T' -> epsilon */
    else ErrorSintactico(lexema,POR,MAS,RPAR,FDF);

    return(Tp_s);
}

```

- F : tiene un único atributo sintetizado y sin heredados, luego también será una función sin parámetros formales. Nótese cómo el lexema del *token* **num** se copia antes de la llamada a “Emparejar”, ya que ésta avanza al siguiente *token* de la entrada.

```

char *F()
{
    char *Num_trad,*E_trad,*F_trad;

    switch ( preanalisis ) {
        case NUM : Num_trad=strdup(lexema); Emparejar(NUM);
                    F_trad=Num_trad; break;
        case LPAR : Emparejar(LPAR); E_trad=E(); Emparejar(RPAR);
                    F_trad=E_trad; break;
        default : ErrorSintactico(lexema,NUM,LPAR);
    }
    return(F_trad);
}

```

Nota de implementación: Es frecuente que un traductor tenga que manejar diferentes atributos sintetizados para un mismo no terminal y que éstos deban ser devueltos, por tanto, por una misma función. Para la manipulación de múltiples atributos se suelen definir registros que contienen los distintos atributos como distintos campos y se le pasa a la función como argumento un puntero a dicho registro para que ésta pueda actualizar los campos que son atributos sintetizados. Al mismo tiempo, se pueden incluir en el registro los atributos heredados, que deberían ser inicializados antes de la llamada con los valores adecuados. Por ejemplo:

```
typedef struct {
    float h1,h2;      /* atributos heredados */
    int atr1;
    char *atr2;
} AtribA;
```

Así, la función para el no terminal *A* tendrá un prototipo como el siguiente: `void A(AtribA *aA);` y en su llamada se pasará un puntero a una variable (por ejemplo `atsA`) que haya sido definida de ese tipo: `A(&atsA)`.

<

7.3 Traductores ascendentes

Las características de un ETDS permiten construir un traductor que evalúe todos los atributos y ejecute todas las acciones semánticas a la vez que realiza el análisis sintáctico. Este tipo de traductores se conocen como *traductores de una sola pasada*. Aunque es más sencillo implementar un ETDS a partir de un analizador sintáctico descendente recursivo, la elección de este tipo de analizador obliga a utilizar una gramática LL(1) como base para diseñar el ETDS, cuando suele ser más sencillo diseñar un ETDS a partir de una gramática que tenga recursividad por la izquierda, por ejemplo. Por otra parte, a veces la semántica obliga a escoger gramáticas recursivas por la izquierda (p. ej. cuando deseamos reflejar en la gramática la asociatividad por la izquierda de un operador binario), que no serían LL(1). La elección de un analizador descendente recursivo obliga a modificar el ETDS para obtener uno equivalente basado en una gramática LL(1) equivalente a la original, y esta tarea suele ser tediosa y puede producir errores difíciles de detectar.

Para evitar rediseñar el ETDS y adaptarlo a una gramática LL(1) se debe emplear analizadores ascendentes, siendo los más utilizados los de la familia LR: SLR, LALR y LR. La implementación de ETDS sobre analizadores LR requiere varias modificaciones en las estructuras de datos y en el algoritmo de análisis por desplazamiento-reducción que se estudiarán más adelante.

Afortunadamente, la aparición de herramientas como YACC y LEX³ ha facilitado considerablemente la implementación de ETDS con analizadores LR, pero es impres-

³Estos programas generan un traductor (basado en un analizador LALR(1)) a partir de una especificación en una notación similar a los ETDS.

cindible que un diseñador de compiladores conozca las técnicas que existen (y que son utilizadas por YACC y LEX) para la implementación de ETDS con analizadores LR.

7.3.1 Equivalencia entre estados y símbolos

En una tabla de análisis LR cada estado representa un *prefijo viable*, que es un prefijo de una parte derecha de una regla que el analizador está intentando completar. Para diseñar traductores basados en analizadores LR se suele establecer una equivalencia entre los estados del analizador y los símbolos de la gramática, de manera que cada estado del analizador (excepto el estado inicial) tiene asociado un símbolo de la gramática. Un estado s se asocia con el símbolo (no terminal o terminal) con el que se llega a dicho estado, ya sea a través de una acción de desplazar o al apilar el estado correspondiente de la tabla “Ir_a” al final de una acción de reducción.

Si se construye el autómata reconocedor de prefijos viables, se puede observar mucho más fácilmente esta equivalencia: el símbolo que etiqueta la transición que lleva a un estado s es el que se asocia con dicho estado. Dicho símbolo es el situado más a la derecha en el prefijo viable que se ha reconocido hasta el momento.

Ejemplo 7.4

Dada la siguiente gramática:

$$\begin{aligned} E &\longrightarrow E \text{ addop } T \\ E &\longrightarrow T \\ T &\longrightarrow \text{num} \\ T &\longrightarrow (E) \end{aligned}$$

el autómata reconocedor de prefijos viables se puede ver en la figura 7.1, y la tabla de análisis SLR es la siguiente⁴:

Prefijo viable	Símbolo	Estado	Acción					Ir_a	
			addop	num	()	\$	E	T
-	-	0		d3	d4			1	2
E	E	1	d5				a!		
T	T	2	r2			r2	r2		
num	num	3	r3			r3	r3		
((4		d3	d4			6	2
$E \text{ addop}$	addop	5		d3	d4				7
(E	E	6	d5			d8			
$E \text{ addop } T$	T	7	r1			r1	r1		
(E))	8	r4			r4	r4		

◁

El proceso de análisis de una cadena del lenguaje fuente consiste en ir construyendo prefijos viables cada vez más largos, hasta completar una parte derecha de una regla; cuando esto sucede, se sustituye la parte derecha por la parte izquierda (si esa sustitución

⁴En la tabla, junto a cada estado se ha puesto el símbolo asociado correspondiente y el prefijo viable.

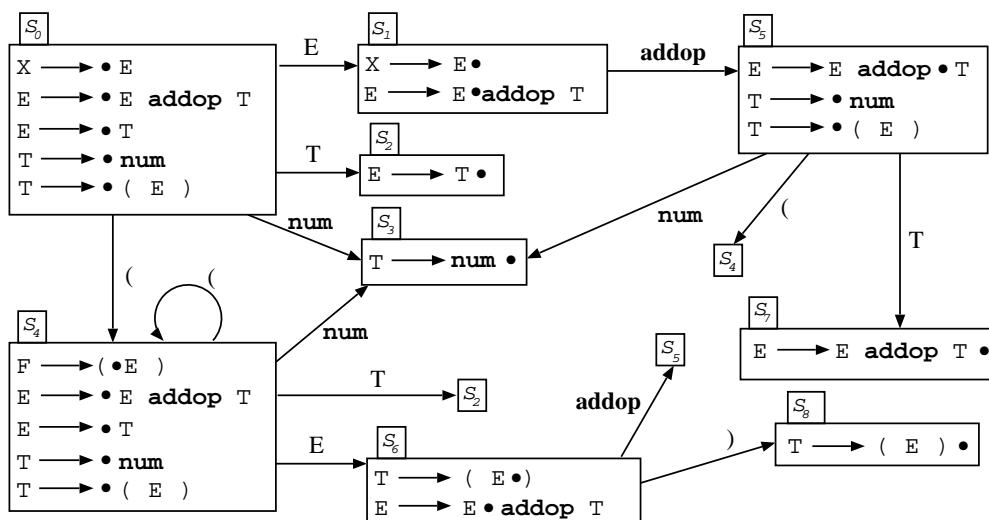


Figura 7.1: Autómata reconocedor de prefijos viables para el ejemplo 7.4.

produce otro prefijo viable). Cuando se hace una traza del análisis de una cadena solamente con estados no se puede apreciar este proceso, pero si se sustituye cada estado en la traza por su símbolo asociado se puede observar perfectamente dicho proceso de análisis.

Ejemplo 7.5

Dada la tabla del ejemplo anterior y la cadena de entrada “7+5”, la traza del análisis de dicha cadena (con el símbolo asociado a cada estado debajo de él) sería:

PILA	ENTRADA	ACCIÓN
0	num addop num \$	d3
-	7 + 5	
0 3	addop num \$	r3
- num	+ 5	
0 2	addop num \$	r2
- T	+ 5	
0 1	addop num \$	d5
- E	+ 5	
0 1 5	num \$	d3
- E addop	5	
0 1 5 3	\$	r3
- E addop num		
0 1 5 7	\$	r1
- E addop T		
0 1	\$	a!
- E		

Teniendo en cuenta que el proceso de análisis es el inverso de una derivación por la derecha de la cadena de entrada, es relativamente sencillo hacer una traza del análisis *sin estados*, usando únicamente los símbolos. Solamente se necesitaría la gramática y no sería necesario, por tanto, construir la tabla de análisis (siempre que la gramática sea LR).

Cuando se tiene que implementar un ETDS sobre un analizador LR es imprescindible conocer con toda exactitud las posibles configuraciones de la pila en un momento dado del análisis. Esto no implica ser capaz de hacer este tipo de trazas mentalmente, pero sí conocer muy bien el funcionamiento del algoritmo de análisis, es decir, la forma de construir poco a poco las partes derechas de las reglas en la pila hasta completar una parte derecha y sustituirla por la parte izquierda correspondiente, que a su vez contribuirá a completar otra parte derecha de otra regla que se había empezado a construir antes.

7.3.2 Implementación de ETDS sencillos

En esta sección hablaremos de cómo implementar ETDS sin atributos heredados y con todas las acciones semánticas situadas al final, que es el caso más sencillo. En posteriores secciones estudiaremos cómo el esquema estudiado en esta sección se puede adaptar para ETDS más complejos.

Para transformar un analizador sintáctico por desplazamiento–reducción (como los de la familia LR) en un traductor hay que resolver las siguientes cuestiones:

1. ¿Dónde se almacenan los atributos?

Los atributos de los no terminales y los de los terminales (como mínimo el lexema) se deben almacenar en alguna estructura de datos, para que puedan ser utilizados en las acciones semánticas del ETDS que queremos implementar. Dada la naturaleza de las gramáticas independientes del contexto y del algoritmo de análisis, lo más razonable es utilizar una pila para almacenar los atributos, ya que en un momento dado del análisis se pueden tener varios símbolos⁵ en la pila pertenecientes a distintas partes derechas de reglas⁶; incluso es posible que el mismo símbolo aparezca varias veces en la pila.

Aunque es posible hacer que las acciones semánticas gestionen directamente una pila de forma transparente al analizador sintáctico, lo más usual es utilizar una pila de atributos (también llamada *pila semántica*) que funciona de forma paralela a la de los estados del analizador. Dependiendo de la implementación se puede utilizar una pila de registros (**struct** en C) en los que cada campo del registro es un atributo (incluidos los atributos de los terminales), o bien una pila de uniones para ahorrar algo de memoria cuando un símbolo tiene menos atributos que los demás:

⁵En adelante hablaremos indistintamente de estados o símbolos en la pila de análisis.

⁶El algoritmo de análisis va construyendo en la parte superior de la pila las partes derechas de las reglas y cuando consigue completar una de ellas (y se cumplen ciertas condiciones), se sustituye toda la parte derecha por la parte izquierda y se sigue intentando buscar otra parte derecha. Este comportamiento hace que en un momento determinado se encuentren en la pila varias partes derechas en construcción.

```

#define MAXPILA ....

typedef struct {
                                /* atributos de los terminales */
    char *lexema;
    int linea,columna;
                                /* atributos de los no terminales */
    char *trad;
    char *otra;
} Atributos;

int PilaEstados[MAXPILA]; /* pila de estados */
Atributos PA[MAXPILA];   /* pila de atributos */
int tope = 0;             /* representa la posición en el array
                           del elemento situado en el tope de la pila */

```

Cuando en la pila de estados se apila un estado, en la pila de atributos se apila el registro con los atributos correspondientes al símbolo de la gramática asociado a ese estado; cuando se desapilan estados, se desapilan registros de atributos de la pila de atributos. Es necesario que la pila se implemente como un *array* puesto que, como veremos más adelante, vamos a acceder a los atributos de todos los símbolos de la parte derecha de la regla, no sólo a los del símbolo situado en el tope de la pila.

Durante el proceso de análisis solamente hay dos situaciones en las que se apila un estado:

1. Se puede demostrar que, cuando se produce un desplazamiento a un estado *s*, el símbolo asociado a ese estado *s* siempre es un terminal, por lo que en la pila de atributos se tendrían que apilar los atributos de dicho terminal (lexema, etc.).
2. En una reducción, se apila la parte izquierda de una regla después de haber desapilado todos los símbolos de la parte derecha. En este caso, lo que se apilaría en la pila de atributos serían los atributos asociados a la parte izquierda de la regla que acabamos de reducir.

2. ¿Cómo se transmite el lexema de los terminales?

En un analizador sintáctico solamente se necesita el lexema del último terminal leído (para producir un posible mensaje de error), pero en un traductor se necesitan los lexemas de todos los terminales de una regla para poder utilizarlos en las acciones semánticas del ETDS (si son necesarios). Por este motivo, el analizador léxico de un traductor debe almacenar el lexema (y posiblemente la línea y la columna asociadas a ese lexema) en algún lugar para que se apilen al desplazar dicho símbolo en la pila de estados. Lo habitual es utilizar una variable global de tipo *Atributos* que llamaremos *AtrTerminal* (en YACC se llama *yylval*), que almacene los atributos del terminal (y deje el resto de atributos sin asignar); esta variable es la que se apilará en la pila de atributos cuando se apile el terminal.

3. ¿En qué momento se ejecutan las acciones semánticas?

Suponiendo que tenemos que implementar un ETDS en el que todas las acciones

semánticas están situadas al final de las reglas (y en el que solamente hay atributos sintetizados), el momento ideal para ejecutar las acciones es cuando estamos a punto de reducir por una regla dada.

Las acciones semánticas en esta clase de ETDS lo que hacen es calcular los atributos de la parte izquierda de la regla en función de los atributos de los símbolos de la parte derecha. Estos atributos de la parte derecha se encuentran en la pila (en el tope se encuentran los del símbolo más a la derecha, debajo del tope los del penúltimo símbolo, etc.), y pueden ser utilizados por la acción semántica. Sin embargo, aunque la posición en la que se van a situar los atributos de la parte izquierda es conocida, no se pueden modificar dichos atributos puesto que corresponden a los del primer símbolo de la parte derecha, y es posible que se utilicen en otra acción semántica asociada también a esta regla. Por este motivo, la solución consiste en utilizar otra variable global (también de tipo **Atributos**), que llamaremos **AtrPI** (llamada **yyval** en YACC, que es distinta de **yyval**) para almacenar los atributos de la parte izquierda mientras se van calculando.

Una vez se han ejecutado todas las acciones semánticas asociadas a la regla, se desapilan los estados/símbolos de la pila de estados (y los atributos correspondientes de la pila de atributos), y, al apilar el estado asociado a la parte izquierda de la regla, se apila el valor de **AtrPI** en la pila de atributos.

Las modificaciones que habría que realizar al algoritmo de análisis del epígrafe 5.2 (figura 5.1) para convertirlo en un traductor están marcadas en negrita:

```

push(0,AtrEstado0)    /* los atributos del estado 0 se ignoran */
a :=analex()
REPETIR
  sea s el estado en el tope de la pila
  SI Accion[s,a] = dj ENTONCES
    push(j,AtrTerminal)
    a :=analex()
  SI NO SI Accion[s,a] = rk ENTONCES
    /* Ejecutar las acciones semánticas de la regla k y dejar los
    atributos de la parte izquierda en AtrPI */
    EjecutarAccionesSemanticas(k)
    PARA i := 1 HASTA Longitud_Parte_Derecha(k) HACER pop()
    /* pop() desapila de las 2 pilas */
    sea p el estado en el tope de la pila
    sea A el símbolo de la parte izquierda de la regla k
    push(Ir_a[p,A], AtrPI)
  SI NO SI Accion[s,a] = aceptar ENTONCES
    fin del analisis /* en el tope de la pila están los
    atributos del símbolo inicial */

  SI NO
    error
  FIN_SI
HASTA fin del analisis

```

Además, el analizador léxico debe rellenar los atributos del token que ha reconocido (modificando **AtrTerminal**) antes de devolver el control al analizador sintáctico.

Ejemplo 7.6

El siguiente ETDS (basado en la gramática del ejemplo 7.4) realiza un proceso simple de traducción, utilizando solamente atributos sintetizados.

- $$\begin{aligned}
 (1) \quad E &\longrightarrow E_1 \text{ addop } T \quad \left\{ \begin{array}{l} \text{si } \text{addop.lexema} = '+' \text{ entonces} \\ E.trad := 'sum(' || E_1.trad || ', ' || T.trad || ')', \\ \text{si no} \\ E.trad := 'res(' || E_1.trad || ', ' || T.trad || ')', \\ \text{fin si} \end{array} \right\} \\
 (2) \quad E &\longrightarrow T \quad \{ E.trad := T.trad \} \\
 (3) \quad T &\longrightarrow \text{num} \quad \{ T.trad := \text{num.lexema} \} \\
 (4) \quad T &\longrightarrow (E) \quad \{ T.trad := E.trad \}
 \end{aligned}$$

Las acciones semánticas que habría que realizar al reducir por cada regla serían, ya traducidas a código en C:

REGLA	ACCIÓN SEMÁNTICA
(1)	<pre> if (!strcmp(PA[tope-1].lexema, "+")) AtrPI.trad = concat("sum(", PA[tope-2].trad, ",", PA[tope].trad, ")"); else AtrPI.trad = concat("res(", PA[tope-2].trad, ",", PA[tope].trad, ")"); </pre>
(2)	AtrPI.trad = PA[tope].trad;
(3)	AtrPI.trad = PA[tope].lexema;
(4)	AtrPI.trad = PA[tope].trad;

La traza del proceso de traducción de la cadena de entrada “7+5” sería la siguiente:

PILA		ENTRADA				ACCIÓN
estado	0	num	addop	num	\$	d3
lexema		7	+	5		
trad						
estado	0	num	addop	num	\$	r3
lexema		7	+	5		acción
trad						(3)
estado	0	<i>T</i>	addop	num	\$	r2
lexema			+	5		acción
trad		7				(2)
estado	0	<i>E</i>	addop	num	\$	d5
lexema			+	5		
trad		7				
estado	0	<i>E</i>	addop	num	\$	d3
lexema			+	5		
trad		7				
estado	0	<i>E</i>	addop	num	\$	r3
lexema			+	5		acción
trad		7				(3)
estado	0	<i>E</i>	addop	<i>T</i>	\$	r1
lexema			+			acción
trad		7		5		(1)
estado	0	<i>E</i>			\$	aceptar
lexema						
trad		sum(7,5)				

7.3.3 Implementación de ETDS más complejos

Aunque lo deseable es tener ETDS sencillos, lo normal es que los ETDS tengan acciones intermedias (acciones situadas antes del final de la parte derecha de la regla) y atributos heredados. En muchas ocasiones, las acciones intermedias asignan valores a atributos heredados, pero también se suelen utilizar para otras tareas no relacionadas directamente con los atributos, como almacenar valores en variables globales (p. ej. en la tabla de símbolos).

Este segundo tipo de acciones intermedias es relativamente sencillo de implementar, pero requiere una pequeña modificación en el esquema estudiado en el apartado anterior: se introduce en la gramática un nuevo no terminal (llamado *marcador*) en la posición en la que está situada la acción semántica, y se añade a la gramática una regla en la que este no terminal deriva a la cadena vacía; al reducir por esta nueva regla se ejecuta la acción semántica. Por cada acción intermedia se introduce un marcador nuevo, ya que las acciones normalmente son distintas (si dos acciones son idénticas se puede aprovechar el marcador previamente introducido).

Esta modificación (introducir marcadores) no debe hacerse directamente sobre el ETDS, sino que se debe hacer sobre la gramática (y las acciones semánticas) al implementar dicho ETDS con un analizador LR. La introducción de uno o más marcadores modifica ligeramente la tabla de análisis: es muy sencillo rehacer el autómata para tener en cuenta el marcador (se añade un nuevo estado), y la tabla de análisis apenas cambia (los siguientes de cada no terminal no cambian), pero debe rehacerse teniendo en cuenta el nuevo autómata.

Normalmente, las acciones intermedias utilizan los atributos de los símbolos situados antes que ella en la parte derecha. En el momento de ejecutar la acción semántica, estos atributos se encuentran en la parte superior de la pila (en el tope están los atributos del símbolo anterior a la acción).

Ejemplo 7.7

El siguiente fragmento de ETDS tiene una acción intermedia que almacena el lexema del identificador en una tabla⁷.

$$\begin{array}{ll} S & \longrightarrow \textit{Tipo LisID} ; \{ \textit{AsignarTipo}(\textit{Tipo.t}) \} \\ \textit{Tipo} & \longrightarrow \textbf{int} \{ \textit{Tipo.t} := \textit{ENTERO} \} \\ \textit{Tipo} & \longrightarrow \textbf{char} \{ \textit{Tipo.t} := \textit{CARACTER} \} \\ \textit{LisID} & \longrightarrow \textbf{id coma} \{ \textit{Guarda}(\textbf{id.lexema}) \} \textit{LisID} \\ \textit{LisID} & \longrightarrow \textbf{id} \{ \textit{Guarda}(\textbf{id.lexema}) \} \end{array}$$

La gramática que se tendría que utilizar para hacer la tabla de análisis sería:

$$\begin{array}{ll} S & \longrightarrow \textit{Tipo LisID} ; \\ \textit{Tipo} & \longrightarrow \textbf{int} \\ \textit{Tipo} & \longrightarrow \textbf{char} \\ \textit{LisID} & \longrightarrow \textbf{id coma} M \textit{LisID} \\ \textit{LisID} & \longrightarrow \textbf{id} \\ M & \longrightarrow \epsilon \end{array}$$

en la que M es un *marcador*, y al reducir por la regla 6 (la del marcador) se ejecutará la acción intermedia ($\textit{Guarda}(\textbf{id.lexema})$). Para acceder al lexema del

⁷Para almacenar los identificadores en una tabla se utiliza la función $\textit{Guarda}()$.

identificador se debe tener en cuenta que en el tope de la pila está la **coma** y debajo de ella el **id** con sus atributos (lexema, etc.). La configuración de la pila en el momento de reducir por la regla del marcador (que es cuando se tiene que ejecutar la acción intermedia) sería la siguiente:

PILA				ACCIÓN
estado	0	...	id coma	reducir $M \rightarrow \epsilon$
lexema			a	
tope			↑	

Por tanto, la acción que se tendría que ejecutar al reducir por el marcador sería `Guarda(PA[tope-1].lexema)`.

<

Los marcadores no solamente ocupan un espacio en la pila de estados, sino que también (igual que el resto de símbolos de la gramática) ocupan un espacio en la pila de atributos. Ese espacio se puede utilizar para almacenar atributos heredados (como veremos más adelante) y también para guardar valores locales al análisis de una regla, es decir, valores que no pueden ser almacenados en variables globales (porque dentro de una regla se puede volver a analizar la misma regla y reescribir ese valor) ni en variables locales.

Ejemplo 7.8

El siguiente fragmento de ETDS tiene una acción intermedia que almacena el lexema del identificador en una tabla, y el valor que devuelve se debe guardar para ser usado al final de la regla.

```

S      → Tipo LisID; { AsignarTipo(Tipo.t) }
Tipo   → int { Tipo.t := ENTERO }
Tipo   → char { Tipo.t := CHARACTER }
LisID  → id coma { vlocal := Guarda(id.lexema) } LisID { Registra(vlocal) }
LisID  → id { v := Guarda(id.lexema); Registra(v) }
```

El valor de la variable *vlocal* se obtiene en la acción intermedia y se utiliza en la acción situada al final de la regla. Si guardamos el valor de *vlocal* en una variable global, al ser una regla recursiva, es posible que (si hay más de dos identificadores) se reescriba ese valor con otro y se pierda por tanto el valor anterior. Para solucionar este problema lo más sencillo es utilizar alguno de los atributos del marcador (p. ej. el llamado “*t*”) para almacenar ese valor, y adaptar la acción situada al final de la regla para que use ese atributo. La gramática que se debe utilizar para construir la tabla de análisis es la misma del ejemplo anterior (7.7), pero las acciones semánticas asociadas a sus reglas cambian. La configuración de la pila en el momento de reducir por el marcador es exactamente la misma que en el ejemplo anterior, por lo que la acción semántica asociada a esa regla será similar. La configuración de la pila en el momento de reducir por la regla

$$LisID \longrightarrow \mathbf{id\ coma} \ M \ LisID$$

sería la siguiente:

PILA						ACCIÓN	
estado	0	...	id	coma	M	$LisID$	reducir $LisID \rightarrow \dots$
lexema			a				
t					$(vlocal)$		
tope						↑	

Por tanto, la acción que se debe ejecutar al reducir por esta regla es **Registra**(PA[tope-1].t).

La siguiente tabla muestra estas producciones junto con las acciones que se deben ejecutar al reducir por ellas:

PRODUCCIÓN	ACCIÓN SEMÁNTICA ASOCIADA
$M \rightarrow \epsilon$	AtrPI.t = Guarda(PA[tope-1].lexema);
$LisID \rightarrow \text{id coma } M LisID$	Registra(PA[tope-1].t);

En el siguiente apartado se muestra cómo escribir este ETDS en la notación de YACC.

<

7.3.4 Notación de YACC

YACC es un programa muy extendido en UNIX (y Linux), que también tiene versiones para otros sistemas operativos: MS-DOS, MacOS, etc. YACC genera, a partir de un fichero con un ETDS (escrito en una notación específica), un traductor basado en una gramática LALR(1). Además, el traductor generado por YACC está preparado para trabajar con los analizadores léxicos generados por LEX. Existe abundante documentación sobre estos dos programas⁸, por lo que aquí solamente estudiaremos muy brevemente la notación de YACC para describir ETDS y cómo se pueden utilizar los marcadores.

Los anteriores ejemplos de ETDS se pueden traducir a la notación de YACC de la siguiente manera:

Ejemplo 7.6: (traducido a la notación de YACC)

```
%%

/* por convención (aunque no es necesario), se suelen poner
   los terminales en mayúsculas y los no terminales en minúsculas */

/* En esta regla (e : e ADDOP t)
   "$$" es la parte izquierda,
   "$1" es la "e" de la parte derecha,
   "$2" es el "ADDOP" y
   "$3" es la "t"

   $$ $1 $2 $3                                     */
e : e ADDOP t { /* código en C */
                if (!strcmp($2.lexema, "+"))
```

⁸Prácticamente todos los libros sobre compiladores dedican un par de capítulos a YACC y LEX.


```

        $$trad=concat("sum(", $1trad, ",", $3trad, ")");
    else
        $$trad=concat("res(", $1trad, ",", $3trad, ")");
    }
| t      { $$trad = $1trad; } /* otra regla de "e" */
;

t : NUM      { $$trad = $1.lexema; }
| LPAR e RPAR { $$trad = $2trad; }
;

```

Lo que hace YACC con una especificación de este tipo es lo siguiente:

1. Traducir las acciones semánticas a código real en C, sustituyendo las referencias con “\$” por la posición correspondiente en la pila, y traduciendo “\$\$” por “yyval”, que es la variable que se utiliza para almacenar los atributos de la parte izquierda de la regla (la que hemos llamado anteriormente “AtrPI”). Las referencias como “\$2” se traducen por algo equivalente a “yypv[tope-lp+(2)]”, donde “yypv” es la pila de atributos y “lp” es el número de símbolos de la parte derecha de la producción que aparecen antes de la acción semántica; si la acción está situada al final de la regla, “lp” es la longitud de la parte derecha de la regla. Por ejemplo, las acciones de las producciones de **t** quedarían de la siguiente manera:

t : NUM	yyval.trad = yypv[tope-1+(1)].lexema;
t : LPAR e RPAR	yyval.trad = yypv[tope-3+(2)].trad

Como se puede observar, el resultado de las operaciones entre los corchetes permite acceder a la posición correcta en la pila.

2. Construir la tabla de análisis LALR(1) para la gramática, y generar el traductor.

Ejemplo 7.7: (traducido a la notación de YACC)

```

%%

s      : tipo lisID PYC      { AsignarTipo($1.t); }
;

tipo : INT                  { $$t = ENTERO; }
; /* se pueden poner todas las reglas de un no terminal
   juntas o separadas */
tipo : CHAR                 { $$t = CARACTER; }
;

lisID : ID COMA      { /* accion intermedia */ Guarda($1.lexema); } lisID
| ID                { Guarda($1.lexema); }
;

/* No es necesario poner marcadores, YACC pone automáticamente el
marcador cuando se trata de una acción intermedia */

```

En este caso, como en el anterior, lo primero que hace YACC es traducir las acciones semánticas. La traducción de todas las acciones situadas al final de la regla es

trivial, y la de la acción intermedia sería: `Guarda(yyvsp[tope-2+(1)].lexema);`. Debe tenerse en cuenta que en el momento de ejecutar la acción intermedia solamente se han analizado dos símbolos de la producción, de ahí el “2” que se le resta a “tope”.

A continuación, YACC sitúa un marcador en el lugar de cada acción semántica (con lo cual quedaría la gramática del ejemplo 7.7) y asocia a cada regla las acciones que previamente ha traducido, de manera que la acción asociada a la reducción por la regla del marcador es la que hemos visto en el párrafo anterior, que es equivalente a la que aparece en el ejemplo 7.7.

Ejemplo 7.8: (traducido a la notación de YACC)

```
%%

s      : tipo lisID PYC      { AsignarTipo($1.t); }
      ;

tipo   : INT                { $$t = ENTERO; }
      | CHAR                { $$t = CARACTER; }
      ;

lisID  : ID COMA    { $$t = Guarda($1.lexema); }
/* Si en una acción intermedia se hace referencia a $$ se están
   utilizando los atributos del marcador, no los de la parte
   izquierda */

      lisID    { Registra($3.t); }
/* Como YACC sustituye las acciones intermedias por un
   marcador, la acción ocupa un lugar:    $1 es "ID",
                                           $2 es "COMA",
                                           $3 es el marcador y
                                           $4 es "lisID"

*/
      | ID      { { int v; /* para declarar variables locales
                           hay que abrir otro bloque */
                     v = Guarda($1.lexema); Registra(v);
                   }
      ;
```

En este caso, la acción intermedia se traduciría por:

```
yyval.t = Guarda(yyvsp[tope-2+(1)].lexema);
```

Al asociar esta acción a la reducción por la regla del marcador, el resultado de la llamada a “Guarda” se almacena en el atributo “t” del marcador; por este motivo, cuando en una acción intermedia se hace referencia a “\$\$” se está accediendo a los atributos del marcador. Esto es posible porque YACC primero traduce las acciones y después construye la gramática y asocia las acciones a las reducciones de las producciones de la gramática.

7.3.5 ETDS con atributos heredados

El principal problema de la implementación de ETDS con atributos heredados es que no existe una posición en la pila para el no terminal que tiene atributos heredados, puesto que está situado a la derecha de la acción y todavía no se ha analizado. De hecho, la posición en la que irá el no terminal es conocida, pero va a ser utilizada por un símbolo de la parte derecha de una regla del no terminal (y a su vez puede ser utilizada por descendientes de dicho símbolo si es un no terminal), por lo que no es posible guardar ninguna información en esa posición, ya que normalmente se perderá.

Ejemplo 7.9

El siguiente fragmento de ETDS tiene una acción intermedia que asigna el tipo de *Tipo* a *LisID* como atributo heredado.

$$\begin{array}{ll} S & \longrightarrow \textit{Tipo} \{ \textit{LisID.th} := \textit{Tipo.t} \} \textit{LisID}; \{ \textit{AsignarTipo}(\textit{Tipo.t}) \} \\ \textit{Tipo} & \longrightarrow \mathbf{int} \{ \textit{Tipo.t} := \textit{ENTERO} \} \\ \textit{Tipo} & \longrightarrow \mathbf{char} \{ \textit{Tipo.t} := \textit{CARACTER} \} \\ \textit{LisID} & \longrightarrow \mathbf{id} \{ \textit{Guarda}(\mathbf{id.lexema}, \textit{LisID.th}) \} \end{array}$$

Si se hace una traza se puede comprobar que, en el momento en que debería ejecutarse la acción intermedia, en la pila solamente está *Tipo*, y el espacio que más tarde ocupará *LisID* está libre y va a ser ocupado por el **id**.

◀

La solución a este problema consiste en estudiar cada regla en la que aparezca en la parte derecha el no terminal en cuestión (en el ejemplo *LisID*); en dichas reglas se debe asignar siempre (excepto casos muy excepcionales) un valor al atributo heredado. Puesto que el espacio para dicho atributo heredado no existe, y casi siempre existe un atributo sintetizado de otro no terminal ya analizado que contiene el valor que queremos darle al atributo heredado (en este caso *Tipo.t*), lo que se debe hacer es *asociar el atributo heredado con dicho atributo sintetizado*. Por tanto, cada atributo heredado estará asociado con uno o más atributos sintetizados de otros símbolos, uno por cada regla en la que aparezca el no terminal en la parte derecha.

Una vez hemos asociado el atributo heredado con el atributo sintetizado que contiene el valor que queremos asignarle, cuando se va a utilizar dicho atributo heredado en una regla es necesario acceder en la pila al atributo sintetizado correspondiente (que estará en alguna posición por debajo del primer símbolo de la parte derecha).

Ejemplo 7.10

En el ejemplo anterior (7.9), el atributo *LisID.th* se puede asociar con *Tipo.t*. Cuando, en la regla de *LisID* se tiene que utilizar el atributo heredado se accede a *Tipo.t*, que está debajo del **id** en la pila. En este caso, la acción semántica intermedia que le asigna valor al atributo heredado se eliminaría y no sería necesario poner ningún marcador. En la notación de YACC, este ETDS se escribiría de la siguiente manera (resolviendo ya el problema del atributo heredado porque YACC no admite atributos heredados):

```

%%

s      : tipo lisID PYC      { AsignarTipo($1.t); }      ;

tipo : INT                  { $$t = ENTERO; }
      | CHAR                { $$t = CARACTER; }
      ;

lisID : ID                  { Guarda($1.lexema,$0.t);

      /* si $1, $2, ... denotan los símbolos de la parte derecha,
         $0 denota el símbolo que hay debajo del primer
         símbolo de la parte derecha (en este caso 'tipo'),
         $-1 el que está dos posiciones por debajo, etc. */

      }

;

```

PILA		ENTRADA	ACCIÓN
estado lexema t	0	int id ; \$ int a	desplazar
estado lexema t	0 int int	id ; \$ a	reducir $Tipo \rightarrow int$
\$	\$0 \$1		\$\$t = ENTERO;
estado lexema t	0 Tipo ENTERO	id ; \$ a	desplazar
estado lexema t	0 Tipo id ENTERO a	; \$	reducir $LisID \rightarrow id$
\$	\$-1 \$0 \$1		Guarda(\$1.lexema,\$0.t);

<

Normalmente, los ETDS con atributos heredados no son tan sencillos de implementar como el ejemplo anterior, en el que no es necesario utilizar marcadores y pueden darse otras situaciones más complejas:

1. Que en dos (o más) reglas distintas los atributos asociados al atributo heredado estén a distinta distancia en la pila,
2. Que el atributo heredado se obtenga como una función de uno o más atributos sintetizados, es decir, que su valor no sea simplemente la copia del valor de un atributo sintetizado,
3. Que el no terminal que tiene el atributo heredado tenga reglas recursivas.

Vamos a estudiar estas situaciones con variaciones del ejemplo anterior.

Ejemplo 7.11

Supongamos que el ETDS fuera el siguiente:

$$\begin{aligned}
 S &\longrightarrow \textit{Tipo} \{ \textit{LisID.th} := \textit{Tipo.t} \} \textit{LisID} ; \{ \textit{AsignarTipo}(\textit{Tipo.t}) \} \\
 S &\longrightarrow \mathbf{var} \textit{Tipo} \mathbf{dosp} \{ \textit{LisID.th} := \textit{Tipo.t} \} \textit{LisID} \mathbf{fvar} \{ \textit{AsignarTipo}(\textit{Tipo.t}) \} \\
 \textit{Tipo} &\longrightarrow \mathbf{int} \{ \textit{Tipo.t} := \textit{ENTERO} \} \\
 \textit{Tipo} &\longrightarrow \mathbf{char} \{ \textit{Tipo.t} := \textit{CARACTER} \} \\
 \textit{LisID} &\longrightarrow \mathbf{id} \{ \textit{Guarda}(\mathbf{id.lexema}, \textit{LisID.th}) \}
 \end{aligned}$$

En este caso, el no terminal *LisID* aparece en dos reglas, y en ambos casos el atributo asociado con *LisID.th* es *Tipo.t* (lo cual no suele ser habitual). Como se puede observar sin necesidad de hacer una traza, la distancia a la que va a estar *Tipo.t* de la regla de *LisID* es distinta según qué regla de las dos de *S* se esté analizando. Si se está analizando la primera regla, *Tipo.t* estará en “\$0”, como en el ejemplo anterior, pero si se está analizando la segunda regla estaría en “\$-1”. En la acción semántica de la regla de *LisID* no se puede saber (sin utilizar una variable global, lo cual es peligroso) qué regla de *S* se está analizando, hay que elegir entre “\$0” y “\$-1”.

Para resolver este problema debemos conseguir que *Tipo.t* esté siempre a la misma distancia del **id** en la pila, y la solución es muy sencilla: en la primera de las dos reglas de *S* se pone un marcador a la derecha de *Tipo* que va a ocupar el hueco del terminal **dosp** en la otra regla, de manera que *Tipo.t* siempre va a estar en “\$-1”:

```

%%

s      :      tipo { /* Marcador */ } lisID PYC      { AsignarTipo($1.t); }
      | VAR tipo      DOSP      lisID FVAR      { AsignarTipo($2.t); }
      ;

tipo : INT          { $$t = ENTERO; }
      | CHAR        { $$t = CARACTER; }
      ;

lisID : ID          { Guarda($1.lexema,$-1.t); }
      ;

```

Las dos posibles configuraciones de la pila al reducir por *LisID* → **id** serían:

estado	0	<i>Tipo</i>	<i>Marcador</i>	id
lexema				a
t		ENTERO		
\$		\$-1	\$0	\$1

estado	0	var	<i>Tipo</i>	dosp	id
lexema					a
t			ENTERO		
\$		\$-2	\$-1	\$0	\$1

En este caso tampoco es necesario sustituir las acciones intermedias que asignan valor a *LisID.th* por marcadores.

Ejemplo 7.12

Supongamos ahora que el ETDS fuera:

$$\begin{aligned}
 S &\longrightarrow \textit{Tipo} \{ \textit{LisID.th} := \textit{GuardaTip}(\textit{Tipo.t}) \} \textit{LisID} ; \{ \textit{AsignarTipo}(\textit{Tipo.t}) \} \\
 \textit{Tipo} &\longrightarrow \mathbf{int} \{ \textit{Tipo.t} := \textit{ENTERO} \} \\
 \textit{Tipo} &\longrightarrow \mathbf{char} \{ \textit{Tipo.t} := \textit{CARACTER} \} \\
 \textit{LisID} &\longrightarrow \mathbf{id} \{ \textit{Guarda}(\mathbf{id.lexema}, \textit{LisID.th}) \}
 \end{aligned}$$

El atributo *LisID.th* se obtiene como el resultado de llamar a la función *GuardaTip()* con *Tipo.t*. En este caso no es posible asociar el atributo heredado con ningún otro atributo sintetizado; además, esa función podría tener efectos secundarios, como modificar el valor de alguna variable global. Por estos dos motivos, lo que se debe hacer en este caso (y en otros similares, aunque sean más sencillos) es poner un marcador en el lugar de la acción semántica y utilizar los atributos del marcador para almacenar el valor del atributo heredado, es decir, asociar un atributo sintetizado del marcador con el atributo heredado:

```

%%
s      : tipo      { $$t = GuardaTip($1.t); } /* $$ aquí se refiere al marcador */
      lisID PYC { AsignarTipo($1.t); }
      ;

tipo : INT      { $$t = ENTERO; }
    | CHAR     { $$t = CARACTER; }
    ;

lisID : ID      { Guarda($1.lexema,$0.t); }
      ;

```

Como se puede observar en la solución al problema en la notación de YACC, el marcador queda situado inmediatamente debajo del **id** en la pila (en “\$0”).

◁

Ejemplo 7.13

Dado el siguiente ETDS:

$$\begin{aligned}
 S &\longrightarrow \textit{Tipo} \{ \textit{LisID.th} := \textit{Tipo.t} \} \textit{LisID} ; \{ \textit{AsignarTipo}(\textit{Tipo.t}) \} \\
 \textit{Tipo} &\longrightarrow \mathbf{int} \{ \textit{Tipo.t} := \textit{ENTERO} \} \\
 \textit{Tipo} &\longrightarrow \mathbf{char} \{ \textit{Tipo.t} := \textit{CARACTER} \} \\
 \textit{LisID} &\longrightarrow \{ \textit{LisID}_1.th := \textit{LisID.th} \} \textit{LisID}_1 \mathbf{coma} \\
 &\quad \mathbf{id} \{ \textit{Guarda}(\mathbf{id.lexema}, \textit{LisID.th}) \} \\
 \textit{LisID} &\longrightarrow \mathbf{id} \{ \textit{Guarda}(\mathbf{id.lexema}, \textit{LisID.th}) \}
 \end{aligned}$$

Hay una regla recursiva por la izquierda para el no terminal *LisID*. Si se hace una traza solamente del análisis, sin tener en cuenta las acciones de traducción, se puede observar que *Tipo* está siempre situado inmediatamente debajo de cualquiera de las dos reglas de *LisID*, independientemente del número de **id** que aparezcan en la cadena de entrada. Las dos posibles configuraciones de la pila al reducir por las reglas de *LisID* serían:

estado	0	<i>Tipo</i>	id		
lexema		ENTERO	a		
t					
\$		\$0	\$1		

estado	0	<i>Tipo</i>	<i>LisID</i>	coma	id
lexema		ENTERO			b
t					
\$		\$0	\$1	\$2	\$3

En este caso, la acción semántica que transmite el atributo heredado de *LisID* a *LisID*₁ no sólo no es necesario sustituirla por un marcador, sino que si introducimos un marcador (o la acción semántica) YACC va a producir un conflicto reducción-reducción en la tabla de análisis⁹, y es probable que el traductor funcione mal. La solución sería eliminar dicha acción semántica, igual que hacemos con la de la regla de *S*:

```
%%

s      : tipo lisID PYC      { AsignarTipo($1.t); }
      ;

tipo   : INT                { $$ .t = ENTERO; }
      | CHAR                { $$ .t = CARACTER; }
      ;

lisID  : lisID COMA ID { Guarda($3.lexema,$0.t); }
      | ID              { Guarda($1.lexema,$0.t); }
      ;
```

<

Ejemplo 7.14

¿Qué ocurre si la recursividad es por la derecha?

$$\begin{aligned}
S &\longrightarrow \textit{Tipo} \{ \textit{LisID.th} := \textit{Tipo.t} \} \textit{LisID} ; \{ \textit{AsignarTipo}(\textit{Tipo.t}) \} \\
\textit{Tipo} &\longrightarrow \mathbf{int} \{ \textit{Tipo.t} := \textit{ENTERO} \} \\
\textit{Tipo} &\longrightarrow \mathbf{char} \{ \textit{Tipo.t} := \textit{CARACTER} \} \\
\textit{LisID} &\longrightarrow \mathbf{id coma} \{ \textit{Guarda}(\mathbf{id.lexema}, \textit{LisID.th}); \\
&\hspace{10em} \textit{LisID}_1.th := \textit{LisID.th} \} \\
&\hspace{10em} \textit{LisID}_1 \\
\textit{LisID} &\longrightarrow \mathbf{id} \{ \textit{Guarda}(\mathbf{id.lexema}, \textit{LisID.th}) \}
\end{aligned}$$

En este caso concreto, si se hace una pequeña traza se puede observar que *Tipo* se va alejando de las reglas de *LisID* conforme añadimos **id** a la pila. La evolución de la pila para la cadena de entrada “**int a,b,c**” sería la siguiente:

⁹En general, si en una especificación para YACC se ponen acciones (o marcadores) al principio de la parte derecha de las reglas es muy probable que aparezcan conflictos reducción-reducción.

estado	0	<i>Tipo</i>	id				
lexema			a				
t		ENTERO					
estado	0	<i>Tipo</i>	id	coma	id		
lexema			a		b		
t		ENTERO					
estado	0	<i>Tipo</i>	id	coma	id	coma	
lexema			a		b		
t		ENTERO					
estado	0	<i>Tipo</i>	id	coma	id	coma	id
lexema			a		b		c
t		ENTERO					

Este alejamiento de *Tipo* hace necesario introducir un marcador en la posición de la acción semántica que vaya transmitiendo el valor de *Tipo.t* a las diferentes *LisID*; y no sólo debe transmitirlo, sino que debe hacer que se encuentre siempre en la misma posición relativa con respecto a las reglas de *LisID*. Además, en este ETDS la acción semántica que transmite el atributo heredado hace también una llamada a la función *Guarda*, por lo que hay doble motivo para poner el marcador:

```
%%

s      : tipo lisID PYC      { AsignarTipo($1.t); }
      ;

tipo : INT      { $$t = ENTERO; }
     | CHAR     { $$t = CARACTER; }
     ;

lisID : ID COMA  { Guarda($1.lexema,$0.t);
                 $$t = $0.t; /* transmitir el atr. heredado */
                 }
       lisID
     | ID      { Guarda($1.lexema,$0.t); }
     ;
```

Las configuraciones de la pila en el momento de ejecutar las acciones semánticas en las producciones de *LisID* serían:

PILA			ACCIÓN
estado	0	<i>Tipo</i>	id
lexema			a
t		ENTERO	
\$	\$0	\$1	
estado	0	<i>... Marcador</i>	id
lexema			a
t		<i>... ENTERO</i>	
\$	\$0	\$1	
estado	0	<i>Tipo</i>	id coma
lexema			b
t		ENTERO	
\$	\$0	\$1 \$2	
estado	0	<i>... Marcador</i>	id coma
lexema			b
t		<i>... ENTERO</i>	
\$	\$0	\$1 \$2	

<

Ejemplo 7.15

Un caso típico en muchos ETDS es el de un atributo heredado que va bajando en el árbol en unas producciones recursivas, hasta llegar al caso base, que debe devolver como atributo sintetizado el atributo heredado que le llega. Dado el siguiente ETDS, que cuenta el número de variables declaradas¹⁰:

$$\begin{aligned}
 S &\longrightarrow \text{var } \{ A.h := 0 \} A \{ S.s := A.s \} \\
 A &\longrightarrow \text{id } \{ A_1.h := A.h + 1 \} A_1 \{ A.s := A_1.s \} \\
 A &\longrightarrow \epsilon \{ A.s := A.h \}
 \end{aligned}$$

Su implementación en YACC sería la siguiente:

```

%%

s  : VAR { /* M1 */ $$h=0; } a { $$s=$3.s; }
    ;

a  : ID { /* M2 */ $$h=$0.h + 1; } a { $$s=$3.s; }
    | /* epsilon */ { $$s=$0.h; }
    ;

```

La traza de este ETDS para una cadena de entrada como “var e f” se puede observar en la figura 7.2. Debemos recordar que al reducir por una producción vacía no existe “\$1”, ni “\$2”, etc., y que no se desapila ningún símbolo y se apila la parte izquierda de la regla. Esto sucede tanto en los marcadores como en $A \rightarrow \epsilon$.

<

¹⁰Evidentemente existen formas más eficientes de contar las variables declaradas, incluso con otros ETDS más sencillos.

PILA		ENTRADA	ACCIÓN
est h s	0	var id id \$	desplazar
est h s	0 VAR	id id \$	reducir $M1 \rightarrow \epsilon$
\$			\$\$h=0;
est h s	0 VAR M1 0	id id \$	desplazar
est h s	0 VAR M1 ID 0	id \$	reducir $M2 \rightarrow \epsilon$
\$	\$0 \$1		\$\$h=\$0.h + 1;
est h s	0 VAR M1 ID M2 0 1	id \$	desplazar
est h s	0 VAR M1 ID M2 ID 0 1	\$	reducir $M2 \rightarrow \epsilon$
\$	\$0 \$1		\$\$h=\$0.h + 1;
est h s	0 VAR M1 ID M2 ID M2 0 1 2	\$	reducir $a \rightarrow \epsilon$
\$	\$0		\$\$s=\$0.h;
est h s	0 VAR M1 ID M2 ID M2 a 0 1 2 2	\$	reducir $a \rightarrow ID\ M2\ a$
\$	\$1 \$2 \$3		\$\$s=\$3.s;
est h s	0 VAR M1 ID M2 a 0 1 2	\$	reducir $a \rightarrow ID\ M2\ a$
\$	\$1 \$2 \$3		\$\$s=\$3.s;
est h s	0 VAR M1 a 0 2	\$	reducir $s \rightarrow VAR\ M1\ a$
\$	\$1 \$2 \$3		\$\$s=\$3.s;
est h s	0 s 2	\$	aceptar

Figura 7.2: Traza del análisis de “var e f”.

7.3.6 Principales usos de los marcadores

Como hemos estudiado en las secciones anteriores, los principales usos de los marcadores son:

1. Conseguir que las acciones semánticas intermedias que sea necesario ejecutar (llamadas a funciones que modifican variables globales, etc) se ejecuten siempre al reducir por una regla.
2. Almacenar valores locales al análisis de una regla.
3. Ocupar huecos en la pila de atributos para conseguir que todos los atributos asociados a un atributo heredado queden situados a la misma distancia de la regla en la que se utiliza.
4. Almacenar el valor de algún atributo heredado que no se encuentra disponible en una posición conocida y fija de la pila, o bien cuando el atributo heredado se obtiene haciendo algún cálculo o llamando a alguna función externa.

También existen algunos casos en los que no es necesario utilizar marcadores:

- Cuando la acción semántica simplemente asocia un atributo heredado con otro sintetizado que está disponible en la pila, y
- Cuando la acción semántica transmite un atributo heredado que no es imprescindible transmitir porque sigue estando disponible en la pila.

Todas las técnicas estudiadas en este tema pueden combinarse y adaptarse adecuadamente a cada caso concreto, pero en general siempre es posible implementar un ETDS con un analizador LR aplicando una o más de estas técnicas.

7.4 Referencias bibliográficas

REFERENCIAS	EPÍGRAFES
[Louden, 1997]	6.2.3 y 6.2.5
[Aho, Sethi y Ullman, 1990]	5.5, 5.3 y 5.6
[Fischer y LeBlanc, 1991]	6.6, 6.7.2, 7.2.5 y 14.1.3

7.5 Ejercicios

Ejercicio 7.1

A partir del siguiente ETDS, aplíquense las técnicas para eliminar la recursividad por la izquierda que se estudian en el epígrafe 7.2.1 e impleméntese el ETDS resultante como un traductor descendente recursivo.

$$\begin{aligned}
 S &\longrightarrow L \textbf{ pyc } \{ S.a := \text{"var" } \parallel L.a \parallel \text{":"} \parallel L.b \parallel \text{";} \} \\
 L &\longrightarrow L_1 \textbf{ coma id } \{ L.a := L_1.a \parallel \text{","} \parallel \textbf{id.lexema}; L.b := L_1.b \} \\
 L &\longrightarrow T \textbf{ id } \{ L.a := \textbf{id.lexema}; L.b := T.t \} \\
 T &\longrightarrow \textbf{int} \{ T.t := \text{"integer"} \} \\
 T &\longrightarrow \textbf{float} \{ T.t := \text{"real"} \}
 \end{aligned}$$

Ejercicio 7.2

Conviértase este ETDS a la notación de YACC.

$$\begin{aligned}
 S &\longrightarrow T \textbf{ var } \{ A.h := T.s \} A \{ S.s := A.s \} \\
 A &\longrightarrow B \{ C.h := A.h \parallel B.s \} C \{ A.s := C.s \} \\
 B &\longrightarrow \textbf{id} \{ B.s := \textbf{id.valex} \} \\
 C &\longrightarrow \epsilon \{ C.s := C.h \} \\
 C &\longrightarrow \textbf{coma} B \{ C_1.h := C.h \parallel \text{","} \parallel B.s \} C_1 \{ C.s := C_1.s \}
 \end{aligned}$$

Ejercicio 7.3

Transfórmese este ETDS a la notación de YACC.

$$\begin{aligned}
 S &\longrightarrow T \textbf{ id } \{ \text{InsertaVar}(\textbf{id.valex}, T.t); L.th := T.t; \} \\
 &\quad L \\
 T &\longrightarrow \textbf{int} \{ T.t := \text{ENTERO}; \} \\
 T &\longrightarrow \textbf{float} \{ T.t := \text{FLOTANTE}; \} \\
 L &\longrightarrow \textbf{, id} \{ \text{InsertaVar}(\textbf{id.valex}, L.th); L_1.th := L.th; \} \\
 &\quad L_1 \\
 L &\longrightarrow \textbf{;}
 \end{aligned}$$

Ejercicio 7.4

Conviértase este ETDS a la notación de YACC. Todos los atributos son cadenas

de caracteres.

S	\longrightarrow	NA	$\{C.prim := A.s\}$
		coma B	$\{C.sec := B.s\}$
		C	$\{S.s := N.s C.s\}$
N	\longrightarrow	ϵ	$\{N.s := ""\}$
N	\longrightarrow	D	$\{C.prim := D.s\}$
		B	$\{C.sec := B.s\}$
		C	$\{N.s := C.s\}$
B	\longrightarrow	id	$\{B.s := id.lexema\}$
A	\longrightarrow	var id	$\{A.s := id.lexema\}$
D	\longrightarrow	func id	$\{D.s := id.lexema\}$
C	\longrightarrow	integer	$\{C.s := "int" C.prim C.sec\}$
C	\longrightarrow	real	$\{C.s := "float" C.prim C.sec\}$

Ejercicio 7.5

Dado el siguiente ETDS, tradúzcase a la notación de YACC.

S	\longrightarrow	decl A	$\{C.t := A.s\}$
		variables C	$\{S.s := C.s\}$
A	\longrightarrow	id	$\{pos := Busca(id.lexema); \}$
		D	$\{C.t := D.s\}$
		C	$\{Guarda(pos, C.s); A.s := C.s\}$
B	\longrightarrow	real	$\{B.s := REAL\}$
B	\longrightarrow	integer	$\{B.s := ENTERO\}$
C	\longrightarrow		$\{C_1.t := C.t\}$
		C_1 id	$\{Guarda(Busca(id.lexema), C_1.s);$ $C.s := C_1.s\}$
C	\longrightarrow	ϵ	$\{C.s := C.t\}$
D	\longrightarrow	var B dospto	$\{C.t := B.s\}$
		C fvar	$\{D.s := C.s\}$

Ejercicio 7.6

Dado el siguiente ETDS, tradúzcase a la notación de YACC.

S	\longrightarrow	main A	$\{B.k := A.a\}$
		B	$\{C.p := A.a ; C.q := B.b\}$
		C	$\{S.s := "principal" C.c\}$
C	\longrightarrow	code	$\{C.c := C.p "codigo" C.q\}$
A	\longrightarrow	$D E$	$\{C.p := D.d ; C.q := E.e\}$
		C	$\{A.a := C.c\}$
B	\longrightarrow	declaration	$\{B.b := B.k "declaracion"\}$
D	\longrightarrow	var	$\{D.d := "var"\}$
E	\longrightarrow	type	$\{E.e := "tipo"\}$

Capítulo 8

Tipos y generación de código

8.1 Introducción

En este capítulo vamos a estudiar cómo emplear los ETDS para obtener información acerca de los tipos y generar código intermedio. Como estudiaremos más adelante, el código intermedio que se genere para un programa fuente determinado depende en gran medida de los tipos utilizados en dicho programa. Por tanto, las fases de análisis semántico y generación de código intermedio están fuertemente relacionadas entre sí, y en la práctica son una única etapa guiada por el análisis sintáctico, como hemos explicado en los capítulos anteriores. Además, estudiaremos los tipos compuestos (*arrays*, registros, ...) y la generación de código para acceder a variables de cualquier tipo.

Un aspecto muy importante en el diseño de un compilador es la elección de la representación intermedia o código intermedio que se va a utilizar. En el epígrafe 8.3 estudiaremos los distintos tipos de representaciones intermedias que se suelen utilizar. Previamente, estudiaremos el módulo de gestión de la tabla de símbolos y los problemas que plantean los ámbitos anidados.

También se estudia en este capítulo la generación de código para expresiones aritméticas y para las instrucciones más comunes de los lenguajes de alto nivel. Durante todo el capítulo se hace una comparación entre los lenguajes C y Pascal, ya que son los precursores de la mayoría de los lenguajes actuales y son lenguajes casi antagónicos en muchos aspectos, especialmente en lo relacionado con el tratamiento de los tipos.

8.2 Gestión de la tabla de símbolos

La tabla de símbolos es una fase (o un módulo) del compilador que interacciona con las demás fases y, aunque suele estar estrechamente ligada al *front end*, a veces también interacciona con algunas fases del *back end*.

La tabla de símbolos es una tabla que se utiliza para almacenar los nombres definidos por el usuario en el programa fuente: variables, nombres de funciones, nombres de tipos, etc. Normalmente, un compilador debe comprobar, por ejemplo, que no se utiliza una variable sin haberla declarado previamente, o que no se declara una variable dos veces. Para ello, el compilador tiene que almacenar el nombre de la variable (y posiblemente su tipo y algún otro dato) en la tabla de símbolos y, cuando se utiliza esa variable en una

expresión, el compilador la busca (eficientemente) en la tabla para comprobar que existe y además para obtener información acerca de ella: tipo, dirección de memoria, etc. La información que se guarda en la tabla depende del tipo de símbolo de que se trate:

TIPO DE SÍMBOLO	INFORMACIÓN ALMACENADA
variable	nombre, tipo, tamaño, dirección de memoria ...
función	nombre, tipo, comienzo del código, ...
tipo definido por el usuario	nombre, tipo, tamaño, ...
constante	nombre, tipo, tamaño, valor, ...

El módulo de la tabla de símbolos proporciona al resto del *front end* dos funciones: añadir un nuevo símbolo a la tabla y buscar un símbolo para obtener su información (tipo, dirección, etc.). Cuando se va a añadir un símbolo lo primero que se hace es buscarlo y comprobar que no está ya declarado (si lo estuviera habría que dar un error), por lo que también en esta función de añadir se realiza una búsqueda. Durante la compilación de un programa se llegan a producir muchas búsquedas, por lo que la eficiencia temporal de estas búsquedas es un factor determinante en la velocidad del compilador. Aunque se suele pensar en la tabla de símbolos como un *array* o una tabla de registros (cuyos campos serían el nombre, el tipo, etc.), lo habitual (excepto en compiladores muy sencillos) es implementar la tabla de símbolos utilizando tablas de dispersión (tablas *hash*) para optimizar el tiempo de búsqueda.

Ejemplo 8.1

Dado el siguiente fragmento de programa en C:

```
int    a,b,c;
float  d,e;
char   f,g;
```

la tabla de símbolos que construiría un compilador podría ser¹:

NOMBRE	TIPO	TAMAÑO	DIRECCIÓN
a	ENTERO	2	100
b	ENTERO	2	102
c	ENTERO	2	104
d	REAL	4	106
e	REAL	4	110
f	CARACTER	1	114
g	CARACTER	1	115

◁

¹La asignación de una dirección de memoria a una variable depende del tipo de representación intermedia que utilice el compilador, y el tamaño de los tipos depende de la máquina objeto.

Cuando se está diseñando el fragmento de ETDS que se encarga de procesar las declaraciones y almacenar los símbolos en la tabla de símbolos, es importante tener en cuenta que el orden en que se deben almacenar los símbolos en la tabla es exactamente el orden en que aparecen en el programa fuente. Por tanto, es fundamental conocer, dado un ETDS, el orden en que se van a ejecutar las acciones semánticas (y para ello se debe conocer el funcionamiento del proceso de análisis sintáctico, que es el que determina el orden de ejecución de las acciones). Según la gramática, conseguir que las acciones se ejecuten en el orden adecuado puede ser una tarea muy sencilla o muy complicada, por lo que es posible que sea necesario rediseñar la gramática para facilitar la tarea del ETDS (en otras partes del *front end* también puede ser conveniente rediseñar la gramática).

Los errores semánticos que se pueden producir en este módulo de gestión de la tabla de símbolos son básicamente de dos clases:

- que se intente utilizar una variable que no haya sido declarada previamente, y
- que se intente declarar dos veces una variable (dentro del mismo *ámbito*, como se explica más adelante).

Además, pueden producirse errores internos del compilador porque hay demasiados símbolos o bien porque las variables declaradas ocupan demasiada memoria (p. ej. si hay *arrays* muy grandes).

Ámbitos anidados

La gestión de una tabla de símbolos parece sencilla a simple vista, pero se complica un poco cuando en el lenguaje fuente se permiten distintos *ámbitos* de declaración de símbolos. Por ejemplo, en el lenguaje C está permitido declarar variables al principio de un bloque entre llaves², y además se permite que dichas variables tengan el mismo nombre que otras variables declaradas previamente en otros bloques anteriores (en otros *ámbitos*), con lo que las nuevas variables están *ocultando* a las anteriores con el mismo nombre: cuando dentro del bloque se utiliza una variable, el compilador primero la busca entre las variables del bloque y, si no la encuentra, la busca en las que había declaradas antes. De esta manera, si se declaran dos variables con el mismo nombre en distintos ámbitos, el compilador encuentra la última que se haya declarado y esté en un ámbito abierto.

Ejemplo 8.2

El siguiente fragmento de programa en C puede servir para demostrar la semántica de las declaraciones en los bloques:

```
{
    int    a,    /* primera 'a' */
          b,c;
```

²En la última versión del estándar ANSI (1999) las variables pueden declararse en cualquier lugar (como en C++) y no exclusivamente al comienzo de un bloque.


```
a = 7;
if (1)
{
    int i,a;    /* segunda 'a' */

    a = 8;      /* asigna un 8 a la segunda 'a' */
    b = a;      /* asigna a 'b' el valor de la segunda 'a' */
}

/* En este punto 'b' vale 8, 'a' vale 7 e 'i' no existe */
}
```

<

Además, cuando se cierra un ámbito, el compilador debe *olvidar* las variables que hubiera declaradas en él, puesto que pueden inducir a error si permanecen en la tabla de símbolos. En algunos compiladores se introduce un nuevo campo en la tabla de símbolos que indica el ámbito o nivel de anidamiento del símbolo en cuestión. El hecho de añadir variables a la tabla de símbolos al abrir un bloque y borrarlas al cerrarlo hace que el funcionamiento de la tabla de símbolos se parezca a una pila, por lo que en algunos casos se suele utilizar una pila de tablas de símbolos (con una tabla para cada bloque) con el objetivo de resolver el problema de los ámbitos.

Como se ha explicado anteriormente, las búsquedas en la tabla de símbolos se realizan tanto al intentar obtener información de un símbolo como al añadir un nuevo símbolo a la tabla (para comprobar que no se ha declarado previamente). Si el lenguaje fuente permite ámbitos anidados, estas búsquedas deben realizarse de forma diferente:

- Cuando se va a añadir un nuevo símbolo, se debe buscar el símbolo únicamente entre los símbolos del mismo ámbito, lo que se puede implementar fácilmente comenzando las búsquedas desde el final de la tabla (si se implementa la tabla de símbolos como una tabla de registros), y terminando cuando se llega al principio de la tabla o cuando se llega a un símbolo cuyo ámbito o nivel de anidamiento es menor que el actual.
- Cuando se intenta obtener información acerca de un símbolo que aparece en una instrucción, se debe buscar el símbolo desde el final de la tabla hasta el principio, de manera que se encontrará el símbolo del ámbito no cerrado más cercano.

Además de la gestión de los símbolos, si el compilador asigna direcciones de memoria a las variables, las direcciones asignadas a las variables del ámbito o bloque deben ser reutilizadas cuando se cierra el bloque, ya que esas variables serán usadas únicamente en el código de ese bloque y cuando se cierre el bloque las variables no serán *visibles*, y por tanto las direcciones que ocupan podrán asignarse a otras variables que se declaren más adelante.

En la mayoría de los lenguajes del tipo de Pascal o C, el cuerpo de una función es considerado un ámbito en sí mismo (los nombres de los argumentos son lo primero que se almacena en la tabla de símbolos), por lo que la gestión de la tabla de símbolos

al compilar el cuerpo de una función es muy similar. Además, Pascal (como otros lenguajes) permite declarar funciones locales a otras funciones, es decir, funciones que son declaradas al mismo nivel que las variables locales de la función; así como las variables locales de una función no son *visibles* fuera de su cuerpo, las funciones locales tampoco. La compilación de las funciones y de las llamadas a funciones se tratará en profundidad en el capítulo siguiente, pero las reglas de ámbitos son básicamente las descritas aquí.

8.3 Representaciones intermedias

La utilización de una representación intermedia permite separar claramente el *front end* del *back end*, y reutilizar ambas partes del compilador para futuros compiladores. Además, la mayoría de los compiladores se diseñan como parte de una familia de compiladores para una serie de procesadores o bien para una serie de lenguajes fuente (C/C++, FORTRAN, Pascal, ...). La elección de la representación intermedia que va a utilizar el compilador depende en gran medida de la “familia”: si se trata de una serie de compiladores para procesadores muy parecidos, lo habitual es utilizar un lenguaje intermedio que sea un superconjunto de los conjuntos de instrucciones de todos los procesadores, de manera que la generación de código objeto real sea prácticamente inmediata a partir del código intermedio. Si por el contrario se trata de un compilador perteneciente a una familia de compiladores cuyos lenguajes fuente son muy parecidos pero cuyos lenguajes objetos son diferentes, lo razonable es utilizar una representación intermedia de más alto nivel, como puede ser un árbol sintáctico abstracto (en inglés *abstract syntax tree*).

Las clases de representaciones intermedias más utilizadas son:

- Código de tres direcciones (*three-address code*), representado mediante tripletas o cuádruplas.

Ejemplo: La expresión aritmética “ $(2+3)*(2+3+5)$ ” se traduciría a la siguiente secuencia de instrucciones en código de tres direcciones:

INSTRUCCIÓN	OPERANDO 1	OPERANDO 2	RESULTADO
ADD	2	3	t_1
ADD	2	3	t_2
ADD	t_2	5	t_3
MUL	t_1	t_3	t_4

Se denomina código de tres direcciones porque, como se puede observar, las instrucciones en este lenguaje tienen (casi todas) 3 direcciones: dos para los operandos y una para el resultado. En el ejemplo, $t_1, t_2 \dots$ son variables temporales que utiliza el compilador para almacenar resultados intermedios.

Nota: El optimizador de código intermedio detectará que t_1 y t_2 contienen el mismo valor, y por tanto eliminará la segunda instrucción y sustituirá t_2 por t_1 en la tercera instrucción. Esta optimización se conoce como *eliminación de subexpresiones comunes*. Es más, en este ejemplo concreto realmente no se llegaría a aplicar esta optimización puesto que antes se habría aplicado otra optimización para evaluar expresiones constantes y se habría sustituido *toda* la expresión por su valor: 50. Lo normal es que el generador de código intermedio no intente generar

un código intermedio muy bueno, sino que intenta generar un código intermedio fácilmente optimizable, y ello implica generar (a veces) código claramente ineficiente.

- Árboles sintácticos abstractos y grafos dirigidos acíclicos.

Ejemplo: La expresión aritmética anterior se representaría con el siguiente árbol sintáctico abstracto de la figura 8.1.

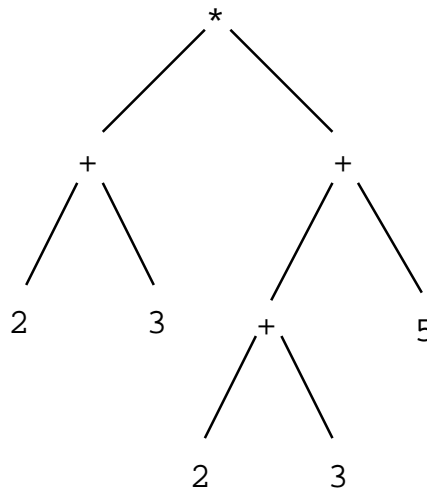


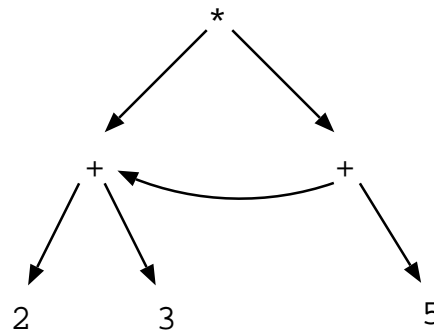
Figura 8.1: Árbol sintáctico abstracto para $(2+3) \cdot (2+3+5)$.

Los grafos dirigidos acíclicos se obtienen cuando el compilador detecta y elimina subexpresiones comunes en un árbol.

Ejemplo: El grafo de la expresión $(2+3) \cdot (2+3+5)$ sería el que se muestra en la figura 8.2. Lo normal sería que el generador de código intermedio del *front end* generase árboles y el optimizador de código intermedio pudiera transformarlos en grafos, por lo que el generador de código objeto tendría que admitir tanto árboles como grafos.

- Código de una máquina virtual, como, por ejemplo, el *P-code* utilizado en muchos compiladores de Pascal, que es el código de una máquina virtual de pila.

Ejemplo: La traducción de la expresión anterior a código para una máquina virtual de pila sería:

Figura 8.2: Grafo dirigido acíclico para “ $(2+3)*(2+3+5)$ ”.

INSTRUCCIÓN	SIGNIFICADO
lda 2	Apilar un 2
lda 3	Apilar un 3
add	Desapilar dos números, sumarlos y apilar el resultado (5)
lda 2	
lda 3	
add	Repetir la operación para el segundo 2+3
lda 5	
add	Sumar el segundo 2+3 con 5 y apilar el resultado (10)
mul	Desapilar dos números (el 5 del primer 2+3 y el 10), multiplicarlos y apilar el resultado

Aunque el código para máquinas virtuales de pila ha tenido mucho éxito debido a su utilización en compiladores de Pascal y a que es un ejemplo muy frecuente en los libros de texto sobre compiladores, es bastante complicado transformar ese código en código para una máquina concreta. Además de las máquinas virtuales de pila se utilizan otros tipos de máquinas virtuales más parecidas a un procesador real, con un número determinado de registros y un conjunto de instrucciones similar al ensamblador de un procesador cualquiera.

Los lenguajes intermedios también se utilizan para construir pseudo-compiladores: tienen un *front end* como el de un compilador normal, que genera código intermedio que después es ejecutado por un intérprete. Los intérpretes de lenguajes de alto nivel no son muy distintos: suelen tener también un *front end* y un intérprete de código intermedio, todo ello en el mismo ejecutable.

Los primeros compiladores de Pascal, debido a problemas de memoria y de velocidad del compilador, generaban *P-code* que posteriormente era interpretado, lo cual permitía ejecutar un programa objeto en todas aquellas máquinas en las que hubiera un intérprete de *P-code*. Esta idea, que estaba últimamente considerada desfasada, se ha vuelto a poner de moda gracias a la aparición del lenguaje Java, que especifica que los compiladores de dicho lenguaje deben generar código intermedio para una máquina virtual (la *Java Virtual Machine* o *JVM*), que después será ejecutado por un intérprete (que por ejemplo llevan incluido en su código los navegadores de Internet). Los famosos *bytecodes* que hay en los ficheros `.class` son instrucciones de este lenguaje intermedio. La gran ventaja de

Java y la principal causa de su éxito es la portabilidad del código generado, ya que puede ejecutarse en cualquier máquina que disponga de un intérprete para dicho lenguaje. También han aparecido compiladores de Java que generan código objeto para máquinas reales, aunque normalmente no utilizan los *bytecodes* como lenguaje intermedio³.

El problema de los tipos y las direcciones de memoria

Los lenguajes de alto nivel utilizan operadores *sobrecargados*, como pueden ser, por ejemplo, los operadores aritméticos o relacionales, que permiten que sus operandos sean enteros, reales, o uno entero y otro real. Sin embargo, y aunque es posible utilizar un lenguaje intermedio con operadores sobrecargados, dado que los lenguajes objeto no permiten la sobrecarga de operadores, lo normal es que el lenguaje intermedio no tenga los operadores sobrecargados, para facilitar la traducción a código objeto. Esto implica que el *front end* debe calcular el tipo de cada operando y generar las instrucciones del lenguaje intermedio de manera adecuada. En el caso de los operadores más habituales:

- si los dos operandos son los dos enteros o los dos reales, se debe generar una operación para enteros o bien una operación para reales,
- si un operando es entero y el otro real, se debe generar código para convertir el operando entero en un valor real y se debe generar la operación para reales.

Ejemplo 8.3

El código de tres direcciones para la expresión “2*3+0.5” sería, con los operadores sobrecargados, el siguiente:

INSTRUCCIÓN	OPERANDO 1	OPERANDO 2	RESULTADO
MUL	2	3	t_1
ADD	t_1	0.5	t_2

y sería el siguiente sin los operadores sobrecargados:

INSTRUCCIÓN	OPERANDO 1	OPERANDO 2	RESULTADO
MUL_i	2	3	t_1
i_TO_r	t_1	-	t_2
ADD_r	t_2	0.5	t_3

En este segundo caso, se utiliza una instrucción específica (MUL_i) para multiplicar enteros y otra para sumar reales (ADD_r). Además, se necesita una instrucción de conversión de entero a real (i_TO_r).

◁

³Un lenguaje intermedio puede ser muy bueno para ser interpretado pero no tan bueno para ser traducido a código objeto de una máquina real (el *P-code* es un buen ejemplo).

Además, otro problema que es posible que tenga que resolver el *front end* es el de asignar o no direcciones de memoria a las variables. Por ejemplo, cuando el lenguaje intermedio es el lenguaje de una máquina virtual, normalmente se asigna una dirección de memoria en la máquina virtual a cada variable (cuando se almacena en la tabla de símbolos), de manera que en el código intermedio solamente aparecen direcciones de memoria. También es posible utilizar una representación intermedia que trabaje con nombres de variables y posponga hasta el *back end* la asignación de direcciones de memoria a las variables.

8.4 Comprobaciones y conversiones de tipos

El compilador debe realizar dos tareas relacionadas con los tipos de los datos: calcular el tipo de cada expresión o construcción del lenguaje (*inferencia de tipos*) y comprobar que en las operaciones y en las instrucciones los tipos son correctos (*comprobación de tipos*). Ambas tareas (y otras menos importantes) constituyen la fase de análisis semántico, y se puede realizar a la vez que se hace el análisis sintáctico, utilizando normalmente un ETDS.

Las comprobaciones semánticas que realiza el compilador dependen siempre de la semántica del lenguaje fuente, del *significado* de cada construcción en dicho lenguaje. La mayoría de los lenguajes imperativos de las familias de Pascal y de C exigen, por ejemplo, que los identificadores (las *variables*) se declaren antes de ser utilizados en una instrucción, y además suelen exigir que el usuario del compilador les asigne un tipo concreto. Por tanto, el compilador debe comprobar, cuando aparece un identificador, que ha sido declarado previamente, y además debe obtener información acerca de él que será imprescindible para la generación de código. Todo esto se hace con ayuda de la tabla de símbolos, en la que se almacenan los identificadores (y el tipo que el usuario les ha asignado) cuando se declaran.

La información acerca del tipo de un identificador es importante por varios motivos, entre ellos los siguientes:

- La generación del código intermedio depende en gran medida de los tipos de los datos: no se genera el mismo código para una suma de valores (constantes numéricas o variables) enteros que para una suma de valores reales. Como hemos estudiado anteriormente en este capítulo, los lenguajes de alto nivel permiten utilizar el mismo operador para sumar enteros y para sumar reales (operadores sobrecargados), y ya hemos comentado que no es razonable que el lenguaje intermedio tenga también los operadores sobrecargados, por lo que lo normal es que el *front end* deba resolver este problema.
- En algunos lenguajes no se permite utilizar valores de cualquier tipo con cualquier operador. Por ejemplo, Pascal no permite sumar un valor real con un valor de tipo carácter, y en este caso el compilador debe producir un error semántico; en este mismo caso, C permite la suma, pero el compilador debe generar código para convertir el valor de tipo carácter en un valor real (tomando su código ASCII como un número real).

- En la instrucción de asignación de Pascal no están permitidas todas las asignaciones; en todos los casos se exige que la expresión a la derecha del operador de asignación sea del mismo tipo que la variable que aparece a la izquierda, y solamente se permite una excepción: que la variable sea de tipo real y la expresión de tipo entero. En C, la asignación es una operación más, como puede ser la suma, y tiene unas reglas de conversión propias: el tipo de la variable es el tipo resultante de la operación y la expresión a la derecha del operador debe convertirse al tipo de la parte izquierda.
- En muchos lenguajes está permitido realizar operaciones aritméticas (suma, resta, multiplicación, división, etc.) o relacionales (comparaciones de igualdad, etc.) mezclando valores enteros con reales. En el caso de que aparezca una operación de un valor entero con un valor real, el tipo resultante de la operación será real y el valor entero debe convertirse a un valor real antes de operarlo. Esta conversión es una de las *conversiones implícitas* que la mayoría de los lenguajes tiene. También se suelen permitir las *conversiones explícitas* (forzadas por el programador); por ejemplo, el lenguaje C permite el uso de operadores para forzar una conversión (como p. ej. `(float)2`) y Pascal utiliza funciones predefinidas para las conversiones explícitas.

Tanto para generar código intermedio como para producir errores semánticos (o generar conversiones de tipos) es necesario que el compilador tenga en cuenta en todo momento el tipo de cada variable o constante del programa fuente, y ese tipo debe transmitirse a las expresiones aritméticas, relacionales o lógicas, para ser utilizado en asignaciones o en otro tipo de instrucciones; por ejemplo, Pascal exige que la condición que aparece después de un “if” sea de tipo booleano.

Ejemplo 8.4

El ETDS de la figura 8.3 reconoce un subconjunto muy pequeño de Pascal y calcula el tipo de cada expresión; además, como hace todo compilador de Pascal, produce un error semántico si se intenta asignar un valor real a una variable entera. Por simplificar, supondremos que solamente hay dos tipos: entero y real.

Como se puede observar en el ETDS, cuando se trata de sumar (o restar, o multiplicar, etc) un valor entero con un valor real, el tipo de la expresión resultante es real (y, como veremos más adelante, la expresión entera debe convertirse a real antes de sumar).

<

Las restricciones semánticas relacionadas con los tipos de un lenguaje se denominan en su conjunto *reglas de tipos* o *sistema de tipos*. El ETDS de la figura 8.3 implementa un subconjunto del sistema de tipos de Pascal. Normalmente, las reglas de tipos se expresan en lenguaje humano, y debe ser el programador del compilador el que interprete dichas reglas y las convierta en acciones semánticas en su ETDS.

$$\begin{array}{ll}
Instr \longrightarrow id := & \{ \text{si } Busca(id.lexema) = NO_ENCONTRADO \\
& \text{entonces } ErrorSemantico(\dots) \\
& fsi \} \\
E & \{ \text{si } Tipo(id.lexema) = ENTERO \text{ y } \\
& E.tipo = REAL \text{ entonces } ErrorSemantico(\dots) \\
& fsi \} \\
E \longrightarrow E_1 + T & \{ \text{si } E_1.tipo = ENTERO \text{ y } T.tipo = ENTERO \text{ entonces } \\
& E.tipo := ENTERO \\
& \text{si no si } E_1.tipo = ENTERO \text{ y } T.tipo = REAL \text{ entonces } \\
& E.tipo := REAL \\
& \text{si no si } E_1.tipo = REAL \text{ y } T.tipo = ENTERO \text{ entonces } \\
& E.tipo := REAL \\
& \text{si no} \\
& E.tipo := REAL \\
& fsi \} \\
E \longrightarrow T & \{ E.tipo := T.tipo \} \\
T \longrightarrow nint & \{ T.tipo := ENTERO \} \\
T \longrightarrow nfix & \{ T.tipo := REAL \} \\
T \longrightarrow id & \{ \text{si } Busca(id.lexema) = NO_ENCONTRADO \\
& \text{entonces } ErrorSemantico(\dots) \\
& \text{si no} \\
& T.tipo := Tipo(id.lexema) \\
& fsi \}
\end{array}$$

Figura 8.3: Esquema de traducción que calcula los tipos y hace algunas comprobaciones semánticas.

8.5 Código intermedio para expresiones

El código que debe generar un compilador para procesar expresiones aritméticas, relacionales o lógicas depende de la semántica del lenguaje fuente, pero también del tipo de representación intermedia que se utilice: si se utiliza un árbol o un grafo, la traducción de la expresión es sencilla; si se utiliza una máquina abstracta de pila, también (la notación de la máquina de pila se parece mucho a la notación postfija, que es muy fácil de generar usando un ETDS). Sin embargo, si se utiliza un código intermedio que necesita temporales, como el código de tres direcciones o el lenguaje `m2r` (descrito en el apéndice C), la traducción se complica un poco, aunque sigue siendo en general bastante sencilla.

Los lenguajes de alto nivel suelen tomar la sintaxis y la semántica de las expresiones del lenguaje Pascal o de C. Los operadores más importantes que aparecen en un lenguaje de alto nivel se suelen clasificar en tres grupos:

Operadores aritméticos: La siguiente tabla muestra los operadores aritméticos de C y Pascal:

LENGUAJE	OPERADORES
C	+ - * / %
Pascal	+ - * / div mod

La diferencia entre ambos lenguajes se manifiesta en el operador de división: mien-

tras que en C la división es una división entera si los dos operandos son enteros, en Pascal el operador “/” se utiliza para la división real (incluso cuando los operandos son enteros), y el operador “div” se utiliza exclusivamente para enteros.

Los operadores “+”, “-” y “*” (y “/” en C) permiten que los dos operandos sean enteros o reales, y también permiten que uno sea real y el otro entero; en este caso, el tipo de la expresión será real y el entero debe convertirse a real antes de operarlo. El ETDS de la figura 8.4 hace este tipo de conversiones y genera código para una máquina abstracta de pila.

$$\begin{array}{ll}
 E \longrightarrow E_1 + T & \{ \text{si } E_1.\text{tipo} = \text{ENTERO y } T.\text{tipo} = \text{ENTERO entonces} \\
 & E.\text{tipo} := \text{ENTERO} \\
 & E.\text{cod} := E_1.\text{cod} || T.\text{cod} || \text{"AddI"} \\
 & \text{si no si } E_1.\text{tipo} = \text{ENTERO y } T.\text{tipo} = \text{REAL entonces} \\
 & E.\text{tipo} := \text{REAL} \\
 & E.\text{cod} := E_1.\text{cod} || \text{"Itor"} || T.\text{cod} || \text{"AddR"} \\
 & \text{si no si } E_1.\text{tipo} = \text{REAL y } T.\text{tipo} = \text{ENTERO entonces} \\
 & E.\text{tipo} := \text{REAL} \\
 & E.\text{cod} := E_1.\text{cod} || T.\text{cod} || \text{"Itor"} || \text{"AddR"} \\
 & \text{si no} \\
 & E.\text{tipo} := \text{REAL} \\
 & E.\text{cod} := E_1.\text{cod} || T.\text{cod} || \text{"AddR"} \\
 & \text{fsi} \} \\
 E \longrightarrow T & \{ E.\text{cod} := T.\text{cod} \\
 & E.\text{tipo} := T.\text{tipo} \} \\
 T \longrightarrow \text{nint} & \{ T.\text{cod} := \text{"LdI"} || \text{nint.lexema} \\
 & T.\text{tipo} := \text{ENTERO} \} \\
 T \longrightarrow \text{nfix} & \{ T.\text{cod} := \text{"LdR"} || \text{nfix.lexema} \\
 & T.\text{tipo} := \text{REAL} \} \\
 T \longrightarrow \text{id} & \{ \text{si Busca}(\text{id.lexema}) = \text{NO_ENCONTRADO} \\
 & \text{entonces ErrorSemantico}(\dots) \\
 & \text{si no} \\
 & T.\text{tipo} := \text{Tipo}(\text{id.lexema}) \\
 & \text{si Tipo}(\text{id.lexema}) = \text{ENTERO entonces} \\
 & T.\text{cod} := \text{"LdaI"} || \text{Direccion}(\text{id.lexema}) \\
 & \text{si no} \\
 & T.\text{cod} := \text{"LdaR"} || \text{Direccion}(\text{id.lexema}) \\
 & \text{fsi} \\
 & \text{fsi} \}
 \end{array}$$

Figura 8.4: Esquema de traducción que genera código para una máquina abstracta de pila utilizando la información de los tipos.

En general, el lenguaje C permite utilizar estos operadores con operandos de casi cualquier tipo, mientras que Pascal exige que sean enteros o reales. Esta característica hace que el compilador de C deba generar muchos más tipos de conversiones (además de la del párrafo anterior), mientras que el compilador de Pascal produce errores semánticos en esos casos.

Operadores relacionales: Los operadores relacionales se utilizan para comparar valores, y son:

LENGUAJE	OPERADORES
C	< <= > >= == !=
Pascal	< <= > >= = <>

Además de utilizar distintos símbolos para los operadores de igualdad y desigualdad, las principales diferencias entre ambos lenguajes aparecen en el tratamiento de los tipos: mientras que en Pascal se exige que al utilizar los operadores <, <=, > y >= los operandos sean del mismo tipo (excepto con enteros y reales, donde sí se permiten distintos tipos, como en otros operadores), en C no existe tal restricción. También el tipo del resultado es diferente: mientras que en C es un entero (que valdrá 0 si la comparación es falsa o un valor distinto de 0 si es cierta), en Pascal es de tipo **boolean**.

La generación de código intermedio para los operadores relacionales no plantea problemas diferentes a los de los operadores aritméticos, excepto los referentes a los tipos que se comentan en el párrafo anterior.

Operadores lógicos: Los operadores lógicos son habitualmente los de la siguiente tabla:

LENGUAJE	OPERADORES
C	&& !
Pascal	or and not

En el caso de Pascal, los operadores exigen que el operando sea de tipo **boolean**, y el resultado es también **boolean**, pero no existen grandes diferencias con los otros operadores en cuanto a la generación de código intermedio. En cambio, en C los operadores “||” y “&&” se evalúan *en cortocircuito*, es decir, primero se evalúa el operando de la izquierda y si su valor ya permite determinar el valor de toda la expresión, no se evalúa el segundo operando. Esto sucede cuando el operando izquierdo se evalúa a cierto y el operador es “||” (en ese caso el valor de la expresión será cierto independientemente del valor del segundo operando), o cuando el operando izquierdo es falso y el operador es “&&” (en ese caso la expresión será falsa). Esta característica implica que el compilador debe traducir estos operadores como si fueran instrucciones condicionales:

$$\begin{array}{lcl} a \ || \ b & \equiv & \text{if } a \text{ then true else } b \\ a \ \&\& \ b & \equiv & \text{if } a \text{ then } b \text{ else false} \end{array}$$

En Pascal se evalúan los dos operandos y posteriormente se realiza la operación lógica con los resultados de evaluarlos. Por este motivo, la generación de código intermedio para estos operadores es similar a la de los otros operadores.

Ejemplo 8.5

El lenguaje **m2r** (véase el apéndice C) utiliza un acumulador y variables temporales para evaluar las expresiones. Por ejemplo, si tenemos dos variables declaradas, una como entera (“a”) y otra como real (“b”), la traducción a **m2r** de la expresión

a + 3*b

sería, suponiendo que la variable “a” tiene asignada la dirección de memoria 10 y “b” tiene la 11,

```

mov 10 100 ; aunque no es necesario, movemos el valor
           ; de 'a' a una temporal
mov #3 101
mov 11 102 ; lo mismo con 'b'
mov 101 A
itor
mov A 103  ; convertir el '3' a real para operarlo con 'b'
mov 103 A  ; poner '3.0' en el acumulador
mulr 102   ; multiplicarlo por 'b'
mov A 104  ; guardar el resultado en otra temporal
mov 100 A
itor
mov A 105  ; convertir el valor de 'a' a real
mov 105 A  ; poner el valor (real) de 'a' en el acumulador
addr 104   ; sumarle el resultado de '3*b'
mov A 106  ; dejar el resultado en una nueva temporal

```

Como se puede observar, la generación de código para los operadores es sencilla: primero se pone el valor del primer operando en el acumulador, a continuación se opera con el segundo operando, y finalmente se almacena el resultado en una temporal. El ETDS de la figura 8.5 muestra cómo generar este código⁴ para la suma, aunque las tareas a realizar son similares para los demás operadores aritméticos, relacionales y lógicos.

◀

8.6 Código intermedio para instrucciones

La semántica de los lenguajes de programación de alto nivel en lo referente a las instrucciones es muy similar. Como en la sección anterior, vamos a describir las distintas clases de instrucciones haciendo a la vez una comparación entre C y Pascal, que son los lenguajes en los que se basan la mayoría de los lenguajes de alto nivel.

Las instrucciones de un lenguaje se pueden clasificar en las siguientes clases o tipos:

Asignación: Aunque en C el operador de asignación es un operador más, lo cual convierte a la asignación en un tipo de expresión más, la mayoría de las veces se emplea únicamente como instrucción, que es como la considera Pascal. La sintaxis en ambos lenguajes es similar:

LENGUAJE	SINTAXIS DE LA ASIGNACIÓN
C	$v = Expr$
Pascal	$v := Expr$

⁴Puesto que en el lenguaje m2r todos los tipos básicos ocupan solamente una posición de memoria (aunque lo normal es que un real ocupe más que un entero), una variable temporal se puede utilizar tanto para almacenar un real como para almacenar un entero.

```

E  →  E1 + T  { tmp := NuevaTemporal()
                  E.tmp := tmp
                  si E1.tipo = ENTERO y T.tipo = ENTERO entonces
                    E.tipo := ENTERO
                    E.cod := E1.cod||T.cod||
                        "mov "||E1.tmp||"A"||
                        "addi "||T.tmp||
                        "mov A "||tmp
                  si no si E1.tipo = ENTERO y T.tipo = REAL entonces
                    E.tipo := REAL
                    tmpconv := NuevaTemporal()
                    E.cod := E1.cod||"mov "||E1.tmp||"A"||
                        "itor"||"mov A"||tmpconv||T.cod||
                        "mov "||tmpconv||"A"||
                        "addr "||T.tmp||
                        "mov A "||tmp
                  si no si E1.tipo = REAL y T.tipo = ENTERO entonces
                    E.tipo := REAL
                    tmpconv := NuevaTemporal()
                    E.cod := E1.cod||T.cod||
                        "mov "||T.tmp||"A"||
                        "itor"||"mov A"||tmpconv||
                        "mov "||E1.tmp||"A"||
                        "addr "||tmpconv||
                        "mov A "||tmp
                  si no
                    E.tipo := REAL
                    E.cod := E1.cod||T.cod||
                        "mov "||E1.tmp||"A"||
                        "addr "||T.tmp||
                        "mov A "||tmp
                  fsi }

E  →  T          { E.cod := T.cod ;  E.tipo := T.tipo ;  E.tmp := T.tmp }
T  →  nint       { T.tipo := ENTERO
                  tmp := NuevaTemporal() ;  T.tmp := tmp
                  T.cod := "mov #"||nint.lexema||tmp }

T  →  nfix       { T.tipo := REAL
                  tmp := NuevaTemporal() ;  T.tmp := tmp
                  T.cod := "mov $"||nfix.lexema||tmp }

T  →  id         { si Busca(id.lexema) = NO_ENCONTRADO
                  entonces ErrorSemantico(...)
                  si no
                    T.tipo := Tipo(id.lexema)
                    tmp := NuevaTemporal() ;  T.tmp := tmp
                    T.cod := "mov "||Direccion(id.lexema)||tmp
                  fsi }

```

Figura 8.5: Esquema de traducción que genera código para el lenguaje m2r.

Como en otras cuestiones, las diferencias más importantes entre C y Pascal en lo referente a la asignación tienen que ver con los tipos de los datos: mientras un compilador de Pascal no permite asignaciones entre tipos distintos⁵, en C se suelen permitir casi todas las asignaciones, produciendo avisos (*warnings*) si es necesario, lo cual implica que el compilador debe generar código para realizar conversiones.

Algo en lo que coinciden ambos lenguajes es en la exigencia de que v sea un *valor izquierdo*, es decir, una variable o una componente de un *array* o registro (o lo que es lo mismo, *algo* que represente una dirección de memoria).

El esquema de la figura 8.6 representa el código que se generaría para una asignación (utilizando `m2r` como lenguaje objeto). Puesto que es posible que sean necesarias conversiones de tipo, en el esquema se supone que el valor de la expresión convertida se quedaría en la temporal t_1 ; si no hubiera que realizar ninguna conversión, en lugar de t_1 se utilizaría `Expr.tmp`.

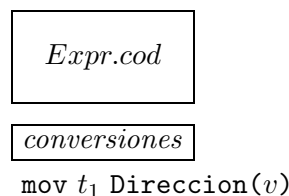


Figura 8.6: Esquema del código para una asignación.

Entrada/Salida: Las sentencias de entrada/salida son imprescindibles en cualquier lenguaje de programación, y son implementadas de formas muy diferentes pero sencillas, por lo que aquí no pondremos más que un pequeño ejemplo del código que habría que generar (en `m2r`) para imprimir el número entero 27 seguido de un cambio de línea:

```
wri #27
wrl
```

Lo más importante que debe tenerse en cuenta al compilar estas sentencias es que las de lectura son muy similares a las asignaciones, lo cual puede complicar el código (como se puede complicar en las asignaciones).

Condicional: La sentencia condicional es indispensable en cualquier lenguaje de programación. La sintaxis de esta sentencia en C y Pascal no es muy diferente:

LENGUAJE	SINTAXIS DE LA SENTENCIA CONDICIONAL
C	if (<i>Expr</i>) <i>Instr</i> if (<i>Expr</i>) <i>Instr</i> ₁ else <i>Instr</i> ₂
Pascal	if <i>Expr</i> then <i>Instr</i> if <i>Expr</i> then <i>Instr</i> ₁ else <i>Instr</i> ₂

⁵Excepto si la expresión es entera y v es real, en cuyo caso se convierte la expresión a real antes de la asignación.

La diferencia entre ambos lenguajes está en los tipos: Pascal exige que la expresión sea de tipo `boolean`, mientras que C no tiene este tipo y por tanto considerará como cierto un valor de la expresión distinto de 0 (sea del tipo que sea; no debe ser necesariamente entero), y falso en otro caso. La figura 8.7 muestra el código (en m2r) que habría que generar para las dos sentencias condicionales (sin `else` y con `else`).

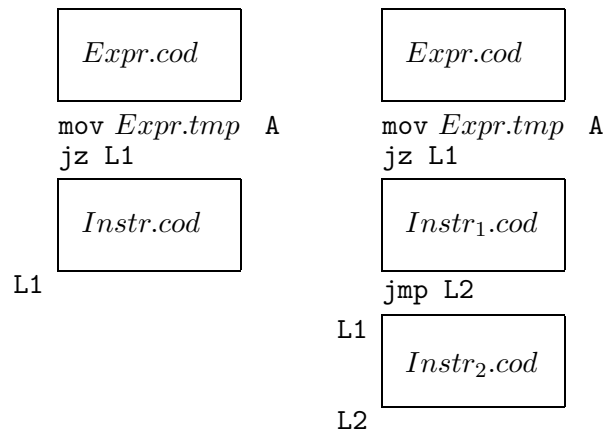


Figura 8.7: Esquema del código para las sentencias condicionales.

Iteración: Los lenguajes de programación suelen definir al menos tres sentencias de iteración:

Sentencias tipo while: Tanto C como Pascal tienen una sentencia de este tipo, cuya sintaxis es, como en el caso de las instrucciones condicionales, muy similar:

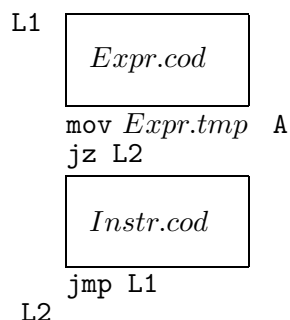
LENGUAJE	SINTAXIS DE LA SENTENCIA <code>while</code>
C	<code>while (Expr) Instr</code>
Pascal	<code>while Expr do Instr</code>

También como en el caso de las instrucciones condicionales, la única diferencia entre ambos lenguajes es el diferente tratamiento de la expresión como consecuencia de la ausencia del tipo `boolean` en C. El esquema de la figura 8.8 muestra el código que habría que generar.

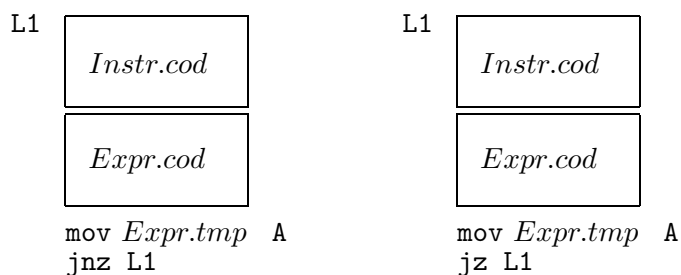
Sentencias de repetición: Este tipo de sentencias ejecutan al menos una vez la instrucción que hay dentro del bucle. La sintaxis en C y Pascal sería:

LENGUAJE	SINTAXIS DE LA SENTENCIA DE REPETICIÓN
C	<code>do Instr while (Expr)</code>
Pascal	<code>repeat Instr until Expr</code>

Además de las diferencias en el léxico y en el tratamiento de las expresiones booleanas, existe además una diferencia semántica: en C el bucle se termina

Figura 8.8: Esquema del código para la sentencia **while**.

cuando la expresión es falsa, mientras que en Pascal se termina cuando la expresión es cierta. La figura 8.9 muestra el esquema del **do-while** a la izquierda y la del **repeat-until** a la derecha; la única diferencia está en el tipo de salto condicional que se utiliza en una y otra sentencias.

Figura 8.9: Esquema del código para las sentencias de repetición **do-while** y **repeat-until**.

Sentencias tipo for: Ambos lenguajes disponen de una sentencia de este tipo, pero la semántica es muy distinta. Mientras que en C un bucle **for** no es muy diferente de un bucle **while**, en Pascal esta sentencia tiene un cometido específico: hacer que un contador tome una serie de valores consecutivos dentro de un rango concreto. La sintaxis de ambas sentencias **for** es la siguiente:

LENGUAJE	SINTAXIS DE LA SENTENCIA for
C	for (E_1 ; E_2 ; E_3) <i>Instr</i>
Pascal	for $v := E_1$ to E_2 do <i>Instr</i>

En cuanto a la semántica, C no pone ninguna restricción con respecto a las expresiones, mientras Pascal exige que tanto v como las dos expresiones sean

de tipo entero. La semántica del **for** de Pascal es muy sencilla: primero, se inicializa v con el valor de E_1 ; a continuación se ejecuta la instrucción, se incrementa en uno el valor de v (el contador), y si el nuevo valor de v no es mayor que el valor de E_2 , se vuelve a ejecutar la instrucción (y el incremento y la comparación, por supuesto). Utilizando “**downto**” en lugar de “**to**” el contador toma valores descendentes en lugar de ascendentes.

En la sentencia **for** de C las tres expresiones que aparecen son opcionales, y el código que se generaría sería equivalente al siguiente código en C:

```

     $E_1$  ;
    while (  $E_2$  )
    {
        Instr
         $E_3$  ;
    }

```

Si la segunda expresión E_2 no aparece, se toma como si fuera cierta (bucle infinito).

En Pascal, la sentencia **for** se puede sustituir por el siguiente código:

```

     $v := E_1$  ;
    repeat
        Instr ;
         $v := v + 1$ 
    until  $v > E_2$ 

```

Además de estas sentencias, en C existen dos instrucciones para alterar el funcionamiento de los bucles: **break** y **continue**. La sentencia **break** se utiliza para salir del bucle en que se encuentra (y también para salir de un **case** en una sentencia **switch**), y la sentencia **continue** hace que el flujo de control salte hasta la expresión del bucle, evitando ejecutar las sentencias intermedias, si es que existen.

Selección: Tanto en C como en Pascal existe una sentencia para la selección. Ambas tienen una expresión como principal argumento (en C se convierte a entero y en Pascal se exige que sea entera), y a continuación contienen una serie de valores enteros con los que se compara el valor de la expresión; cada valor tiene una secuencia de instrucciones asociada, que se ejecuta cuando el valor de la expresión coincide con dicho valor.

Ejemplo 8.6

Un ejemplo de sentencia **switch** en C y su equivalente sentencia **case** en Pascal podría ser el siguiente:

C	PASCAL
<pre> switch(E) { case 1: case 2: case 3: /* secuencia de instrucciones */ break; case 4: /* otra secuencia */ break; default: /* instrucciones por defecto */ } </pre>	<pre> case E of 1..3: (* secuencia de instrucciones *) 4: (* otra secuencia *) default: (* instrucciones por defecto *) end </pre>

<

Como se puede observar en el ejemplo, las diferencias entre ambas sentencias son mínimas. La más importante es la utilización de la sentencia **break** en C para terminar de ejecutar una secuencia de instrucciones; si no se pone **break**, la ejecución sigue por la siguiente sentencia (aunque pertenezca a un **case** diferente). Esta característica permite agrupar varios casos, lo cual se consigue en Pascal utilizando rangos de números.

La generación de código intermedio para una sentencia de selección es una de las decisiones más difíciles que debe tomar el compilador. Una vez compilada cada secuencia de instrucciones de cada **case**, y suponiendo que tienen asignada cada una de ellas una etiqueta, el compilador tiene varias posibilidades:

1. Generar una secuencia de comparaciones y saltos (algo parecido a la traducción de una secuencia de **if-else** anidados). Cuando el número de casos es relativamente grande, esta aproximación es muy ineficiente.
2. Construir una tabla ordenada de pares *valor-etiqueta*, y generar código que realice una búsqueda binaria del valor de la expresión entre los casos posibles y que cuando lo encuentre salte a la etiqueta correspondiente.
3. Cuando los posibles valores están agrupados en un rango relativamente pequeño (p. ej. 0–255), lo más eficaz es construir una *tabla de saltos* para todos los valores del rango, poniendo la etiqueta del **default** en los casos que no aparecen.

Un buen compilador debe, en función de la sentencia **switch** o **case** que esté procesando, elegir una de las posibilidades anteriores (o bien una combinación eficiente de las mismas).

Reciclado de variables temporales

En un lenguaje que utiliza variables temporales (como el código de tres direcciones o el **m2r**) es necesario establecer algún tipo de estrategia de reciclado de variables temporales. Aunque sería posible reciclar o reutilizar las variables temporales en las expresiones

(utilizando por ejemplo la temporal de uno de los operandos para almacenar el resultado), esta posibilidad dificultaría mucho la tarea del optimizador (sería más difícil la eliminación de subexpresiones comunes, por ejemplo). Por tanto, parece razonable reciclar las temporales al nivel de las instrucciones, es decir, una vez se ha generado todo el código de una instrucción (incluido el código de las expresiones o instrucciones que contenga), las variables temporales ya no son necesarias y se pueden reutilizar en la siguiente instrucción.

8.7 Tipos compuestos

Los lenguajes de programación permiten utilizar tipos más complejos que los tipos simples o básicos, que se definen en función de otros tipos. Los más comunes son: punteros, vectores o tablas (*arrays*) y registros.

En el libro de Aho, Sethi y Ullman (1990) se utiliza una notación para especificar el tipo de un componente de un programa, como, por ejemplo, una variable o una expresión. En la siguiente tabla se muestran los principales elementos de dicha notación, que permite escribir las llamadas *expresiones de tipos* para los elementos del lenguaje:

NOTACIÓN	EJEMPLOS	
puntero(t)	<code>int *p;</code>	puntero(entero)
array(rango,t)	<code>float **q;</code>	puntero(puntero(real))
	<code>int c[10];</code>	array(0..9,entero)
	<code>int d[3][6];</code>	array(0..2,array(0..5,entero))
	<code>float *e[5];</code>	array(0..4,puntero(real))
registro(campos)	<code>struct {int a,b;};</code>	registro((a:entero) × (b:entero))
	<code>struct {int *c; float d[4];};</code>	registro((c:puntero(entero)) × (d:array(0..3,real)))
(argumentos) → t	<code>int f(int a,float b);</code>	(int × float) → int
	<code>char *strdup(char *);</code>	puntero(char) → puntero(char)

La mayoría de los lenguajes permiten definir otros tipos de datos compuestos, además de los anteriores, como pueden ser los tipos enumerados (**enum**) y las uniones (**union**) en C, pero no suponen muchas dificultades añadidas, por lo que no hablaremos de ellos. Aunque en los lenguajes orientados a objetos como C++ y Java, el tipo **class** no es muy distinto de un registro, la compilación de este tipo de lenguajes es muy compleja y se sale del ámbito de este libro.

Las principales características de los tipos más comunes son:

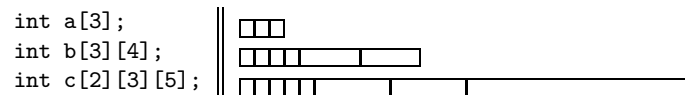
Punteros: Son fundamentales en cualquier lenguaje de alto nivel, pero también son posiblemente la mayor fuente de errores de cualquier programa que los utilice. Probablemente por esto, algunos lenguajes recientemente diseñados (como Java) no permiten al programador utilizar punteros, aunque internamente el compilador sí los utiliza. En cuanto a la tarea del compilador, una variable de tipo puntero no es más que una variable que puede contener una dirección de memoria en la que hay datos. El tamaño que el compilador asigna a una variable de este tipo depende de la arquitectura de la máquina objeto: si utiliza, por ejemplo, direccionamiento de 32 bits, se utilizan 4 bytes para un puntero.

Arrays: También son tipos fundamentales en cualquier lenguaje. Normalmente, se pueden definir *arrays* de una dimensión (también llamados vectores), de dos dimensiones (matrices), etc. En general, no suele haber límite en el número de dimensiones de un *array*: depende del espacio que ocupe en memoria (que sí suele estar limitado).

La forma de representar un *array* en memoria suele ser en todos los lenguajes la misma, y se basa en considerar un *array* n -dimensional como un *array* de *arrays* $(n - 1)$ -dimensionales.

Ejemplo 8.7

La siguiente tabla muestra cómo se almacenarían en memoria algunos *arrays* en C.



El *array* “b” es un *array* de 3 *arrays* de 4 enteros cada uno, y su *expresión de tipos* sería:

```
array(0..2,array(0..3,entero))
```

◁

Normalmente, el índice de un *array* es un valor entero⁶ (o se convierte a un valor entero, como ocurre en C). El rango de posibles valores del índice depende del lenguaje: en C, el rango va de 0 al tamaño de la dimensión correspondiente menos 1, mientras que en Pascal el rango de posibles valores se define al declarar el *array* y puede por tanto comenzar en valores distintos de 0 (e incluye los dos límites del rango).

Ejemplo 8.8

La siguiente tabla muestra los rangos de valores posibles para el índice en algunos *arrays* en C y en Pascal.

DECLARACIÓN	LENGUAJE	RANGO
<code>int a[3];</code>	C	0 – 2
<code>float b[15];</code>	C	0 – 14
<code>c:array [1..7] of char</code>	Pascal	1–7
<code>d:array [50..67] of real</code>	Pascal	50 – 67

◁

Mientras que en C, por cuestiones de eficiencia, no se comprueba que el valor de un índice esté incluido en el rango de valores posibles, Pascal exige que el índice pertenezca a ese rango. Puesto que, en general, el compilador no puede conocer el valor del índice, debe generar código para que esta comprobación se realice en

⁶Algunos lenguajes permiten otros tipos como índices. Por ejemplo, AWK permite que el índice de un *array* sea una cadena (*arrays* asociativos).

tiempo de ejecución (lo cual ralentiza el programa objeto). Existen multitud de librerías para C que realizan estas comprobaciones y otras similares relacionadas con los punteros, ya que la gran mayoría de los errores de ejecución de los programas en C (y C++) están relacionados con esta comprobación y con el uso indebido de punteros.

Registros: Este tipo de datos es también indispensable en cualquier lenguaje de alto nivel, aunque algunos lenguajes orientados a objetos lo sustituyen por el tipo `class`. Un registro se almacena en memoria simplemente guardando consecutivamente todos los campos del registro⁷, por lo que el tamaño de un registro no suele ser más que la suma de los tamaños de sus campos.

Una de las tareas que debe realizar el compilador es almacenar eficientemente los campos de un registro. Para ello existen las siguientes posibilidades:

1. Crear una tabla de símbolos nueva para cada registro que se declare (pero no para cada variable de tipo registro).
2. Utilizar una tabla de símbolos específica para almacenar todos los campos de todos los registros, distinta de la utilizada para almacenar variables y funciones. Esto implica añadir un campo más a dicha tabla para enlazar cada campo de un registro con el siguiente campo de dicho registro.
3. Almacenar los campos de los registros en la tabla de símbolos global, para lo cual es necesario enlazar los símbolos globales entre sí, y enlazar también los campos de los registros. Esta alternativa es muy ineficiente pero puede ser necesario utilizarla cuando la memoria disponible sea escasa.

Independientemente de la alternativa que se utilice, la dirección de los campos de un registro debe ser relativa al comienzo de la variable de tipo registro (puede haber más de una variable declarada con ese tipo), por lo que la dirección del primer campo siempre debe ser 0, y las siguientes deben ser consecutivas (según el tamaño del campo anterior).

8.7.1 Tabla de tipos

En la implementación de un compilador no es posible utilizar las expresiones de tipos presentadas anteriormente para representar los tipos de las variables. Para almacenar eficientemente los tipos compuestos se utiliza una tabla de tipos, en la que inicialmente se almacenan los tipos básicos, y a la que se van añadiendo los tipos compuestos según aparecen en el programa fuente. Una vez almacenado un tipo en la tabla de tipos, las variables declaradas con ese tipo se pueden almacenar en la tabla de símbolos utilizando un índice a la tabla de tipos como tipo.

⁷Esta afirmación no es del todo cierta: algunos compiladores, para no tener que trabajar con direcciones de memoria impares (lo cual es más ineficiente en la mayoría de los procesadores actuales), rellenan con campos ficticios el registro para que la dirección de todos los campos sea par, e incluso se puede llegar a utilizar esta técnica para que las direcciones sean múltiplo de 4.

Ejemplo 8.9

Las siguientes declaraciones en C:

```
int a;
float **b;
char c[10];
int d[4][7];
float *e[15];
struct {
    float f;
    int g;
} h;
int funcion(char,float,int);
```

se almacenarían de la siguiente manera en la tabla de tipos y en la tabla de símbolos⁸:

TABLA DE TIPOS

NÚM.	TIPO	D/T	T.BASE
0	ENTERO		
1	REAL		
2	CARÁCTER		
3	PUNTERO		1
4	PUNTERO		3
5	ARRAY	10	2
6	ARRAY	7	0
7	ARRAY	4	6
8	PUNTERO		1
9	ARRAY	15	8
10	REGISTRO		TS1
11	P.CART.	2	1
12	P.CART.	11	0
13	FUNCIÓN	12	0

TABLA DE SÍMBOLOS GLOBAL

NOMBRE	TIPO	DIRECCIÓN
a	0	100
b	4	102
c	5	104
d	7	114
e	9	170
h	10	200
funcion	13	-

TABLA DE SÍMBOLOS TS1

NOMBRE	TIPO	DIRECCIÓN
f	1	0
g	0	4

Observaciones:

- Como se puede observar en el caso de los punteros a **float** (entradas 3 y 8 en la tabla de tipos), cuando se va a añadir un tipo a la tabla de tipos no se suele perder el tiempo buscándolo, sino que directamente se añade al final.
- En los *arrays* se puede almacenar la dimensión del *array* (como en el ejemplo) o bien, si el lenguaje fuente fuera Pascal o alguno similar, el rango de valores posibles.
- En el caso de los registros, en la tabla de tipos se puede almacenar un puntero a la tabla de símbolos del registro, o bien la posición en la tabla de símbolos

⁸Para calcular las direcciones de memoria se ha supuesto que los caracteres ocupan una posición de memoria, los enteros y los punteros ocupan dos posiciones y los reales 4. Asimismo, se supone que las variables globales se almacenan a partir de la dirección 100.

específica (o en la global) en la que se almacenan los campos del registro, según la alternativa que se haya utilizado para almacenar los campos.

- Las funciones se almacenan guardando primero los productos cartesianos de sus argumentos (de izquierda a derecha), y después almacenando la función haciendo referencia al tipo de los argumentos y al tipo que devuelve.

◁

La tabla de tipos, como la tabla de símbolos, es una variable global del compilador y, por tanto, es necesario definir funciones para añadir tipos nuevos y para obtener información acerca de éstos.

Ejemplo 8.10

El ETDS de la figura 8.10 procesa declaraciones de variables simples y *arrays* en C, creando los tipos en la tabla de tipos y almacenando después las variables en la tabla de símbolos. Las funciones que utiliza este ETDS son:

```
int NuevoTipoArray(int dim,int tbase)
```

Crea un nuevo tipo *array* en la tabla de tipos (con *dim* y *tbase* como la dimensión y el tipo base del *array* respectivamente) y devuelve el número del nuevo tipo dentro de la tabla.

```
void GuardaSimb(char *nom,int tipo,int tam)
```

Almacena en la tabla de símbolos la variable de nombre *nom* con su tipo y su tamaño. Por simplificar, supondremos que esta función comprueba que la variable no esté repetida y, además, se encarga de asignar una dirección de memoria a la nueva variable al guardarla en la tabla de símbolos.

◁

8.7.2 Ámbitos y tabla de tipos

La gestión de la tabla de símbolos cuando el lenguaje fuente permite diferentes ámbitos de declaración se ha explicado en la sección correspondiente a la tabla de símbolos. Cuando se cierra el ámbito, el compilador elimina de la tabla de símbolos todos los que se hayan declarado dentro del ámbito (los *olvida*), puesto que dichos símbolos ya no son accesibles. De igual manera, los tipos que se han añadido a la tabla de tipos durante la compilación del ámbito se pueden también eliminar sin ningún problema (excepto quizá en la compilación de funciones, que estudiaremos en el capítulo siguiente).

8.7.3 Equivalencia de tipos

Las expresiones de tipos se utilizan para especificar de una manera independiente del programa fuente el tipo de una construcción. Además, la mayoría de los lenguajes permiten que el usuario defina sus propios tipos de datos⁹. Cuando el compilador debe

⁹Los identificadores utilizados por los usuarios para definir nuevos tipos se suelen almacenar en la tabla de símbolos o bien en una tabla especial (depende del lenguaje).

D	\longrightarrow	$D \text{ Var}$	
D	\longrightarrow	Var	
Var	\longrightarrow	Tipo	$\{ L.th := \text{Tipo}.t ; L.tamh := \text{Tipo}.tam \}$
		$L ;$	
Tipo	\longrightarrow	int	$\{ \text{Tipo}.t := ENTERO ; \text{Tipo}.tam := 2 \}$
Tipo	\longrightarrow	float	$\{ \text{Tipo}.t := REAL ; \text{Tipo}.tam := 4 \}$
Tipo	\longrightarrow	char	$\{ \text{Tipo}.t := CHARACTER ; \text{Tipo}.tam := 1 \}$
L	\longrightarrow	$L_1 ,$	$\{ L_1.th := L.th ; L_1.tamh := L.tamh \}$
		V	$\{ V.th := L.th ; V.tamh := L.tamh \}$
L	\longrightarrow	V	$\{ V.th := L.th ; V.tamh := L.tamh \}$
V	\longrightarrow	id	$\{ A.th := V.th ; A.tamh := V.tamh \}$
		A	$\{ \text{GuardaSimb}(\text{id.lexema}, A.tipo, A.tam) \}$
A	\longrightarrow	$[\text{nint}]$	$\{ A_1.th := A.th ; A_1.tamh := A.tamh \}$
		A_1	$\{ A.tipo := \text{NuevoTipoArray}(\text{nint.valex}, A_1.tipo);$
			$A.tam := A_1.tam * \text{nint.valex} \}$
A	\longrightarrow	ϵ	$\{ A.tipo := A.th ; A.tam := A.tamh \}$

Figura 8.10: Esquema de traducción para procesar declaraciones de variables simples y *arrays*.

comprobar si dos elementos tienen tipos iguales o equivalentes (en una asignación o en una llamada a una función, por ejemplo) dicha comprobación puede hacerse de dos maneras:

- con respecto al nombre de los tipos (*equivalencia de nombre*): dos tipos son considerados iguales o equivalentes si tienen el exactamente mismo nombre.

Ejemplo: Dadas las siguientes variables “a” y “b”,

```
int *a;
typedef int *punteroEntero;
punteroEntero b;
```

sus tipos no son equivalentes si el lenguaje utiliza la equivalencia de nombres, como hace por ejemplo Pascal.

- con respecto a la estructura de los tipos (*equivalencia estructural*): dos tipos son considerados iguales o equivalentes si tienen la misma estructura. En el ejemplo anterior los tipos de las variables “a” y “b” serían considerados iguales, aunque tuvieran distinto nombre (las dos son punteros a enteros). El lenguaje C utiliza este tipo de equivalencia.

8.8 Código intermedio para *arrays*

Además de procesar las declaraciones de *arrays* y crear los tipos correspondientes en la tabla de tipos, el compilador debe generar código intermedio para acceder a una posición

concreta de un *array*. Aunque las referencias a una posición de un *array* pueden aparecer como parte de una expresión o en la parte izquierda de una asignación, el código que debe generar el compilador en ambos casos es similar: primero generará código para obtener (en tiempo de ejecución) la dirección de memoria correspondiente a la posición del *array*, y después generará código para obtener el valor que hay almacenado en esa dirección (si la referencia aparece en una expresión) o bien para almacenar un valor en esa dirección.

Las referencias a posiciones de *arrays* están formadas por el nombre de la variable de tipo *array* y uno o más índices entre corchetes (o con una sintaxis similar); puesto que cualquier expresión (de tipo entero o convertida a tipo entero) puede ser el índice de un *array*, y dado que el compilador no siempre puede conocer el valor de una expresión antes de que se ejecute el programa objeto, el *front end* genera código para calcular la dirección de memoria asociada a la posición del *array*, independientemente del valor de los índices. El optimizador, en algunos casos (cuando detecta que los índices tienen un valor constante), puede ahorrar ese cálculo y sustituir todo el código que se genera en el *front end* por la dirección de memoria correspondiente.

Para explicar cómo se debe generar código para acceder a una posición de un *array*, vamos a estudiar el caso concreto de un *array* tridimensional (los cálculos y el código a generar son generalizables para *arrays* *n*-dimensionales). Dada una declaración de un *array* en C como la siguiente:

```
int a[D1][D2][D3];
```

donde D_1 , D_2 y D_3 son las dimensiones del *array*, si en el programa fuente aparece una referencia como:

```
a[i][j][k]
```

la dirección de memoria asociada a esa posición del *array* es:

$$\begin{aligned} \text{DirBase}(a) &+ i \times (D_2 \times D_3 \times \text{Tam}(\text{int})) \\ &+ j \times (D_3 \times \text{Tam}(\text{int})) \\ &+ k \times \text{Tam}(\text{int}) \end{aligned}$$

Todos los productos de las dimensiones del *array* son independientes de los índices y por tanto pueden realizarse en tiempo de compilación e incluso se pueden almacenar en la tabla de tipos. Es relativamente sencillo implementar esta fórmula en un compilador, pero existe una fórmula recursiva para realizar este cálculo que simplifica la generación de código. La implementación de dicha fórmula recursiva sería en este caso:

$$\begin{aligned} t_1 &:= 0 \\ t_2 &:= t_1 \times D_1 + i \\ t_3 &:= t_2 \times D_2 + j \\ t_4 &:= t_3 \times D_3 + k \\ t_5 &:= \text{DirBase}(a) + t_4 \times \text{Tam}(\text{int}) \end{aligned}$$

Como se puede observar, ambas formulaciones son equivalentes. En el caso base de la recursión se asigna un 0 a una variable temporal y en el caso general se multiplica la

temporal anterior por la dimensión correspondiente y se le suma el índice. Finalmente, una vez terminada la recursión, se multiplica la temporal resultante por el tamaño del tipo básico del *array* y se le suma la dirección base de comienzo del *array*.

De forma simultánea a este cálculo, el compilador debe realizar algunas comprobaciones semánticas:

1. Se debe comprobar que la subreferencia a la que se pone índice es efectivamente un *array*, es decir, que no se le ponen índices a una variable que no ha sido declarada como *array* ni se ponen más índices que los que necesita el *array*.
2. En la mayoría de los lenguajes, se exige además que una referencia a una posición de un *array* tenga todos los índices que necesita según su declaración, es decir, si el *array* se declara como tridimensional (como el caso que estamos tratando), el compilador debe comprobar que todas las referencias tienen exactamente 3 índices.
3. Los lenguajes del estilo de Pascal exigen que el índice sea de tipo entero (en C se convierte a entero si no lo es).
4. El lenguaje Pascal exige que el índice se encuentre dentro del rango de posibles valores de la declaración. Como se ha comentado en la sección anterior, esta comprobación solamente puede hacerse en tiempo de ejecución, por lo que un compilador de Pascal genera código objeto para realizar esta comprobación.

Ejemplo 8.11

El fragmento de ETDS de la figura 8.11 realiza algunas de estas comprobaciones semánticas¹⁰ a la vez que genera código **m2r**, utilizando la fórmula recursiva (que también es válida para referencias a variables que no sean de tipo *array*). Este ETDS debería además realizar comprobaciones semánticas y posiblemente conversiones de tipos en la regla de la asignación, pero no se muestran aquí por simplificar. Las funciones de la tabla de símbolos y la tabla de tipos que se utilizan en este ETDS son:

<code>int TipoSimb(char *nom)</code>	Devuelve el tipo almacenado en la tabla de símbolos para la variable de nombre <code>nom</code> .
<code>int DirSimb(char *nom)</code>	Devuelve la dirección de memoria almacenada en la tabla de símbolos para la variable <code>nom</code> .
<code>int DimTipoArray(int t)</code>	Devuelve la dimensión almacenada en la tabla de tipos para el tipo <i>array</i> <code>t</code> .
<code>int TBaseTipoArray(int t)</code>	Devuelve el tipo base guardado en la tabla de tipos para el tipo <i>array</i> <code>t</code> .
<code>int TAMTIPO(int t)</code>	Devuelve el tamaño del tipo (simple) <code>t</code> .

Suponiendo una variable declarada como “`int a[10][5]`” y una referencia como “`a[7][4]`”, la figura 8.12 muestra el árbol sintáctico con los valores de los atributos de los no terminales. En esta figura se supone que la dirección de “`a`” es 100

¹⁰La generación de código para comprobar que el índice se encuentra dentro del rango de valores posibles no se muestra en el ETDS para evitar que sea excesivamente largo y complejo.

y que las variables temporales empiezan en la dirección 1000. Un buen optimizador se daría cuenta que los valores de los índices son constantes y calcularía la dirección completa de la referencia, sustituyendo todo el código de *Factor* por una instrucción; a pesar de esto, el generador de código intermedio debe generar todo el código, puesto que confía en el optimizador para mejorarlo cuando sea posible (como en este caso).

<

Debe tenerse en cuenta que, si el lenguaje fuente es Pascal (o uno similar), los *arrays* se declaran con un rango de valores posibles, es decir, los índices no comienzan necesariamente en la posición 0 (como en C). Además, ambos tanto el límite inferior como el superior del rango están incluidos dentro del rango y son valores válidos. Para *arrays* declarados en Pascal, de la siguiente forma:

```
var a:array[LI1..LS1,LI2..LS2,LI3..LS3];
```

las fórmulas anteriores quedarían de la siguiente manera:

$$\begin{aligned} \text{DirBase}(a) &+ (i - LI_1) \times ((LS_2 - LI_2 + 1) \times (LS_3 - LI_3 + 1) \times \text{Tam}(\text{int})) \\ &+ (j - LI_2) \times ((LS_3 - LI_3 + 1) \times \text{Tam}(\text{int})) \\ &+ (k - LI_3) \times \text{Tam}(\text{int}) \end{aligned}$$

La fórmula recursiva sería:

$$\begin{aligned} t_1 &:= 0 \\ t_2 &:= t_1 \times (LS_1 - LI_1 + 1) + (i - LI_1) \\ t_3 &:= t_2 \times (LS_2 - LI_2 + 1) + (j - LI_2) \\ t_4 &:= t_3 \times (LS_3 - LI_3 + 1) + (k - LI_3) \\ t_5 &:= \text{DirBase}(a) + t_4 \times \text{Tam}(\text{int}) \end{aligned}$$

8.9 Código intermedio para registros

Un lenguaje fuente que permitiera solamente tipos simples y registros no plantearía un gran problema para el compilador, ya que la dirección de cualquier campo de cualquier registro se podría calcular en tiempo de compilación simplemente sumando la dirección base del registro con el desplazamiento relativo del campo. Por tanto, el código intermedio para acceder a un campo de un registro sería exactamente el mismo que habría que generar para acceder a una variable de tipo simple.

Sin embargo, lo normal es que un lenguaje fuente permita tanto *arrays* como registros, lo cual implica que no se puede calcular la dirección de un campo (en tiempo de compilación) en todos los casos, por lo que el *front end* generaría código para acceder tanto a referencias de variables simples, como a campos de registros y a posiciones de *arrays*.

En el ETDS de la figura 8.11, se utilizan dos atributos (*tmp* y *cod*) en las reglas de *Ref* para calcular el desplazamiento dentro del *array*, mientras que se utiliza otro atributo para la dirección base, que siempre es constante (se obtiene de la tabla de

```

Ref      →  id      { Ref.tipo := TipoSimb(id.lexema)
                    Ref.dbase := DirSimb(id.lexema)
                    tmp := NuevaTemporal(); Ref.tmp := tmp
                    Ref.cod := "mov #0 "||tmp }

Ref      →  Ref1 [ { si no EsArray(Ref1.tipo) entonces
                    ErrorSemántico(...)
                    fsi }
                E ] { si no EsEntero(E.tipo) entonces
                    ErrorSemántico(...)
                    si no
                        Ref.tipo := TBaseTipoArray(Ref1.tipo)
                        Ref.dbase := Ref1.dbase
                        tmp := NuevaTemporal(); Ref.tmp := tmp
                        Ref.cod := Ref1.cod||E.cod||
                            "mov "||Ref1.tmp||"A"||
                            "mul #0||DimTipoArray(Ref1.tipo)||
                            "addi "||E.tmp||
                            "mov A"||tmp
                        fsi }

Factor   →  Ref      { si EsArray(Ref.tipo) entonces
                    ErrorSemántico(...)
                    si no
                        Factor.tipo := Ref.tipo
                        tmp := NuevaTemporal(); Factor.tmp := tmp
                        Factor.cod := Ref.cod||"mov "||Ref.tmp||"A"||
                            "mul #0||TAMTIPO(Ref.tipo)||
                            "addi #0||Ref.dbase||
                            "mov @A"||tmp
                        fsi }

Instr    →  Ref :=   { si EsArray(Ref.tipo) entonces
                    ErrorSemántico(...)
                    fsi }
                E      { /* comprobaciones semanticas con Ref y E */
                    Instr.cod := Ref.cod||E.cod||
                        "mov "||Ref.tmp||"A"||
                        "mul #0||TAMTIPO(Ref.tipo)||
                        "addi #0||Ref.dbase||
                        "mov "||E.tmp||"@A"
                    }

```

Figura 8.11: Esquema de traducción que genera código m2r para acceder a una posición de un *array*.

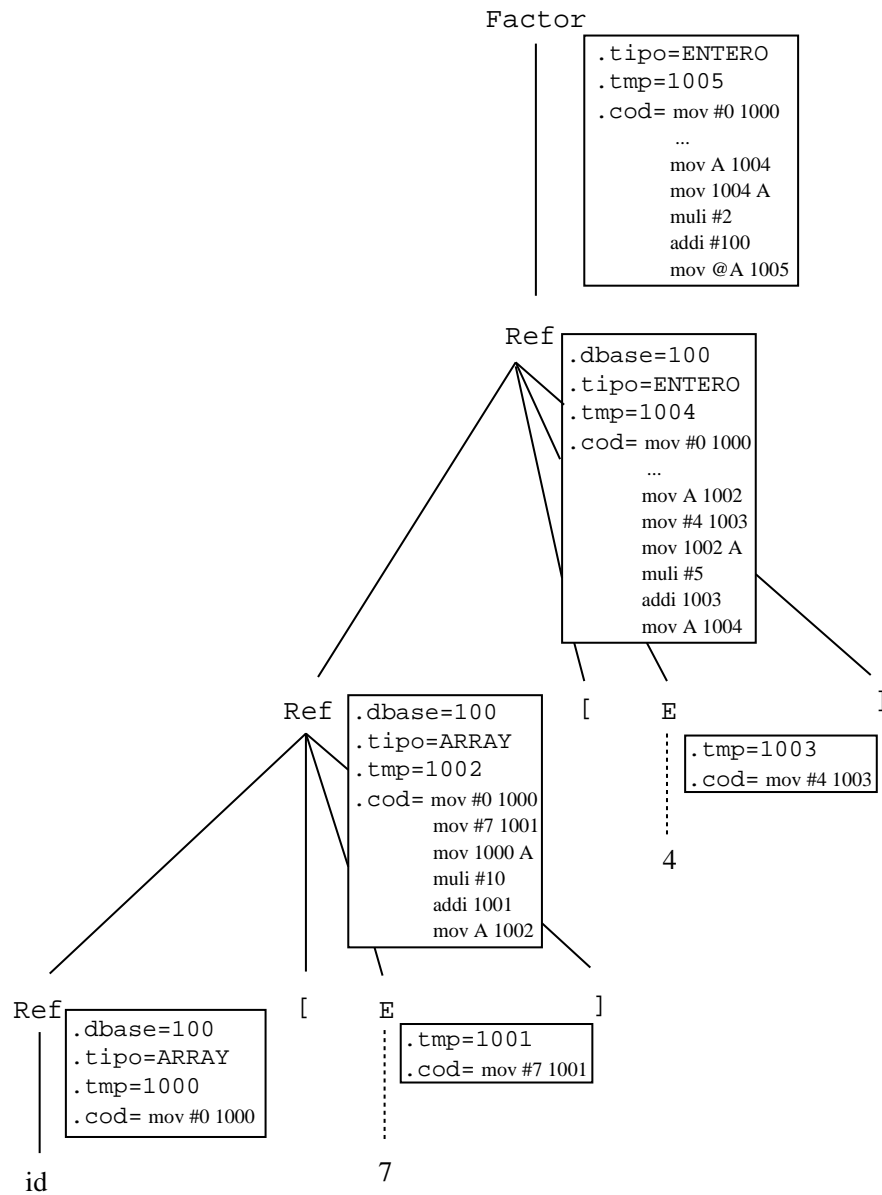


Figura 8.12: Árbol sintáctico con los valores de los atributos para “a[7][4]”.

símbolos). Si el lenguaje fuente permite también los registros, el principal cambio en dicho ETDS sería que la dirección base dejaría de ser una constante (un *array* puede ser un campo de un registro, y se pueden declarar *arrays* de registros), por lo que se debe generar código para calcular la dirección base en tiempo de ejecución. Esto implicaría sustituir el atributo que contiene la dirección base por dos atributos (uno para el código y otro para una variable temporal). Además, habría que añadir al ETDS una regla para permitir las referencias a campos de registros:

$$Ref \longrightarrow Ref . id$$

Ejemplo 8.12

Dada la siguiente declaración en C

```
struct {
    int a,b;
    char c[10];
} d[5];
```

y dada la referencia

```
d[i].c[j]
```

suponiendo que los enteros ocupan dos posiciones de memoria y los caracteres una, y suponiendo que el *array* “d” está almacenado a partir de la dirección de memoria 100, los valores de los atributos de *Ref* para las distintas subreferencias que componen la referencia deberían ser (utilizando un esquema similar al de la figura 8.11) los que se muestran en la figura 8.13. En las reglas del *Factor* y de las instrucciones se debe generar código para hacer la suma del desplazamiento con la dirección base.

REFERENCIA	ATRIBUTOS DE <i>Ref</i>
d	<pre> .tipo = 4 /* array(0..4, registro(...)) */ .tamaño = 70 .dbase = t₁ .cbase = mov #100 t₁ .tmp = t₂ .cod = mov #0 t₂ </pre>
d[i]	<pre> .tipo = 3 /* registro(...) */ .tamaño = 14 .dbase = t₁ .cbase = mov #100 t₁ .tmp = t₃ .cod = mov #0 t₂ mov t₂ A muli #5 /* dimension del array */ addi E.tmp /* E ≡ i */ mov A t₃ </pre>
d[i].c	<pre> .tipo = 2 /* array(0..9, char) */ .tamaño = 10 .dbase = t₄ .cbase = mov #100 t₁ mov #0 t₂ mov t₂ A muli #5 addi E.tmp mov A t₃ mov t₃ A muli #14 /* tamaño del registro */ addi t₁ addi #4 /* offset de c */ mov A t₄ .tmp = t₅ .cod = mov #0 t₅ </pre>
d[i].c[j]	<pre> .tipo = 1 /* char */ .tamaño = 1 .dbase = t₄ .cbase =tmp = t₆ .cod = mov #0 t₅ mov t₅ A muli #10 /* dimension del array */ addi E.tmp /* E ≡ j */ mov A t₆ </pre>

Figura 8.13: Generación de código intermedio para *arrays* y registros.

8.10 Referencias bibliográficas

REFERENCIAS	EPÍGRAFES
[Louden, 1997]	6.3, 3.3, 8.1, 6.4, 8.1, 8.2.1, 8.4, 6.4.1, 6.4.3 y 8.3
[Aho, Sethi y Ullman, 1990]	2.7, 7.6, 2.8, 8.1, 6.1, 6.2, 6.3, 6.4, 8.3, 8.4 y 8.5
[Bennett, 1990]	5.1.1, 9.1 y 10.3.3
[Fischer y LeBlanc, 1991]	8.3, 7.3, 14.2, 8.4.1, 11.2.3 y 11.3.1

8.11 Ejercicios

Ejercicio 8.1

Dada la siguiente declaración de variables en C:

```
struct {
    int a;
    float b[10][5];
    char **c[4];
    struct {
        float a;
        int *c;
    } d,e;
} f,g;
```

1. Escribanse las expresiones de tipos correspondientes a las variables declaradas.
2. Describanse brevemente los contenidos de la tabla de tipos y de la tabla de símbolos global.
3. Suponiendo que los tipos básicos (`int`, `float` y `char`) ocupan todos lo mismo (una posición de memoria), y que los punteros ocupan también una posición de memoria, ¿cuál sería la dirección de `g.e.a` si la dirección donde comienza `f` es la 100? ¿Es posible calcular esta dirección durante la compilación? ¿Y la de `g.b[f.a][g.a]`?

Ejercicio 8.2

Dada la siguiente declaración de variables en C:

```
int a,*b[10];
int principal(int numargs,char **argumentos);
struct {
    float c[4][5];
    struct {
        float x,y;
    } puntos[15];
} d,f;
```

y suponiendo que los tipos básicos y los punteros ocupan sólo una posición de memoria,

1. Escribanse las expresiones de tipos de cada uno de los símbolos que aparecen en esas declaraciones.
2. Dibújese la tabla de tipos y la tabla de símbolos en el estado en que quedarían después de procesar las declaraciones.

3. Explíquese cómo se calcularía la posición de memoria en la que está `d.puntos[7].y`.
4. ¿Puede el compilador calcular *en todos los casos* esa posición de memoria u otras con distinto índice para `puntos`? ¿por qué?

Ejercicio 8.3

Escríbase un ETDS que realice la siguiente traducción:

Lenguaje fuente: expresiones aritméticas con enteros, reales, identificadores de variables *predeclaradas*, los operadores “+”, “-”, “(”, “)”, “[” i “]”, donde los operadores tienen la misma precedencia y asociatividad que en C o en Pascal (excepto los corchetes, que funcionan como los paréntesis pero entregan la parte entera de la expresión que engloban; por ejemplo, `[3.5+1]` vale 4). La gramática que describe este lenguaje es muy similar a las utilizadas en los ejemplos de este y otros capítulos.

Lenguaje objeto: un lenguaje basado en enteros, reales, identificadores y las funciones siguientes:

<code>int si(int, int)</code>	suma entera
<code>int ri(int, int)</code>	resta entera
<code>float sr(float, float)</code>	suma real
<code>float rr(float, float)</code>	resta real
<code>int t(float)</code>	truncamiento a parte entera
<code>float r(int)</code>	conversión de entero a real

Al contrario que en el lenguaje fuente, en el lenguaje objeto no se permiten las conversiones implícitas de tipos, y por tanto los parámetros de estas funciones deben ser del tipo indicado en el prototipo¹¹.

Restricciones semánticas: Cuando un operador encuentra dos operandos enteros, el resultado es entero; si alguno de los operandos es real, el resultado tiene que ser real.

Ejemplos: si *a* es real y *b* es entero, la siguiente tabla muestra las traducciones de algunas expresiones:

<code>a+b-3.5</code>	<code>rr(sr(a,r(b)),3.5)</code>
<code>a+[b-3.5]</code>	<code>sr(a,r(t(rr(r(b),3.5))))</code>

Ejercicio 8.4

Diséñese un ETDS que genere código objeto `m2r` a partir de un lenguaje fuente con *arrays* que se declaran y utilizan como en el lenguaje Pascal.

La parte del ETDS que se encarga de crear los tipos en la tabla de tipos y de almacenar los símbolos en la tabla de símbolos se debe suponer que ya está hecha. Solamente se debe hacer la parte del ETDS que genera código para acceder a las posiciones de un *array* o variable simple.

¹¹Es decir, no se puede hacer una llamada como `sr(si(3,4.5),3.0)`.

Ejemplo: Suponiendo las siguientes declaraciones de variables globales,

```
var
  entero : a,b;
  tabla 4..7 de real : ar;
  tabla 10..11 de tabla 1..6 de entero : abe;
fvar
```

la parte del ETDS encargada de procesar las declaraciones almacenaría los tipos y los símbolos en las tablas correspondientes, que quedarían de esta manera:

TABLA DE TIPOS					
POSICIÓN	TIPO	TIPO BÁSICO	LÍMITE INFERIOR	LÍMITE SUPERIOR	TAMAÑO
0	ENTERO	-	-	-	-
1	REAL	-	-	-	-
2	ARRAY	1	4	7	4
3	ARRAY	0	1	6	6
4	ARRAY	3	10	11	2

TABLA DE SÍMBOLOS			
POSICIÓN	NOMBRE	TIPO	DIRECCIÓN
0	a	0	0
1	b	0	1
2	ar	2	2
3	abe	4	6

En este ETDS se supone que los tipos básicos (entero y real) ocupan solamente una posición de memoria, y debe tenerse en cuenta que en las tablas tanto el límite inferior como el superior están incluidos en el rango de valores posibles. Las funciones para acceder a los valores de las tablas de tipos y símbolos son:

```
booleano TT_EsArray(entero:posicion)
entero TT_TipoBasico(entero:posicion)
entero TT_LimiteInferior(entero:posicion)
entero TT_LimiteSuperior(entero:posicion)
entero TT_Tamanyo(entero:posicion)
entero TS_BuscaId(cadena:identificador) // devuelve la posicion o -1
                                         si no se ha declarado
entero TS_TipoId(cadena:identificador)
entero TS_DireccionId(cadena:identificador)
```

El fragmento de ETDS que se debe hacer es el asociado a las siguientes reglas:

$$\begin{aligned}
 Instr &\longrightarrow Ref \mathbf{assop} Expr \\
 Factor &\longrightarrow Ref \\
 Ref &\longrightarrow \mathbf{id} \\
 Ref &\longrightarrow Ref \mathbf{lcor} E\mathbf{simple} \mathbf{rcor}
 \end{aligned}$$

Se supone que *Factor*, *Esimple* y *Expr* tienen (al menos) tres atributos sintetizados: **cod**, **tipo** y **tmp**, con el mismo significado que en los ejemplos de este capítulo. Las restricciones de tipos son las mismas que en Pascal y, aunque un compilador de Pascal genera código para comprobar que el índice de un *array* está dentro del rango permitido, en este ETDS no se debe hacer.

Capítulo 9

Compilación de subprogramas

9.1 Introducción

En el capítulo anterior se explica cómo se implementan en un compilador elementos tan importantes como variables, instrucciones de control, registros o arrays. Hay, sin embargo, un aspecto que aún no hemos tratado explícitamente y que constituye una parte fundamental de muchos compiladores: el *entorno de ejecución*.¹ Aunque este entorno condiciona elementos tan importantes de muchos lenguajes como la gestión de la memoria dinámica o la implementación de ámbitos, en este capítulo lo abordaremos únicamente desde el punto de vista de su aplicación a la gestión de funciones y procedimientos.

El entorno de ejecución, en cualquier caso, no es nuevo para nosotros. Hasta ahora, hemos asumido la existencia de un determinado entorno de ejecución; la novedad es que su gestión se complica cuando queremos que el lenguaje permita el uso de funciones y procedimientos.

Antes de continuar, debemos señalar que en lo sucesivo utilizaremos el término *función* para referirnos en general tanto a *procedimientos* como a *funciones*. Un procedimiento puede considerarse como una forma sencilla de función en la que no se devuelve valor alguno. La adaptación de lo discutido aquí para funciones al caso de los procedimientos es, por lo tanto, inmediata.

El diseñador de un compilador debe plantearse tres aspectos fundamentales del diseño del esquema de tratamiento de funciones: la organización de la memoria en tiempo de ejecución (normalmente impuesta por el lenguaje), la compilación del cuerpo de las funciones, y la compilación de las llamadas a funciones. Lo veremos en este capítulo.

9.1.1 El entorno de ejecución

Los libros de programación suelen distinguir dos zonas en el entorno de ejecución de un programa: la zona de *datos* y la zona de *código*; esta última suele permanecer inalterada durante la ejecución y, por ello, es posible conocer en tiempo de compilación la dirección

¹El entorno de ejecución (*runtime environment*) viene dado por la estructura de memoria (incluyendo los registros de la unidad central de proceso, si los hubiera) y la forma de gestionarla que permite desarrollar adecuadamente el proceso de ejecución de un programa.

de cada instrucción del código ejecutable, en especial el punto de comienzo de cada función.

Lo anterior no suele ser del todo cierto para la zona de datos: pensemos en un lenguaje como C en el que se pueden definir funciones recursivas con variables locales; en un determinado instante de la ejecución de un programa pueden coexistir varias *instancias* de una misma función y cada una de ellas debe almacenar sus argumentos y sus variables locales en una posición diferente de la memoria. No hay, en consecuencia, una relación biunívoca entre variables y posiciones de memoria en este caso: los únicos datos que tienen una posición única en memoria son las variables globales y las estáticas.

Los entornos de ejecución utilizados en la gran mayoría de los lenguajes de programación pueden clasificarse en las tres siguientes categorías:

- **Entornos completamente estáticos** en los que la dirección exacta de cada dato se conoce en tiempo de compilación;² FORTRAN 77, por ejemplo, utiliza un entorno de este tipo.
- **Entornos basados en pila**, requeridos por lenguajes como C, C++, Pascal o Ada, en los que los datos se almacenan en una pila en memoria; a menudo, la arquitectura de la máquina objeto facilita el uso de esta pila mediante instrucciones o registros especiales.
- **Entornos completamente dinámicos** en los que tanto el código como los datos pueden modificarse durante la ejecución del programa; esto permite a lenguajes (normalmente interpretados) como Lisp o Prolog la generación dinámica de funciones y la modificación de las existentes durante la ejecución del programa.

Evidentemente, también son posibles enfoques híbridos, que aúnen características de más de una de las categorías anteriores.

Un entorno completamente estático no permite, en general, la definición de funciones recursivas debido a varias razones, entra las que está la imposibilidad de determinar de antemano cuántas instancias de cada función pueden coexistir durante la ejecución del programa para, de esta forma, reservar espacio suficiente en memoria para los símbolos utilizados en cada una de ellas (variables locales, variables temporales y parámetros de la función); además, al compilar el cuerpo de la función no sabríamos qué dirección asignar a los símbolos locales, puesto que es posible que se utilicen en varias instancias de la función. Un entorno basado en pila, por otro lado, sí que permite la ejecución de funciones recursivas, ya que cada vez que se llama a una función se reserva espacio en la pila para sus variables, espacio que se libera adecuadamente cuando la función termina.³

Aunque la política de gestión de una pila es adecuada para la reserva de memoria de las variables asociadas a una función, no lo es para el caso de la gestión de la *memoria dinámica*, un aspecto de los lenguajes de programación que no trataremos en este libro.

²Los sistemas operativos actuales exigen normalmente que los programas del usuario sean *reubicables*, es decir, que todas las direcciones de memoria sean relativas a una *dirección base* asignada por el sistema operativo antes de cada ejecución. Un entorno completamente estático no es incompatible con este modelo: en este caso, son los desplazamientos relativos los que se conocen en tiempo de compilación.

³Un aspecto de algunos lenguajes que complica ligeramente la implementación de un entorno basado en pila es el relacionado con la declaración anidada de funciones; lo trataremos más adelante.

valor devuelto
dirección de retorno
...
parámetros
⋮
datos locales
⋮
temporales locales
⋮

Figura 9.1: Modelo general de registro de activación (RA). El orden de distribución de los distintos elementos en el RA puede variar de un compilador a otro.

Nótese cómo en este caso no hay un orden preestablecido entre la memoria reservada dinámicamente y su liberación; normalmente se reserva una zona especial denominada *montículo* (*heap*)⁴ para los bloques de memoria reservados dinámicamente.

9.1.2 Registro de activación

Con lo discutido anteriormente queda claro que hay un conjunto de datos asociados a cada función⁵ que deben ser guardados en algún lugar de la memoria. Estos datos incluyen: los parámetros de la función, las variables locales y temporales, el valor devuelto, y unos cuantos valores adicionales que deben salvaguardarse antes de que la función se ejecute y restaurarse a su finalización; dentro de este último grupo es de especial importancia la *dirección de retorno*, que indica la dirección de la instrucción siguiente a la instrucción de salto que llama a la función y a la que debe hacerse un salto al terminar esta para que la ejecución continúe tras el punto de llamada. Se denomina *registro de activación* (RA) (*activation record*) a la estructura en memoria que guarda estos datos. Un ejemplo de RA puede verse en la figura 9.1.

Algunas zonas del RA tienen el mismo tamaño para todas las funciones; otras, como las reservadas a parámetros, variables locales y temporales, pueden tener tamaños diferentes para cada función. El RA se almacena de manera distinta según la organización del entorno de ejecución del lenguaje: así, el RA se guarda en el área estática en FORTRAN 77; en la pila en C o Pascal; y en el montículo en el caso de lenguajes como Lisp. Cuando el RA se guarda en la pila de ejecución (el caso más común), también se le conoce como *marco de pila* (*stack frame*).

9.1.3 Secuencias de llamada y retorno

Una parte importante en el diseño de un entorno de ejecución relacionada con el procesamiento de funciones es la especificación de la secuencia de pasos que se producen cuando se invoca una función y el orden en que estos se ejecutan. Esta serie de pasos

⁴No confundir con el tipo abstracto de datos homónimo con el que no guarda ninguna relación.

⁵En general, a cada *llamada a función*, pero en eso entraremos más adelante.

tienen lugar siempre, independientemente del contenido de la función correspondiente, y es habitual dividirlos en dos categorías, según el momento en el que tienen lugar: secuencia de llamada (*call sequence*) y secuencia de retorno (*return sequence*).

La *secuencia de llamada* se produce antes de la ejecución del código del cuerpo de la función y es responsable de cosas como la reserva de memoria para el RA, el almacenamiento de la dirección de retorno, el cálculo y almacenamiento de los parámetros, y la asignación de valores a los registros necesarios.⁶ La *secuencia de retorno* se produce cuando la función acaba y se encarga, entre otras cosas, de almacenar en el lugar adecuado el valor devuelto por la función y devolver el control al punto de llamada.

Determinadas tareas de ambas secuencias pueden realizarse tanto en el llamador como en la función invocada. La implementación en la parte del llamador, sin embargo, sobrecarga excesivamente el tamaño del código objeto al repetirse el mismo código en cada punto de invocación a una función.

9.1.4 Comprobaciones semánticas

Hay una serie de comprobaciones semánticas que deben realizarse en el ETDS cuando trabajamos con funciones:

- Las comprobaciones básicas son asegurar que no se utilizan paréntesis con identificadores que no son funciones y, a la inversa, que no se utilizan como variables identificadores que corresponden a funciones. Algunos lenguajes como C pueden relajar este último requisito.
- Cuando se invoca una función, el número de argumentos ha de coincidir con el indicado en su declaración. De nuevo, en algunos lenguajes como C esto puede no ser obligatorio.
- Según la especificación del lenguaje en lo referente al tratamiento de los tipos, debe comprobarse la equivalencia entre los tipos declarados para los argumentos de la función y los utilizados en la invocación y realizar, en su caso, las conversiones de tipo adecuadas. Esta conversión también deberá realizarse en el caso del valor devuelto por la función.

9.1.5 Implementación con m2r

A continuación se muestra el código de la máquina virtual m2r que implementa algunos de los pasos de las secuencias de llamada y retorno, asumiendo un entorno de ejecución estático como el que se discutirá en la sección 9.2

El comienzo del código objeto de cada función en memoria viene indicado por una etiqueta. Supongamos que estamos compilando el programa en C de la figura 9.2, que define dos funciones, g1 y g2, y el cuerpo principal⁷ o punto de entrada del programa.

⁶En determinadas arquitecturas es habitual que en la secuencia de llamada se salvaguarde también el contenido de todos los registros de la unidad central de proceso; el repertorio de instrucciones suele tener una instrucción para hacerlo.

⁷Los compiladores de C tratan la función *main* como una función más; en este capítulo, sin embargo, la consideraremos como un programa principal a la manera de Pascal.

```

int a;
int g1 ()
{
    return 10;
}
int g2 (int a,int b)
{
    int x,y,z;
    return a*b;
}
main ()
{
    a=g2(3,5);
}

```

Figura 9.2: Programa fuente en C utilizado en los ejemplos del apartado 9.1.5.

	jmp L3
L1	código de g1
	⋮
L2	código de g2
	⋮
L3	programa principal
	⋮

Figura 9.3: La instrucción de salto al comienzo del código permite saltar sobre el código de las funciones y acceder al programa principal.

La distribución del código en memoria es la que se muestra en la figura 9.3. Como puede verse, es necesario que la primera instrucción del código objeto salte sobre la zona de código de las funciones hasta la etiqueta donde comienza el programa principal.

Si los RA siguen el formato indicado en la figura 9.1 y suponemos que el RA de la función g2 comienza en la posición de memoria 100, la dirección de retorno tendrá que ser almacenada por el llamador (dentro de la función main) en la posición 101, como puede verse en la figura 9.4. El código que ha de ejecutar el llamador, por tanto, será similar a:

```

mvetq L4 101
jmp L2
L4 ...

```

El código que sigue a L4 se encargará de recoger del lugar correspondiente del RA el valor devuelto por la función y almacenarlo donde corresponda; en este caso, en la dirección de la variable global a que, por ahora, indicaremos mediante d(a):

```

L4 mov 100 d(a)

```


100	valor devuelto
101	dirección de retorno
102	parámetro a
	parámetro b
104	variable local x
	variable local y
	variable local z
107	temporales locales
	⋮

Figura 9.4: Ejemplo de RA para la función `g2` del código de la figura 9.2. Se asume que todos los datos ocupan una posición de memoria, independientemente de su tipo.

El código de la función `g2` se encargará de multiplicar el valor de los parámetros `a` y `b`, guardar el resultado en una temporal (las temporales se sitúan a partir de la dirección de memoria 107),

```

mov 102 107
mov 103 108
mov 107 A
mul 108
mov A 109

```

y, finalmente, como parte de la secuencia de retorno, copiar el valor de esa temporal en la zona del RA reservada para el valor devuelto y saltar a la etiqueta de retorno (que, en este caso, es `L4` y está almacenada en el RA en la dirección 101):

```

mov 109 100
mov 101 A
jmp @A

```

9.1.6 Valor devuelto

La manera de determinar el valor devuelto por una función depende del lenguaje de programación considerado. Así, C utiliza la instrucción `return` seguida de la expresión cuyo valor se ha de devolver (si lo hay); por otro lado, lenguajes como Pascal consideran que el valor a devolver es el último asignado a una variable cuyo nombre coincide con el de la función (variable que se declara implícitamente). El siguiente código muestra cómo quedaría en Pascal una función equivalente a la función `g2` de la figura 9.2:

```

function g2 (a: integer; b:integer) : integer;
var x,y,z: integer;
begin
  g2 := a*b
end

```

ENTRADA	TIPO	T. BASE/ ARG.	ARG./ RET.	
0	entero			
1	real			
2	carácter			
3	prod. cartesiano	0	2	$\text{int} \times \text{float}$
4	prod. cartesiano	3	1	$(\text{int} \times \text{float}) \times \text{char}$
5	función	4	1	$[(\text{int} \times \text{float}) \times \text{char}] \rightarrow \text{float}$

Figura 9.5: Tabla de tipos para la función `foo` del apartado 9.1.7.

9.1.7 Gestión de la tabla de tipos

El tipo de una función viene definido por el tipo de cada uno de sus argumentos junto al tipo del valor devuelto. Siguiendo con la notación definida en el capítulo anterior para caracterizar los tipos de datos, una función definida como

```
float foo (int i,float f,char c)
```

se representaría mediante la expresión de tipos

```
int × float × char → float
```

La tabla de tipos almacena el tipo de los argumentos como una sucesión de productos cartesianos, como puede estudiarse en la figura 9.5. El campo *Tipo base/Argumento* indica la entrada anterior del producto cartesiano o el tipo del primer argumento; el campo *Argumento/Retorno* guarda el tipo de los argumentos posteriores y, en el caso de la última entrada, el tipo del valor devuelto por la función.

La tabla de símbolos contendrá en este caso una entrada para `foo` con tipo 5, que se corresponde con el registro número 5 de la tabla de tipos de la figura 9.5. Además, un nuevo campo de la tabla de símbolos servirá para almacenar la dirección o etiqueta de comienzo del código de cada función.

La política para tratar las variables locales de funciones en la tabla de símbolos es muy parecida a la mostrada en el capítulo anterior para el tratamiento de las declaraciones de variables dentro de ámbitos: los argumentos y variables locales de cada función se van guardando en la tabla de símbolos conforme se procesa su declaración y el espacio que ocupan se recupera cuando finaliza la compilación de la función; más adelante se muestra un ejemplo de esto.

9.1.8 Parámetros y argumentos

Aunque en muchos casos se puede usar indistintamente un término u otro, conviene mencionar aquí que un argumento y un parámetro no son exactamente lo mismo. Los *argumentos* forman parte de la definición de la función, mientras que los *parámetros* son los valores reales indicados para ellos en la llamada a la función. La función `foo` del apartado 9.1.7 tiene tres argumentos y una llamada como `foo(0,0,'a')` tiene tres parámetros.

```
int a;
int f1 (int b)
{
    int c;
    c=b*b;
    return c;
}
int b;
int f2 (int a,int b)
{
    int c;
    c=f1(a)+f1(b);
    return c;
}
int c;
main ()
{
    a=7;
    a=f2(a,3);
}
```

Figura 9.6: Programa fuente en C utilizado en el apartado 9.2.

9.2 Funciones sin recursividad

Un entorno de ejecución completamente estático es adecuado para lenguajes que no permiten punteros, ni reserva dinámica de memoria, ni (lo que aquí nos importa) *recursividad* en las funciones.⁸ La memoria de datos se organiza en este caso de manera que se reserva una zona para los datos globales⁹ y una zona para el RA de cada función definida en el programa. Un programa fuente suele presentar intercaladas declaraciones de variables globales y de funciones, como ocurre en el programa de la figura 9.6. Si la compilación se hace en una sola pasada, lo normal es que también aparezcan intercalados en memoria espacios para los datos globales y para los RA correspondientes; la figura 9.7 muestra la disposición en memoria de estos elementos.

El código objeto generado para el programa de la figura 9.6 y la máquina m2r se muestra profusamente comentado en la figura 9.8. Con todo lo discutido hasta aquí en este capítulo y con ayuda de la figura 9.9 (una versión más detallada de la figura 9.7), este código no debería plantear grandes dificultades para su comprensión.

Dado que las declaraciones se suelen procesar en el orden en el que se definen en el programa fuente, es posible conocer cuando se termina de compilar una función el espacio requerido para sus temporales, que corresponderá con el máximo número de

⁸Cuando en este capítulo nos refiramos de forma general a *recursividad*, lo haremos tanto en el sentido de recursividad *directa* (una función se llama a sí misma), como en el de recursividad *indirecta* (una secuencia de llamadas entre distintas funciones termina generando una llamada a la función inicial).

⁹En el lenguaje C los datos globales son las variables globales propiamente dichas y las variables de tipo *static*.

variable global a
RA de f1
⋮
variable global b
RA de f2
⋮
variable global c

Figura 9.7: Distribución en memoria del programa de la figura 9.6 en un entorno completamente estático.

temporales utilizadas en una instrucción. Este dato se usa para determinar el espacio ocupado por el RA de la función correspondiente.

Solo nos queda estudiar la gestión de la tabla de símbolos durante la compilación del programa anterior. Como ya se ha dicho, el cuerpo de la función se considera como un ámbito y los argumentos de la función y sus variables locales se incluyen en la tabla de símbolos mediante una política idéntica a la estudiada en el capítulo anterior para el manejo de ámbitos: se insertan en la tabla de símbolos al iniciar la compilación de la función y se eliminan cuando esta finaliza. La entrada correspondiente a la función en sí, sin embargo, no se elimina nunca, ya que la función puede invocarse desde cualquier punto del programa.

La figura 9.10 muestra la configuración de la tabla de símbolos en tres momentos distintos de la compilación del programa de la figura 9.6: cuando se están compilando las instrucciones de la función `f1`, cuando se están compilando las de la función `f2` y, finalmente, cuando se está compilando el programa principal.¹⁰ En el caso de un entorno de ejecución completamente estático los datos que deben almacenarse en la tabla de símbolos para una función son su nombre, su entrada asociada en la tabla de tipos, la dirección de comienzo de su RA en memoria y la dirección o etiqueta de comienzo del código objeto de la función.

A modo de resumen indicaremos a continuación los principales pasos a seguir para compilar la declaración y la llamada a una función.

9.2.1 Compilación del cuerpo de la función

Podemos resumir los aspectos a tener en cuenta al compilar el cuerpo de una función en un entorno completamente estático como sigue:

- a) En relación a la tabla de símbolos: insertar las entradas adecuadas para la función¹¹ en la tabla de tipos y en la tabla de símbolos; salvaguardar la posición de comienzo en la tabla de símbolos de las declaraciones locales a la función; insertar en la tabla

¹⁰En el caso de que el lenguaje permita un único tipo simple para las variables, no es necesario utilizar la tabla de tipos; la tabla de símbolos puede guardar simplemente el número de argumentos de la función, la dirección de comienzo de su RA y la etiqueta de comienzo de su código.

¹¹El tipo de la función no se puede construir hasta que se han analizado los argumentos y se conoce su tipo.

```

        jmp L1      ; salta al programa principal
; ----- código de f1:
L2  mov 3 5      ; t1=b
    mov 3 6      ; t2=b
    mov 5 A
    muli 6       ; b*b
    mov A 7      ; t3=b*b
    mov 7 4      ; c=t3
    mov 4 5      ; t1=c
    mov 5 2      ; valor devuelto=t1
    mov 1 A
    jmp @A       ; return c
; ----- código de f2:
L3  mov 11 14    ; t1=a
    mov 14 3     ; parámetro b de f1=t1
    mvetq L4 1   ; guarda etiqueta de retorno
    jmp L2      ; salta al código de f1
L4  mov 2 15     ; t2=valor devuelto por f1
    mov 12 16    ; t3=b
    mov 16 3     ; parámetro b de f1=t3
    mvetq L5 1   ; guarda etiqueta de retorno
    jmp L2      ; salta al código de f1
L5  mov 2 17     ; t4=valor devuelto por f1
    mov 15 A
    addi 17     ; f1(a)+f1(b)
    mov A 18    ; t5=f1(a)+f1(b)
    mov 18 13   ; c=t5
    mov 13 14   ; t1=c
    mov 14 10   ; valor devuelto=t1
    mov 9 A
    jmp @A      ; return c
; ----- código del programa principal:
L1  mov #7 15000 ; t1=7
    mov 15000 0  ; a=7
    mov 0 15000  ; t1=a
    mov 15000 11 ; parámetro a de f2=t1
    mov #3 15001 ; t2=3
    mov 15001 12 ; parámetro b de f2=t2
    mvetq L6 9   ; guarda etiqueta de retorno
    jmp L3       ; salta al código de f2
L6  mov 10 15002 ; t3=valor devuelto por f2
    mov 15002 0  ; a=f2(a,3)
    halt

```

Figura 9.8: Código objeto para m2r del programa de la figura 9.6 bajo un entorno de ejecución estático.

de símbolos los argumentos de la función y las variables locales conforme estas se declaren; al terminar de compilar la función, restaurar la posición libre de la tabla de símbolos al valor guardado al comienzo de su compilación.

- b) Generar código para copiar el valor devuelto por la función a la posición adecuada del RA; comprobar, si corresponde, que el tipo del valor devuelto coincide con el

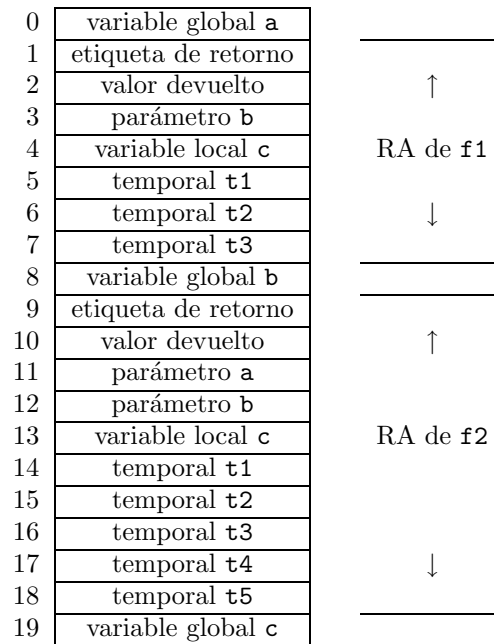


Figura 9.9: Versión detallada del mapa de memoria de la figura 9.7.

indicado en el prototipo de la función y generar las conversiones de tipo adecuadas en caso contrario.

- c) Generar código para devolver el control al llamador, tomando la dirección de retorno del lugar apropiado del RA.
- d) Según el lenguaje, puede ser necesario comprobar que se ha establecido un valor de retorno o asignar uno por defecto. En general, se debe generar código para devolver al final de la función el control al llamador, independientemente de si existe una instrucción específica para ello en el cuerpo de la función (secuencia de retorno por defecto).

9.2.2 Compilación de la llamada a la función

A la hora de compilar una llamada a una función deberemos tener en cuenta:

- a) Comprobaciones semánticas: comprobar que el número y tipo de los parámetros se corresponde con los indicados en la definición de la función y, en su caso, generar las conversiones de tipo adecuadas; comprobar que el tipo del valor devuelto por la función es adecuado para la expresión en la que figura la llamada y generar código para la conversión de tipo, si es necesario.
- b) Generar código para evaluar los parámetros y poner cada uno de ellos en su posición del RA.
- c) Almacenar la dirección o etiqueta de retorno en el lugar adecuado del RA.

IDENTIFICADOR	TIPO	DIRECCIÓN	ETIQUETA
a	entero	0	
f1	$t(f1)$	1	L2
b	entero	3	
c	entero	4	

(a) Mientras se procesa la función **f1**.

IDENTIFICADOR	TIPO	DIRECCIÓN	ETIQUETA
a	entero	0	
f1	$t(f1)$	1	L2
b	entero	8	
f2	$t(f2)$	9	L3
a	entero	11	
b	entero	12	
c	entero	13	

(b) Mientras se procesa la función **f2**.

IDENTIFICADOR	TIPO	DIRECCIÓN	ETIQUETA
a	entero	0	
f1	$t(f1)$	1	L2
b	entero	8	
f2	$t(f2)$	9	L3
c	entero	19	

(c) Mientras se procesa el programa principal.

Figura 9.10: Evolución de la tabla de símbolos en distintos momentos de la compilación del programa de la figura 9.6. Mediante $t(f)$ se indica la posición en la tabla de tipos de la entrada correspondiente a la función **f**.

- d) Generar código para saltar al comienzo de la función, accediendo a su dirección o etiqueta de inicio en la tabla de símbolos.
- e) Recuperar y utilizar el valor devuelto por la función (almacenándolo en una variable temporal, por ejemplo).

9.3 Funciones con recursividad

Si permitimos recursividad en las funciones de un lenguaje, el espacio de los RA no puede reservarse estáticamente como se hacía en el caso de los entornos de ejecución completamente estáticos. Pensemos, por ejemplo, en una función para calcular el factorial de un número entero tal como la de la figura 9.11. Si seguimos la política de la sección anterior, tendremos un único RA para la función. Dado que la recursividad

```

int a;
int fact (int n)
{
    if (n <= 1)
        return 1;
    else
        return fact(n-1)*n;
}
main ()
{
    a=fact(3);
}

```

Figura 9.11: Código en C de la funcion factorial utilizada en la seccion 9.3.

hace que en un determinado momento puedan haber varias instancias de la funcion `fact` ejecutándose, estas competirían por el RA, modificarían valores de otras instancias y harían que el comportamiento del programa fuera completamente erróneo.

En este caso, es necesario considerar un entorno de ejecución basado en una pila, denominada *pila de ejecución (runtime stack)*, donde se almacenan los RA. Cada llamada a una función apila un RA en la pila de ejecución y cuando la función acaba se desapila el RA correspondiente.¹² En un determinado instante puede haber en la pila de ejecución más de un RA correspondiente a una única función.

Ya que se utiliza una zona de memoria nueva y diferente para cada llamada o activación de la función y dado que puede haber más de un RA en la pila para la misma función, los campos del RA no están en una dirección única; para poder acceder a ellos se utiliza *direccionamiento indirecto* con respecto a un determinado registro índice que apunta al RA actual.

Para que el esquema anterior funcione, deben mantenerse como mínimo dos punteros: uno al RA actual, llamado *puntero de cuadro (frame pointer)*, y otro, llamado *enlace de control (control link)*, que apunta al RA anterior en la pila y permite restituirlo cuando la llamada acabe. El puntero de cuadro se guarda normalmente en un registro, ya que se usa intensivamente para acceder a las variables locales, a las temporales y a los parámetros de la función. Por otro lado, el enlace de control se almacena en el RA y salvaguarda el valor del puntero de cuadro del RA precedente (es decir, de la función que realizó la llamada a la función actual). Finalmente, el puntero de cuadro apunta a la posición del primer parámetro de la función en el RA¹³ y, si la función no tiene argumentos, a la posición en el RA de la primera variable local o temporal. En la figura 9.12 se muestra como queda un RA con esta modificación. En el caso de `m2r`, el puntero de cuadro se almacena en el registro B por lo que el RA puede también representarse como se hace en la figura 9.13.

En la figura 9.13 se observa que, suponiendo que todos los tipos de datos ocupan sólo una posición de memoria, el valor devuelto por la función se almacena en `@B-3`,

¹²Las variables de tipo *static* de C no se guardan en el RA, ya que, de hacerlo, su contenido se perdería al desapilar el RA; este tipo de variables se almacenan en la zona global.

¹³El puntero de cuadro puede, según el compilador, apuntar a otras posiciones del RA.

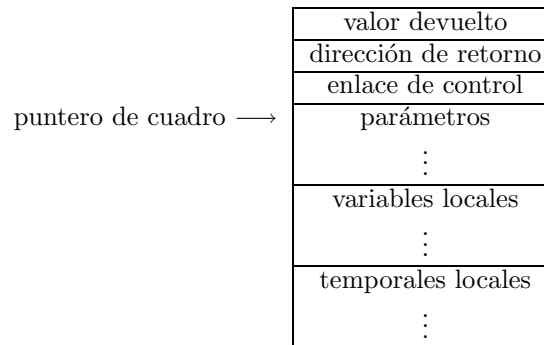


Figura 9.12: Modelo de registro de activación (RA) para un entorno basado en pila. El puntero de cuadro apunta al primer parámetro de la función y el enlace de control salvaguarda el puntero de cuadro del RA inmediatamente anterior en la pila.

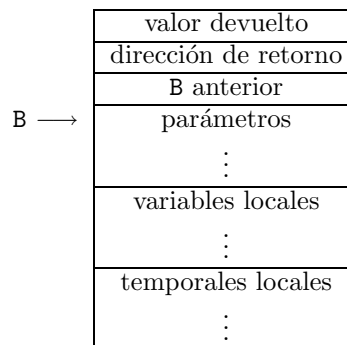


Figura 9.13: Forma del RA en el caso de la máquina m2r. El puntero de cuadro se guarda en el registro B.

la dirección o etiqueta de retorno en $@B-2$ y el valor anterior de B en $@B-1$. El primer argumento está en $@B+0$ o, lo que es lo mismo, en B; el segundo está en $@B+1$, y así sucesivamente. Supongamos que el RA es el de una función con dos argumentos; en ese caso, la primera variable local se guardará en $@B+2$, la segunda en $@B+3$, y así sucesivamente. Las temporales se almacenarán en el RA a continuación de las variables locales.

Las variables globales se suelen almacenar en un espacio de memoria separado de la pila de ejecución y se accede a ellas mediante direccionamiento absoluto.

9.3.1 Compilación del cuerpo de la función

Cuando el compilador encuentra la definición de una función bajo un entorno de ejecución basado en pila ha de llevar a cabo algunas tareas, que se indican a continuación:

- Guardar la función (sin tipo todavía) en la tabla de símbolos; abrir un nuevo ámbito en la tabla de símbolos y hacer que la dirección del primer argumento sea 0 (relativa al puntero de cuadro, B).

- b) Al mismo tiempo que los argumentos se van almacenando en la tabla de símbolos se va creando el tipo de la función en la tabla de tipos; finalmente, el tipo de la función se almacena en la entrada correspondiente de la tabla de símbolos.
- c) Hacer que las temporales de la función comiencen a continuación de la última variable local.
- d) Cuando se acabe el cuerpo de la función deben eliminarse los símbolos locales de la tabla de símbolos (esto es, cerrar el ámbito) y generar la secuencia de retorno por defecto.

En un ETDS con las dos reglas siguientes para definir la sintaxis de una declaración de función, las acciones anteriores se situarían donde se indica:

$$\begin{aligned} \text{Func} &\longrightarrow \text{Tipo id lpar } \{a\} \text{ Args rpar } \{b\} \text{ Bloque } \{d\} \\ \text{Bloque} &\longrightarrow \text{lbra BDecl } \{c\} \text{ SeqInstr rbra} \end{aligned}$$

9.3.2 Compilación de instrucciones

En cuanto a la compilación de las instrucciones, debe tenerse en cuenta:

- a) En el acceso a variables debe distinguirse entre símbolos locales y globales para usar direccionamiento indirecto o absoluto.
- b) Normalmente las variables temporales del programa principal se acceden mediante direccionamiento absoluto, mientras que las de las funciones se acceden con direccionamiento indirecto.
- c) La instrucción de retorno debe almacenar el valor de la expresión a devolver en el lugar adecuado del RA y efectuar el salto a la dirección de retorno previamente almacenada en el RA.

9.3.3 Compilación de la llamada

Mientras se compila una llamada a una función es necesario tener en cuenta los siguientes aspectos:

- a) Reservar el espacio para el RA de la nueva llamada a función (el número de posiciones a reservar será la suma de 3 con el número de argumentos de la función) antes de empezar a procesar sus parámetros.
- b) Comprobar el tipo de cada parámetro, realizar las conversiones de tipo adecuadas y generar código para situar cada parámetro en su posición en el RA. Normalmente las temporales necesarias para calcular el valor de las expresiones que conforman los parámetros se cogen de la zona destinada a variables locales y temporales del nuevo RA; más adelante se muestra un ejemplo.
- c) En general, hay que comprobar que el número de parámetros utilizados en la llamada a la función coincide con el indicado en su definición, aunque esto puede depender del lenguaje considerado.

- d) Generar código para: guardar el valor de B (enlace de control) actual, actualizar el valor del registro B para que apunte al nuevo RA, guardar la etiqueta de retorno en el nuevo RA, dar el salto a la función y, una vez que esta devuelva el control, restaurar el registro B a su valor anterior (esto último se puede hacer en la secuencia de retorno de la función llamada, con lo cual se genera menos código y el programa objeto resulta ligeramente más pequeño).

En un ETDS las acciones anteriores se situarían donde se indica en la siguiente regla de ejemplo:

$$Factor \longrightarrow \text{id lpar } \{a\} \text{ Par } \{b\} \text{ rpar } \{c\} \{d\}$$

9.3.4 Traducción a m2r de la función factorial

Las figuras 9.14 a 9.17 ejemplifican todas las ideas anteriores. En ellas se muestra la traducción de la función factorial a código objeto de m2r. Para hacer más sencillo su seguimiento, la figura 9.18 muestra la pila de ejecución justo antes de ejecutar la instrucción `jmp L2` que efectúa la llamada `fact(2)`.

9.4 Funciones locales

En este apartado consideraremos el tratamiento de las funciones en lenguajes como Pascal o Ada, que no sólo permiten la recursividad, sino también la declaración anidada de funciones. La figura 9.19 muestra un ejemplo de programa que declara varias funciones anidadas de este tipo. Nótese cómo para la función `hijo` la variable `v_abu` no es *ni local ni global*, sino que pertenece al ámbito de su función abuelo. La existencia de estas referencias (*nonlocal references*) que no son globales ni locales complica considerablemente la compilación de las funciones y procedimientos.

Consideremos en primer lugar cómo podríamos tratar las funciones anidadas con el modelo descrito en la sección anterior para el caso de funciones recursivas. En el momento que el programa estuviera ejecutando el cuerpo de la función `hijo`, la pila de ejecución tendría el aspecto de la figura 9.20. Si utilizamos únicamente la información del RA de la función `hijo` (con una forma como la de la figura 9.12), no será posible acceder a la variable `v_abu` directamente; la única forma de hacerlo será *saltando* a través de los enlaces de control hasta llegar al RA de `abuelo`. Esta técnica se conoce como *encadenamiento* (*chaining*).

El encadenamiento tal como se ha descrito en el párrafo anterior presenta un inconveniente adicional: no es posible conocer en tiempo de compilación el número de saltos (4 en el caso anterior) a realizar para llegar al RA deseado, ya que, en general, las funciones se pueden llamar desde más de un lugar del programa y los saltos podrían depender de ese punto de llamada (imaginemos, por ejemplo, que la función `padre` llamara directamente a `hijo`). La solución a este problema complicaría tanto el diseño del compilador que nunca se sigue este enfoque para el tratamiento de funciones locales. Son otros enfoques, como el *encadenamiento de accesos* o el *display*, que veremos a continuación, los que se usan en la práctica.

PROG. FUENTE	PROGRAMA OBJETO
	<pre> mov #10000 B ; temporales del programa principal jmp L6 ; salta al programa principal </pre>
<code>int a;</code>	<pre> ; - reserva la posicion 0 para la ; variable 'a' </pre>
<code>int fact (</code>	<pre> ; - guarda 'fact' en la tabla de ; símbolos (TS) , con la etiqueta L1 para ; indicar el comienzo del codigo de la ; función ; - marca el principio de la TS para ; saber dónde empiezan los símbolos locales </pre>
<code>int n) {</code>	<pre> ; - guarda 'n' en la TS como símbolo local, ; con la dirección (relativa) 0 ; - guarda el tipo de la funcion en la tabla ; de tipos </pre>
<code>if (n<=1)</code>	<pre> L1 mov @B+0 @B+1 ; guarda 'n' (@B+0) ; en una temporal (@B+1) mov #1 @B+2 ; guarda un '1' en otra (@B+2) mov @B+1 A leqi @B+2 ; 'n <= 1' mov A @B+3 ; guarda el resultado en @B+3 mov @B+3 A jz L3 ; 'if (n<=1) ...' </pre>

Figura 9.14: Ejemplo de traducción a código de m2r de la función `fact` (continua en la figura 9.15).

9.4.1 Encadenamiento de accesos

La técnica de encadenamiento de accesos (*access chaining*) consiste básicamente en añadir un campo nuevo, llamado *enlace de acceso* (*access link*), a la información guardada en el RA. El enlace de acceso apunta al RA de la función padre según esté definida esta en el programa fuente.¹⁴ El nuevo RA tiene la forma general que se muestra en la figura 9.21 y la forma para la máquina m2r mostrada en la 9.22. Como se observa en ambas figuras, el puntero de cuadro apunta ahora al enlace de acceso del RA actual.

En el caso de que la variable a acceder se encuentre definida en la función padre (esto es, en el ámbito externo más cercano), basta con acceder a su RA a través del enlace

¹⁴De manera distinta al enlace de control, que apunta al RA de la función llamadora.

PROG. FUENTE	PROGRAMA OBJETO
<pre>return 1;</pre>	<pre>mov #1 @B+4 ; guarda un '1' en @B+4 mov @B+4 @B-3 ; pone el '1' (@B+4) en la ; posición reservada al valor ; que devuelve la función mov @B-2 A ; coge la posición de programa ; (o etiqueta) a la que hay que ; volver jmp @A ; salto de retorno jmp L4 ; cuando termina la parte ; 'then', hay que saltar al ; final del 'if'</pre>
<pre>else return fact(</pre>	<pre>; - llamada a otra función: reserva sitio en ; las temporales para el registro de activación ; (RA) de la función que se va a llamar: ; 3 posiciones (fijas) + 1 argumento; ; reserva desde @B+5 hasta @B+8</pre>
<pre>n-1</pre>	<pre>L3 mov @B+0 @B+9 ; guarda 'n' (@B+0) en una ; temporal (@B+9) mas alla del ; nuevo RA mov #1 @B+10 ; guarda un '1' en @B+10 mov @B+9 A subi @B+10 ; 'n - 1' mov A @B+11 ; guarda el resultado en @B+11 mov @B+11 @B+8 ; pone el primer parametro ; en la posición reservada para ; él en el nuevo RA</pre>

Figura 9.15: Ejemplo de traducción a código de m2r de la función `fact` (continuación de la figura 9.14; continua en la 9.16).

de acceso y acceder a la variable mediante un desplazamiento conocido en tiempo de compilación. En el caso general, habrá que *saltar* repetidamente a través de los enlaces de acceso hasta llegar al RA adecuado. Así, el código en m2r que permite acceder a la variable `v_abu` en el programa de la figura 9.19 sería el siguiente:

```
mov @B+0 A ; salto al RA de la función padre
mov @A A ; salto al RA de la función abuelo
addi #1 ; desplazamiento de la variable
mov @A t1 ; guarda la variable en una temporal
```

donde `t1` representa la dirección de memoria de una variable temporal.

Para que el encadenamiento de accesos funcione correctamente es necesario poder determinar en tiempo de compilación el número de saltos o encadenamientos a realizar

PROG. FUENTE	PROGRAMA OBJETO
)	<pre> ; --- llamada a 'fact' mov B @B+7 ; guarda B anterior mov B A addi #8 ; 8 = 4(temporales usadas) ; +3(campos fijos RA)+1 mov A B ; B apunta al nuevo RA mvetq L2 @B-2 ; pone etiqueta de retorno jmp L1 ; salto a la funcion fact L2 mov @B-1 B ; al volver, deja la B como estaba ; el valor devuelto por la función ; queda en @B+5 y el resto de ; temporales (@B+6,...) se ; reciclan </pre>
*n;	<pre> mov @B+0 @B+6 ; guarda 'n' (@B+0) en @B+6 mov @B+5 A ; valor devuelto por fact(n-1) mul @B+6 ; 'n' * (valor devuelto) mov A @B+7 ; sobrescribe el antiguo valor de ; retorno mov @B+7 @B-3 ; devuelve el resultado del ; producto mov @B-2 A jmp @A ; return fact(n-1)*n </pre>
}	<pre> L4 mov @B-2 A ; fin del 'if (n <= 1) ...' jmp @A ; secuencia de retorno por defecto, ; por si se alcanza el final de la ; función sin haber hecho 'return' </pre>

Figura 9.16: Ejemplo de traducción a código de m2r de la función `fact` (continuación de la figura 9.16; continua en la figura 9.17).

para llegar a la variable deseada. Esto se consigue asociando un *nivel de anidamiento* a cada declaración por medio de una variable global que se inicializa a cero y que se incrementa o decrementa conforme se entra o sale de una función durante la compilación. El nivel de anidamiento se almacena en la tabla de símbolos. En el ejemplo de la figura 9.19 la función `abuelo` tiene nivel 0; las funciones `otrotio`, `padre` y `tio`, y las variables locales de `abuelo`, nivel 1; las funciones `hijo` y `hermano`, y las variables locales de `otrotio`, `padre` y `tio`, nivel 2; y, finalmente, las variables locales de `hijo` y `hermano`, nivel 3.

El número de encadenamientos o saltos¹⁵ necesarios para acceder a una variable que

¹⁵Como *saltos* consideraremos tanto la primera instrucción `mov @B+0 A`, como las posteriores instrucciones de la forma `mov @A A`.

PROG. FUENTE	PROGRAMA OBJETO
<pre>int main () { a=fact(</pre>	<pre>; llamada a 'factorial': ; hay que reservar 3+1 ; posiciones para el RA, ; desde @B+0 a @B+3</pre>
<pre> 3);</pre>	<pre>L6 mov #3 @B+4 ; guarda un '3' en una temporal mov @B+4 @B+3 ; pone el primer parametro ; (el '3') en la posición ; reservada para él en el RA mov B @B+2 ; guarda B anterior mov B A addi #3 mov A B ; pone la nueva B mvetq L5 @B-2 ; pone etiqueta de retorno jmp L1 ; salta a la función L5 mov @B-1 B ; deja la B como estaba y ; fin de la llamada mov @B+0 0 ; asignacion a la variable 'a' ; (dir: 0) del valor que ; devuelve la función, que ha ; quedado en @B+0</pre>
<pre>}</pre>	<pre>halt</pre>

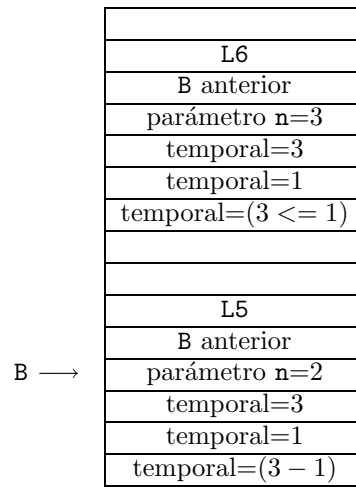
Figura 9.17: Ejemplo de traducción a código de m2r de la función `fact` (continuación de la figura 9.16).

no es local ni global se obtiene mediante la diferencia entre el nivel del ámbito en el que se realiza el acceso y el nivel de la variable accedida; en el ejemplo que venimos siguiendo hasta ahora, el número de encadenamientos desde el interior de la función `hijo` para acceder a las variables locales de la función `abuelo` sería de $3 - 1 = 2$. El número de instrucciones generadas del tipo `mov @A A` depende, por lo tanto, de esta diferencia.

La secuencia de llamada debe modificarse para considerar el almacenamiento del puntero de cuadro del padre en el campo del enlace de acceso del RA; en el caso de m2r, esto consiste en poner la B del padre en el nuevo RA. Cuando la función invocada es una función hijo o una función hermano, el acceso a la B de su padre es inmediato.¹⁶ Por otro lado, cuando se invoca una función de otro tipo,¹⁷ el acceso es más complicado, ya que debe recorrerse la cadena de enlaces de acceso hasta encontrar la B del padre de dicha

¹⁶Todas las llamadas a funciones del programa de la figura 9.19 son de este tipo.

¹⁷Este sería el caso si, por ejemplo, se invocara la función `otrotio` desde el cuerpo de la función `hermano` del programa de la figura 9.19.

Figura 9.18: Pila de ejecución inmediatamente antes de la llamada a `fact(2)`.

```

int abuelo ()
{
    int v_abu;
    int otrotio ()
    {
    }
    int padre ()
    {
        int hijo ()
        {
            v_abu=7;
        }
        int hermano ()
        {
            hijo();
        }
        hermano();
    }
    int tio ()
    {
        padre();
    }
    tio();
}

```

Figura 9.19: Programa con sintaxis de C con declaraciones anidadas de funciones.

función; esto implica una secuencia de instrucciones `mov @A A` similar a la generada en el caso del acceso a variables que no son ni locales ni globales.

Cuando el número de niveles de anidamiento es muy elevado, el código generado

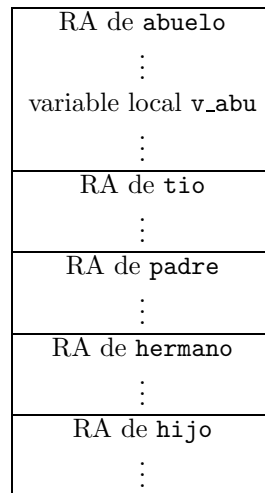


Figura 9.20: Aspecto de la pila de ejecución cuando se llama a la función `hijo` en el programa de la figura 9.19.

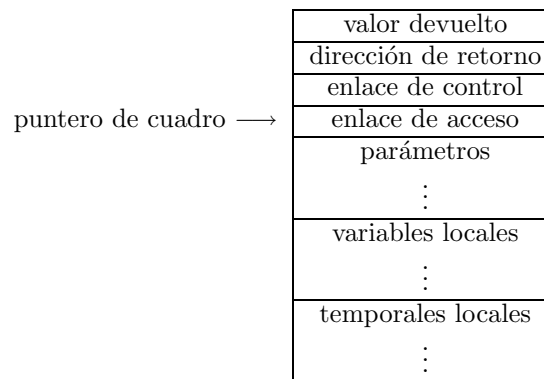


Figura 9.21: Modelo de registro de activación (RA) para un entorno basado en pila con funciones locales bajo el esquema de encadenamiento de accesos. El puntero de cuadro apunta al enlace de acceso, que apunta a su vez al enlace de acceso del RA de la función padre.

para acceder a las variables definidas en otras funciones y para invocar determinadas funciones se sobrecarga excesivamente.¹⁸ La estructura de datos conocida como *display* evita esta sobrecarga.

9.4.2 Display

Una alternativa al encadenamiento de accesos es mantener (en tiempo de ejecución) los enlaces de acceso en un vector que se guarda en una zona de memoria separada de la

¹⁸En cualquier caso, los programas habituales no suelen tener más de 2 o 3 niveles de anidamiento.

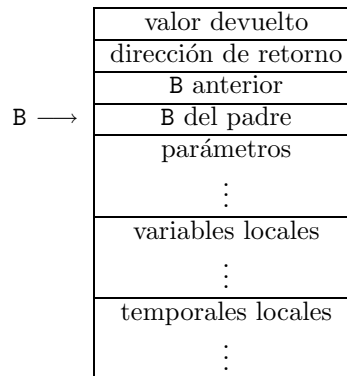


Figura 9.22: Forma del RA en el caso de la máquina `m2r` cuando se permiten funciones locales bajo el esquema de encadenamiento de accesos. El puntero de cuadro se guarda en el registro B.

pila de ejecución y se indiza por medio del nivel de anidamiento. A este vector se le conoce con el nombre de *display*.

Para acceder a una variable que no es local ni global se determina, consultando la tabla de símbolos, el nivel de anidamiento n de la función en la que se declara dicha variable; como el puntero de cuadro del RA correspondiente está guardado en `display[n]`, el acceso a la variable es inmediato y se evita la secuencia de saltos del método de encadenamiento de accesos.

Si consideramos que el nivel de anidamiento asignado a la función `abuelo` es 0 y que el *display* está almacenado a partir de la posición 100 (con lo que el valor de `display[0]` está guardado en esa misma dirección), el código de `m2r` necesario para acceder a la variable `v_abu` desde la función `hijo` es el siguiente:

```
mov 100 A
addi #1
mov @A t1
```

Como puede observarse, el número de instrucciones a ejecutar es independiente de la diferencia de niveles de anidamiento.

Cuando se invoca una función de nivel n , en `display[n]` se guarda la dirección de su RA. Ya que el valor anterior de `display[n]` puede ser necesario en el código que sigue a la llamada, este valor debe salvaguardarse convenientemente. El mejor sitio donde hacerlo es en el RA de la función invocada, de manera que pueda ser restaurado cuando esta termine. Como resultado de lo anterior, el RA adoptará (en el caso de la máquina `m2r`) la forma de la figura 9.23.

La secuencia de llamada debe modificarse, por tanto, para que guarde en el RA la posición del *display* de la función a llamar y almacene en esa posición el valor del puntero de cuadro de la función llamada. La secuencia de retorno debe encargarse de dejar la posición correspondiente del *display* como estaba antes de la llamada a la función.

Consideremos la pila de ejecución de la figura 9.20. En este momento en `display[0]` se guarda el puntero de cuadro del RA de la función `abuelo`, en `display[1]` el de la

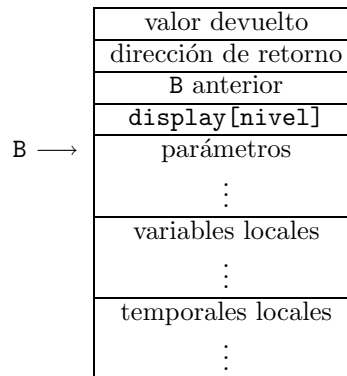


Figura 9.23: Forma del RA para la máquina `m2r` cuando se permiten funciones locales gestionadas mediante el *display*. Con `nivel` se indica el nivel de anidamiento de la función a la que pertenece el RA.

función `padre` y en `display[2]` el de la función `hijo`. Cuando la función `hijo` termine y devuelva el control a la función `hermano`, el valor de `display[2]` se restaurará y contendrá el puntero de cuadro del RA de la función `hermano`.

9.5 Paso de parámetros

Hay varias técnicas de paso de parámetros a una función; las más importantes son:

Paso por valor: el valor del parámetro se copia en el RA de la función durante la llamada; de esta manera el acceso se realiza igual que si de una variable local se tratara. Cualquier modificación en el parámetro dentro de la función no afecta a su valor en el exterior de esta (parámetro de *entrada*). Es el tipo de paso de parámetros que hemos utilizado en los ejemplos de este capítulo y el único que permite el lenguaje C.

Paso por referencia: en este caso lo que se copia en el RA es la dirección del parámetro y no su valor; así, cualquier acceso al valor del parámetro desde el interior de la función se lleva a cabo mediante la desreferencia oportuna. Los parámetros pasados de esta forma son, por lo tanto, de *entrada* y de *salida*.¹⁹

Paso por valor-copia: este tipo de paso de parámetros está a medio camino de los dos anteriores: el valor del parámetro se copia al RA durante la llamada por lo que se puede acceder como una variable local; pero, por otro lado, como parte de la secuencia de retorno, el valor posiblemente modificado del parámetro se copia a la variable utilizada en la llamada. Es también, por tanto, un caso de parámetros de *entrada* y *salida*.

¹⁹En C++ se indica que un parámetro es de entrada y salida anteponiendo a su nombre el carácter `&`.

9.6 Referencias bibliográficas

REFERENCIAS	EPÍGRAFES
[Louden, 1997]	7.1, 7.2, 7.3, 7.5, ejercicio 7.10 y 8.5
[Aho, Sethi y Ullman, 1990]	7.2, 7.3, 7.4, 7.5 y 8.7
[Bennett, 1990]	2.2.2 y 10.3.2
[Fischer y LeBlanc, 1991]	9.1, 9.2 y 13.2

9.7 Ejercicios

Ejercicio 9.1

Dado el siguiente programa en C, dibújese el contenido del registro de activación de la función **Pepito** (incluidas las variables temporales) en el momento inmediatamente anterior a la ejecución del **return** (donde indica el comentario).

```
int Pepito(int a,float b)
{
    float c;

    a = a+b;
    c = 2.0*a;

    /* aqui */
    return c;
}

main()
{
    Pepito(7,3.9);
}
```

Ejercicio 9.2

Dado el siguiente fragmento de programa en C, dibújese el contenido de los registros de activación de las llamadas activas en el momento en que se va a ejecutar la instrucción **return num**; *por tercera vez*. Indíquese en los registros solamente los valores de los parámetros y de las variables locales cuyo valor sea conocido. No hace falta especificar cuántas temporales se utilizan ni qué valor tienen. Se debe suponer que para la función **main** no se crea un registro de activación.

```
int a;
int potencia(int num,int aque)
{
    int uno,dos;
    uno = aque/2;
    dos = aque - uno;
    if (aque == 0)      return 1;
    else if (aque == 1) return num;
    else                return potencia(num,uno)*potencia(num,dos);
}
main()
{
    a = 2+potencia(2,5);
}
```

Ejercicio 9.3

Escribir el código en **m2r** que se generaría para el siguiente fragmento de programa en C.

```
int a;
int RestaUno(int n) {
    return n-1;
}
int factorial(int n) {
    if (n>1) return n*factorial(RestaUno(n));
    else return 1;
}
main() {
    a = 2+factorial(5);
}
```

Ejercicio 9.4

Escribir el código en m2r que generaría un compilador para el siguiente fragmento de programa en C. Indíquese también qué valor se almacena en la variable **a** declarada en la función **main**. Debe suponerse que la función **main** no tiene registro de activación (al estilo del programa principal en Pascal), aunque en C sea una función más.

```
float Abs(float f)
{
    if (f<0)
        return -f;
    else
        return f;
}
char Suma(int i,float f)
{
    int a[2][5];

    a[1][3] = Abs(i+f);
    return a[1][3];
}
main() {
    int a;
    a = Suma(10.987,1024);
}
```

Apéndice A

Soluciones a los ejercicios

A.1 Soluciones a los ejercicios del capítulo 2

Ejercicio 2.1

Diseñar un analizador léxico que utilice un DT construido a partir de las expresiones regulares de los patrones de los tokens involucrados en expresiones algebraicas (los operadores son “*”, “+”, “-”, “/”, “(” y “)”) en las que intervengan números enteros y reales (en notación no exponencial) sin signo y variables expresadas mediante identificadores.

Las expresiones regulares no triviales son:

Números enteros: dígito^+

Números reales: $\text{dígito}^+(\cdot \text{dígito}^+)?$

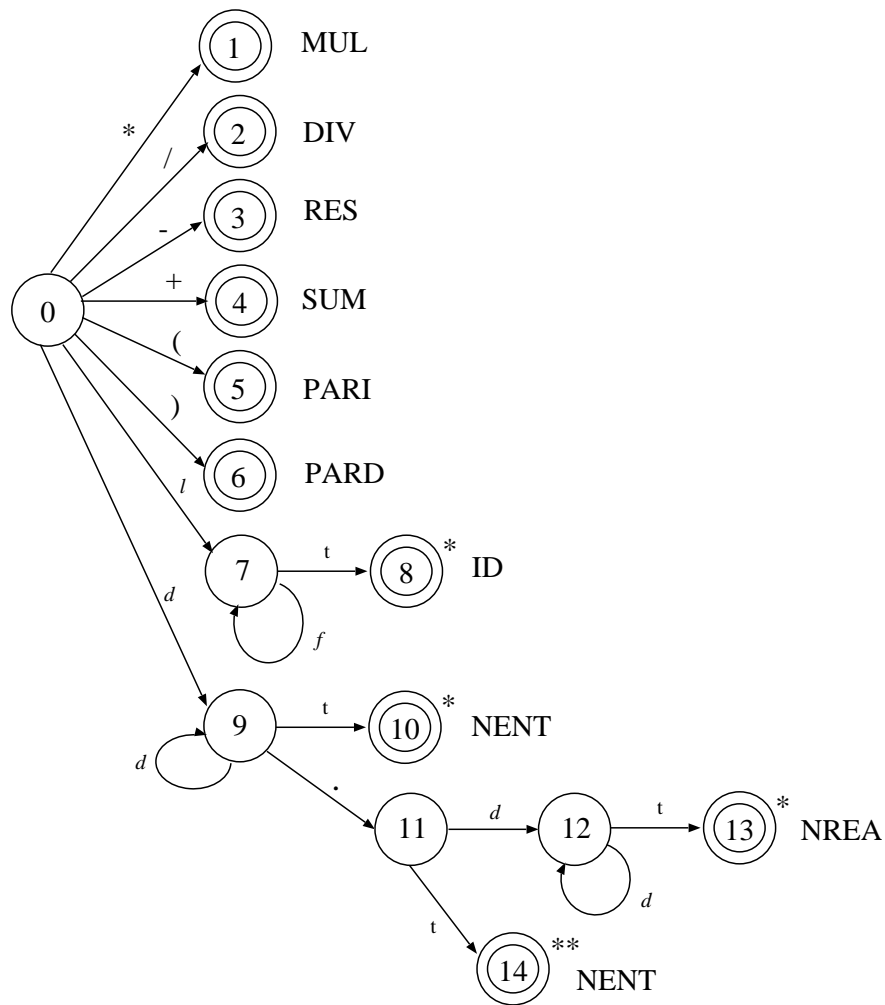
Identificadores: $\text{letra}(\text{letra} \mid \text{dígito})^*$

Solución:

Se pueden calcular los AFD a partir de las definiciones de los operadores (cadenas específicas) y de las expresiones regulares (cadenas no específicas) y luego se combinan todos, a partir de un mismo estado inicial, de manera que se obtiene el diagrama de transiciones de la figura A.1.

Como se observa en la figura A.1 el patrón de los enteros es un prefijo del patrón de los reales, por lo que se puede construir para ambos una misma rama del autómata en la que aparecerá un estado de aceptación que surge de un estado intermedio para los enteros (10) y otro final para los reales (13). En este caso, todos los estados de aceptación llevan asociado el reconocimiento de un *token* distinto. Aquellos que están marcados con “*” se debe a que el reconocimiento para esos *tokens* se produce cuando se ha leído uno o más caracteres más allá del final del lexema correspondiente a dicho patrón, por lo que llevan aparejada como acción asociada el retroceso de la marca de análisis sobre el *buffer* de entrada.

En la tabla de transiciones A.1, las celdas marcadas con “0” son transiciones que dan error. Se puede observar la poca eficiencia del almacenamiento en la tabla.



Notación: d = dígito; l = letra; f = alfanumérico (dígito | letra); t = otro.

Figura A.1: Diagrama de transiciones

Ejercicio 2.2

Hacer lo mismo que en el ejercicio anterior pero implementándolo “a mano” en C.

Solución:

Se supone que `lexema` es una variable global de tipo `char[]`.

```

int analex(void)
{
    int cl=0;
    do
    {
        c = obtenercaracter();
    }
}
  
```

	*	/	+	-	()	l	d	.	t	Token	Dev
0	1	2	4	3	5	6	7	9	0	0	-	-
1	-	-	-	-	-	-	-	-	-	-	MUL	0
2	-	-	-	-	-	-	-	-	-	-	DIV	0
3	-	-	-	-	-	-	-	-	-	-	RES	0
4	-	-	-	-	-	-	-	-	-	-	SUM	0
5	-	-	-	-	-	-	-	-	-	-	PARI	0
6	-	-	-	-	-	-	-	-	-	-	PARD	0
7	8	8	8	8	8	8	7	7	8	8	-	-
8	-	-	-	-	-	-	-	-	-	-	ID	1
9	10	10	10	10	10	10	10	9	11	10	-	-
10	-	-	-	-	-	-	-	-	-	-	NENT	1
11	14	14	14	14	14	14	14	12	14	14	-	-
12	13	13	13	13	13	13	13	12	13	13	-	-
13	-	-	-	-	-	-	-	-	-	-	NREA	1
14	-	-	-	-	-	-	-	-	-	-	NENT	2

Tabla A.1: Tabla de transiciones (los 0 son error)

```

switch (c)
{
    case ' ' :
    case '\t':
    case '\n': /* blancos */ break;
    case '*' : strcpy(lexema,"*");
                return(MUL);
    case '/' : strcpy(lexema,"/");
                return(DIV);
    case '-' : strcpy(lexema,"-");
                return(RES);
    case '+' : strcpy(lexema,"+");
                return(SUM);
    case '(' : strcpy(lexema,"(");
                return(PARI);
    case ')' : strcpy(lexema,")");
                return(PARD);
    default:
        if (ESNUMERO(c))
        {
            while (ESNUMERO(c))
            {
                lexema[cl]=c;
                cl++;
                c=obtenercaracter();
            }
            if (c=='.' || c==',')
            {
                lexema[cl]=c;
                cl++;
            }
        }
    }
}

```

```

        c=obtenercaracter();
        if (ESNUMERO(c))
        {
            while (ESNUMERO(c))
            {
                lexema[cl]=c;
                cl++;
                c=obtenercaracter();
            }
            devolvercaracter(c);
            lexema[cl]='\0';
            return(NREA);
        }
        else
        {
            devolvercaracter(lexema[cl-1]); /* devuelvo el '.' */
            devolvercaracter(c); /* devuelvo el siguiente
                                   carácter leído */
            lexema[cl-1]='\0';
            return(NENT);
        }
    }
    else
    {
        devolvercaracter(c); /* devuelvo el punto */
        lexema[cl]='\0';
        return(NENT);
    }
}
else if (ESLETRA(c))
{
    while((ESLETRA(c)) || (ESNUMERO(c)))
    {
        lexema[cl]=c;
        cl++;
        c=obtenercaracter();
    }
    devolvercaracter(c);
    lexema[cl]='\0';
    return(ID);
}
} /* del switch */
} while (1); /* bucle infinito para eliminar blancos */
} /* de analex */

```

Ejercicio 2.3

Diseñar un diagrama de transiciones determinista para reconocer los siguientes componentes léxicos:

- while** la palabra reservada “while” (en minúsculas).
- when** la palabra reservada “when” (en minúsculas).
- ident** cualquier secuencia de letras (mayúsculas y minúsculas) y dígitos que

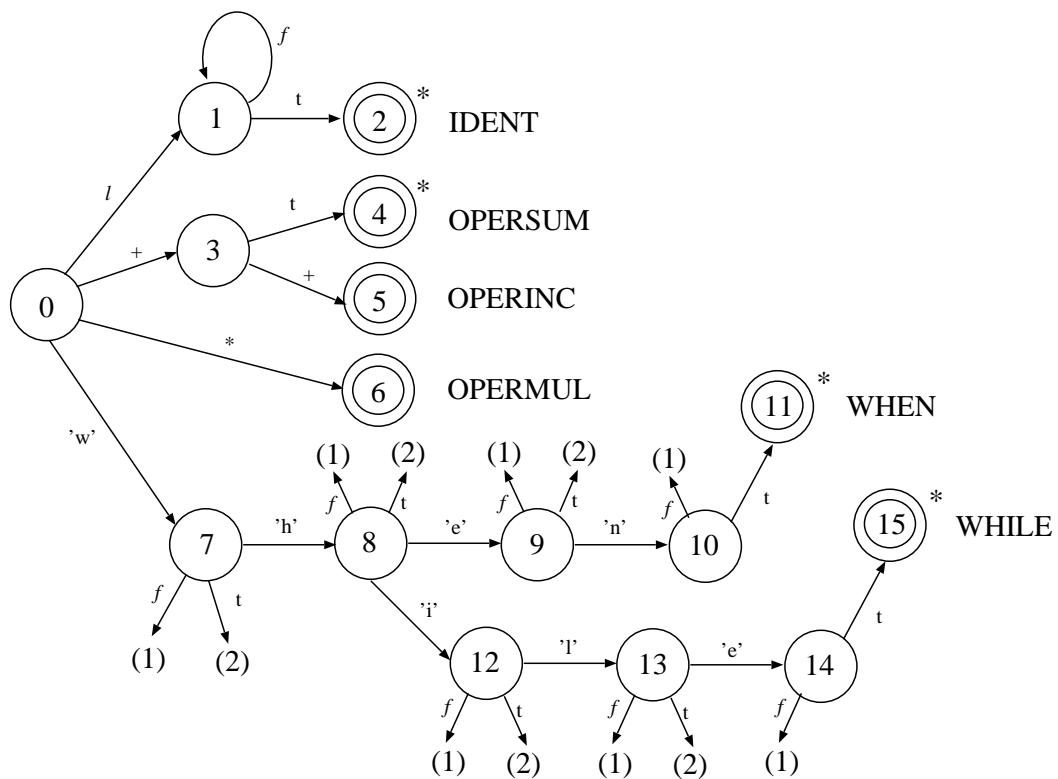
empiece por una letra, y que no coincida con ninguna de las palabras reservadas.

opersum el símbolo '+’.

opermul el símbolo '*’.

operinc el símbolo '++’.

Solución:



Notación: l = letra (mayúscula o minúscula); f = alfanumérico (dígito | letra); t = otro.

Ejercicio 2.4

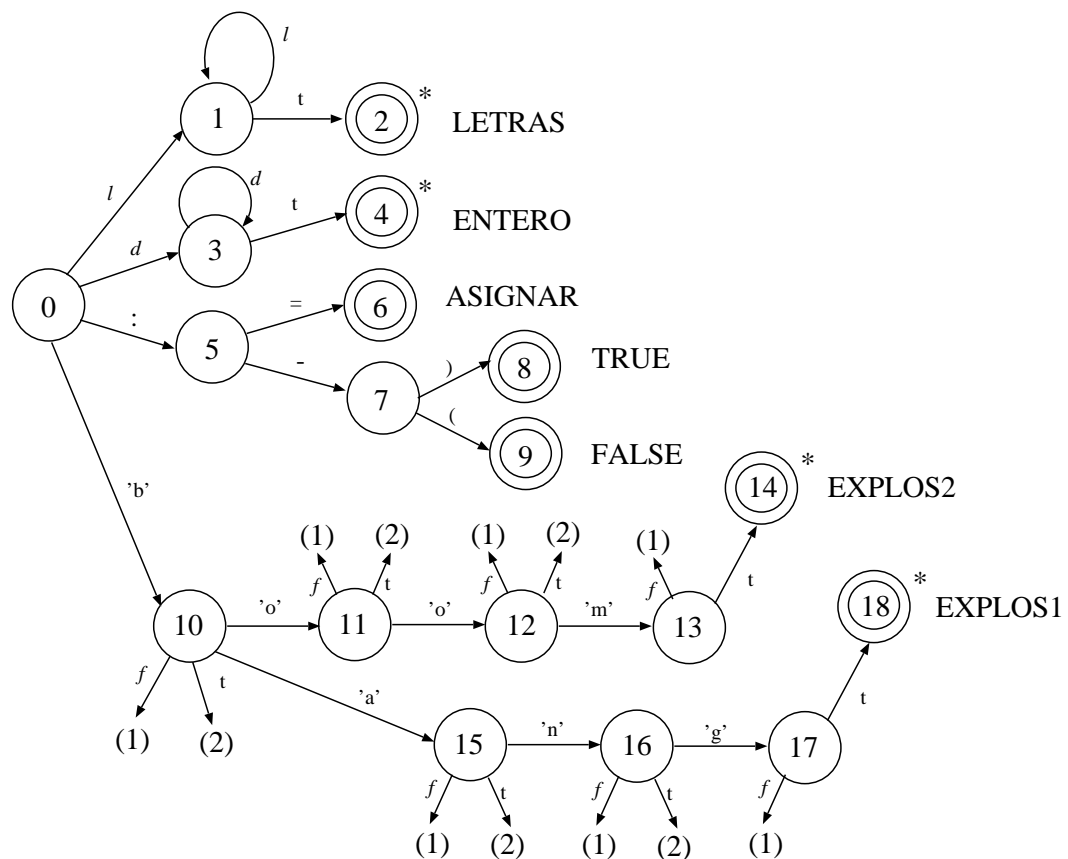
Diseñar un diagrama de transiciones determinista para reconocer los siguientes componentes léxicos:

letras cualquier secuencia de una o más letras (mayúsculas y minúsculas);

entero cualquier secuencia de uno o más dígitos;

explos1 la palabra reservada “bang” (en minúsculas)

explos2	la palabra reservada “boom” (en minúsculas)
true	la secuencia “:-)”
false	la secuencia “:- (“
asignar	la secuencia “:=”

Solución:

Notación: d = dígito; l = letra (mayúscula o minúscula); f = alfanumérico (dígito | letra); t = otro.

Ejercicio 2.5

Diseñar un diagrama de transiciones determinista para reconocer los siguientes componentes léxicos:

read	la palabra reservada ‘read’.
print	la palabra reservada ‘print’.

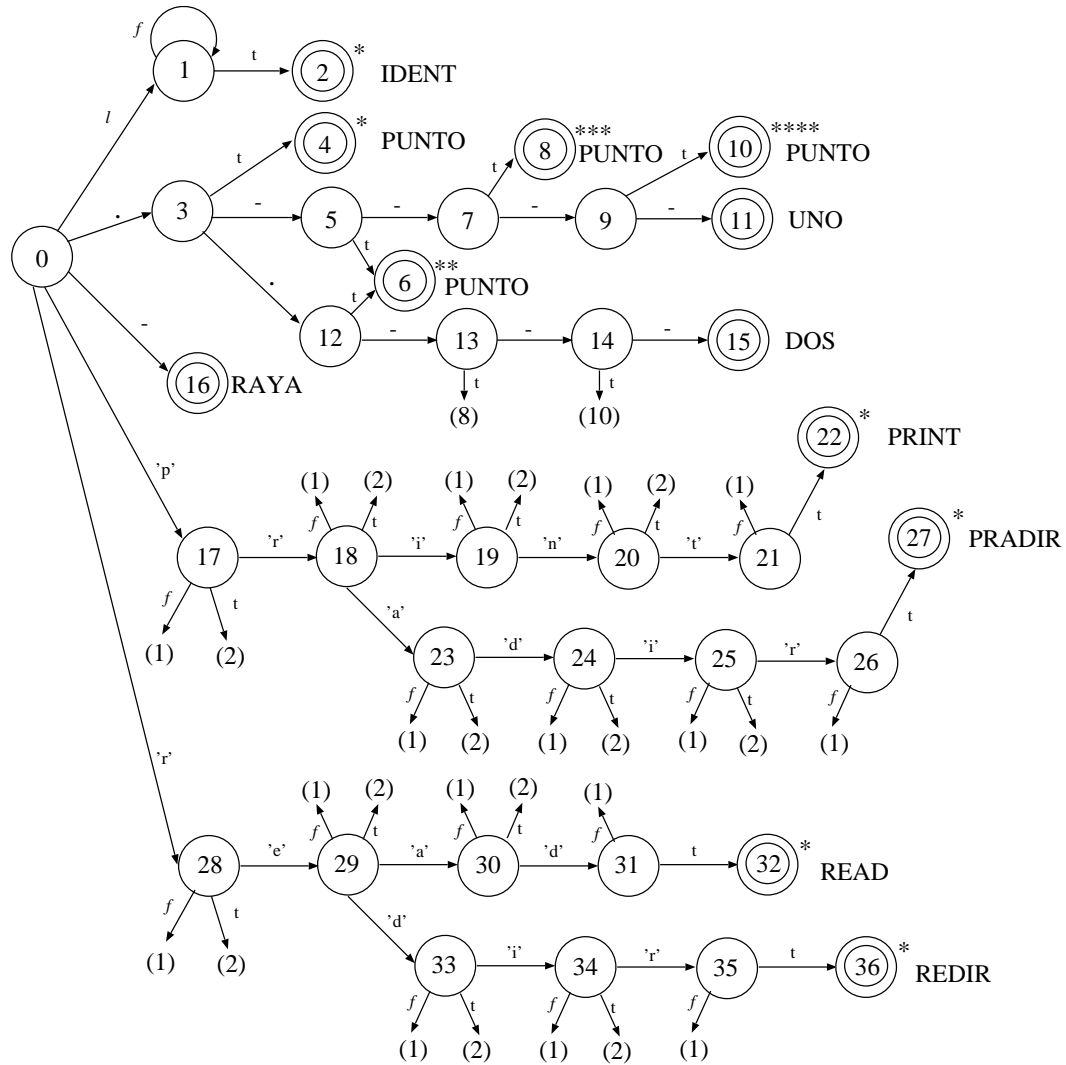
pradir	la palabra reservada ‘pradir’.
redir	la palabra reservada ‘redir’.
ident	cualquier secuencia de letras y dígitos que empiece por una letra y no coincida con ninguna de las palabras reservadas.
raya	el símbolo ‘-’.
punto	el símbolo ‘.’.
uno	el símbolo ‘.----’.
dos	el símbolo ‘..---’.

E indíquese cómo separa este analizador la secuencia de entrada
“pradir6dire..--.-”.

Solución:

La secuencia de entrada “pradir6dire..--.-” se tokeniza siguiendo el diagrama de transiciones de la figura A.2 como:

pradir6dire	→	IDENT
.	→	PUNTO
.	→	PUNTO
-	→	RAYA
-	→	RAYA
.	→	PUNTO
-	→	RAYA



Notación: d = dígito; l = letra (mayúscula o minúscula); f = alfanumérico (dígito | letra); t = otro.

Figura A.2: Diagrama de transiciones

A.2 Soluciones a los ejercicios del capítulo 3

Ejercicio 3.1

Diseñar una gramática *no ambigua* para el lenguaje de las expresiones que se pueden construir en Zaskal usando únicamente “**true**”, “**false**” y operadores booleanos. Los operadores de Zaskal son: “**or**” (binario, infijo), “**and**” (binario, infijo), “**not**” (unario, *postfijo*), “(” y “)”. Sin contar los paréntesis, la precedencia relativa de los operadores es

$$\text{not} > \text{and} > \text{or};$$

además, “**and**” y “**or**” son asociativos por la derecha. Por ejemplo, la expresión

$$(\text{true and false not}) \text{ or false and true not not}$$

sería una expresión correcta en Zaskal (que se evalúa como **true**, por cierto).

Solución:

$$\begin{aligned} E &\longrightarrow T \text{ or } E \\ E &\longrightarrow T \\ T &\longrightarrow F \text{ and } T \\ T &\longrightarrow F \\ F &\longrightarrow (E) \\ F &\longrightarrow F \text{ not} \\ F &\longrightarrow \text{true} \mid \text{false} \end{aligned}$$

Ejercicio 3.2

Diseñar una gramática *no ambigua* para el lenguaje de las expresiones regulares que se pueden construir con el alfabeto $\{0,1\}$. Los operadores que se usan para construir expresiones regulares son los siguientes (ordenados de menor a mayor precedencia):

$a \mid b$	unión	binario	asociatividad por la izquierda
ab	concatenación	binario	asociatividad por la derecha
a^+, a^*	clausura	unarios	

Algunas expresiones regulares que deben poder ser generadas por la gramática son:

$$\begin{aligned} &010 \\ &(01^*|(0|1^+)^*|1)1 \\ &(0(1^+)0|1(0^+)1)^*0011 \\ &(1100^{+*})^{+*} \end{aligned}$$

Solución:

$$\begin{aligned}
E &\longrightarrow E \mid T \\
E &\longrightarrow T \\
T &\longrightarrow F T \\
T &\longrightarrow F \\
F &\longrightarrow (E) \\
F &\longrightarrow F^+ \\
F &\longrightarrow F^* \\
F &\longrightarrow \mathbf{0} \\
F &\longrightarrow \mathbf{1}
\end{aligned}$$

Ejercicio 3.3

Diseñar una gramática *no ambigua* para los lenguajes que permiten escribir cualquier número de declaraciones de variables enteras, caracteres o reales en Pascal y C.

Solución:**Gramática para declaraciones en C:**

Ejemplo de declaraciones que debe admitir:

```
int i,i1; char c,e;
```

$$\begin{aligned}
L &\longrightarrow D L \\
L &\longrightarrow D \\
D &\longrightarrow T V \text{ punto} \\
V &\longrightarrow \text{ident coma } V \\
V &\longrightarrow \text{ident} \\
T &\longrightarrow \text{int} \\
T &\longrightarrow \text{char} \\
T &\longrightarrow \text{float}
\end{aligned}$$

Gramática para declaraciones en Pascal:

Ejemplo de declaraciones que debe admitir:

```
var i,i1: integer; c,e: char;
```

$$\begin{array}{ll}
LD & \longrightarrow \text{var } L \\
L & \longrightarrow D L \\
L & \longrightarrow D \\
D & \longrightarrow V \text{dospunt } T \text{puntco} \\
V & \longrightarrow \text{ident coma } V \\
V & \longrightarrow \text{ident} \\
T & \longrightarrow \text{integer} \\
T & \longrightarrow \text{char} \\
T & \longrightarrow \text{real}
\end{array}$$

Ejercicio 3.4

Las reglas siguientes definen el lenguaje LogPro. Escribanse las expresiones regulares que definen los *tokens* de este lenguaje y después diseñese una gramática para él.

- Un programa en el lenguaje LogPro consta de una secuencia de cero o más hechos o reglas y una única pregunta.
- Un hecho es un predicado seguido de un punto.
- Una regla es un predicado seguido del símbolo <-, la parte derecha de la regla y un punto.
- Una pregunta empieza con el símbolo <- seguido de la parte derecha de una regla y termina en un punto.
- Un predicado tiene un nombre (que es una secuencia de letras, dígitos y el símbolo _ (el caracter de subrayado) que empieza por una letra minúscula) y cero o más argumentos separados por comas y encerrados entre paréntesis (al contrario que en C, si no tiene argumentos, no se ponen los paréntesis).
- Un argumento puede ser un número (una secuencia de uno más dígitos sin signo), una variable (una secuencia de letras, dígitos y el símbolo _, que empieza por una letra mayúscula o por _) o un predicado.
- La parte derecha de una regla es una expresión booleana, y está formada por una secuencia de términos booleanos combinados con los operadores ', ' (una coma, representa la operación lógica *and*) y ', ' (un punto y coma, representa al *or* lógico). Los dos operandos tienen la asociatividad por la izquierda, y el ', ' tiene menor precedencia (se evalúa después) que el ', '. Se pueden utilizar paréntesis para agrupar expresiones, de la misma manera que se utilizan en expresiones de C o Pascal.
- Un término booleano puede ser un predicado o una expresión relacional. Una expresión relacional está formada por dos expresiones aritméticas separadas por un operador relacional, que puede ser uno de estos símbolos: ==, !=, <, <=, >, >=.
- Una expresión aritmética está formada por números, variables, paréntesis y los operadores +, -, * y /, con la misma precedencia y asociatividad que

tienen en lenguajes como C o Pascal (una expresión aritmética en **LogPro** es sintácticamente correcta en C o Pascal).

- De igual forma que en C o Pascal, se pueden utilizar espacios en blanco, tabuladores y cambios de línea para mejorar la legibilidad de un programa en **LogPro**, pero no son elementos del lenguaje.
- Un argumento de un predicado puede ser también una lista, que está formada por un corchete izquierdo, una secuencia de elementos separados por comas, una cola opcional y un corchete derecho. Cualquier tipo de argumento de un predicado puede ser un elemento de una lista (se puede construir una lista de listas), y la cola (que puede aparecer o no) está formada por el símbolo `|` seguido de una lista. La cola, si aparece, va situada inmediatamente antes del corchete derecho (aunque puede haber blancos entre estos elementos).

Ejemplo:

```
preD1      .
preD2(23,_23,f(a)) <- (eurt,eslaf;cierto), 2+3*4<= 3, X == 5.
preD3(Predicado, [l,i,s,t,a
                |[c,o,l,a]]).
<-preD4(preD2(Y,2,f(X)),preD1)
.
```

Solución:

Tabla de tokens con sus respectivas expresiones regulares

Token	Expresión regular
separa	"<-"
punto	"."
coma	","
punto	"."
lpar	"("
rpar	")"
lcor	"["
rcor	"]"
barra	" "
nompred	[a-z] [a-zA-Z0-9_]*
numero	[0-9]+
variable	[A-Z_] [a-zA-Z0-9_]*
oprel	(!= == > = <= <)
mulop	(* /)
addop	(+ -)

Gramática que reconoce el lenguaje **LogPro**:

$Prog$	\longrightarrow	$LH\ Preg$
LH	\longrightarrow	$HR\ LH \mid \epsilon$
HR	\longrightarrow	$H \mid R$
H	\longrightarrow	$Pred\ \mathbf{punto}$
R	\longrightarrow	$Pred\ \mathbf{separa}\ PD\ \mathbf{punto}$
$Preg$	\longrightarrow	$\mathbf{separa}\ PD\ \mathbf{punto}$
$Pred$	\longrightarrow	$\mathbf{nompred}\ LA$
LA	\longrightarrow	$\mathbf{lpar}\ Arg\ FArg \mid \epsilon$
Arg	\longrightarrow	$\mathbf{numero} \mid \mathbf{variable} \mid Pred \mid Lista$
$FArg$	\longrightarrow	$\mathbf{coma}\ Arg\ FArg \mid \mathbf{rpar}$
PD	\longrightarrow	$PD\ \mathbf{puntco}\ EB \mid EB$
EB	\longrightarrow	$EB\ \mathbf{coma}\ EBf \mid EBf$
EBf	\longrightarrow	$\mathbf{lpar}\ PD\ \mathbf{rpar} \mid Pred \mid ER$
ER	\longrightarrow	$EA\ \mathbf{oprel}\ EA$
EA	\longrightarrow	$EA\ \mathbf{addop}\ EAf \mid EAf$
EAf	\longrightarrow	$EAf\ \mathbf{mulop}\ EAf \mid EAf$
EAf	\longrightarrow	$\mathbf{lpar}\ EA\ \mathbf{rpar} \mid \mathbf{numero} \mid \mathbf{variable}$
$Lista$	\longrightarrow	$\mathbf{lcor}\ LE\ Cola\ \mathbf{rcor}$
LE	\longrightarrow	$Arg\ LArg$
$LArg$	\longrightarrow	$\mathbf{coma}\ Arg\ LArg \mid \epsilon$
$Cola$	\longrightarrow	$\mathbf{barra}\ Lista \mid \epsilon$

Ejercicio 3.5

Diseñar una gramática *no ambigua* que genere el lenguaje G definido por las siguientes frases:

1. Un programa en G es una secuencia de uno o más métodos.
2. Un método está formado por una declaración de variables y una secuencia de cero o más mensajes acabada en una sentencia de retorno.
3. Una declaración de variables está formada por una barra vertical ('|'), una secuencia de cero o más identificadores y otra barra vertical.
4. Un mensaje puede ser una asignación, una expresión o un mensaje de método.
5. Un mensaje de asignación está formado por una variable, el símbolo ':= ' y un mensaje.
6. Las expresiones pueden contener números enteros, variables, los operadores '+', '-', '*', '/', con la misma asociatividad que en C o Pascal, pero *sin precedencia*. Una expresión puede ser un mensaje entre paréntesis.
7. Un mensaje de método está formado por un identificador, un corchete izquierdo, una secuencia de cero o más expresiones separadas por un punto y coma entre cada dos expresiones, y un corchete derecho.
8. Una sentencia de retorno está formada por el símbolo '^' y una expresión.

Solución:

G	\longrightarrow	$Met\ G \mid Met$
Met	\longrightarrow	$Decl\ SMen\ Ret$
$SMen$	\longrightarrow	$Men\ SMen \mid \epsilon$
$Decl$	\longrightarrow	barra Sid barra
Sid	\longrightarrow	ident $Sid \mid \epsilon$
Men	\longrightarrow	$Asig \mid Expr \mid Memet$
$Asig$	\longrightarrow	variable asig Men
$Expr$	\longrightarrow	lpar Men rpar $\mid Expr\ op\ F \mid F$
F	\longrightarrow	numero \mid variable
$Memet$	\longrightarrow	ident lcor SE rcor
SE	\longrightarrow	$Expr\ FSE \mid \epsilon$
FSE	\longrightarrow	punto $Expr\ FSE \mid \epsilon$
Ret	\longrightarrow	circunflejo $Expr$

Ejercicio 3.6

Diseñar una gramática *no ambigua* que genere (carácter a carácter) el lenguaje de todos los números enteros sin signo que sean pares, considerando que el 0 es par también. Algunos números que tendría que generar esta gramática son:

0
234
11112
2350
00078

Solución:

N	\longrightarrow	$D\ N$
N	\longrightarrow	$DigPar$
D	\longrightarrow	0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
$DigPar$	\longrightarrow	0 \mid 2 \mid 4 \mid 6 \mid 8

A.3 Soluciones a los ejercicios del capítulo 4

Ejercicio 4.1

Compruébese que la siguiente gramática es LL(1) sin modificarla (en esta y en el resto de gramáticas de estos ejercicios se supondrá que el símbolo inicial es el primero).

$$\begin{aligned}
 A &\longrightarrow B C D \\
 B &\longrightarrow \mathbf{a} C \mathbf{b} \\
 B &\longrightarrow \epsilon \\
 C &\longrightarrow \mathbf{c} A \mathbf{d} \\
 C &\longrightarrow \mathbf{e} B \mathbf{f} \\
 C &\longrightarrow \mathbf{g} D \mathbf{h} \\
 C &\longrightarrow \epsilon \\
 D &\longrightarrow \mathbf{i}
 \end{aligned}$$

Solución:

A simple vista esta gramática no presenta ninguna característica que nos indique que no es LL(1), ya que no tiene ninguna regla con recursividad por la izquierda ni símbolos comunes por la izquierda en ninguna producción.

A continuación pasamos a comprobar si cumple la condición LL(1). Para ello en primer lugar se calculan los conjuntos de primeros para cada no terminal de la gramática¹:

$$\begin{aligned}
 \text{PRIM}(D) &= \text{PRIM}(\mathbf{i}) = \{\mathbf{i}\} \\
 \text{PRIM}(C) &= \text{PRIM}(\mathbf{c} A \mathbf{d}) \cup \text{PRIM}(\mathbf{e} B \mathbf{f}) \cup \text{PRIM}(\mathbf{g} D \mathbf{h}) \cup \text{PRIM}(\epsilon) = \\
 &\quad \{\mathbf{c}, \mathbf{e}, \mathbf{g}, \epsilon\} \\
 \text{PRIM}(B) &= \text{PRIM}(\mathbf{a} C \mathbf{b}) \cup \text{PRIM}(\epsilon) = \{\mathbf{a}, \epsilon\} \\
 \text{PRIM}(A) &= \text{PRIM}(B C D) = (\text{PRIM}(B) - \{\epsilon\}) \cup \text{PRIM}(C D) = \\
 &\quad \{\mathbf{a}\} \cup \text{PRIM}(C D) = \\
 &\quad \{\mathbf{a}\} \cup (\text{PRIM}(C) - \{\epsilon\}) \cup \text{PRIM}(D) = \\
 &\quad \{\mathbf{a}, \mathbf{c}, \mathbf{e}, \mathbf{g}\} \cup \text{PRIM}(D) = \{\mathbf{a}, \mathbf{c}, \mathbf{e}, \mathbf{g}, \mathbf{i}\}
 \end{aligned}$$

Como se puede ver, para calcular los primeros de A , se comprueba si entre los primeros de B se encuentra ϵ ; como es así, ϵ no se incluye en $\text{PRIM}(A)$, pero se añaden los conjuntos de primeros de los símbolos que siguen a B en la regla; en este caso C . Con $\text{PRIM}(C)$ ocurre lo mismo por lo que se añade el conjunto $\text{PRIM}(C) - \{\epsilon\}$, y se hace la misma comprobación para el conjunto de primeros de D .

En segundo lugar se calculan los conjuntos de siguientes para cada no terminal de la gramática.

$$\begin{aligned}
 \text{SIG}(A) &= \{\mathbf{\$}\} \cup \{\mathbf{d}\} = \{\mathbf{\$}, \mathbf{d}\} \\
 \text{SIG}(B) &= \text{PRIM}(C D) \cup \{\mathbf{f}\} = \{\mathbf{c}, \mathbf{e}, \mathbf{g}, \mathbf{i}, \mathbf{f}\} \\
 \text{SIG}(C) &= \text{PRIM}(D) \cup \{\mathbf{b}\} = \{\mathbf{i}, \mathbf{b}\} \\
 \text{SIG}(D) &= \{\mathbf{h}\} \cup \text{SIG}(A) = \{\mathbf{h}, \mathbf{\$}, \mathbf{d}\}
 \end{aligned}$$

¹El conjunto de primeros de cada terminal es el conjunto formado por el propio terminal.

El conjunto de siguientes del símbolo inicial siempre contendrá “\$”. El resto de elementos de este conjunto se calcula igual que el conjunto de siguientes de cualquier otro no terminal. Para ello se observan las partes derechas de aquellas reglas donde aparezca el no terminal. En este ejemplo el único conjunto de siguientes que presenta alguna dificultad es el de D , ya que se le añade el conjunto $SIG(A)$, porque D es el último símbolo de la regla de A .

Por último se calculan los conjuntos de predicción para cada regla de la gramática:

$$\begin{aligned}
 PRED(A \rightarrow B C D) &= PRIM(B C D) = \{ \mathbf{a}, \mathbf{c}, \mathbf{e}, \mathbf{g}, \mathbf{i} \} \\
 PRED(B \rightarrow \mathbf{a} C \mathbf{b}) &= PRIM(\mathbf{a} C \mathbf{b}) = \{ \mathbf{a} \} \\
 PRED(B \rightarrow \epsilon) &= SIG(B) = \{ \mathbf{c}, \mathbf{e}, \mathbf{g}, \mathbf{i}, \mathbf{f} \} \\
 PRED(C \rightarrow \mathbf{c} A \mathbf{d}) &= PRIM(\mathbf{c} A \mathbf{d}) = \{ \mathbf{c} \} \\
 PRED(C \rightarrow \mathbf{e} B \mathbf{f}) &= PRIM(\mathbf{e} B \mathbf{f}) = \{ \mathbf{e} \} \\
 PRED(C \rightarrow \mathbf{g} D \mathbf{h}) &= PRIM(\mathbf{g} D \mathbf{h}) = \{ \mathbf{g} \} \\
 PRED(C \rightarrow \epsilon) &= SIG(C) = \{ \mathbf{i}, \mathbf{b} \} \\
 PRED(D \rightarrow \mathbf{i}) &= PRIM(\mathbf{i}) = \{ \mathbf{i} \}
 \end{aligned}$$

La única dificultad para calcular los conjuntos de predicción se presenta en las reglas del tipo $B \rightarrow \epsilon$. En estos casos el conjunto de predicción de la regla se calcula como $(PRIM(\epsilon) - \{ \epsilon \}) \cup SIG(B)$; es decir, solamente $SIG(B)$.

Como podemos observar la gramática es LL(1), ya que los conjuntos de predicción que corresponden a reglas con la misma parte izquierda son disjuntos entre sí.

Ejercicio 4.2

¿Es LL(1) la siguiente gramática?

$$\begin{aligned}
 A &\longrightarrow B C D \\
 B &\longrightarrow \mathbf{b} \mid \epsilon \\
 C &\longrightarrow \mathbf{c} \mid \epsilon \\
 D &\longrightarrow \mathbf{d} \mid \epsilon
 \end{aligned}$$

Solución:

Cálculo de los conjuntos de primeros:

$$\begin{aligned}
 PRIM(D) &= \{ \mathbf{d}, \epsilon \} \\
 PRIM(C) &= \{ \mathbf{c}, \epsilon \} \\
 PRIM(B) &= \{ \mathbf{b}, \epsilon \} \\
 PRIM(A) &= \{ \mathbf{b}, \mathbf{c}, \mathbf{d}, \epsilon \}
 \end{aligned}$$

Como podemos observar en el conjunto de primeros de A se incluye ϵ ; esto se debe a que todos los símbolos de la parte derecha de la regla de A derivan a ϵ .

Cálculo de los conjuntos de siguientes:

$$\begin{aligned}
 SIG(A) &= \{ \$ \} \\
 SIG(B) &= \{ \mathbf{c}, \mathbf{d}, \$ \} \\
 SIG(C) &= \{ \mathbf{d}, \$ \} \\
 SIG(D) &= \{ \$ \}
 \end{aligned}$$

Cálculo de los conjuntos de predicción:

$$\begin{aligned}
 \text{PRED}(A \rightarrow B C D) &= \{ \mathbf{b}, \mathbf{c}, \mathbf{d}, \$ \} \\
 \text{PRED}(B \rightarrow \mathbf{b}) &= \{ \mathbf{b} \} \\
 \text{PRED}(B \rightarrow \epsilon) &= \{ \mathbf{c}, \mathbf{d}, \$ \} \\
 \text{PRED}(C \rightarrow \mathbf{c}) &= \{ \mathbf{c} \} \\
 \text{PRED}(C \rightarrow \epsilon) &= \{ \mathbf{d}, \$ \} \\
 \text{PRED}(D \rightarrow \mathbf{d}) &= \{ \mathbf{d} \} \\
 \text{PRED}(D \rightarrow \epsilon) &= \{ \$ \}
 \end{aligned}$$

Ejercicio 4.3

De un simple vistazo se puede comprobar que la siguiente gramática no es LL(1):

$$\begin{aligned}
 S &\longrightarrow S \mathbf{inst} \\
 S &\longrightarrow S \mathbf{var} D \\
 S &\longrightarrow \epsilon \\
 D &\longrightarrow D \mathbf{ident} E \\
 D &\longrightarrow D \mathbf{ident} \mathbf{sep} \\
 D &\longrightarrow \mathbf{int} \\
 D &\longrightarrow \mathbf{float} \\
 E &\longrightarrow S \mathbf{fproc}
 \end{aligned}$$

Elimínese la recursividad por la izquierda y los factores comunes por la izquierda y compruébese si la gramática equivalente resultante cumple la condición LL(1).

Solución:

En primer lugar eliminamos la recursividad por la izquierda de la gramática, aplicando la regla general:

$$\begin{array}{ll}
 \begin{array}{l}
 S \longrightarrow S \mathbf{inst} \\
 S \longrightarrow S \mathbf{var} D \\
 S \longrightarrow \epsilon
 \end{array}
 & \Rightarrow \begin{array}{l}
 S \longrightarrow S' \\
 S' \longrightarrow \mathbf{inst} S' \\
 S' \longrightarrow \mathbf{var} D S' \\
 S' \longrightarrow \epsilon
 \end{array} \\
 \\
 \begin{array}{l}
 D \longrightarrow D \mathbf{ident} E \\
 D \longrightarrow D \mathbf{ident} \mathbf{sep} \\
 D \longrightarrow \mathbf{int} \\
 D \longrightarrow \mathbf{float}
 \end{array}
 & \Rightarrow \begin{array}{l}
 D \longrightarrow \mathbf{int} D' \\
 D \longrightarrow \mathbf{float} D' \\
 D' \longrightarrow \mathbf{ident} E D' \\
 D' \longrightarrow \mathbf{ident} \mathbf{sep} D' \\
 D' \longrightarrow \epsilon
 \end{array}
 \end{array}$$

A continuación eliminamos los factores comunes por la izquierda que han quedado al eliminar la recursividad por la izquierda:

$$\begin{array}{ll}
 \begin{array}{l}
 D' \longrightarrow \mathbf{ident} E D' \\
 D' \longrightarrow \mathbf{ident} \mathbf{sep} D'
 \end{array}
 & \Rightarrow \begin{array}{l}
 D' \longrightarrow \mathbf{ident} D'' \\
 D'' \longrightarrow E D' \\
 D'' \longrightarrow \mathbf{sep} D'
 \end{array}
 \end{array}$$

Pasemos ahora a comprobar que efectivamente es LL(1).

Cálculo de los conjuntos de primeros:

$$\begin{aligned} \text{PRIM}(S) &= \{ \text{inst}, \text{var}, \epsilon \} \\ \text{PRIM}(S') &= \{ \text{inst}, \text{var}, \epsilon \} \\ \text{PRIM}(D) &= \{ \text{int}, \text{float} \} \\ \text{PRIM}(D') &= \{ \text{ident}, \epsilon \} \\ \text{PRIM}(D'') &= \{ \text{inst}, \text{var}, \text{fproc}, \text{sep} \} \\ \text{PRIM}(E) &= \{ \text{inst}, \text{var}, \text{fproc} \} \end{aligned}$$

Cálculo de los conjuntos de siguientes:

$$\begin{aligned} \text{SIG}(S) &= \{ \$, \text{fproc} \} \\ \text{SIG}(S') &= \{ \$, \text{fproc} \} \\ \text{SIG}(D) &= \{ \text{inst}, \text{var}, \$, \text{fproc} \} \\ \text{SIG}(D') &= \{ \text{inst}, \text{var}, \$, \text{fproc} \} \\ \text{SIG}(D'') &= \{ \text{inst}, \text{var}, \$, \text{fproc} \} \\ \text{SIG}(E) &= \{ \text{ident}, \text{inst}, \text{var}, \$, \text{fproc} \} \end{aligned}$$

Cálculo de los conjuntos de predicción:

$$\begin{aligned} \text{PRED}(S \rightarrow S') &= \{ \text{inst}, \text{var}, \$, \text{fproc} \} \\ \text{PRED}(S' \rightarrow \text{inst } S') &= \{ \text{inst} \} \\ \text{PRED}(S' \rightarrow \text{var } D S') &= \{ \text{var} \} \\ \text{PRED}(S' \rightarrow \epsilon) &= \{ \$, \text{fproc} \} \\ \text{PRED}(D \rightarrow \text{int } D') &= \{ \text{int} \} \\ \text{PRED}(D \rightarrow \text{float } D') &= \{ \text{float} \} \\ \text{PRED}(D' \rightarrow \text{ident } D'') &= \{ \text{ident} \} \\ \text{PRED}(D' \rightarrow \epsilon) &= \{ \text{inst}, \text{var}, \$, \text{fproc} \} \\ \text{PRED}(D'' \rightarrow E D') &= \{ \text{inst}, \text{var}, \text{fproc} \} \\ \text{PRED}(D'' \rightarrow \text{sep } D') &= \{ \text{sep} \} \\ \text{PRED}(E \rightarrow S \text{ fproc}) &= \{ \text{inst}, \text{var}, \text{fproc} \} \end{aligned}$$

La nueva gramática cumple la condición LL(1).

Ejercicio 4.4

Dada la siguiente gramática:

$$\begin{aligned} S &\longrightarrow A B \\ A &\longrightarrow \text{begin } S \text{ end } B \text{ theend} \\ A &\longrightarrow \epsilon \\ B &\longrightarrow \text{var } L : \text{tipo} \\ B &\longrightarrow B \text{ fvar} \\ B &\longrightarrow \epsilon \\ L &\longrightarrow L, id \\ L &\longrightarrow id \end{aligned}$$

1. Háganse las transformaciones necesarias para eliminar la recursividad por la izquierda.

2. Calcúlense los conjuntos de primeros y siguientes de cada no terminal.
3. Compruébese que la gramática modificada cumple la condición LL(1).
4. Constrúyase la tabla de análisis sintáctico LL(1) para esa nueva gramática.
5. Finalmente hágase la traza del análisis de la cadena:

```

begin
  var a,b:tipo
  var c:tipo
  fvar
end
  var d:tipo
theend

```

comprobando que las derivaciones son correctas mediante la construcción del árbol de análisis sintáctico.

Solución:

Eliminación de la recursión por la izquierda:

$$\begin{array}{ll}
 B \longrightarrow \mathbf{var} \, L : \mathbf{tipo} & B \longrightarrow \mathbf{var} \, L : \mathbf{tipo} \, B' \\
 B \longrightarrow B \, \mathbf{fvar} & \Rightarrow B \longrightarrow B' \\
 S \longrightarrow \epsilon & B' \longrightarrow \mathbf{fvar} \, B' \\
 & B' \longrightarrow \epsilon
 \end{array}$$

$$\begin{array}{ll}
 L \longrightarrow L \, , \, \mathbf{id} & L \longrightarrow \mathbf{id} \, L' \\
 L \longrightarrow \mathbf{id} & \Rightarrow L' \longrightarrow , \, \mathbf{id} \, L' \\
 & L' \longrightarrow \epsilon
 \end{array}$$

Cálculo de los conjuntos de primeros:

$$\begin{array}{ll}
 \text{PRIM}(S) = & \{ \mathbf{begin} , \mathbf{var} , \mathbf{fvar} , \epsilon \} \\
 \text{PRIM}(A) = & \{ \mathbf{begin} , \epsilon \} \\
 \text{PRIM}(B) = & \{ \mathbf{var} , \mathbf{fvar} , \epsilon \} \\
 \text{PRIM}(B') = & \{ \mathbf{fvar} , \epsilon \} \\
 \text{PRIM}(L) = & \{ \mathbf{id} \} \\
 \text{PRIM}(L') = & \{ , , \epsilon \}
 \end{array}$$

Cálculo de los conjuntos de siguientes:

$$\begin{array}{ll}
 \text{SIG}(S) = & \{ \$, \mathbf{end} \} \\
 \text{SIG}(A) = & \{ \mathbf{var} , \mathbf{fvar} , \$, \mathbf{end} \} \\
 \text{SIG}(B) = & \{ \mathbf{theend} , \$, \mathbf{end} \} \\
 \text{SIG}(B') = & \{ \mathbf{theend} , \$, \mathbf{end} \} \\
 \text{SIG}(L) = & \{ : \} \\
 \text{SIG}(L') = & \{ : \}
 \end{array}$$

Cálculo de los conjuntos de predicción:

(1)	$\text{PRED}(S \rightarrow A B)$	$= \{ \text{begin}, \text{var}, \text{fvar}, \$, \text{end} \}$
(2)	$\text{PRED}(A \rightarrow \text{begin } S \text{ end } B \text{ theend})$	$= \{ \text{begin} \}$
(3)	$\text{PRED}(A \rightarrow \epsilon)$	$= \{ \text{var}, \text{fvar}, \$, \text{end} \}$
(4)	$\text{PRED}(B \rightarrow \text{var } L : \text{tipo } B')$	$= \{ \text{var} \}$
(5)	$\text{PRED}(B \rightarrow B')$	$= \{ \text{fvar}, \text{theend}, \$, \text{end} \}$
(6)	$\text{PRED}(B' \rightarrow \text{fvar } B')$	$= \{ \text{fvar} \}$
(7)	$\text{PRED}(B' \rightarrow \epsilon)$	$= \{ \text{theend}, \$, \text{end} \}$
(8)	$\text{PRED}(L \rightarrow \text{id } D')$	$= \{ \text{id} \}$
(9)	$\text{PRED}(L' \rightarrow , \text{id } L')$	$= \{ , \}$
(10)	$\text{PRED}(L' \rightarrow \epsilon)$	$= \{ : \}$

La tabla de análisis correspondiente es:

	begin	end	theend	var	:	tipo	fvar	id	,	\$
S	(1)	(1)		(1)			(1)			(1)
A	(2)	(3)		(3)			(3)			(3)
B		(5)	(5)	(4)			(5)			(5)
B'		(7)	(7)				(6)			(7)
L								(8)		
L'					(10)				(9)	

La traza de la cadena de entrada “begin var a,b:tipo var c:tipo fvar end var d:tipo theend” es como sigue:

PILA	ENTRADA	ACCIÓN
$\$S$	begin var a,b: tipo ...	(1)
$\$B A$	begin var a,b: tipo ...	(2)
$\$B \text{ theend } B \text{ end } S \text{ begin}$	begin var a,b: tipo ...	emparejar begin
$\$B \text{ theend } B \text{ end } S$	var a,b:tipo var ...	(1)
$\$B \text{ theend } B \text{ end } B A$	var a,b:tipo var ...	(3)
$\$B \text{ theend } B \text{ end } B$	var a,b:tipo var ...	(4)
$\$B \text{ theend } B \text{ end } B' \text{ tipo } : L \text{ var}$	var a,b:tipo var ...	emparejar var
$\$B \text{ theend } B \text{ end } B' \text{ tipo } : L$	a,b:tipo var c ...	(8)
$\$B \text{ theend } B \text{ end } B' \text{ tipo } : L' \text{ id}$	a,b:tipo var c ...	emparejar id
$\$B \text{ theend } B \text{ end } B' \text{ tipo } : L'$,b:tipo var c: ...	(9)
$\$B \text{ theend } B \text{ end } B' \text{ tipo } : L' \text{ id } ,$,b:tipo var c: ...	emparejar ,
$\$B \text{ theend } B \text{ end } B' \text{ tipo } : L' \text{ id}$	b:tipo var c: ...	emparejar id
$\$B \text{ theend } B \text{ end } B' \text{ tipo } : L'$:tipo var c: ...	(10)
$\$B \text{ theend } B \text{ end } B' \text{ tipo } :$:tipo var c: ...	emparejar :
$\$B \text{ theend } B \text{ end } B' \text{ tipo}$	tipo var c: ...	emparejar tipo
$\$B \text{ theend } B \text{ end } B'$	var c: ...	error

El error que se debe generar en este caso es:

“Error: encontrado 'var' esperaba 'fvar', 'theend', 'end' o fin de fichero”

Ejercicio 4.5

Dada la siguiente gramática:

$$\begin{aligned}
 S &\longrightarrow S \text{ inst} \\
 S &\longrightarrow T R V \\
 T &\longrightarrow \text{tipo} \\
 T &\longrightarrow \epsilon \\
 R &\longrightarrow \text{blq } V \text{ fblq} \\
 R &\longrightarrow \epsilon \\
 V &\longrightarrow \text{id } S \text{ fin} \\
 V &\longrightarrow \text{id } ; \\
 V &\longrightarrow \epsilon
 \end{aligned}$$

Constrúyase un analizador sintáctico descendente recursivo (ASDR) para la gramática LL(1) equivalente a esta gramática. Supóngase que ya existen las funciones **analex** (analizador léxico), **emparejar** (función de emparejamiento) y **error** (emisión de error sintáctico). La función **error** se debe llamar con una cadena de caracteres que indique exactamente qué lexema ha provocado el error y qué símbolos se esperaban en lugar del encontrado en la entrada.

Solución:

Eliminación de la recursión por la izquierda:

$$\begin{array}{ll}
 S \longrightarrow S \text{ inst} & S \longrightarrow T R V S' \\
 S \longrightarrow T R V & \Rightarrow S' \longrightarrow \text{inst } S' \\
 & S' \longrightarrow \epsilon
 \end{array}$$

Eliminación de factores comunes por la izquierda:

$$\begin{array}{ll}
 V \longrightarrow \text{id } S \text{ fin} & V \longrightarrow \text{id } V' \\
 V \longrightarrow \text{id } ; & \Rightarrow V \longrightarrow \epsilon \\
 V \longrightarrow \epsilon & V' \longrightarrow S \text{ fin} \\
 & V' \longrightarrow ;
 \end{array}$$

Pasemos ahora a comprobar que efectivamente es LL(1).

Cálculo de los conjuntos de primeros:

$$\begin{aligned}
 \text{PRIM}(S) &= \{ \text{tipo}, \text{blq}, \text{id}, \text{inst}, \epsilon \} \\
 \text{PRIM}(S') &= \{ \text{inst}, \epsilon \} \\
 \text{PRIM}(T) &= \{ \text{tipo}, \epsilon \} \\
 \text{PRIM}(R) &= \{ \text{blq}, \epsilon \} \\
 \text{PRIM}(V) &= \{ \text{id}, \epsilon \} \\
 \text{PRIM}(V') &= \{ \text{tipo}, \text{blq}, \text{id}, \text{inst}, ;, \text{fin} \}
 \end{aligned}$$

Cálculo de los conjuntos de siguientes:

$$\begin{aligned}
 \text{SIG}(S) &= \{ \$, \text{fin} \} \\
 \text{SIG}(S') &= \{ \$, \text{fin} \} \\
 \text{SIG}(T) &= \{ \text{blq}, \text{id}, \text{inst}, \$, \text{fin} \} \\
 \text{SIG}(R) &= \{ \text{id}, \text{inst}, \$, \text{fin} \} \\
 \text{SIG}(V) &= \{ \text{inst}, \$, \text{fin}, \text{fbloq} \} \\
 \text{SIG}(V') &= \{ \text{inst}, \$, \text{fin}, \text{fbloq} \}
 \end{aligned}$$

Cálculo de los conjuntos de predicción:

$$\begin{aligned}
 \text{PRED}(S \rightarrow T R V S') &= \{ \text{tipo}, \text{blq}, \text{id}, \text{inst}, \$, \text{fin} \} \\
 \text{PRED}(S' \rightarrow \text{inst } S') &= \{ \text{inst} \} \\
 \text{PRED}(S' \rightarrow \epsilon) &= \{ \$, \text{fin} \} \\
 \text{PRED}(T \rightarrow \text{tipo}) &= \{ \text{tipo} \} \\
 \text{PRED}(T \rightarrow \epsilon) &= \{ \text{blq}, \text{id}, \text{inst}, \$, \text{fin} \} \\
 \text{PRED}(R \rightarrow \text{blq } V \text{fbloq}) &= \{ \text{blq} \} \\
 \text{PRED}(R \rightarrow \epsilon) &= \{ \text{id}, \text{inst}, \$, \text{fin} \} \\
 \text{PRED}(V \rightarrow \text{id } V') &= \{ \text{id} \} \\
 \text{PRED}(V \rightarrow \epsilon) &= \{ \text{inst}, \text{fbloq}, \$, \text{fin} \} \\
 \text{PRED}(V' \rightarrow S \text{fin}) &= \{ \text{tipo}, \text{blq}, \text{id}, \text{inst}, \text{fin} \} \\
 \text{PRED}(V' \rightarrow ;) &= \{ ; \}
 \end{aligned}$$

El ASDR correspondiente sería:

```

void S(void);

void V2(void)
{
    if (preanalisis==PUNTCO)
        Emparejar(PUNTCO);
    else
        if ((preanalisis==TIPO)|| (preanalisis==BLQ)||
            (preanalisis==ID)|| (preanalisis==INST)|| (preanalisis==FIN))
            S(); Emparejar(FIN);
        else
            error(lexema,PUNTCO,TIPO,BLQ,ID,INST,FIN);
}

void V(void)
{
    if (preanalisis==ID)
        Emparejar(ID); V2();
    else
        if ((preanalisis==INST)|| (preanalisis==FBLOQ)||
            (preanalisis==FIN)|| (preanalisis==FINDEFICHERO))
            break;
        else
            error(lexema,ID,INST,FBLOQ,FIN,FINDEFICHERO);
}

```

```
void R(void)
{
    if (preanalisis==BLQ)
        Emparejar(BLQ); V(); Emparejar(FBLOQ);
    else
        if ((preanalisis==ID)||(preanalisis==INST)||
            (preanalisis==FIN)||(preanalisis==FINDEFICHERO))
            break;
        else
            error(lexema,BLQ,ID,INST,FIN,FINDEFICHERO);
}

void T(void)
{
    if (preanalisis==TIPO)
        Emparejar(TIPO);
    else
        if ((preanalisis==BLQ)||(preanalisis==ID)||
            (preanalisis==INST)||(preanalisis==FIN)||
            (preanalisis==FINDEFICHERO))
            break;
        else
            error(lexema,TIPO,BLQ,ID,INST,FIN,FINDEFICHERO);
}

void S2(void)
{
    if (preanalisis==INST)
        Emparejar(INST); S2();
    else
        if ((preanalisis==FIN)||(preanalisis==FINDEFICHERO))
            break;
        else
            error(lexema,INST,FIN,FINDEFICHERO);
}

void S(void)
{
    if ((preanalisis==TIPO)||(preanalisis==BLQ)||(preanalisis==ID)||
        (preanalisis==INST)||(preanalisis==FIN)||(preanalisis==FINDEFICHERO))
        T(); R(); V(); S2();
    else error(lexema,TIPO,BLQ,ID,INST,FIN,FINDEFICHERO);
}
```

Ejercicio 4.6

Háganse las mismas operaciones que en el ejercicio anterior para la gramática:

$$\begin{aligned} E &\longrightarrow [L \\ E &\longrightarrow \mathbf{a} \\ L &\longrightarrow E Q \\ Q &\longrightarrow , L \\ Q &\longrightarrow] \end{aligned}$$

Con los mismos supuestos y condiciones que en aquel caso. Escribese la secuencia de llamadas recursivas a las funciones que haría el ASDR durante su ejecución, incluidas las llamadas a la función de emparejamiento, para la cadena de entrada “[a,a]”.

Solución:

Cálculo de los conjuntos de primeros:

$$\begin{aligned} \text{PRIM}(E) &= \{ [, \mathbf{a} \} \\ \text{PRIM}(L) &= \{ [, \mathbf{a} \} \\ \text{PRIM}(Q) &= \{ , ,] \} \end{aligned}$$

Cálculo de los conjuntos de siguientes:

$$\begin{aligned} \text{SIG}(E) &= \{ \$, ,] \} \\ \text{SIG}(L) &= \{ \$, ,] \} \\ \text{SIG}(Q) &= \{ \$, ,] \} \end{aligned}$$

Cálculo de los conjuntos de predicción:

$$\begin{aligned} \text{PRED}(E \rightarrow [L) &= \{ [\} \\ \text{PRED}(E \rightarrow \mathbf{a}) &= \{ \mathbf{a} \} \\ \text{PRED}(L \rightarrow E Q) &= \{ [, \mathbf{a} \} \\ \text{PRED}(Q \rightarrow , L) &= \{ , \} \\ \text{PRED}(Q \rightarrow]) &= \{] \} \end{aligned}$$

Como podemos comprobar esta gramática es LL(1).

La secuencia de llamadas que haría su correspondiente ASDR para la cadena [a,a] es:

```
E();
Emparejar(LCOR); L();
E(); Emparejar(a); Q();
Emparejar(COMA); L();
E(); Emparejar(a); Q();
Emparejar(RCOR);
```

Ejercicio 4.7

Dada la siguiente gramática:

$$\begin{aligned}
 P &\longrightarrow D S \\
 D &\longrightarrow D V \\
 D &\longrightarrow \epsilon \\
 S &\longrightarrow S I \\
 S &\longrightarrow \epsilon \\
 V &\longrightarrow \text{decl id ;} \\
 V &\longrightarrow \text{decl id (} P \text{) ;} \\
 V &\longrightarrow \text{decl [} D \text{] id ;} \\
 I &\longrightarrow \text{id ;} \\
 I &\longrightarrow \text{begin } P \text{ end}
 \end{aligned}$$

1. Háganse las transformaciones necesarias para que cumpla la condición LL(1).
2. Constrúyase la tabla de análisis sintáctico LL(1) para esa nueva gramática.
3. Finalmente hágase la traza del análisis de las cadenas:

decl id (begin id ;)

decl id (decl [decl id ;] id ;) ; id ;

Si durante el análisis se produjera algún error sintáctico indíquese qué símbolos podrían esperarse en lugar del *token* de preanálisis que entró.

Solución:

Eliminación de la recursión por la izquierda:

$$\begin{array}{ll}
 D \longrightarrow D V & D \longrightarrow D' \\
 D \longrightarrow \epsilon & \Rightarrow D' \longrightarrow V D' \\
 & D' \longrightarrow \epsilon
 \end{array}$$

$$\begin{array}{ll}
 S \longrightarrow S I & S \longrightarrow S' \\
 S \longrightarrow \epsilon & \Rightarrow S' \longrightarrow I S' \\
 & S' \longrightarrow \epsilon
 \end{array}$$

Eliminación de factores comunes por la izquierda:

$$\begin{array}{ll}
 V \longrightarrow \text{decl id ;} & V \longrightarrow \text{decl } V' \\
 V \longrightarrow \text{decl id (} P \text{) ;} & V' \longrightarrow \text{id } V'' \\
 V \longrightarrow \text{decl [} D \text{] id ;} & \Rightarrow V' \longrightarrow \text{[} D \text{] id ;} \\
 & V'' \longrightarrow \text{;} \\
 & V'' \longrightarrow \text{(} P \text{) ;}
 \end{array}$$

Pasemos ahora a comprobar que efectivamente es LL(1).

Cálculo de los conjuntos de primeros:

$$\begin{aligned}
 \text{PRIM}(P) &= \{ \text{decl}, \text{id}, \text{begin}, \epsilon \} \\
 \text{PRIM}(D) &= \{ \text{decl}, \epsilon \} \\
 \text{PRIM}(D') &= \{ \text{decl}, \epsilon \} \\
 \text{PRIM}(S) &= \{ \text{id}, \text{begin}, \epsilon \} \\
 \text{PRIM}(S') &= \{ \text{id}, \text{begin}, \epsilon \} \\
 \text{PRIM}(V) &= \{ \text{decl} \} \\
 \text{PRIM}(V') &= \{ \text{id}, [\} \\
 \text{PRIM}(V'') &= \{ ;, (\} \\
 \text{PRIM}(I) &= \{ \text{id}, \text{begin} \}
 \end{aligned}$$

Cálculo de los conjuntos de siguientes:

$$\begin{aligned}
 \text{SIG}(P) &= \{ \$,), \text{end} \} \\
 \text{SIG}(D) &= \{ \text{id}, \text{begin}, \$,), \text{end},] \} \\
 \text{SIG}(D') &= \{ \text{id}, \text{begin}, \$,), \text{end},] \} \\
 \text{SIG}(S) &= \{ \$,), \text{end} \} \\
 \text{SIG}(S') &= \{ \$,), \text{end} \} \\
 \text{SIG}(V) &= \{ \text{decl}, \text{id}, \text{begin}, \$,), \text{end},] \} \\
 \text{SIG}(V') &= \{ \text{decl}, \text{id}, \text{begin}, \$,), \text{end},] \} \\
 \text{SIG}(V'') &= \{ \text{decl}, \text{id}, \text{begin}, \$,), \text{end},] \} \\
 \text{SIG}(I) &= \{ \text{id}, \text{begin}, \$,), \text{end} \}
 \end{aligned}$$

Cálculo de los conjuntos de predicción:

$$\begin{aligned}
 (1) \quad \text{PRED}(P \rightarrow D S) &= \{ \text{decl}, \text{id}, \text{begin}, \$,), \text{end} \} \\
 (2) \quad \text{PRED}(D \rightarrow D') &= \{ \text{decl}, \text{id}, \text{begin}, \$,), \text{end},] \} \\
 (3) \quad \text{PRED}(D' \rightarrow V D') &= \{ \text{decl} \} \\
 (4) \quad \text{PRED}(D' \rightarrow \epsilon) &= \{ \text{id}, \text{begin}, \$,), \text{end},] \} \\
 (5) \quad \text{PRED}(S \rightarrow S') &= \{ \text{id}, \text{begin}, \$,), \text{end} \} \\
 (6) \quad \text{PRED}(S \rightarrow I S') &= \{ \text{id}, \text{begin} \} \\
 (7) \quad \text{PRED}(S' \rightarrow \epsilon) &= \{ \$,), \text{end} \} \\
 (8) \quad \text{PRED}(V \rightarrow \text{decl } V') &= \{ \text{decl} \} \\
 (9) \quad \text{PRED}(V' \rightarrow \text{id } V'') &= \{ \text{id} \} \\
 (10) \quad \text{PRED}(V' \rightarrow [D] \text{id } ;) &= \{ [\} \\
 (11) \quad \text{PRED}(V'' \rightarrow ;) &= \{ ; \} \\
 (12) \quad \text{PRED}(V'' \rightarrow (P) ;) &= \{ (\} \\
 (13) \quad \text{PRED}(I \rightarrow \text{id } ;) &= \{ \text{id} \} \\
 (14) \quad \text{PRED}(I \rightarrow \text{begin } P \text{end}) &= \{ \text{begin} \}
 \end{aligned}$$

La tabla de análisis correspondiente es:

	decl	id	[]	;	()	begin	end	\$
<i>P</i>	(1)	(1)					(1)	(1)	(1)	(1)
<i>D</i>	(2)	(2)		(2)			(2)	(2)	(2)	(2)
<i>D'</i>	(3)	(4)		(4)			(4)	(4)	(4)	(4)
<i>S</i>		(5)					(5)	(5)	(5)	(5)
<i>S'</i>		(6)					(7)	(6)	(7)	(7)
<i>V</i>	(8)									
<i>V'</i>		(9)	(10)							
<i>V''</i>					(11)	(12)				
<i>I</i>		(13)						(14)		

La traza de la cadena de entrada “decl id (begin id ;)” es como sigue:

PILA	ENTRADA	ACCIÓN
<i>\$P</i>	decl id (begin ...	(1)
<i>\$S D</i>	decl id (begin ...	(2)
<i>\$S D'</i>	decl id (begin ...	(3)
<i>\$S D' V</i>	decl id (begin ...	(8)
<i>\$S D' V' decl</i>	decl id (begin ...	emparejar decl
<i>\$S D' V'</i>	id (begin id ...	(9)
<i>\$S D' V'' id</i>	id (begin id ...	emparejar id
<i>\$S D' V''</i>	(begin id ; ...	(12)
<i>\$S D' ;) P (</i>	(begin id ; ...	emparejar (
<i>\$S D' ;) P</i>	begin id ;) ...	(1)
<i>\$S D' ;) S D</i>	begin id ;) ...	(2)
<i>\$S D' ;) S D'</i>	begin id ;) ...	(4)
<i>\$S D' ;) S</i>	begin id ;) ...	(5)
<i>\$S D' ;) S'</i>	begin id ;) ...	(6)
<i>\$S D' ;) S' I</i>	begin id ;) ...	(14)
<i>\$S D' ;) S' end P begin</i>	begin id ;) ...	emparejar begin
<i>\$S D' ;) S' end P</i>	id ;) \$	(1)
<i>\$S D' ;) S' end S D</i>	id ;) \$	(2)
<i>\$S D' ;) S' end S D'</i>	id ;) \$	(4)
<i>\$S D' ;) S' end S</i>	id ;) \$	(5)
<i>\$S D' ;) S' end S'</i>	id ;) \$	(6)
<i>\$S D' ;) S' end S' I</i>	id ;) \$	(13)
<i>\$S D' ;) S' end S' ; id</i>	id ;) \$	emparejar id
<i>\$S D' ;) S' end S' ;</i>	;) \$	emparejar ;
<i>\$S D' ;) S' end S'</i>) \$	(7)
<i>\$S D' ;) S' end</i>) \$	error

El error que se debe generar en este caso es:

“Error: encontrado ')’ esperaba 'end’”

La traza de la cadena de entrada “decl id (decl [decl id ;] id ;) ; id ;” es como sigue:

PILA	ENTRADA	ACCIÓN
$\$P$	decl id (decl [...	(1)
$\$S D$	decl id (decl [...	(2)
$\$S D'$	decl id (decl [...	(3)
$\$S D' V$	decl id (decl [...	(8)
$\$S D' V' \text{ decl}$	decl id (decl [...	emparejar decl
$\$S D' V'$	id (decl [decl ...	(9)
$\$S D' V'' \text{ id}$	id (decl [decl ...	emparejar id
$\$S D' V''$	(decl [decl id ...	(12)
$\$S D' ;) P ($	(decl [decl id ...	emparejar (
$\$S D' ;) P$	decl [decl id ; ...	(1)
$\$S D' ;) S D$	decl [decl id ; ...	(2)
$\$S D' ;) S D'$	decl [decl id ; ...	(3)
$\$S D' ;) S D' V$	decl [decl id ; ...	(8)
$\$S D' ;) S D' V' \text{ decl}$	decl [decl id ; ...	emparejar decl
$\$S D' ;) S D' V'$	[decl id ;] ...	(10)
$\$S D' ;) S D' ; \text{ id }] D [$	[decl id ;] ...	emparejar [
$\$S D' ;) S D' ; \text{ id }] D$	decl id ;] id ...	(2)
$\$S D' ;) S D' ; \text{ id }] D'$	decl id ;] id ...	(3)
$\$S D' ;) S D' ; \text{ id }] D' V$	decl id ;] id ...	(8)
$\$S D' ;) S D' ; \text{ id }] D' V' \text{ decl}$	decl id ;] id ...	emparejar decl
$\$S D' ;) S D' ; \text{ id }] D' V'$	id ;] id ; ...	(9)
$\$S D' ;) S D' ; \text{ id }] D' V'' \text{ id}$	id ;] id ; ...	emparejar id
$\$S D' ;) S D' ; \text{ id }] D' V''$;] id ;) ...	(11)
$\$S D' ;) S D' ; \text{ id }] D' ;$;] id ;) ...	emparejar ;
$\$S D' ;) S D' ; \text{ id }] D'$] id ;) ; ...	(4)
$\$S D' ;) S D' ; \text{ id }]$] id ;) ; ...	emparejar]
$\$S D' ;) S D' ; \text{ id}$	id ;) ; ...	emparejar id
$\$S D' ;) S D' ;$;) ; id ; ...	emparejar id
$\$S D' ;) S D' ;$;) ; id ; ...	emparejar ;
$\$S D' ;) S D'$) ; id ; \$	(4)
$\$S D' ;) S$) ; id ; \$	(5)
$\$S D' ;) S'$) ; id ; \$	(7)
$\$S D' ;)$) ; id ; \$	emparejar)
$\$S D' ;$; id ; \$	emparejar ;
$\$S D'$	id ; \$	(4)
$\$S$	id ; \$	(5)
$\$S'$	id ; \$	(6)
$\$S' I$	id ; \$	(13)
$\$S' ; \text{ id}$	id ; \$	emparejar id
$\$S' ;$; \$	emparejar ;
$\$S'$	\$	(7)
$\$$	\$	Aceptar

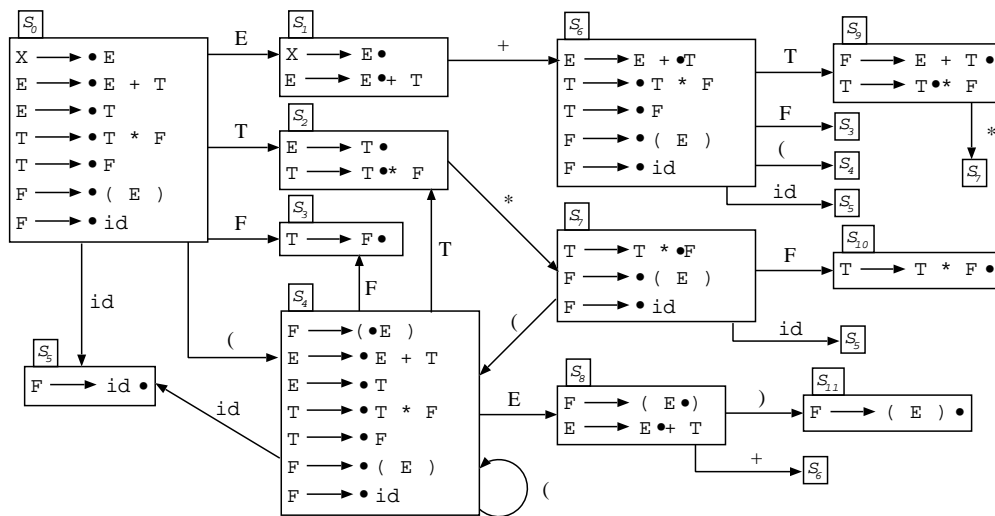
A.4 Soluciones a los ejercicios del capítulo 5

Ejercicio 5.1

A partir de la siguiente gramática (de expresiones aritméticas), construir la tabla de análisis SLR a partir del autómata reconocedor de prefijos viables y, a continuación, hacer la traza del análisis ascendente de la cadena “**id * id + id**”:

$$\begin{aligned}
 E &\longrightarrow E + T \\
 E &\longrightarrow T \\
 T &\longrightarrow T * F \\
 T &\longrightarrow F \\
 F &\longrightarrow (E) \\
 F &\longrightarrow \text{id}
 \end{aligned}$$

Solución:



Cálculo de los conjuntos de siguientes:

$$\begin{aligned}
 \text{SIG}(E) &= \{ \$, +,) \} \\
 \text{SIG}(T) &= \{ \$, +,), * \} \\
 \text{SIG}(F) &= \{ \$, +,), * \}
 \end{aligned}$$

La tabla de análisis correspondiente es:

Estado	Acción						Ir_a		
	id	+	*	()	\$	E	T	F
0	d5			d4			1	2	3
1		d6				aceptar			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

La traza de la cadena de entrada “id*id+id” es como sigue:

PILA	ENTRADA	ACCIÓN
0	id * id + id \$	d5
0 5	* id + id \$	r6 ($F \rightarrow \text{id}$)
0 3	* id + id \$	r4 ($T \rightarrow F$)
0 2	* id + id \$	d7
0 2 7	id + id \$	d5
0 2 7 5	+ id \$	r6 ($F \rightarrow \text{id}$)
0 2 7 10	+ id \$	r3 ($T \rightarrow T * F$)
0 2	+ id \$	r2 ($E \rightarrow T$)
0 1	+ id \$	d6
0 1 6	id \$	d5
0 1 6 5	\$	r6 ($F \rightarrow \text{id}$)
0 1 6 3	\$	r4 ($T \rightarrow F$)
0 1 6 9	\$	r1 ($E \rightarrow E + T$)
0 1	\$	aceptar

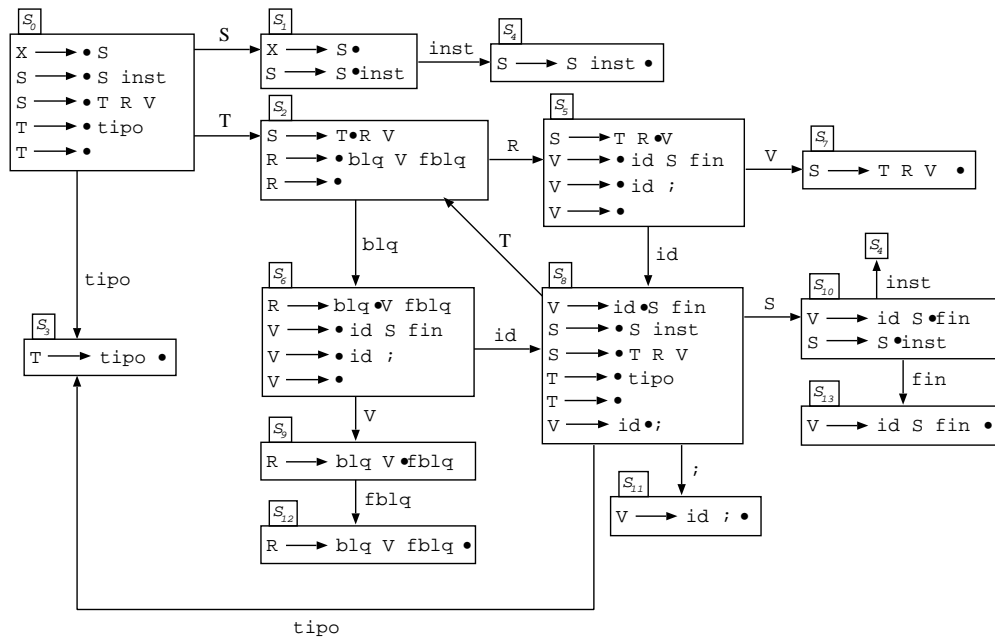
Ejercicio 5.2

Dada la siguiente gramática:

$$\begin{aligned}
 S &\longrightarrow S \text{ inst} \\
 S &\longrightarrow T R V \\
 T &\longrightarrow \text{tipo} \\
 T &\longrightarrow \epsilon \\
 R &\longrightarrow \text{blq } V \text{ fblq} \\
 R &\longrightarrow \epsilon \\
 V &\longrightarrow \text{id } S \text{ fin} \\
 V &\longrightarrow \text{id ;} \\
 V &\longrightarrow \epsilon
 \end{aligned}$$

Escribid la tabla de análisis sintáctico SLR para esta gramática y haced la traza del funcionamiento del analizador SLR para las cadenas “tipo id blq id ; fblq fin inst” e “id tipo id fin ;”

Solución:



Cálculo de los conjuntos de siguientes:

$$\begin{aligned}
 \text{SIG}(S) &= \{ \$, \text{inst}, \text{fin} \} \\
 \text{SIG}(T) &= \{ \text{blq}, \text{id}, \$, \text{inst}, \text{fin} \} \\
 \text{SIG}(R) &= \{ \text{id}, \$, \text{inst}, \text{fin} \} \\
 \text{SIG}(V) &= \{ \text{fblq}, \$, \text{inst}, \text{fin} \}
 \end{aligned}$$

La tabla de análisis correspondiente es:

Estado	Acción								Ir_a			
	inst	tipo	blq	fblq	id	fin	;	\$	S	T	R	V
0	r4	d3	r4		r4	r4		r4	1	2		
1	d4							aceptar				
2	r6		d6		r6	r6		r6			5	
3	r3		r3		r3	r3		r3				
4	r1					r1		r1				
5	r9			r9	d8	r9		r9				7
6	r9			r9	d8	r9		r9				9
7	r2					r2		r2				
8	r4	d3	r4		r4	r4	d11	r4	10	2		
9				d12								
10	d4					d13						
11	r8			r8		r8		r8				
12	r5				r5	r5		r5				
13	r7			r7		r7		r7				

La traza de la cadena de entrada “tipo id blq id ; fblq fin inst” es como sigue:

PILA	ENTRADA	ACCIÓN
0	tipo id blq id ; fblq fin inst \$	d3
0 3	id blq id ; fblq fin inst \$	r3 ($T \rightarrow \text{tipo}$)
0 2	id blq id ; fblq fin inst \$	r6 ($R \rightarrow \epsilon$)
0 2 5	id blq id ; fblq fin inst \$	d8
0 2 5 8	blq id ; fblq fin inst \$	r4 ($T \rightarrow \epsilon$)
0 2 5 8 2	blq id ; fblq fin inst \$	d6
0 2 5 8 2 6	id ; fblq fin inst \$	d8
0 2 5 8 2 6 8	; fblq fin inst \$	d11
0 2 5 8 2 6 8 11	fblq fin inst \$	r8 ($V \rightarrow \text{id ;}$)
0 2 5 8 2 6 9	fblq fin inst \$	d12
0 2 5 8 2 6 9 12	fin inst \$	r5 ($R \rightarrow \text{blq } V \text{ fblq}$)
0 2 5 8 2 5	fin inst \$	r9 ($V \rightarrow \epsilon$)
0 2 5 8 2 5 7	fin inst \$	r2 ($S \rightarrow T R V$)
0 2 5 8 10	fin inst \$	d13
0 2 5 8 10 13	inst \$	r7 ($V \rightarrow \text{id } S \text{ fin}$)
0 2 5 7	inst \$	r2 ($S \rightarrow T R V$)
0 1	inst \$	d4
0 1 4	\$	r1 ($S \rightarrow S \text{ inst}$)
0 1	\$	aceptar

La traza de la cadena de entrada “id tipo id fin ;” es como sigue:

PILA	ENTRADA	ACCIÓN
0	id tipo id fin ; \$	r4 ($T \rightarrow \epsilon$)
0 2	id tipo id fin ; \$	r6 ($R \rightarrow \epsilon$)
0 2 5	id tipo id fin ; \$	d8
0 2 5 8	tipo id fin ; \$	d3
0 2 5 8 3	id fin ; \$	r3 ($T \rightarrow \mathbf{tipo}$)
0 2 5 8 2	id fin ; \$	r6 ($R \rightarrow \epsilon$)
0 2 5 8 2 5	id fin ; \$	d8
0 2 5 8 2 5 8	fin ; \$	r4 ($T \rightarrow \epsilon$)
0 2 5 8 2 5 8 2	fin ; \$	r6 ($R \rightarrow \epsilon$)
0 2 5 8 2 5 8 2 5	fin ; \$	r9 ($V \rightarrow \epsilon$)
0 2 5 8 2 5 8 2 5 7	fin ; \$	r2 ($S \rightarrow T R V$)
0 2 5 8 2 5 8 10	fin ; \$	d13
0 2 5 8 2 5 8 10 13	; \$	error

El error que se debe generar en este caso es:

“Error: encontrado ‘;’ esperaba ‘inst’, ‘fblq’, ‘fin’ o fin de fichero”

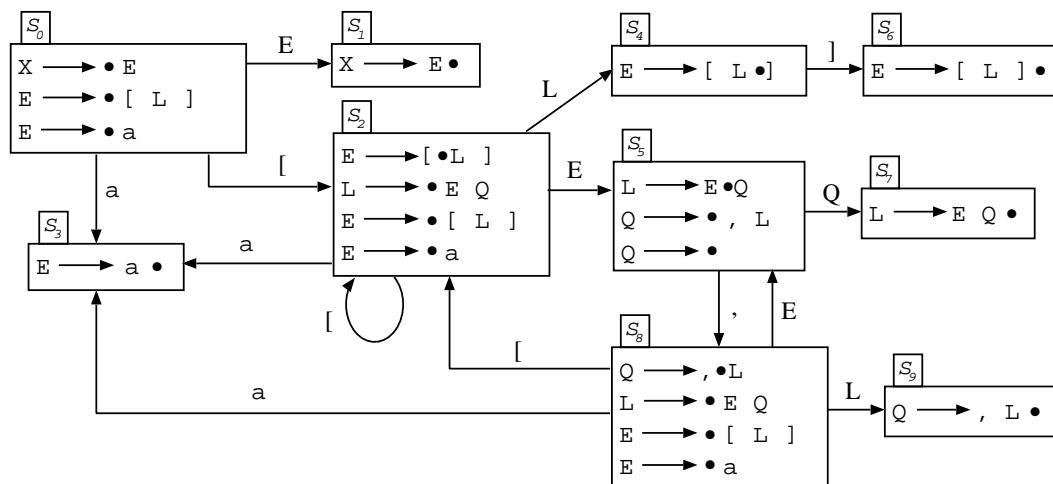
Ejercicio 5.3

Dada la siguiente gramática:

$$\begin{aligned}
 E &\longrightarrow [L] \\
 E &\longrightarrow \mathbf{a} \\
 L &\longrightarrow E Q \\
 Q &\longrightarrow , L \\
 Q &\longrightarrow \epsilon
 \end{aligned}$$

Escribid la tabla de análisis sintáctico SLR para esta gramática y haced la traza del funcionamiento del analizador SLR para la cadena “[a, [a, a],]”.

Solución:



Cálculo de los conjuntos de siguientes:

$$\begin{aligned}\text{SIG}(E) &= \{ \$, , ,] \} \\ \text{SIG}(L) &= \{] \} \\ \text{SIG}(Q) &= \{] \}\end{aligned}$$

La tabla de análisis correspondiente es:

Estado	Acción					Ir_a		
	[]	a	,	\$	E	L	Q
0	d2		d3			1		
1	d2				aceptar			
2	d2		d3			5	4	
3		r2		r2	r2			
4		d6						
5		r5		d8				7
6		r1		r1	r1			
7		r3						
8	d2		d3			5	9	
9		r4						

La traza de la cadena de entrada “[a , [a , a] ,]” es como sigue:

PILA	ENTRADA	ACCIÓN
0	[a , [a , a] ,] \$	d2
0 2	a , [a , a] ,] \$	d3
0 2 3	, [a , a] ,] \$	r2 ($E \rightarrow a$)
0 2 5	, [a , a] ,] \$	d8
0 2 5 8	[a , a] ,] \$	d2
0 2 5 8 2	a , a] ,] \$	d3
0 2 5 8 2 3	, a] ,] \$	r2 ($E \rightarrow a$)
0 2 5 8 2 5	, a] ,] \$	d8
0 2 5 8 2 5 8	a] ,] \$	d3
0 2 5 8 2 5 8 3] ,] \$	r2 ($E \rightarrow a$)
0 2 5 8 2 5 8 5] ,] \$	r5 ($Q \rightarrow \epsilon$)
0 2 5 8 2 5 8 5 7] ,] \$	r3 ($L \rightarrow E Q$)
0 2 5 8 2 5 8 9] ,] \$	r4 ($Q \rightarrow , L$)
0 2 5 8 2 5 7] ,] \$	r3 ($L \rightarrow E Q$)
0 2 5 8 2 4] ,] \$	d6
0 2 5 8 2 4 6	,] \$	r1 ($E \rightarrow [L]$)
0 2 5 8 5	,] \$	d8
0 2 5 8 5 8] \$	error

El error que se debe generar en este caso es:

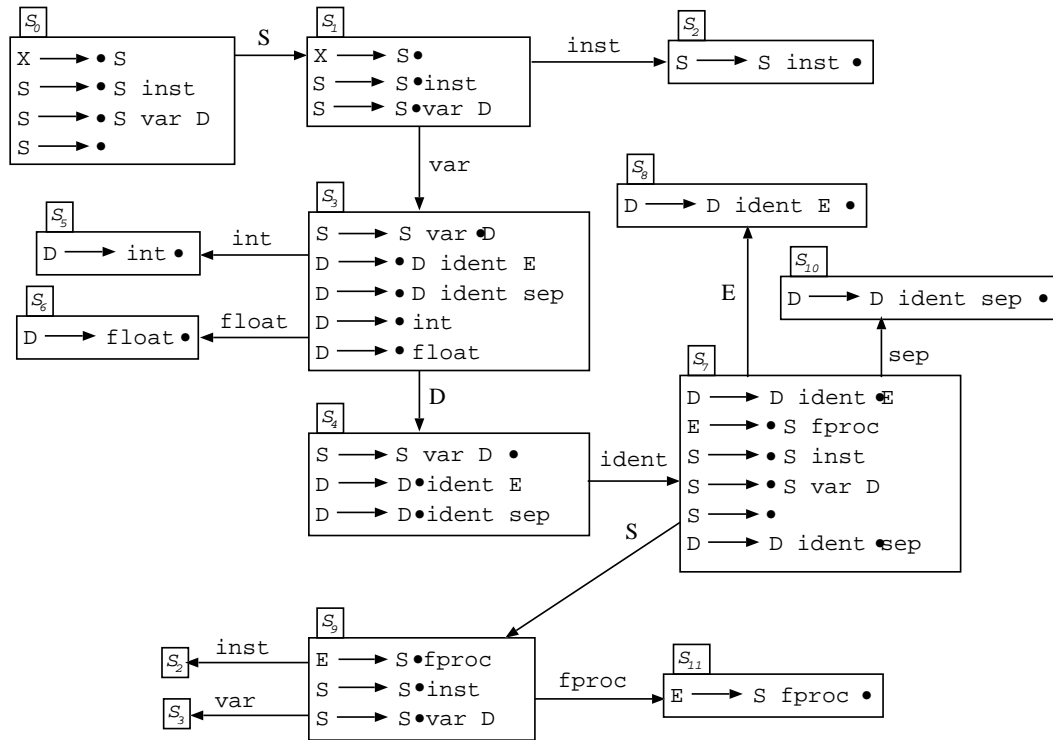
“Error: encontrado ’]’ esperaba ’[’ o ’a’”

Ejercicio 5.4

Dada la siguiente gramática:

$$\begin{aligned}
 S &\longrightarrow S \text{ inst} \\
 S &\longrightarrow S \text{ var } D \\
 S &\longrightarrow \epsilon \\
 D &\longrightarrow D \text{ ident } E \\
 D &\longrightarrow D \text{ ident sep} \\
 D &\longrightarrow \text{int} \\
 D &\longrightarrow \text{float} \\
 E &\longrightarrow S \text{ fproc}
 \end{aligned}$$

Construye una tabla de análisis SLR para ella y haz la traza del funcionamiento del analizador para las cadenas “var int ident inst fproc inst” e “inst fproc”.

Solución:

Cálculo de los conjuntos de siguientes:

$$\begin{aligned}
 \text{SIG}(S) &= \{ \$, \text{inst}, \text{var}, \text{fproc} \} \\
 \text{SIG}(D) &= \{ \$, \text{inst}, \text{var}, \text{fproc}, \text{ident} \} \\
 \text{SIG}(E) &= \{ \$, \text{inst}, \text{var}, \text{fproc}, \text{ident} \}
 \end{aligned}$$

La tabla de análisis correspondiente es:

Estado	Acción								Ir_a		
	inst	var	ident	sep	int	float	fproc	\$	<i>S</i>	<i>D</i>	<i>E</i>
0	r3	r3					r3	r3	1		
1	d2	d3						aceptar			
2	r1	r1					r1	r1			
3					d5	d6				4	
4	r2	r2	d7				r2	r2			
5	r6	r6	r6				r6	r6			
6	r7	r7	r7				r7	r7			
7	r3	r3		d10			r3	r3	9		8
8	r4	r4	r4				r4	r4			
9	d2	d3					d11				
10	r5	r5	r5				r5	r5			
11	r8	r8	r8				r8	r8			

La traza de la cadena de entrada “var int ident inst fproc inst” es como sigue:

PILA	ENTRADA	ACCIÓN
0	var int ident inst fproc inst \$	r3 ($S \rightarrow \epsilon$)
0 1	var int ident inst fproc inst \$	d3
0 1 3	int ident inst fproc inst \$	d5
0 1 3 5	ident inst fproc inst \$	r6 ($D \rightarrow \text{int}$)
0 1 3 4	ident inst fproc inst \$	d7
0 1 3 4 7	inst fproc inst \$	r3 ($S \rightarrow \epsilon$)
0 1 3 4 7 9	inst fproc inst \$	d2
0 1 3 4 7 9 2	fproc inst \$	r1 ($S \rightarrow S \text{ inst}$)
0 1 3 4 7 9	fproc inst \$	d11
0 1 3 4 7 9 11	inst \$	r8 ($E \rightarrow S \text{ fproc}$)
0 1 3 4 7 8	inst \$	r4 ($D \rightarrow D \text{ ident } E$)
0 1 3 4	inst \$	r2 ($S \rightarrow S \text{ var } D$)
0 1	inst \$	d2
0 1 2	\$	r1 ($S \rightarrow S \text{ inst}$)
0 1	\$	aceptar

La traza de la cadena de entrada “inst fproc” es como sigue:

PILA	ENTRADA	ACCIÓN
0	inst fproc \$	r3 ($S \rightarrow \epsilon$)
0 1	inst fproc \$	d2
0 1 2	fproc \$	r1 ($S \rightarrow S \text{ inst}$)
0 1	fproc \$	error

El error que se debe generar en este caso es:

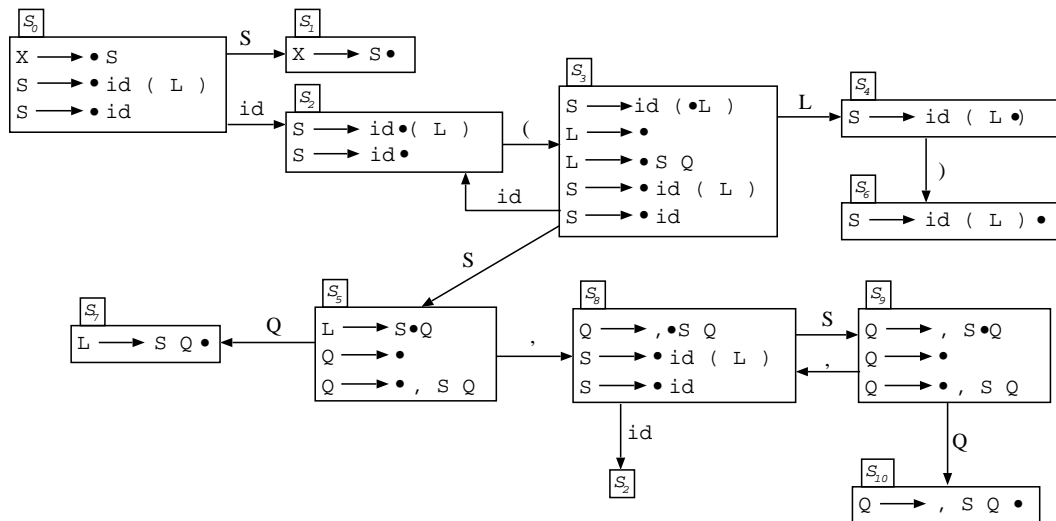
“Error: encontrado ‘fproc’ esperaba ‘inst’, ‘var’ o fin de fichero”

Ejercicio 5.5

Escribe la tabla de análisis sintáctico SLR para la gramática siguiente:

$$\begin{aligned}
 S &\longrightarrow \text{id} (L) \\
 S &\longrightarrow \text{id} \\
 L &\longrightarrow \epsilon \\
 L &\longrightarrow S Q \\
 Q &\longrightarrow \epsilon \\
 Q &\longrightarrow , S Q
 \end{aligned}$$

Haz también la traza de las cadenas “a(b())” y “c(d,e,())”.

Solución:

Cálculo de los conjuntos de siguientes:

$$\begin{aligned}
 \text{SIG}(S) &= \{ \$, , ,) \} \\
 \text{SIG}(L) &= \{) \} \\
 \text{SIG}(Q) &= \{) \}
 \end{aligned}$$

La tabla de análisis correspondiente es:

Estado	Acción					Ir_a		
	id	()	,	\$	S	L	Q
0	d2					1		
1					aceptar			
2		d3	r2	r2	r2			
3	d2		r3			5	4	
4			d6					
5			r5	d8				7
6			r1	r1	r1			
7			r4					
8	d2					9		
9			r5	d8				10
10			r6					

La traza de la cadena de entrada “a (b ())” es como sigue:

PILA	ENTRADA	ACCIÓN
0	a (b ()) \$	d2
0 2	(b ()) \$	d3
0 2 3	b ()) \$	d2
0 2 3 2	()) \$	d3
0 2 3 2 3)) \$	r3 ($L \rightarrow \epsilon$)
0 2 3 2 3 4)) \$	d6
0 2 3 2 3 4 6) \$	r1 ($S \rightarrow \text{id} (L)$)
0 2 3 5) \$	r5 ($Q \rightarrow \epsilon$)
0 2 3 5 7) \$	r4 ($L \rightarrow L Q$)
0 2 3 4) \$	d6
0 2 3 4 6	\$	r1 ($S \rightarrow \text{id} (L)$)
0 1	\$	aceptar

La traza de la cadena de entrada “c (d , e , ())” es como sigue:

PILA	ENTRADA	ACCIÓN
0	c (d , e , ()) \$	d2
0 2	(d , e , ()) \$	d3
0 2 3	d , e , () \$	d2
0 2 3 2	, e , () \$	r2 ($S \rightarrow \text{id}$)
0 2 3 5	, e , () \$	d8
0 2 3 5 8	e , () \$	d2
0 2 3 5 8 2	, () \$	r2 ($S \rightarrow \text{id}$)
0 2 3 5 8 9	, () \$	d8
0 2 3 5 8 9 8	() \$	error

El error que se debe generar en este caso es:

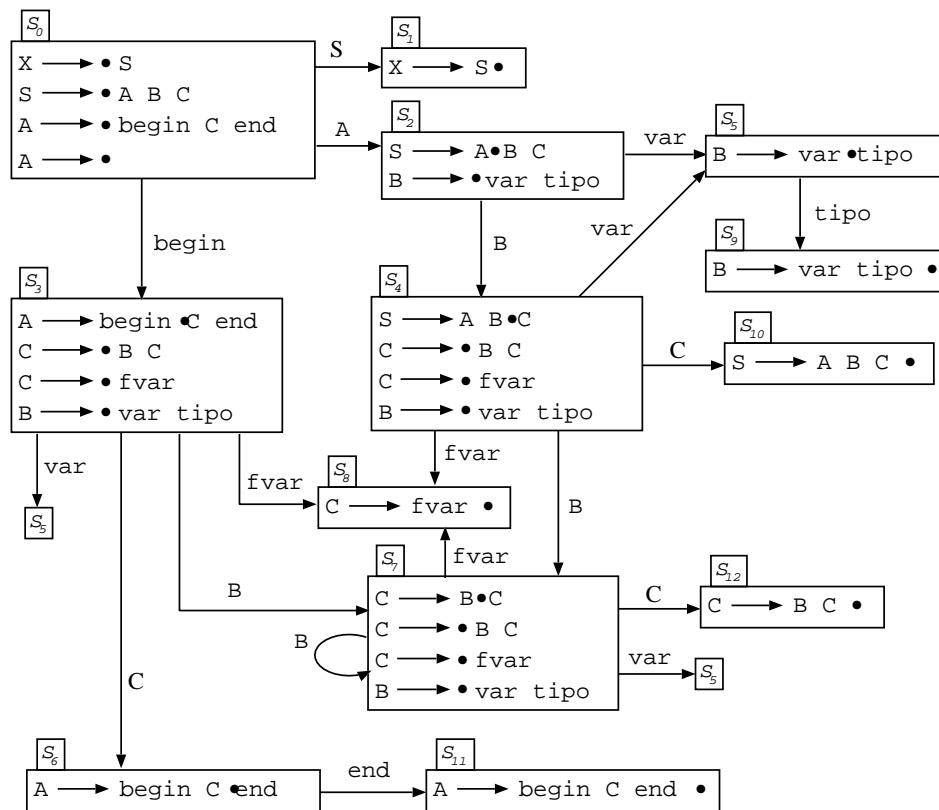
“Error: encontrado ‘(’ esperaba un identificador”

Ejercicio 5.6

Dada la siguiente gramática:

$$\begin{aligned}
 S &\longrightarrow A B C \\
 A &\longrightarrow \text{begin } C \text{ end} \\
 A &\longrightarrow \epsilon \\
 B &\longrightarrow \text{var tipo} \\
 C &\longrightarrow B C \\
 C &\longrightarrow \text{fvar}
 \end{aligned}$$

Sin modificar en absoluto la gramática (excepto la modificación que prescribe el algoritmo, la de añadir una regla al principio), diseña un analizador SLR para esa gramática y haz la traza de la cadena “begin var tipo fvar end var tipo fvar”.

Solución:

Cálculo de los conjuntos de siguientes:

$$\begin{aligned}
 \text{SIG}(S) &= \{ \$ \} \\
 \text{SIG}(A) &= \{ \text{var} \} \\
 \text{SIG}(B) &= \{ \text{var}, \text{fvar} \} \\
 \text{SIG}(C) &= \{ \text{end}, \$ \}
 \end{aligned}$$

La tabla de análisis correspondiente es:

Estado	Acción						Ir_a			
	begin	end	var	tipo	fvar	\$	S	A	B	C
0	d3		r3				1	2		
1						aceptar				
2			d5						4	
3			d5		d8				7	6
4			d5		d8				7	10
5				d9						
6		d11								
7			d5		d8				7	12
8		r6				r6				
9			r4		r4					
10						r1				
11			r2							
12		r5				r5				

La traza de la cadena de entrada “begin var tipo fvar end var tipo fvar” es como sigue:

PILA	ENTRADA	ACCIÓN
0	begin var tipo fvar end var tipo fvar \$	d3
0 3	var tipo fvar end var tipo fvar \$	d5
0 3 5	tipo fvar end var tipo fvar \$	d9
0 3 5 9	fvar end var tipo fvar \$	r4 ($B \rightarrow \text{var tipo}$)
0 3 7	fvar end var tipo fvar \$	d8
0 3 7 8	end var tipo fvar \$	r6 ($C \rightarrow \text{fvar}$)
0 3 7 12	end var tipo fvar \$	r5 ($C \rightarrow B C$)
0 3 6	end var tipo fvar \$	d11
0 3 6 11	var tipo fvar \$	r2 ($A \rightarrow \text{begin } C \text{ end}$)
0 2	var tipo fvar \$	d5
0 2 5	tipo fvar \$	d9
0 2 5 9	fvar \$	r4 ($B \rightarrow \text{var tipo}$)
0 2 4	fvar \$	d8
0 2 4 8	\$	r6 ($C \rightarrow \text{fvar}$)
0 2 4 10	\$	r1 ($S \rightarrow A B C$)
0 1	\$	aceptar

A.5 Soluciones a los ejercicios del capítulo 6

Ejercicio 6.1

Diseñar un esquema de traducción para asignar tipo a una lista de variables declaradas en Pascal, siendo la sintaxis la siguiente.

<i>Variables</i>	→	var <i>Declaración</i>
<i>Declaración</i>	→	identificador <i>Lista</i>
<i>Lista</i>	→	coma identificador <i>Lista</i>
<i>Lista</i>	→	dosptos <i>Tipo</i> ptocomas
<i>Tipo</i>	→	integer real

Solución:

<i>Variables</i>	→	var <i>Declaración</i>
<i>Declaración</i>	→	identificador { <i>Guarda</i> (identificador.lexema) } <i>Lista</i> { <i>AsignarTipo</i> (identificador.lexema , <i>Lista.ts</i>) }
<i>Lista</i>	→	coma identificador { <i>Guarda</i> (identificador.lexema) } <i>Lista</i> ₁ { <i>AsignarTipo</i> (identificador.lexema , <i>Lista</i> ₁ . <i>ts</i>); <i>Lista.ts</i> := <i>Lista</i> ₁ . <i>ts</i> }
<i>Lista</i>	→	dosptos <i>Tipo</i> ptocomas { <i>Lista.ts</i> := <i>Tipo.ts</i> }
<i>Tipo</i>	→	integer { <i>Tipo.ts</i> := <i>integer</i> }
<i>Tipo</i>	→	real { <i>Tipo.ts</i> := <i>real</i> }

Nota: Los identificadores se almacenan sin tipo en la tabla de símbolos según aparecen. La función *AsignarTipo* los busca en la tabla y les asigna el tipo.

Ejercicio 6.2

Diseñar un esquema de traducción para asignar tipo a una lista de variables declaradas en C, siendo la sintaxis la siguiente.

<i>Declaración</i>	→	<i>Tipo</i> identificador <i>Lista</i> ptocomas
<i>Lista</i>	→	coma identificador <i>Lista</i>
<i>Lista</i>	→	ε
<i>Tipo</i>	→	int float

Solución:

<i>Declaración</i>	→	<i>Tipo</i> identificador { <i>GuardaconTipo</i> (identificador.lexema , <i>Tipo.ts</i>); <i>Lista.h</i> := <i>Tipo.ts</i> } <i>Lista</i> ptocomas
<i>Lista</i>	→	coma identificador { <i>GuardaconTipo</i> (identificador.lexema , <i>Lista.h</i>); <i>Lista</i> ₁ . <i>h</i> := <i>Lista.h</i> } <i>Lista</i> ₁
<i>Lista</i>	→	ε
<i>Tipo</i>	→	int { <i>Tipo.ts</i> := <i>int</i> }
<i>Tipo</i>	→	float { <i>Tipo.ts</i> := <i>float</i> }


```

                                puntero : enteras : a
                                fvar

float **a;                    variables
                                puntero : puntero : reales : a
                                fvar

```

El lenguaje fuente está generado por la siguiente gramática:

$$\begin{aligned}
 S &\longrightarrow T D ; \\
 D &\longrightarrow D , V \\
 D &\longrightarrow V \\
 V &\longrightarrow E \\
 V &\longrightarrow V [\text{ nint }] \\
 E &\longrightarrow * E \\
 E &\longrightarrow \text{ id } \\
 T &\longrightarrow \text{ int } \mid \text{ float } \mid \text{ char }
 \end{aligned}$$

Solución:

$$\begin{aligned}
 S &\longrightarrow T \{ D.th := T.trad \} D ; \{ S.trad := \text{ "variables" } \parallel D.trad \parallel \text{ "fvar" } \} \\
 D &\longrightarrow \{ D_1.th := D.th \} D_1 , \{ V.th := D.th \} V \{ \\
 &\quad \text{ if } (V.tab == NULL) \\
 &\quad \quad D.trad := D_1.trad \parallel \text{ ";" } \parallel V.trad \\
 &\quad \text{ else } \\
 &\quad \quad D.trad := D_1.trad \parallel \text{ ";" } \parallel V.tab \parallel V.trad \\
 &\quad \} \\
 D &\longrightarrow \{ V.th := D.th \} V \{ \\
 &\quad \text{ if } (V.tab == NULL) \\
 &\quad \quad D.trad := V.trad \\
 &\quad \text{ else } \\
 &\quad \quad D.trad := V.tab \parallel V.trad \\
 &\quad \} \\
 V &\longrightarrow \{ E.th := V.th \} E \{ V.trad := E.trad; V.tab := NULL \} \\
 V &\longrightarrow \{ V_1.th := V.th \} V_1 [\text{ nint }] \{ \\
 &\quad V.trad := V_1.trad; \\
 &\quad \text{ if } (V_1.tab == NULL) \\
 &\quad \quad V.tab := \text{ "tabla [" } \parallel \text{ nint.lexema } \parallel \text{ "]" : " } \\
 &\quad \text{ else } \\
 &\quad \quad V.tab := V_1.tab \parallel \text{ "tabla [" } \parallel \text{ nint.lexema } \parallel \text{ "]" : " } \\
 &\quad \} \\
 E &\longrightarrow * \{ E_1.th := E.th \} E_1 \{ E.trad := \text{ "puntero: " } \parallel E_1.trad \} \\
 E &\longrightarrow \text{ id } \{ E.trad := E.th \parallel \text{ id.lexema } \} \\
 T &\longrightarrow \text{ int } \{ T.trad := \text{ "enteras : " } \} \\
 T &\longrightarrow \text{ float } \{ T.trad := \text{ "reales : " } \} \\
 T &\longrightarrow \text{ char } \{ T.trad := \text{ "caracteres : " } \}
 \end{aligned}$$

Ejercicio 6.5

Diseñar e implementar un ETDS para la traducción de los siguientes ejemplos:

```

int a,b;          a,b : integer;

struct s {        d,e : registro [
    int a,b;      a(s),b(s):integer;
    float c;      c(s):real;
} d,e;            ];

struct s1 {       h : registro [
    int a,b,b2;   a(s1),b(s1),b2(s1):integer;
    struct s2 {   e(s1),f(s1):registro [
        float c,d;      c(s2),d(s2):real;
    } e,f;              ];
} h;                ];

```

El lenguaje fuente está generado por la siguiente gramática:

$$\begin{aligned}
 S &\longrightarrow D \\
 D &\longrightarrow T \text{ id } L \ ; \\
 T &\longrightarrow \text{int} \\
 T &\longrightarrow \text{float} \\
 T &\longrightarrow \text{char} \\
 T &\longrightarrow \text{struct id lbra } D \ M \ \text{rbra} \\
 M &\longrightarrow D \ M \\
 M &\longrightarrow \epsilon \\
 L &\longrightarrow \ , \ \text{id } L \\
 L &\longrightarrow \epsilon
 \end{aligned}$$

Solución:

```

S  → { D.vh := NULL } D { S.trad := D.trad }
D  → T id { L.vh := D.vh } L ; {
      if (D.vh == NULL)
        D.trad := id.lexema || L.trad || T.trad || “;”
      else
        D.trad := id.lexema || “(” || D.vh || “)” || L.trad || T.trad || “;”
      }
T  → int { T.trad := “integer” }
T  → float { T.trad := “real” }
T  → char { T.trad := “char” }
T  → struct id lbra { D.vh := id.lexema } D { M.vh := id.lexema } M rbra {
      T.trad := “registro [” || D.trad || M.trad || “]” }
M  → { D.vh := M.vh } D { M1.vh := M.vh } M1 {
      if (M1.trad == NULL)
        M.trad := D.trad
      else
        M.trad := D.trad || M1.trad
      }
M  → ε { M.trad := NULL }
L  → , id { L1.vh := L.vh } L1 {
      if (L.vh == NULL)
        L.trad := “,” || id.lexema || L1.trad
      else
        L.trad := “,” || id.lexema || “(” || L.vh || “)” || L1.trad
      }
L  → ε { L.trad := “:” }

```

Ejercicio 6.6

¿Por qué no se puede escribir un ETDS directamente a partir de la siguiente DDS (sin modificarla, solamente introduciendo las acciones semánticas en la parte derecha de las reglas)?

PRODUCCIÓN	REGLA SEMÁNTICA
$A \rightarrow B C$	$B.h := C.s$
$B \rightarrow B_1 \mathbf{b}$	if $B.h \geq 0$ then $B_1.h := B.h - 1;$ $mark(\mathbf{b}.dir);$ endif
$B \rightarrow \epsilon$	
$C \rightarrow \mathbf{c} C_1$	$C.s := C_1.s + 1$
$C \rightarrow \epsilon$	$C.s := 0$

Solución:

La primera acción semántica, $B.h := C.s$, impide que la DDS se pueda implementar como un ETDS, ya que al atributo heredado $B.h$ se le debe asignar valor antes del símbolo al que pertenece, B , pero el valor que se le asigna es el atributo sintetizado $C.s$ que tendrá valor después de analizar el símbolo C de la regla. Es decir, esta acción implica una herencia de derecha a izquierda en el árbol, lo cual hace que la DDS incumpla las restricciones de una GAI (que un ETDS debe cumplir).

La segunda acción semántica tampoco se puede implementar, ya que se debe situar antes del símbolo B_1 , y está actuando sobre un atributo, $\mathbf{b.dir}$, que todavía no tiene valor, ya que aún no se ha leído \mathbf{b} .

Ejercicio 6.7

Transformar la siguiente DDS en un ETDS, situando las acciones semánticas en los lugares apropiados en las partes derechas de las reglas.

PRODUCCIÓN	REGLA SEMÁNTICA
$S \rightarrow \mathbf{lbra} \ B \ \mathbf{rbra}$	$B.bh := \text{NULL}; S.cs := B.cs;$ $B.ch := \text{NULL};$
$B \rightarrow I; B_1$	$B_1.bh := B.bh; B.cs := I.cs \parallel B_1.cs;$ $I.bh := B.bh; B_1.ch := B.ch;$ $I.ch := B.ch;$
$B \rightarrow \epsilon$	$B.cs := \text{NULL};$
$I \rightarrow \mathbf{lbra} \ B \ \mathbf{rbra}$	$B.bh := I.bh; I.cs := B.cs;$ $B.ch := I.ch;$
$I \rightarrow \mathbf{while} \ (\ E) \ I_1$	$I_1.bh := e1; e1 := \text{etiquueva}();$ $I_1.ch := e2; e2 := \text{etiquueva}();$ $I.cs := \text{gen}(e2 \ " : ") \parallel E.cs$ $\parallel \text{gen}(\text{"jumpz"} \ e1) \parallel I_1.cs$ $\parallel \text{gen}(e1 \ " : ") ;$
$I \rightarrow \mathbf{print} \ E$	$I.cs := \text{gen}(\text{"print"} \ ") \parallel E.cs;$
$I \rightarrow \mathbf{break}$	if $I.bh \neq \text{NULL}$ then $I.cs := \text{gen}(\text{"jump"} \ I.bh)$ else $\text{error}()$ endif
$I \rightarrow \mathbf{continue}$	if $I.ch \neq \text{NULL}$ then $I.cs := \text{gen}(\text{"jump"} \ I.ch)$ else $\text{error}()$ endif
$E \rightarrow \mathbf{a}$	$E.cs := \text{gen}(\text{" a"});$

Solución:

```

S  →  lbra { B.bh := NULL; B.ch := NULL } B rbra { S.cs := B.cs }
B  →  { I.bh := B.bh; I.ch := B.ch } I ; { B1.bh := B.bh; B1.ch := B.ch } B1 {
      B.cs := I.cs || B1.cs }
B  →  ε { B.cs := NULL }
I  →  lbra { B.bh := I.bh; B.ch := I.ch } B rbra { I.cs := B.cs }
I  →  while ( E ) { e1 := etiqnueva(); I1.bh := e1; e2 := etiqnueva(); I1.ch := e2; }
      I1 { I.cs := gen(e2“:”) || E.cs || gen(“jumpz ”e1) || I1.cs || gen(e1“:”) }
I  →  print E { I.cs := gen(“print ”) || E.cs }
I  →  break {
      if (I.bh != NULL) then
        I.cs := gen(“jump ”I.bh)
      else
        error()
      endif }
I  →  continue {
      if (I.ch != NULL) then
        I.cs := gen(“jump ”I.ch)
      else
        error()
      endif }
E  →  a { E.cs := gen(“ a ”) }

```

Ejercicio 6.8

Explicar si la siguiente DDS se puede convertir en un ETDS (situando las acciones semánticas donde corresponda sin modificar la gramática en absoluto).

PRODUCCIÓN	REGLAS SEMÁNTICAS
$D \rightarrow T L$	$L.tip := T.tipo$
$T \rightarrow \mathbf{int}$	$T.tipo := \text{integer}$
$T \rightarrow \mathbf{real}$	$T.tipo := \text{real}$
$L \rightarrow L_1 , I$	$L_1.tip := L.tip ; I.tip := L.tip$
$L \rightarrow I$	$I.tip := L.tip$
$I \rightarrow I_1 [\mathbf{num}]$	$I_1.tip := \text{array}(\mathbf{num.val}, I.tip)$
$I \rightarrow \mathbf{id}$	$\text{introtip}(\mathbf{id.lexema}, I.tip)$

Solución:

No se puede convertir en un ETDS porque la acción $I_1.tip := \text{array}(\mathbf{num.val}, I.tip)$ se debe situar antes del símbolo I_1 , pero está utilizando el valor de un atributo sintetizado, $\mathbf{num.val}$, que todavía no tiene valor porque \mathbf{num} no ha sido analizado.

A.6 Soluciones a los ejercicios del capítulo 7

Ejercicio 7.1

A partir del siguiente ETDS, aplíquense las técnicas para eliminar la recursividad por la izquierda que se estudian en el epígrafe 7.2.1 e implementese el ETDS resultante como un traductor descendente recursivo.

$$\begin{aligned}
 S &\longrightarrow L \textbf{ pyc } \{ S.a := \text{"var " } \parallel L.a \parallel \text{":" } \parallel L.b \parallel \text{";} \} \\
 L &\longrightarrow L_1 \textbf{ coma id } \{ L.a := L_1.a \parallel \text{"," } \parallel \textbf{id.lexema}; L.b := L_1.b \} \\
 L &\longrightarrow T \textbf{ id } \{ L.a := \textbf{id.lexema}; L.b := T.t \} \\
 T &\longrightarrow \textbf{int} \{ T.t := \text{"integer"} \} \\
 T &\longrightarrow \textbf{float} \{ T.t := \text{"real"} \}
 \end{aligned}$$

Solución:

$$\begin{aligned}
 S &\longrightarrow L \textbf{ pyc } \{ S.a := \text{"var " } \parallel L.a \parallel \text{":" } \parallel L.b \parallel \text{";} \} \\
 L &\longrightarrow T \textbf{ id } \{ L'.ha := \textbf{id.lexema}; L'.hb := T.t \} L' \{ L.a := L'.sa; \\
 &\quad L.b := L'.sb \} \\
 L' &\longrightarrow \textbf{coma id} \{ L'_1.ha := L'.ha \parallel \text{"," } \parallel \textbf{id.lexema}; L'_1.hb := L'.hb \} \\
 &\quad L'_1 \{ L'.sa := L'_1.sa; L'.sb := L'_1.sb \} \\
 L' &\longrightarrow \epsilon \{ L'.sa := L'.ha; L'.sb := L'.hb \} \\
 T &\longrightarrow \textbf{int} \{ T.t := \text{"integer"} \} \\
 T &\longrightarrow \textbf{float} \{ T.t := \text{"real"} \}
 \end{aligned}$$

En primer lugar se definen registros de atributos para los símbolos L y L' :

```

typedef struct{
    char *a,*b;
}AtrL;

typedef struct{
    char *ha,*hb;
    char *sa,*sb;
}AtrLp;

```

A continuación se escriben las funciones correspondientes a cada no terminal:

```

char *S(void)
{
    char *S_a;
    AtrL atl;

    L(&atl);
    emparejar(PYC);
    S_a=concat("var ",atl.a,":",atl.b,";");
    return(S_a);
}

void L(AtrL *atl)

```

```

{
    char *T_t,*Id_lexema;
    AtrLp atlp;

    T_t=T();
    Id_lexema=strdup(lexema);
    emparejar(ID);
    atlp.ha=Id_lexema;
    atlp.hb=T_t;
    Lp(&atlp);
    atl->a=atlp.sa;
    atl->b=atlp.sb;
}

void Lp(AtrLp *atlp)
{
    char *Id_lexema;
    AtrLp atlp1;

    if (preanalisis==COMA)
    {
        emparejar(COMA);
        Id_lexema=strdup(lexema);
        emparejar(ID);
        atlp1.ha=concat(atlp->ha,"",Id_lexema);
        atlp1.hb=atlp->hb;
        Lp(&atlp1);
        atlp->sa=atlp1.sa;
        atlp->sb=atlp1.sb;
    }
    else
    {
        if (preanalisis==PYC)
        {
            atlp->sa=atlp->ha;
            atlp->sb=atlp->hb;
        }
        else ErrorSintactico(lexema,COMA,PYC);
    }
}

char *T(void)
{
    char *T_t;

    if (preanalisis==INT)
        T_t=strdup("integer");
    else
        if (preanalisis==FLOAT)
            T_t=strdup("real");
        else ErrorSintactico(lexema,INT,FLOAT);
}

```



```

    return(T_t);
}

```

Ejercicio 7.2

Conviértase este ETDS a la notación de YACC.

$$\begin{aligned}
 S &\longrightarrow T \textbf{ var } \{A.h := T.s\} A \{S.s := A.s\} \\
 T &\longrightarrow \textbf{ integer } \{T.s := \text{"int "}\} \\
 T &\longrightarrow \textbf{ real } \{T.s := \text{"float "}\} \\
 A &\longrightarrow B \{C.h := A.h \parallel B.s\} C \{A.s := C.s\} \\
 B &\longrightarrow \textbf{ id } \{B.s := \textbf{id.lexema}\} \\
 C &\longrightarrow \epsilon \{C.s := C.h\} \\
 C &\longrightarrow \textbf{ coma } B \{C_1.h := C.h \parallel \text{" , " } \parallel B.s\} C_1 \{C.s := C_1.s\}
 \end{aligned}$$
Solución:

```

s : t VAR a {$.s = $3.s;}
;
t : INTEGER {$.s = strdup("int ");}
  | REAL {$.s = strdup("float ");}
;
a : b {$.s = concat($-1.s,$1.s);} c {$.s = $3.s;}
;
b : ID {$.s = strdup($1.lexema); }
;
c : COMA b {$.s = concat($0.s,"",$2.s);} c {$.s = $4.s;}
  | {$.s = $0.s;}
;

```

Ejercicio 7.3

Transfórmese este ETDS a la notación de YACC.

$$\begin{aligned}
 S &\longrightarrow T \textbf{ id } \{ \text{InsertaVar}(\textbf{id.lexema}, T.t); L.th = T.t; \} \\
 &\quad L \\
 T &\longrightarrow \textbf{ int } \{ T.t := \text{ENTERO}; \} \\
 T &\longrightarrow \textbf{ float } \{ T.t := \text{FLOTANTE}; \} \\
 L &\longrightarrow \textbf{ , id } \{ \text{InsertaVar}(\textbf{id.lexema}, L.th); L_1.th = L.th; \} \\
 &\quad L_1 \\
 L &\longrightarrow ;
 \end{aligned}$$
Solución:

```

s : t ID {InsertaVar($2.lexema,$1.t); $.t = $1.t;} 1

```

```

;
t : INT    {$$t = ENTERO;}
  | FLOAT {$$t = FLOTANTE;}
;
1 : COMA ID {InsertaVar($2.lexema,$0.t); $$t = $0.t;} 1
  | PUNTCO
;

```

Ejercicio 7.4

Conviértase este ETDS a la notación de YACC. Todos los atributos son cadenas de caracteres.

S	\longrightarrow	NA	$\{C.prim := A.s\}$
		coma B	$\{C.sec := B.s\}$
		C	$\{S.s := N.s C.s\}$
N	\longrightarrow	ϵ	$\{N.s := ""\}$
N	\longrightarrow	D	$\{C.prim := D.s\}$
		B	$\{C.sec := B.s\}$
		C	$\{N.s := C.s\}$
B	\longrightarrow	id	$\{B.s := id.lexema\}$
A	\longrightarrow	var id	$\{A.s := id.lexema\}$
D	\longrightarrow	func id	$\{D.s := id.lexema\}$
C	\longrightarrow	integer	$\{C.s := "int " C.prim C.sec\}$
C	\longrightarrow	real	$\{C.s := "float " C.prim C.sec\}$

Solución:

Una posible solución sería:

```

s : n a COMA {$$prim = $2.s;} b c {$$s = concat($1.s,$6.s);}
;
n : d {$$prim = $1.s;} b c {$$s = $4.s;}
  | {$$s = strdup("");}
;
b : ID {$$s = strdup($1.lexema);}
;
a : VAR ID {$$s = strdup($2.lexema);}
;
c : INTEGER {$$s = concat("int ",$-1.prim,$0.s);}
  | REAL {$$s = concat("float ",$-1.prim,$0.s);}
;
d : FUNC ID {$$s = strdup($2.lexema);}
;

```

Otra posible solución es:

```

s : n a COMA b c  {$$$.s = concat($1.s,$5.s);}
;
n :   d { }   b c  {$$$.s = $4.s;}
    | {$$$.s = strdup("");}
;
b : ID  {$$$.s = strdup($1.lexema);}
;
a : VAR ID  {$$$.s = strdup($2.lexema);}
;
c : INTEGER {$$$.s = concat("int ",$-2.s,$0.s);}
    | REAL   {$$$.s = concat("float ",$-2.s,$0.s);}
;
d : FUNC ID  {$$$.s = strdup($2.lexema);}
;

```

Ejercicio 7.5

Dado el siguiente ETDS, tradúzcase a la notación de YACC.

S	\longrightarrow	decl A	$\{C.t := A.s\}$
		variables C	$\{S.s := C.s\}$
A	\longrightarrow	id	$\{pos := Busca(id.lexema);\}$
		D	$\{C.t := D.s\}$
		C	$\{Guarda(pos, C.s); A.s := C.s\}$
B	\longrightarrow	real	$\{B.s := REAL\}$
B	\longrightarrow	integer	$\{B.s := ENTERO\}$
C	\longrightarrow		$\{C_1.t := C.t\}$
		C_1 id	$\{Guarda(Busca(id.lexema), C_1.s);$
			$C.s := C_1.s\}$
C	\longrightarrow	ϵ	$\{C.s := C.t\}$
D	\longrightarrow	var B dospto	$\{C.t := B.s\}$
		C fvar	$\{D.s := C.s\}$

Solución:

Este ejercicio, como el anterior, tiene varias soluciones. Una de ellas es la siguiente:

```

s : DECL a VARIABLES  {$$$.t = $2.s;}  c  {$$$.s = $5.s;}
;
a : ID  {$$$.pos = Busca($1.lexema);}  d  {$$$.t = $3.s;}  c
                                     {Guarda($2.pos,$5.s);
                                     $$$$.s = $5.s;}
;
b : REAL    {$$$.s = REAL;}
    | INTEGER {$$$.s = ENTERO;}
;
c : c ID  {Guarda(Busca($2.lexema), $1.s);}
    | {$$$.s = $0.t;}

```

```

;
d : VAR d DOSPTO  {$.t = $.s;}  c  FVAR  {$.s = $.s;}
;

```

Otra posible solución es:

```

s : DECL a VARIABLES c  {$.s = $.s;}
;
a : ID  {$.pos = Busca($1.lexema);} d { } c
                                     {Guarda($2.pos,$5.s);
                                      $.s = $.s;}
;
b : REAL    {$.s = REAL;}
    | INTEGER {$.s = ENTERO;}
;
c : c ID  {Guarda(Busca($2.lexema),$1.s);}
    | {$.s = $.s;}
;
d : VAR d DOSPTO c FVAR  {$.s = $.s;}
;

```

Ejercicio 7.6

Dado el siguiente ETDS, tradúzcase a la notación de YACC.

S	\rightarrow	main A	$\{B.k := A.a\}$
		B	$\{C.p := A.a ; C.q := B.b\}$
		C	$\{S.s := \text{"principal"} \parallel C.c\}$
C	\rightarrow	code	$\{C.c := C.p \parallel \text{"codigo"} \parallel C.q\}$
A	\rightarrow	$D E$	$\{C.p := D.d ; C.q := E.e\}$
		C	$\{A.a := C.c\}$
B	\rightarrow	declaration	$\{B.b := B.k \parallel \text{"declaracion"}\}$
D	\rightarrow	var	$\{D.d := \text{"var"}\}$
E	\rightarrow	type	$\{E.e := \text{"tipo"}\}$

Solución:

```

s : MAIN a b  {$.p = $.a; $.q = $.b;}  c
               {$.s = concat("principal ", $.c);}
;
c : CODE  {$.c = concat($0.p, " codigo ", $0.q);}
;
a : d e {$.p = $.d; $.q = $.e;}  c  {$.a = $.c;}
;
b : DECLARATION  {$.b = concat($0.a, "declaracion ");}
;
d : VAR  {$.d = strdup("var");}

```

```
    ;  
    e : TYPE  {$$e = strdup("tipo");}  
    ;
```

A.7 Soluciones a los ejercicios del capítulo 8

Ejercicio 8.1

Dada la siguiente declaración de variables en C:

```
struct {
    int a;
    float b[10][5];
    char **c[4];
    struct {
        float a;
        int *c;
    } d,e;
} f,g;
```

1. Escribanse las expresiones de tipos correspondientes a las variables declaradas.
2. Describanse brevemente los contenidos de la tabla de tipos y de la tabla de símbolos global.
3. Suponiendo que los tipos básicos (`int`, `float` y `char`) ocupan todos lo mismo (una posición de memoria), y que los punteros ocupan también una posición de memoria, ¿cuál sería la dirección de `g.e.a` si la dirección donde comienza `f` es la 100? ¿Es posible calcular esta dirección durante la compilación? ¿Y la de `g.b[f.a][g.a]`?

Solución:

1. El tipo de `f` y `g` es exactamente el mismo, y podría definirse con la siguiente expresión de tipos:

```
registro((a : entero) ×
  (b : array(0..9, array(0..4, real))) ×
  (c : array(0..3, puntero(puntero(carácter)))) ×
  (d : registro((a : real) × (c : puntero(entero)))) ×
  (e : registro((a : real) × (c : puntero(entero)))))
```

2. La tabla de tipos y las tablas de símbolos de los registros son:

TABLA DE TIPOS

NÚM.	TIPO	D/T	T.BASE
0	ENTERO		
1	REAL		
2	CARÁCTER		
3	ARRAY	5	1
4	ARRAY	10	3
5	PUNTERO		2
6	PUNTERO		5
7	ARRAY	4	6
8	PUNTERO		0
9	REGISTRO		TS1
10	REGISTRO		TS2

TABLA DE SÍMBOLOS TS1

NOMBRE	TIPO	DIRECCIÓN
a	1	0
c	8	1

TABLA DE SÍMBOLOS TS2

NOMBRE	TIPO	DIRECCIÓN
a	0	0
b	4	1
c	7	51
d	9	55
e	9	57

La tabla de símbolos sería:

TABLA DE SÍMBOLOS GLOBAL

NOMBRE	TIPO	DIRECCIÓN
f	10	100
g	10	159

3. La dirección de `g.e.a` sería:

$$159(g.) + 57(e.) + 0(a) = 216$$

Puesto que todas las direcciones son conocidas para el compilador, sería posible calcular esa dirección en tiempo de compilación. En realidad, el *front end* generaría código para realizar ese cálculo en tiempo de ejecución, pero el optimizador se daría cuenta de que es posible realizar el cálculo en tiempo de compilación y sustituiría ese código por el valor calculado.

Sin embargo, la dirección de `g.b[f.a][g.a]` en principio no podría calcularse en tiempo de compilación, puesto que es muy difícil o incluso imposible conocer el valor que tendrían `f.a` y `g.a` en tiempo de ejecución. De todas maneras, el *front end* haría lo mismo que en el caso anterior, generando código para realizar el cálculo en tiempo de ejecución. Si el optimizador puede averiguar con total seguridad el valor de `f.a` y `g.a`, podría calcular la dirección y sustituir todo el código por el valor calculado, como por ejemplo si aparece la siguiente secuencia de instrucciones:

```
f.a = 7;
g.a = 2;
g.b[f.a][g.a] = 27 ; /* el optimizador lo convierte en la
                        posición de memoria de 'g.b[7][2]' */
```

Ejercicio 8.2

Dada la siguiente declaración de variables en C:

```
int a,*b[10];
int principal(int numargs,char **argumentos);
```

```

struct {
    float c[4][5];
    struct {
        float x,y;
    } puntos[15];
} d,f;

```

y suponiendo que los tipos básicos y los punteros ocupan sólo una posición de memoria,

1. Escribanse las expresiones de tipos de cada uno de los símbolos que aparecen en esas declaraciones.
2. Dibújese la tabla de tipos y la tabla de símbolos en el estado en que quedarían después de procesar las declaraciones.
3. Explíquese cómo se calcularía la posición de memoria en la que está `d.puntos[7].y`.
4. ¿Puede el compilador calcular *en todos los casos* esa posición de memoria u otras con distinto índice para `puntos`? ¿por qué?

Solución:

1. Las expresiones de tipos serían:

a	entero
b	<code>array(0..9, puntero(entero))</code>
principal	<code>(entero × puntero(puntero(carácter))) → entero</code>
d	<code>registro((c : array(0..3, array(0..4, real))) × (puntos : array(0..14, registro((x : real) × (y : real))))</code>
f	(el mismo tipo que d)

2. La tabla de tipos y las tablas de símbolos de los registros son:

NÚM.	TIPO	D/T	T.BASE
0	ENTERO		
1	REAL		
2	CARÁCTER		
3	PUNTERO		0
4	ARRAY	10	3
5	PUNTERO		2
6	PUNTERO		5
7	PCART	0	6
8	FUNCIÓN	7	0
9	ARRAY	5	1
10	ARRAY	4	9
11	REGISTRO		TS1
12	ARRAY	15	11
13	REGISTRO		TS2

NOMBRE	TIPO	DIRECCIÓN
x	1	0
y	1	1

NOMBRE	TIPO	DIRECCIÓN
c	10	0
puntos	12	20

La tabla de símbolos sería:

TABLA DE SÍMBOLOS GLOBAL		
NOMBRE	TIPO	DIRECCIÓN
a	0	100
b	4	101
principal	8	Etiqu.
d	13	111
f	13	161

3. La dirección de “d.puntos[7].y” se calcularía de la siguiente manera:

$$111(d.) + 20(\text{puntos}) + 7 * 2(\text{tam. registro}) + 1(y) = 146$$

4. En este caso, el *front end* podría calcular esa dirección, pero genera código para que ese cálculo se haga en tiempo de ejecución. Sin embargo, si el índice de `puntos` es constante (o puede ser constante), el optimizador sustituye todo el código generado por el *front end* por la dirección ya calculada.

Ejercicio 8.3

Escríbase un ETDS que realice la siguiente traducción:

Lenguaje fuente: expresiones aritméticas con enteros, reales, identificadores de variables *predeclaradas*, los operadores “+”, “-”, “(”, “)”, “[” i “]”, donde los operadores tienen la misma precedencia y asociatividad que en C o en Pascal (excepto los corchetes, que funcionan como los paréntesis pero entregan la parte entera de la expresión que engloban; por ejemplo, [3.5+1] vale 4). La gramática que describe este lenguaje es muy similar a las utilizadas en los ejemplos de este y otros capítulos.

Lenguaje objeto: un lenguaje basado en enteros, reales, identificadores y las funciones siguientes:

<code>int si(int, int)</code>	suma entera
<code>int ri(int, int)</code>	resta entera
<code>float sr(float, float)</code>	suma real
<code>float rr(float, float)</code>	resta real
<code>int t(float)</code>	truncamiento a parte entera
<code>float r(int)</code>	conversión de entero a real

Al contrario que en el lenguaje fuente, en el lenguaje objeto no se permiten las conversiones implícitas de tipos, y por tanto los parámetros de estas funciones deben ser del tipo indicado en el prototipo².

Restricciones semánticas: Cuando un operador encuentra dos operandos enteros, el resultado es entero; si alguno de los operandos es real, el resultado tiene que ser real.

²Es decir, no se puede hacer una llamada como `sr(si(3,4.5),3.0)`.

Ejemplos: si a es real y b es entero, la siguiente tabla muestra las traducciones de algunas expresiones:

$a+b-3.5$	$rr(sr(a,r(b)),3.5)$
$a+[b-3.5]$	$sr(a,r(t(rr(r(b),3.5))))$

Solución:

La solución de este ejercicio es:

$E \longrightarrow E_1 \text{ addop } T$	$\{$ si $E_1.tipo = ENTERO$ y $T.tipo = ENTERO$ entonces $E.tipo := ENTERO$ $E.cod := \text{addop.trad} \parallel "i(" \parallel E_1.cod \parallel ", " \parallel T.cod \parallel ")"$ si no si $E_1.tipo = ENTERO$ y $T.tipo = REAL$ entonces $E.tipo := REAL$ $E.cod := \text{addop.trad} \parallel "r(r(" \parallel E_1.cod \parallel "), " \parallel T.cod \parallel ")"$ si no si $E_1.tipo = REAL$ y $T.tipo = ENTERO$ entonces $E.tipo := REAL$ $E.cod := \text{addop.trad} \parallel "r(" \parallel E_1.cod \parallel ", r(" \parallel T.cod \parallel ")"$ si no $E.tipo := REAL$ $E.cod := \text{addop.trad} \parallel "r(" \parallel E_1.cod \parallel ", " \parallel T.cod \parallel ")"$ $\}$
$E \longrightarrow T$	$\{$ $E.cod := T.cod$ $E.tipo := T.tipo$ $\}$
$T \longrightarrow \text{nint}$	$\{$ $T.cod := \text{nint.lexema}$ $T.tipo := ENTERO$ $\}$
$T \longrightarrow \text{nfix}$	$\{$ $T.cod := \text{nfix.lexema}$ $T.tipo := REAL$ $\}$
$T \longrightarrow \text{id}$	$\{$ $T.cod := \text{id.lexema}$ $T.tipo := TipoVar(\text{id.lexema})$ $\}$
$T \longrightarrow (E)$	$\{$ $T.cod := E.cod$ $T.tipo := E.tipo$ $\}$
$T \longrightarrow [E]$	$\{$ si $E.tipo = ENTERO$ entonces $T.cod := E.cod$ si no $T.cod := "t(" \parallel E.cod \parallel ")"$ $T.tipo := ENTERO$ $\}$

Notas:

1. El atributo **addop.trad** es la traducción de los operadores "+" y "-" a "s" y "r", respectivamente.
2. La función *TipoVar* devuelve el tipo con el que se declaró la variable.

Ejercicio 8.4

Diséñese un ETDS que genere código objeto **m2r** a partir de un lenguaje fuente con *arrays* que se declaran y utilizan como en el lenguaje Pascal.

La parte del ETDS que se encarga de crear los tipos en la tabla de tipos y de almacenar los símbolos en la tabla de símbolos se debe suponer que ya está hecha. Solamente se debe hacer la parte del ETDS que genera código para acceder a las posiciones de un *array* o variable simple.

Ejemplo: Suponiendo las siguientes declaraciones de variables globales,

```
var
    entero : a,b;
    tabla 4..7 de real : ar;
    tabla 10..11 de tabla 1..6 de entero : abe;
fvar
```

la parte del ETDS encargada de procesar las declaraciones almacenaría los tipos y los símbolos en las tablas correspondientes, que quedarían de esta manera:

TABLA DE TIPOS					
POSICIÓN	TIPO	TIPO BÁSICO	LÍMITE INFERIOR	LÍMITE SUPERIOR	TAMAÑO
0	ENTERO	-	-	-	-
1	REAL	-	-	-	-
2	ARRAY	1	4	7	4
3	ARRAY	0	1	6	6
4	ARRAY	3	10	11	2

TABLA DE SÍMBOLOS			
POSICIÓN	NOMBRE	TIPO	DIRECCIÓN
0	a	0	0
1	b	0	1
2	ar	2	2
3	abe	4	6

En este ETDS se supone que los tipos básicos (entero y real) ocupan solamente una posición de memoria, y debe tenerse en cuenta que en las tablas tanto el límite inferior como el superior están incluidos en el rango de valores posibles. Las funciones para acceder a los valores de las tablas de tipos y símbolos son:

```
booleano TT_EsArray(entero:posicion)
entero TT_TipoBasico(entero:posicion)
entero TT_LimiteInferior(entero:posicion)
entero TT_LimiteSuperior(entero:posicion)
entero TT_Tamanyo(entero:posicion)
entero TS_BuscaId(cadena:identificador) // devuelve la posicion o -1
                                         si no se ha declarado
entero TS_TipoId(cadena:identificador)
entero TS_DireccionId(cadena:identificador)
```

El fragmento de ETDS que se debe hacer es el asociado a las siguientes reglas:

$$\begin{aligned}
 Instr &\longrightarrow Ref \mathbf{assop} Expr \\
 Factor &\longrightarrow Ref \\
 Ref &\longrightarrow \mathbf{id} \\
 Ref &\longrightarrow Ref \mathbf{lcor} E\mathbf{simple} \mathbf{rcor}
 \end{aligned}$$

Se supone que *Factor*, *Esimple* y *Expr* tienen (al menos) tres atributos sintetizados: **cod**, **tipo** y **tmp**, con el mismo significado que en los ejemplos de este capítulo. Las restricciones de tipos son las mismas que en Pascal y, aunque un compilador de Pascal genera código para comprobar que el índice de un *array* está dentro del rango permitido, en este ETDS no se debe hacer.

Solución:

La solución se muestra en la figura A.3, empezando por las reglas más sencillas.

<i>Ref</i>	\longrightarrow	id	<pre> { Ref.dbase := TS_DireccionId(id.lexema); tmp := NuevaTemporal(); Ref.tipo := TS_TipoId(id.lexema); Ref.cod := "mov #0" tmp; Ref.tmp := tmp; Ref.tam := TT_Tamanyo(Ref.tipo); } </pre>
<i>Ref</i>	\longrightarrow	<i>Ref</i> ₁ lcor	<pre> { si no TT_EsArray(Ref₁.tipo) entonces ErrorSemántico("la referencia no es array o sobran índices") } </pre>
		<i>Esimple</i> rcor	<pre> { si Esimple.tipo != ENTERO entonces ErrorSemántico("el índice debe ser entero") Ref.dbase := Ref₁.dbase; tmp := NuevaTemporal(); Ref.tmp := tmp; Ref.tipo := TT_TipoBasico(Ref₁.tipo); Ref.tam := TT_Tamanyo(Ref.tipo); Ref.cod := Ref₁.cod Esimple.cod "mov " Ref₁.tmp "A" "mul # " Ref₁.tam "addi " Esimple.tmp "subi " TT_LimiteInferior(Ref₁.tipo) "mov A " tmp } </pre>
<i>Factor</i>	\longrightarrow	<i>Ref</i>	<pre> { si TT_EsArray(Ref.tipo) entonces ErrorSemántico("faltan índices") tmp := NuevaTemporal(); Factor.tmp := tmp; Factor.tipo := Ref.tipo Factor.cod := Ref.cod "mov " Ref.tmp "A" "mul # " Ref.tam "addi " Ref.dbase "mov @A " tmp } </pre>
<i>Instr</i>	\longrightarrow	<i>Ref</i> assop	<pre> { si TT_EsArray(Ref₁.tipo) entonces ErrorSemántico("faltan índices") } </pre>
		<i>Expr</i>	<pre> { si Ref.tipo = ENTERO y Expr.tipo = REAL entonces ErrorSemántico("no se puede asignar un real a una variable entera") si Ref.tipo = REAL y Expr.tipo = ENTERO entonces tmp := NuevaTemporal(); Instr.cod := Ref.cod Expr.cod "mov " Expr.tmp "A" "itor" "mov A " tmp "mov " Ref.tmp "A" "mul # " Ref.tam "addi " Ref.dbase "mov " tmp "@A" si no Instr.cod := Ref.cod Expr.cod "mov " Ref.tmp "A" "mul # " Ref.tam "addi " Ref.dbase "mov " Expr.tmp "@A" fsi } </pre>

Figura A.3: Solución del ejercicio 8.4.

A.8 Soluciones a los ejercicios del capítulo 9

Ejercicio 9.1

Dado el siguiente programa en C, dibújese el contenido del registro de activación de la función **Pepito** (incluidas las variables temporales) en el momento inmediatamente anterior a la ejecución del **return** (donde indica el comentario).

```
int Pepito(int a,float b)
{
    float c;

    a = a+b;
    c = 2.0*a;

    /* aqui */
    return c;
}

main()
{
    Pepito(7,3.9);
}
```

Solución:

(valor devuelto)	
(dirección de retorno)	
(B anterior)	
a	10
b	3.9
c	20.0
t_1	2.0
t_2	10
t_3	10.0
t_4	20.0
...	

Debe tenerse en cuenta que, aunque “a” sea un argumento, es posible modificar su valor como si de una variable local se tratara. Las temporales que se muestran son únicamente las utilizadas en la asignación a “c”, ya que las de la primera asignación se habrán reutilizado en esta segunda asignación.

Ejercicio 9.2

Dado el siguiente fragmento de programa en C, dibújese el contenido de los registros

de activación de las llamadas activas en el momento en que se va a ejecutar la instrucción `return num;` *por tercera vez*. Indíquese en los registros solamente los valores de los parámetros y de las variables locales cuyo valor sea conocido. No hace falta especificar cuántas temporales se utilizan ni qué valor tienen. Se debe suponer que para la función `main` no se crea un registro de activación.

```
int a;
int potencia(int num,int aque)
{
    int uno,dos;
    uno = aque/2;
    dos = aque - uno;
    if (aque == 0)      return 1;
    else if (aque == 1) return num;
    else                return potencia(num,uno)*potencia(num,dos);
}
main()
{
    a = 2+potencia(2,5);
}
```

Solución:

(valor devuelto)	
(dirección de retorno)	
(B anterior)	
num	2
aque	5
uno	2
dos	3
...	
(valor devuelto)	
(dirección de retorno)	
(B anterior)	
num	2
aque	3
uno	1
dos	2
...	
(valor devuelto)	
(dirección de retorno)	
(B anterior)	
num	2
aque	1
uno	0
dos	1
...	

⇒ potencia(2,5)

⇒ potencia(2,3)

⇒ potencia(2,1)

Ejercicio 9.3

Escribir el código en m2r que se generaría para el siguiente fragmento de programa en C.

```
int a;
int RestaUno(int n) {
    return n-1;
}
int factorial(int n) {
    if (n>1) return n*factorial(RestaUno(n));
    else return 1;
}
main() {
    a = 2+factorial(5);
}
```

Solución:

```
        jmp L1          ; salto al 'main'

L2  ; comienza el código de 'RestaUno'
    mov @B+0 @B+1  ; 'n' a una temporal
    mov #1  @B+2  ; '1' a otra temporal
    mov @B+1 A
    subi @B+2
    mov A  @B+3  ; 'n-1'
    mov @B+3 @B-3 ; poner el valor devuelto
    mov @B-2 A   ; coger dirección de retorno
    jmp @A       ; salir de la función

L3  ; comienza el código de 'factorial'
    mov @B+0 @B+1
    mov #1  @B+2
    mov @B+1 A
    gtri @B+2
    mov A  @B+3  ; 'n>1'
    mov @B+3 A
    jz L4      ; 'if (n>1)'
    mov @B+0 @B+1 ; 'n*...'
    ; llamada a 'factorial': se reserva de @B+2 a @B+5
    ; llamada a 'RestaUno' (de @B+6 a @B+9)
    mov @B+0 @B+10 ; '...(n)'
    mov @B+10 @B+9 ; primer parámetro de 'RestaUno'
    mov B @B+8    ; guardar 'B' anterior
    mov B A
    addi #9
    mov A B       ; poner nueva 'B' (en @B+9)
    mvetq L5 @B-2 ; guardar dirección de retorno
    jmp L2        ; salto a 'RestaUno'
```



```

L5  mov @B-1 B      ; dejar 'B' como estaba
    ; fin llamada a 'RestaUno' (resultado en @B+6)
    mov @B+6 @B+5   ; lo que devuelve 'RestaUno' es el
                    ; parámetro de 'factorial'
    mov B @B+4      ; guardar 'B' anterior
    mov B A
    addi #5
    mov A B          ; poner nueva 'B' (en @B+5)
    mvetq L6 @B-2
    jmp L3           ; llamada recursiva a 'factorial'
L6  mov @B-1 B      ; dejar 'B' como estaba
    ; fin llamada recursiva a 'factorial(RestaUno(n))' (resultado en @B+2)
    mov @B+1 A
    muli @B+2
    mov A @B+3       ; 'n (@B+1) * factorial ... (@B+2)'
    mov @B+3 @B-3
    mov @B-2 A
    jmp @A           ; 'return factorial(...'
    jmp L7           ; fin de la primera parte del 'if'
L4  mov #1 @B+4     ; 'else return 1'
    mov @B+4 @B-3
    mov @B-2 A
    jmp @A
L7  mov @B-2 A      ; retorno por defecto
    jmp @A

L1  ; comienza el código del programa principal
    mov #10000 B     ; las temporales y los RA empiezan en la 10000
    mov #2 @B+0      ; '2+...'
    ; llamada a 'factorial(5)': se reserva de @B+1 a @B+4
    mov #5 @B+5
    mov @B+5 @B+4    ; primer parámetro de 'factorial'
    mov B @B+3       ; guardar 'B' anterior
    mov B A
    addi #4
    mov A B          ; poner nueva 'B' (en @B+4)
    mvetq L8 @B-2
    jmp L3           ; llamada a 'factorial'
L8  mov @B-1 B      ; dejar 'B' como estaba
    ; fin llamada a 'factorial(5)' (resultado en @B+1)
    mov @B+0 A
    addi @B+1
    mov A @B+2       ; '2+factorial(5)'
    mov @B+2 100     ; suponiendo que 'a' está en la dirección 100
    halt

```

Ejercicio 9.4

Escribir el código en m2r que generaría un compilador para el siguiente fragmento de programa en C. Indíquese también qué valor se almacena en la variable `a` declarada en la función `main`. Debe suponerse que la función `main` no tiene registro

de activación (al estilo del programa principal en Pascal), aunque en C sea una función más.

```
float Abs(float f)
{
    if (f<0)
        return -f;
    else
        return f;
}
char Suma(int i,float f)
{
    int a[2][5];

    a[1][3] = Abs(i+f);
    return a[1][3];
}
main() {
    int a;
    a = Suma(10.987,1024);
}
```

Solución:

```
        jmp L1          ; salto a 'main'

L2  ; código de 'Abs'
    mov @B+0 @B+1      ; 'f' a una temporal
    mov #0 @B+2
    mov @B+2 A
    itor
    mov A @B+3          ; convertir el '0' en '0.0'
    mov @B+1 A
    lssr @B+3
    mov A @B+4          ; 'f<0'
    mov @B+4 A
    jz L3               ; 'if (...)'
    mov @B+0 @B+5
    mov $0.0 A
    subr @B+5
    mov A @B+6          ; '-f'
    mov @B+6 @B-3
    mov @B-2 A
    mov @B-1 B          ; en este ejercicio se restaura la 'B'
                        ; en el código de la función llamada
    jmp @A              ; 'return -f'
    jmp L4
L3  mov @B+0 @B+7
    mov @B+7 @B-3
```

```

    mov @B-2 A
    mov @B-1 B
    jmp @A          ; 'return f'
L4  mov @B-2 A      ; retorno por defecto
    mov @B-1 B
    jmp @A

L5  ; código de 'Suma' (el array 'a' va de @B+2 a @B+11)
    mov #0 @B+12    ; desplazamiento 'a'
    mov #1 @B+13    ; '[1]'
    mov @B+12 A
    muli #2
    addi @B+13
    mov A @B+14      ; desplazamiento 'a[1]'
    mov #3 @B+15    ; '[3]'
    mov @B+14 A
    muli #5
    addi @B+15
    mov A @B+16      ; desplazamiento 'a[1][3]'
    ; llamada a 'Abs': se reserva de @B+17 a @B+20
    mov @B+0 @B+21   ; 'i' a una temporal
    mov @B+1 @B+22   ; 'f' a otra temporal
    mov @B+21 A
    itor
    mov A @B+23      ; convertir 'i' a real
    mov @B+23 A
    addr @B+22
    mov A @B+24      ; 'i+f'
    mov @B+24 @B+20  ; poner primer parámetro
    mov B @B+19      ; guardar 'B' anterior
    mov B A
    addi #20
    mov A B          ; poner nueva 'B' (en @B+20)
    mvetq L6 @B-2
    jmp L2

L6  ; fin llamada a 'Abs' (resultado en @B+17)
    mov @B+17 A
    rtoi
    mov A @B+18      ; convertir a entero para asignarlo a 'a[1][3]'
    mov @B+16 A
    muli #1          ; multiplicar por el tamaño del tipo entero
    addi B
    addi #2          ; calcular dirección completa de 'a[1][3]'
    mov @B+18 @A     ; 'a[1][3] = Abs(i+f)'
    ; segunda instrucción 'return a[1][3]', se reciclan las temporales
    mov #0 @B+12    ; desplazamiento 'a'
    mov #1 @B+13    ; '[1]'
    mov @B+12 A
    muli #2
    addi @B+13
    mov A @B+14      ; desplazamiento 'a[1]'

```

```

    mov #3 @B+15    ; '[3]'
    mov @B+14 A
    muli #5
    addi @B+15
    mov A @B+16      ; desplazamiento 'a[1][3]'
    mov @B+16 A
    muli #1          ; multiplicar por el tamaño del tipo entero
    addi B
    addi #2          ; dirección completa de 'a[1][3]'
    mov @A @B+17     ; poner el valor contenido en 'a[1][3]' en una temporal
    mov @B+17 A
    modi #256
    mov A @B+18      ; convertir el entero 'a[1][3]' en 'char',
    mov @B+18 @B-3   ; porque 'Suma' devuelve 'char'
    mov @B-2 A
    mov @B-1 B
    jmp @A           ; retornar desde 'Suma'

L1  ; código del programa principal
    mov #10000 B
    mov #0 @B+0      ; desplazamiento de 'a'
    ; llamada a 'Suma': se reserva de @B+1 a @B+5
    mov $10.987 @B+6
    mov @B+6 A       ; como 'i' es entero, hay que convertir
    rtoi            ; el '10.987' a entero
    mov A @B+7
    mov @B+7 @B+4     ; poner el primer parámetro
    mov #1024 @B+8
    mov @B+8 A       ; lo contrario sucede con 'f' (real) y '1024' (entero)
    itor
    mov A @B+9
    mov @B+9 @B+5     ; poner el segundo parámetro
    mov B @B+3       ; guardar 'B' anterior
    mov B A
    addi #4
    mov A B          ; poner nueva 'B' (en @B+4)
    mvetq L7 @B-2
    jmp L5           ; salto a 'Suma'
L7  ; fin llamada a 'Suma' (resultado en @B+1, como real)
    mov @B+1 A
    rtoi
    mov A @B+2       ; convertir a entero porque se va a asignar a 'a'
    mov @B+0 A
    muli #1          ; multiplicar por el tamaño del tipo entero
    addi #100        ; calcular dirección completa de 'a' (100)
    mov @B+2 @A      ; 'a = Suma(...)'
    halt

```

Notas:

1. En este ejercicio se restaura el valor anterior de 'B' en el código de la función

llamada, mientras que en el ejercicio anterior esto se hace en el código de la función llamadora.

2. Se supone que la variable `a` declarada en la función `main` es global, y su dirección es 100.
 3. En la llamada `Suma(10.987,1024)`, 10.987 se convierte en 10 (el valor que tendrá `"i"`), y 1024 en 1024.0 (el valor de `"f"`). La suma `"i+f"` vale 1034, pero como la función devuelve un `"char"`, ese 1034 se convierte en 10, que es lo que finalmente se almacena en `'a'`.
-

Apéndice B

Ejercicios prácticos propuestos

En cualquier libro sobre compiladores se propone la construcción de un compilador “de juguete” como la mejor manera de aprender los conocimientos y las técnicas que se explican a lo largo del libro. Aunque un compilador real es un programa tremendamente complejo, es relativamente sencillo llevar a la práctica muchos de los conceptos tratados en este libro.

A continuación se proponen una serie de ejercicios prácticos con la idea de tratar los aspectos más importantes del diseño de un compilador. Todos ellos especifican un formato estricto para la salida, lo cual permite corregir automáticamente casi todos los ejercicios, excepto alguna parte (los mensajes de error, por ejemplo).

B.1 Analizador léxico

Este ejercicio práctico consiste en construir un analizador léxico que reconozca los componentes léxicos o *tokens* de la tabla B.1

Se sugiere que el programa esté implementado en C. El programa tiene que leer un fichero (cuyo nombre se le pasará como argumento) y analizar léxicamente su contenido. Para comprobar que funciona correctamente, el programa principal debe llamar a la función que implementa el analizador léxico para cada *token* y debe imprimir por la salida estándar (`stdout`) una línea por cada *token* reconocido, con el siguiente formato:

(línea) (columna) (nombre token) (lexema)

Ejemplo B.1

Dado el siguiente fichero de entrada:

```
program Programa + 34          / 2.3e*7
```

el analizador léxico debe generar la siguiente salida:

```
1 1 program program
1 9 id Programa
1 18 addop +
1 20 nint 34
```

EXPRESIÓN REGULAR	COMPONENTE LÉXICO	VALOR LÉXICO ENTREGADO
<code>[\n\t]+</code>	(ninguno)	
<code>program</code>	program	(palabra reservada)
<code>begin</code>	begin	(palabra reservada)
<code>end</code>	end	(palabra reservada)
<code>var</code>	var	(palabra reservada)
<code>integer</code>	integer	(palabra reservada)
<code>write</code>	write	(palabra reservada)
<code>read</code>	read	(palabra reservada)
<code>if</code>	if	(palabra reservada)
<code>then</code>	then	(palabra reservada)
<code>else</code>	else	(palabra reservada)
<code>endif</code>	endif	(palabra reservada)
<code>while</code>	while	(palabra reservada)
<code>do</code>	do	(palabra reservada)
<code>div</code>	mulop	(palabra reservada)
<code>mod</code>	mulop	(palabra reservada)
<code>[A-Za-z] [0-9A-Za-z]*</code>	id	(nombre del ident.)
<code>[0-9]+</code>	nint	(valor numérico)
<code>([0-9]+) " . " ([0-9]+)</code>	nfix	(valor numérico)
<code>([0-9]+) " . " ([0-9]+) [eE] ([-+])? ([0-9]+)</code>	nflo	(valor numérico)
<code>,</code>	coma	
<code>;</code>	pyc	
<code>.</code>	punto	
<code>:</code>	dosp	
<code>(</code>	lpar	
<code>)</code>	rpar	
<code>=</code>	relop	<code>=</code>
<code><></code>	relop	<code><></code>
<code><</code>	relop	<code><</code>
<code><=</code>	relop	<code><=</code>
<code>></code>	relop	<code>></code>
<code>>=</code>	relop	<code>>=</code>
<code>+</code>	addop	<code>+</code>
<code>-</code>	addop	<code>-</code>
<code>*</code>	mulop	<code>*</code>
<code>/</code>	mulop	<code>/</code>
<code>:=</code>	assop	

Tabla B.1: *Tokens* que debe reconocer el analizador léxico.

```

2 7 mulop /
2 9 nfix 2.3
2 12 id e
2 13 mulop *
2 14 nint 7

```

Notas:

- El analizador léxico debe ignorar los espacios en blanco, los tabuladores y los finales de línea, y debe devolver un *token* especial cuando encuentre el final del fichero, con el lexema vacío.
- Se debe leer el fichero solamente una vez, y carácter a carácter. Para ello es recomendable utilizar la función “`fgetc`” (pero teniendo en cuenta que lo que devuelve es un entero, y que devuelve `-1` cuando llega al final del fichero). Además, no se debe utilizar la función “`ungetc`” ya que sólo permite devolver un carácter al *buffer* de entrada, y en algunos casos es necesario devolver más de un carácter. Por tanto, será necesario implementar algún tipo de *buffer* para almacenar estos caracteres y una función auxiliar que lea del *buffer* o, si está vacío, lea con “`fgetc`”.
- Cuando el analizador encuentre un carácter que no sea el comienzo de ningún componente léxico, debe mostrar un error (por `stderr`) como el siguiente:

Error (1,5) : caracter '\$' incorrecto

- La línea, la columna y el lexema se deben almacenar en variables globales. El tipo de *token* será un entero que devuelva la función que implementa el analizador léxico.
- De todos los métodos comentados en el capítulo 2 se debe elegir el más adecuado para implementar este analizador léxico, aunque un enfoque mixto (con diagramas de transiciones para los números y los operadores) es probablemente el mejor.

B.2 Analizador sintáctico descendente recursivo

Dada la siguiente gramática:

S	\longrightarrow	program id pyc BDecl Bloque punto
$Bloque$	\longrightarrow	begin SeqInstr end
$BDecl$	\longrightarrow	ϵ
$BDecl$	\longrightarrow	var Decl pyc
$Decl$	\longrightarrow	$Decl$ pyc DVar
$Decl$	\longrightarrow	$DVar$
$DVar$	\longrightarrow	$LIdent$ dosp integer
$LIdent$	\longrightarrow	$LIdent$ coma id
$LIdent$	\longrightarrow	id
$SeqInstr$	\longrightarrow	$SeqInstr$ pyc Instr
$SeqInstr$	\longrightarrow	$Instr$
$Instr$	\longrightarrow	$Bloque$
$Instr$	\longrightarrow	id assop Expr
$Instr$	\longrightarrow	write lpar LExpr rpar
$LExpr$	\longrightarrow	$LExpr$ coma Expr
$LExpr$	\longrightarrow	$Expr$
$Instr$	\longrightarrow	read lpar id rpar
$Instr$	\longrightarrow	if Expr then Instr endif
$Instr$	\longrightarrow	if Expr then Instr else Instr endif
$Instr$	\longrightarrow	while Expr do Instr
$Expr$	\longrightarrow	$Esimple$ relop Esimple
$Expr$	\longrightarrow	$Esimple$
$Esimple$	\longrightarrow	$Esimple$ addop Term
$Esimple$	\longrightarrow	$Term$
$Term$	\longrightarrow	$Term$ mulop Factor
$Term$	\longrightarrow	$Factor$
$Factor$	\longrightarrow	id
$Factor$	\longrightarrow	nint
$Factor$	\longrightarrow	nfix
$Factor$	\longrightarrow	nflo
$Factor$	\longrightarrow	lpar Expr rpar

este ejercicio práctico consiste en:

1. Eliminar la recursividad por la izquierda y los factores comunes por la izquierda, obteniendo una gramática equivalente a la anterior.
2. Calcular los conjuntos de predicción y comprobar que la gramática resultante es LL(1).
3. Utilizando el analizador léxico del ejercicio práctico anterior, construir un analizador sintáctico descendente recursivo que imprima por pantalla una derivación por la izquierda del programa de entrada (imprimiendo un número de regla por línea).

Además, si el programa fuente contiene algún error sintáctico, el analizador debe producir un mensaje de error y terminar. El mensaje de error para un programa de entrada como el del ejemplo B.1 debería ser el siguiente:

Error (1,18): encontrado '+', esperaba ';'.

Debe indicar la línea, la columna y el lexema del *token* incorrecto y debe decir qué *tokens* se esperaban en su lugar (en un lenguaje comprensible para un usuario que no conoce el funcionamiento de los compiladores).

Ejemplo B.2

Si la gramática fuera por ejemplo la siguiente:

<i>S</i>	→	<i>Prog Bloque punto</i>
<i>Bloque</i>	→	begin <i>SI</i> end
<i>Prog</i>	→	program <i>id</i> pyc
<i>SI</i>	→	<i>Instr</i>
<i>Instr</i>	→	read <i>lpar id rpar</i>
<i>Instr</i>	→	write <i>lpar Expr rpar</i>
<i>Expr</i>	→	id
<i>Expr</i>	→	nint
<i>Expr</i>	→	nfix

y el programa fuente fuera:

```
program EjASDR;
begin
  write(25.7)
end.
```

la salida del analizador debería ser:

```
1
3
2
4
6
9
```

◁

B.3 Analizador SLR

Este ejercicio práctico consiste en, a partir de la gramática del apartado anterior, hacer lo siguiente:

1. Construir el autómata reconocedor de prefijos viables para dicha gramática.

2. Construir la tabla de análisis SLR.
3. Construir, utilizando el analizador léxico del primer ejercicio práctico, un analizador SLR que imprima por la salida estándar una derivación por la derecha del programa de entrada¹. Si el programa contiene algún error sintáctico, el analizador debe producir un mensaje de error con el mismo formato que en el ejercicio práctico anterior y terminar.

B.4 Traductor descendente recursivo

A partir del analizador descendente recursivo construido anteriormente, se debe construir un traductor que implemente el siguiente proceso de traducción de un programa en el lenguaje fuente (muy parecido a un subconjunto de Pascal) a C.

Ejemplo B.3

Dado el programa fuente de la izquierda, se debe producir el programa que aparece a la derecha:

program EjTDR;	/* EjTDR */
var a,b,c:integer;	int a,b,c;
e,f:integer;	int e,f;
g:integer;	int g;
begin	main() {
a := 76;	a=76;
write(a,23.5,2.3E1);	printf("%d %f %f\n",a,23.5,2.3E1);
read(b);	scanf("%d\n",b);
if a<>b then	if (a!=b)
begin	{
while a-b>0 do	while (a-b>0)
a:=a-1;	a=a-1;
while a-b<0 do	while (a-b<0)
a:=a+1;	a=a+1;
g:=1	g=1;
end	}
else	else
g:=1	g=1;
endif	
end.	}

◁

Notas:

- La traducción del programa fuente debe irse construyendo como una cadena de caracteres en memoria dinámica. Para ello se recomienda utilizar funciones para

¹La secuencia de reducciones que realiza un analizador SLR es *la inversa* de una derivación por la derecha, no la propia derivación por la derecha.

duplicar un *array* de caracteres en memoria (**strdup**) y funciones para concatenar cadenas de caracteres (no se recomienda utilizar **strcat** sin haber hecho **malloc** o **realloc** previamente).

- Los mensajes de error léxico y sintáctico deben ser los que ya tenía implementados el analizador descendente recursivo.

B.5 Traductor ascendente

A partir de lo explicado en el capítulo 7, este ejercicio consiste en implementar dos ETDS basados en analizadores SLR. Para ello se puede reutilizar gran parte del código del analizador SLR que se ha propuesto en el apartado B.3.

Primer ETDS

Los componentes léxicos del lenguaje fuente son un subconjunto de las del primer ejercicio práctico:

nint:	un número entero sin signo.
id:	un identificador.
addop:	los operadores de suma “+” o resta “-”.
coma:	la coma, “,”.
pyc:	el carácter punto y coma, “;”.
Palabras reservadas:	“var”

El ETDS a implementar es el siguiente:

$S \longrightarrow D I$	$\{S.trad := D.trad \parallel I.trad\}$
$D \longrightarrow V D_1$	$\{D.trad := V.trad \parallel D_1.trad\}$
$D \longrightarrow \epsilon$	$\{D.trad := ''\}$
$V \longrightarrow \text{var } L \text{ pyc}$	$\{V.trad := 'int' \parallel L.trad \parallel ';;'\}$
$L \longrightarrow L_1 \text{ coma id}$	$\{L.trad := L_1.trad \parallel ', ' \parallel id.lexema\}$
$L \longrightarrow \text{id}$	$\{L.trad := id.lexema\}$
$I \longrightarrow I_1 \text{ pyc } E$	$\{I.trad := I_1.trad \parallel ';;' \parallel E.trad\}$
$I \longrightarrow E$	$\{I.trad := E.trad\}$
$E \longrightarrow E_1 \text{ addop } T$	$\{E.trad = ' [' \parallel addop.trad \parallel ' ' \parallel E_1.trad \parallel ' ' \parallel T.trad \parallel '] '\}$
$E \longrightarrow T$	$\{E.trad = T.trad\}$
$T \longrightarrow \text{nint}$	$\{T.trad = \text{nint.lexema}\}$
$T \longrightarrow \text{id}$	$\{T.trad = \text{id.lexema}\}$

Notas:

1. El atributo **addop.trad** es la traducción de “+” y “-” a “sum” y “res” respectivamente.
2. El ETDS a implementar no debe ser modificado (ni simplificado, ni mejorado) en absoluto.

3. Antes de implementar el ETDS se debe construir el analizador sintáctico SLR para el lenguaje fuente y comprobar que analiza correctamente los programas de éste. Una vez se haya comprobado que el analizador sintáctico funciona bien, se puede proceder a modificar el algoritmo de análisis para que incorpore las acciones de traducción.

Ejemplo B.4

Ejemplo de traducción:

<code>var a,b,c;</code>	<code>int a,b,c;</code>
<code>var d;</code>	<code>int d;</code>
<code>a+b-10;</code>	<code>[res [sum a b] 10] ;</code>
<code>a+b-10+c</code>	<code>[sum [res [sum a b] 10] c]</code>

◁

Segundo ETDS

En este segundo ETDS, los componentes léxicos son los mismos del anterior ETDS con algunas ampliaciones:

nint:	un número entero sin signo.
id:	un identificador.
addop:	los operadores de suma “+” o resta “-”.
coma:	la coma, “,”.
lbra:	la llave izquierda, “{”
rbra:	la llave derecha, “}”
pyc:	el carácter punto y coma, “;”.
Palabras reservadas:	“var”, “func”

Como se puede observar, se han añadido las llaves y la palabra reservada “func” con respecto al analizador léxico del primer ETDS.

El ETDS a implementar es el de la figura B.1.

Notas:

1. El ETDS no debe ser modificado. Como se puede observar al comparar este ETDS con el anterior, cuando se utiliza un analizador ascendente para construir el traductor no es recomendable eliminar la recursión por la izquierda de la gramática.
2. Antes de implementar el ETDS es necesario estudiar qué marcadores es necesario introducir en la gramática, y después se debe implementar el analizador sintáctico SLR para la gramática con marcadores.

Ejemplo B.5

Ejemplo de traducción:

<code>var a,b,c;</code>	<code>int a,b,c;</code>
<code>func fu {</code>	<code>int fu () {</code>
<code>func kung {</code>	<code>int fu_kung () {</code>

S	\longrightarrow	$\{ D.idh := ' ' \}$
	$D I$	$\{ S.trad := D.trad \parallel I.trad \}$
D	\longrightarrow	$\{ V.idh := D.idh \}$
	V	$\{ D_1.idh := D.idh \}$
	D_1	$\{ D.trad := V.trad \parallel D_1.trad \}$
D	\longrightarrow	$\{ F.idh := D.idh \}$
	F	$\{ D_1.idh := D.idh \}$
	D_1	$\{ D.trad := F.trad \parallel D_1.trad \}$
D	\longrightarrow	ϵ
F	\longrightarrow	$\{ D.idh := F.idh \parallel \mathbf{id.lexema} \parallel ' _ ' \}$
	$D I rbra$	$\{ F.trad := ' \mathbf{int} ' \parallel F.idh \parallel \mathbf{id.lexema} \parallel ' () \{ ' \parallel D.trad \parallel I.trad \parallel ' \} ' \}$
V	\longrightarrow	\mathbf{var}
	$L \mathbf{pyc}$	$\{ L.idh := V.idh \}$
L	\longrightarrow	\mathbf{id}
	Lp	$\{ V.trad := ' \mathbf{int} ' \parallel L.trad \parallel ' ; ' \}$
Lp	\longrightarrow	$\mathbf{coma id}$
	Lp_1	$\{ Lp.idh := L.idh \}$
Lp	\longrightarrow	ϵ
I	\longrightarrow	$I_1 \mathbf{pyc} E$
I	\longrightarrow	E
E	\longrightarrow	T
	Ep	$\{ L.trad := L.idh \parallel \mathbf{id.lexema} \parallel Lp.trad \}$
Ep	\longrightarrow	$\mathbf{addop} T$
	Ep_1	$\{ Lp_1.idh := Lp.idh \}$
Ep	\longrightarrow	ϵ
T	\longrightarrow	\mathbf{nint}
T	\longrightarrow	\mathbf{id}

Figura B.1: Segundo ETDS para implementar con un analizador SLR.

```

var c, d;
7-d
}
var a, b;
2+a-b+d
}
a+b-10;
a+b-10+c

int fu_kung_c, fu_kung_d;
[ res 7 d ]
}
int fu_a, fu_b;
[ sum [ res [ sum 2 a ] b ] d ]
}
[ res [ sum a b ] 10 ];
[ sum [ res [ sum a b ] 10 ] c ]

```

<

B.6 Compilador para un lenguaje sencillo

Este ejercicio práctico consiste en realizar (utilizando YACC (o **bison**) y LEX (o **flex**)) un compilador para el lenguaje fuente que se describe más adelante, que genere código

para el lenguaje objeto **m2r** (véase el apéndice C para una descripción completa de este lenguaje).

El lenguaje fuente es un subconjunto de Pascal, y la semántica es similar a la de dicho lenguaje. El lenguaje tiene tres tipos simples: booleanos, enteros y reales. La sintaxis del lenguaje fuente puede ser representada por la gramática de la figura B.2. La especificación léxica de los símbolos terminales de la gramática es la de la tabla de la figura B.3.

<i>S</i>	→	program <i>id</i> pyc <i>BDecl</i> <i>Bloque</i> punto
<i>Bloque</i>	→	begin <i>SeqInstr</i> end
<i>BDecl</i>	→	ε
<i>BDecl</i>	→	<i>BlVar</i> <i>BDecl</i>
<i>BlVar</i>	→	var <i>Decl</i> pyc
<i>Decl</i>	→	<i>Decl</i> pyc <i>DVar</i>
<i>Decl</i>	→	<i>DVar</i>
<i>DVar</i>	→	<i>LIdent</i> dosp <i>Tipo</i>
<i>Tipo</i>	→	boolean
<i>Tipo</i>	→	integer
<i>Tipo</i>	→	real
<i>LIdent</i>	→	<i>LIdent</i> coma <i>id</i>
<i>LIdent</i>	→	id
<i>SeqInstr</i>	→	<i>SeqInstr</i> pyc <i>Instr</i>
<i>SeqInstr</i>	→	<i>Instr</i>
<i>Instr</i>	→	<i>Bloque</i>
<i>Instr</i>	→	<i>Ref</i> assop <i>Expr</i>
<i>Instr</i>	→	wri lpar <i>LExpr</i> rpar
<i>LExpr</i>	→	<i>LExpr</i> coma <i>Expr</i>
<i>LExpr</i>	→	<i>Expr</i>
<i>Instr</i>	→	read lpar <i>LRef</i> rpar
<i>LRef</i>	→	<i>LRef</i> coma <i>Ref</i>
<i>LRef</i>	→	<i>Ref</i>
<i>Instr</i>	→	if <i>Expr</i> then <i>Instr</i>
<i>Instr</i>	→	if <i>Expr</i> then <i>Instr</i> else <i>Instr</i>
<i>Instr</i>	→	while <i>Expr</i> do <i>Instr</i>
<i>Expr</i>	→	<i>Esimple</i> relop <i>Esimple</i>
<i>Expr</i>	→	<i>Esimple</i>
<i>Esimple</i>	→	<i>Esimple</i> addop <i>Term</i>
<i>Esimple</i>	→	<i>Esimple</i> obool <i>Term</i>
<i>Esimple</i>	→	<i>Term</i>
<i>Esimple</i>	→	addop <i>Term</i>
<i>Term</i>	→	<i>Term</i> mulop <i>Factor</i>
<i>Term</i>	→	<i>Term</i> ybool <i>Factor</i>
<i>Term</i>	→	<i>Factor</i>
<i>Factor</i>	→	<i>Ref</i>
<i>Factor</i>	→	nint
<i>Factor</i>	→	nfix
<i>Factor</i>	→	ctebool
<i>Factor</i>	→	nobool <i>Factor</i>
<i>Factor</i>	→	lpar <i>Expr</i> rpar
<i>Factor</i>	→	trunc lpar <i>Esimple</i> rpar
<i>Ref</i>	→	id

Figura B.2: Gramática que genera el lenguaje de la sección B.6.

EXPRESIÓN REGULAR	COMPONENTE LÉXICO	VALOR LÉXICO ENTREGADO
[\n\t]+	(ninguno)	
program	program	(palabra reservada)
begin	begin	(palabra reservada)
end	end	(palabra reservada)
var	var	(palabra reservada)
boolean	boolean	(palabra reservada)
integer	integer	(palabra reservada)
real	real	(palabra reservada)
writeln	wri	(palabra reservada)
write	wri	(palabra reservada)
read	read	(palabra reservada)
if	if	(palabra reservada)
then	then	(palabra reservada)
else	else	(palabra reservada)
while	while	(palabra reservada)
do	do	(palabra reservada)
and	ybool	(palabra reservada)
or	obool	(palabra reservada)
div	mulop	(palabra reservada)
mod	mulop	(palabra reservada)
true	ctebool	(palabra reservada)
false	ctebool	(palabra reservada)
not	nobool	(palabra reservada)
trunc	trunc	(palabra reservada)
[A-Za-z][0-9A-Za-z]*	id	(nombre del ident.)
[0-9]+	nint	(valor numérico)
([0-9]+)"."([0-9]+)	nfix	(valor numérico)
,	coma	
;	pyc	
.	punto	
:	dosp	
(lpar	
)	rpar	
=	relop	=
<>	relop	<>
<	relop	<
<=	relop	<=
>	relop	>
>=	relop	>=
+	addop	+
-	addop	-
*	mulop	*
/	mulop	/
:=	assop	

Figura B.3: Componentes léxicos del ejercicio práctico de la sección B.6.

Notas:

1. En Pascal (y en este lenguaje), al contrario que en C, no se distingue entre letras mayúsculas y minúsculas. Por tanto, “BeGiN”, “bEGIn” y “Begin” son sólo algunas de las posibles formas de escribir la palabra reservada “begin”; lo mismo sucede con

el resto de palabras reservadas que aparecen en la especificación léxica. Además, lo mismo sucede con los identificadores de variables: es posible referirse a una variable utilizando indistintamente letras mayúsculas o minúsculas, por lo que por ejemplo la variable “`pep`” se puede escribir también como “`Pep`”, “`PEP`” o “`pEp`”.

2. El analizador léxico debe ignorar los comentarios, que en este ejercicio empiezan con la secuencia de caracteres “`(*`” y terminan con la secuencia “`*)`”. Un comentario puede ocupar varias líneas y no se permiten los comentarios anidados.

B.6.1 Especificación semántica

Las reglas semánticas de este lenguaje son similares a las de Pascal, y son las siguientes:

Declaración de variables

- No es posible declarar dos veces una variable con el mismo identificador, aunque sea con el mismo tipo (recuérdese que no se debe distinguir entre mayúsculas y minúsculas). El identificador que aparece después de la palabra reservada “`program`” no se debe almacenar en la tabla de símbolos.
- No es posible utilizar una variable sin haberla declarado previamente.
- Si al declarar una variable el espacio ocupado por ésta sobrepasa el tamaño máximo de memoria para variables (siempre inferior a 16384, que es el tamaño de la memoria de la máquina virtual), el compilador debe producir un mensaje de error indicando el lexema exacto de la variable que ya no cabe en memoria.

Instrucciones

Asignación: en esta instrucción tanto la referencia que aparece a la izquierda del operador “`:=`” como la expresión de la derecha deben ser del mismo tipo. Solamente hay una excepción a esta regla, y consiste en que está permitida también la asignación cuando la referencia es real y la expresión es entera.

Lectura y escritura: aunque en Pascal existe otra instrucción para lectura (“`readln`”) y es posible especificar el formato en las instrucciones de escritura, en este lenguaje solamente se permite una sentencia de lectura, “`read`”, y dos sentencias de escritura, “`write`” y “`writeln`”, que se diferencian únicamente en que la segunda escribe un carácter de nueva línea (un “`\n`” de C) después de haber escrito todas las expresiones.

La lectura de valores enteros y reales no plantea ningún problema especial. Por simplificar, si lo que se debe leer es de tipo booleano se seguirá la siguiente norma (que no existe en Pascal): se leerá un carácter y si dicho carácter es una “`t`” (minúscula), se entenderá que el valor leído es “`true`”; si el carácter leído es cualquier otro, se entenderá que se ha leído “`false`”. Al escribir un valor booleano se escribirá el carácter “`t`” para “`true`” y el carácter “`f`” para “`false`”.

Control de flujo: las instrucciones “if” y “while” tienen una semántica similar a la de C, con la única excepción de que se exige que el tipo de la expresión sea booleano; en caso contrario, se debe producir un error semántico.

Expresiones

Precedencia de operadores: la siguiente tabla muestra los operadores de este lenguaje ordenados de menor a mayor precedencia (todos aquellos que aparecen en la misma fila tiene la misma precedencia):

OPERADORES
= <> > >= < <=
+ - or
* / div mod and
not

La gramática de la especificación sintáctica refleja esta tabla de precedencias, por lo que no es aconsejable modificarla (al menos las reglas de las expresiones).

Operadores booleanos: los operadores “or”, “and” y “not” son equivalentes a los operadores “||”, “&&” y “!” de C, con la restricción de que en este lenguaje los operandos deben ser booleanos (en C no existe esa restricción porque no existe el tipo booleano). Además, debe tenerse en cuenta la precedencia de los operadores al construirse una expresión booleana: la expresión en C “2<3 && 3<4” se debe escribir en este lenguaje como “(2<3) and (3<4)”, ya que los operadores relacionales tienen menor precedencia que el operador “and”.

Operadores relacionales: son equivalentes a los de C, excepto el operador de igualdad “=” (equivalente al “==” de C) y el de desigualdad “<>” (equivalente a “!=”). Estos operadores permiten comparar valores numéricos (enteros o reales) entre sí, y también valores booleanos entre sí (en cuyo caso se considera que “false” es menor que “true”). No se permite comparar valores de distinto tipo (excepto valores enteros y reales, por supuesto).

Operadores aritméticos: solamente pueden utilizarse con valores enteros o reales. Los operadores de suma “+”, resta “-” y producto “*” son similares a los de C. En cambio, el operador de división “/” realiza solamente divisiones reales, independientemente de si los operandos son reales o enteros. Para realizar una división entera se debe utilizar el operador “div”, en cuyo caso ambos operandos deben ser enteros. Además, es posible obtener el módulo de una división entera con el operando “mod”, que también exige que los operandos sean enteros.

Funciones predefinidas

En Pascal existen muchas funciones predefinidas, pero en este ejercicio solamente es necesario implementar una de ellas, que se ha incluido para facilitar la conversión de tipos:

trunc Esta función admite un argumento numérico (real o entero) y devuelve un entero que es la parte entera del argumento.

B.7 Compilador con tipos complejos

Este ejercicio práctico consiste en ampliar el ejercicio anterior para admitir la declaración y utilización de *arrays* y registros. Como en el ejercicio anterior, el código objeto debe ser el lenguaje **m2r**.

El lenguaje fuente seguirá siendo un subconjunto de Pascal y la semántica también será similar a la de dicho lenguaje, con una excepción: aunque el compilador de Pascal genera código para comprobar que los índices de un *array* están dentro del rango permitido, este compilador no debe generar dicho código, para que el código generado sea más legible y sencillo de depurar.

La sintaxis del lenguaje fuente puede ser representada por la gramática de la figura B.2 añadiéndole las siguientes reglas:

$$\begin{array}{ll} \textit{Tipo} & \longrightarrow \textbf{record Decl end} \\ \textit{Tipo} & \longrightarrow \textbf{array lcor Rango rcor of Tipo} \\ \textit{Rango} & \longrightarrow \textbf{nint ptopto nint} \end{array}$$

para declarar *arrays* y registros, y

$$\begin{array}{ll} \textit{Ref} & \longrightarrow \textbf{id} \\ \textit{Ref} & \longrightarrow \textit{Ref} \textbf{lcor LisExpr rcor} \\ \textit{Ref} & \longrightarrow \textit{Ref} \textbf{punto id} \\ \textit{LisExpr} & \longrightarrow \textit{LisExpr} \textbf{coma Expr} \\ \textit{LisExpr} & \longrightarrow \textit{Expr} \end{array}$$

para acceder a posiciones de *arrays* y a campos de registros. Aunque el no terminal *LisExpr* genera exactamente el mismo lenguaje que el no terminal *LExpr* de la gramática de la tercera práctica, no es aconsejable simplificar la gramática unificando ambos no terminales, ya que el código que se debe generar en cada caso es muy diferente. Esta modificación de la gramática permitiría referencias como “a[1,2][3,4]”, que, por simplificar, no están permitidas en este lenguaje y por tanto el compilador debe producir un error semántico si aparece una referencia de este tipo. En cambio, sí están permitidas referencias como “a[1,2].b[3,4]”, y es para permitir este tipo de referencias por lo que es necesaria una gramática como la anterior.

La especificación léxica de los símbolos terminales que se añaden a la gramática es:

EXPRESIÓN REGULAR	COMPONENTE LÉXICO	VALOR LÉXICO ENTREGADO
record	record	(palabra reservada)
array	array	(palabra reservada)
of	of	(palabra reservada)
[lcor	
]	rcor	
..	ptopto	

B.7.1 Especificación semántica

Las reglas semánticas del ejercicio práctico anterior siguen vigentes en este ejercicio y se añaden las siguientes reglas.

Declaración de *arrays*

- En la declaración de un *array*, el segundo número que aparece en el rango entre corchetes debe ser mayor o igual que el primero. Debe tenerse en cuenta que ambos valores son válidos como índices (al contrario que en C, que no incluye el valor que aparece entre corchetes en el rango de valores posibles de los índices del *array*).
- Al declarar una variable de tipo *array* es posible que se agote la memoria disponible en la máquina objeto, por lo que debe producirse un error semántico. En ningún caso está permitido que se reserve más memoria que la que sea imprescindible para un *array*.

Acceso a componentes de *arrays* y a campos de registros

- En este lenguaje no es posible hacer referencia a un componente de un *array* sin poner tantos índices como sean necesarios según la declaración de la variable. Tanto si faltan como si sobran índices se debe producir un error semántico (lo antes posible, especialmente cuando sobran índices). De esta manera, el no terminal *Ref* debe devolver un tipo básico (entero, real o booleano) al resto del compilador, de manera que, si está bien diseñado, no debe ser necesario modificar apenas el código del ejercicio práctico anterior.
- No es posible poner una referencia a un registro sin poner a continuación el punto y el nombre de un campo. Por supuesto, el identificador que aparece después del punto debe ser el de un campo del registro.
- No está permitido poner corchetes (índices) a una referencia que no sea de tipo *array*. Tampoco está permitido poner un punto después de una referencia que no sea un registro.
- El índice de un *array* debe ser de tipo entero; en caso contrario, se debe producir un error.
- Se debe recordar que es posible utilizar referencias a *arrays* en la parte izquierda de una asignación, en una sentencia de lectura y en una expresión. En los dos primeros casos el código que se debe generar es similar, y es ligeramente distinto del tercer caso.

B.8 Compilador con funciones

Este ejercicio práctico consiste en implementar declaraciones y llamadas a funciones a partir de una versión simplificada del ejercicio práctico de la sección B.6, que solamente incluirá los tipos entero y booleano (aunque los parámetros de las funciones solamente

podrán ser enteros). Como en los ejercicios anteriores, el código objeto será el `m2r`. El lenguaje fuente seguirá siendo un subconjunto de Pascal y la semántica también será similar a la de dicho lenguaje. Opcionalmente, se podrán implementar funciones locales a otras funciones.

La sintaxis del lenguaje fuente puede ser representada por la gramática de la figura B.4². La especificación léxica de este lenguaje incluye casi todos los símbolos terminales del ejercicio práctico de la sección B.6 y la palabra reservada **function**.

B.8.1 Especificación semántica

Las reglas semánticas del ejercicio práctico de la sección B.6 siguen vigentes en esta ejercicio (las que se puedan aplicar, obviamente), y se añaden las siguientes reglas:

Declaración de funciones

- En el lenguaje de este ejercicio práctico existen dos ámbitos: el de los símbolos (variables o funciones) globales, y el de los símbolos locales a una función (argumentos o variables locales). Cuando se declara una función, el primer argumento (o la primera variable local, si no hay argumentos) se considera que abre el ámbito de los símbolos locales a la función. Por tanto, es posible que algún parámetro o variable local tenga el mismo nombre que una variable o función declarada previamente en el ámbito global, pero no está permitido declarar dos símbolos con el mismo nombre dentro del mismo ámbito (sea global o local).
- Cuando la función se termina de compilar, los símbolos pertenecientes al ámbito local de la función se deben *olvidar*, eliminándolos de la tabla de símbolos.
- Puesto que las funciones se almacenan en el ámbito global, no es posible declarar una función con el mismo nombre que una variable o función declarada previamente.

Nota de implementación: Por simplificar, aunque en la gramática existen dos tipos básicos (enteros y booleanos), las funciones solamente pueden devolver enteros y sus parámetros deben ser de tipo entero. Por tanto, no es necesario implementar una tabla de tipos para almacenar el tipo de la función junto con el de sus argumentos, es suficiente con almacenar en la tabla de símbolos cuántos argumentos tiene.

Llamadas a funciones

- No está permitido hacer una llamada a una función sin poner tantos parámetros como argumentos tenga. Debe tenerse en cuenta que si la función se ha declarado sin argumentos, la llamada simplemente consiste en poner el nombre de la función (sin paréntesis), por lo que gramaticalmente es idéntica a una referencia a una variable.

²Las reglas que se han añadido con respecto a la gramática de la figura B.2 están marcadas con una flecha.

- No está permitido poner parámetros entre paréntesis a un identificador que no sea una función.
- Como se dice más arriba, todos los parámetros de una función deben ser expresiones enteras (no están permitidos los argumentos booleanos).
- Solamente está permitido poner el nombre de una función en la parte izquierda de una asignación en un caso: dentro del cuerpo de la propia función. El valor de la expresión a la derecha del “:=” será el valor que devuelva la función (cuando llegue al final de su código). Está permitido asignar más de una vez valor al nombre de la función (dentro de su cuerpo, por supuesto)³, y no se debe comprobar que el usuario del compilador hace una asignación de este tipo en cada función (puede ser muy complicado en algunos casos).
- Los parámetros se pasan por valor, y aunque dentro de una función se modifique el valor de un parámetro (operación que, por lo tanto, está permitida), dicha modificación no tendrá efecto fuera de esa función. Los argumentos son todos, por tanto, de entrada y no pueden ser de salida.
- Por supuesto, el compilador debe permitir las llamadas recursivas sin ningún tipo de limitación, aunque la especificación sintáctica solamente permite la recursividad directa (que una función se llame a sí misma).

B.8.2 Ampliación opcional: funciones locales

Para realizar esta ampliación habría que modificar la regla de la declaración de funciones, de manera que quedase de esta forma:

$$DFun \longrightarrow \text{function id } Arg \text{ dosp integer pyc} \\ BDecl \text{ Bloque pyc}$$

y eliminar las reglas del no terminal $BDFun$. Esta modificación de la gramática permitiría declarar funciones locales a otras funciones, y por tanto habría más ámbitos además de el ámbito local y el global. El gran problema de esta ampliación es el acceso a variables situadas en aquellos ámbitos que no son ni el ámbito local ni el ámbito global; además, la gestión de la tabla de símbolos se complica un poco con respecto a la del ejercicio sin la ampliación.

³En Pascal no existe una instrucción como el “return” de C, por lo que las funciones tienen que llegar al final de su código.

	<i>S</i>	→	program id pyc BDecl Bloque punto
	<i>Bloque</i>	→	begin SeqInstr end
	<i>BDecl</i>	→	ε
	<i>BDecl</i>	→	<i>BlVar BDecl</i>
→	<i>BDecl</i>	→	<i>DFun BDecl</i>
	<i>BlVar</i>	→	var Decl pyc
	<i>Decl</i>	→	<i>Decl pyc DVar</i>
	<i>Decl</i>	→	<i>DVar</i>
	<i>DVar</i>	→	<i>LIdent dosp Tipo</i>
→	<i>DFun</i>	→	function id Arg dosp integer pyc <i>BDFun Bloque pyc</i>
→	<i>Arg</i>	→	ε
→	<i>Arg</i>	→	lpar LArg rpar
→	<i>LArg</i>	→	<i>LArg pyc UnArg</i>
→	<i>LArg</i>	→	<i>UnArg</i>
→	<i>UnArg</i>	→	id dosp integer
→	<i>BDFun</i>	→	ε
→	<i>BDFun</i>	→	<i>BlVar BDFun</i>
	<i>Tipo</i>	→	boolean
	<i>Tipo</i>	→	integer
	<i>LIdent</i>	→	<i>LIdent coma id</i>
	<i>LIdent</i>	→	id
	<i>SeqInstr</i>	→	<i>SeqInstr pyc Instr</i>
	<i>SeqInstr</i>	→	<i>Instr</i>
	<i>Instr</i>	→	<i>Bloque</i>
	<i>Instr</i>	→	<i>Ref assop Expr</i>
	<i>Instr</i>	→	wri lpar LExpr rpar
	<i>LExpr</i>	→	<i>LExpr coma Expr</i>
	<i>LExpr</i>	→	<i>Expr</i>
	<i>Instr</i>	→	read lpar LRef rpar
	<i>LRef</i>	→	<i>LRef coma Ref</i>
	<i>LRef</i>	→	<i>Ref</i>
	<i>Instr</i>	→	if Expr then Instr
	<i>Instr</i>	→	if Expr then Instr else Instr
	<i>Instr</i>	→	while Expr do Instr
	<i>Expr</i>	→	<i>Esimple relop Esimple</i>
	<i>Expr</i>	→	<i>Esimple</i>
	<i>Esimple</i>	→	<i>Esimple addop Term</i>
	<i>Esimple</i>	→	<i>Esimple obool Term</i>
	<i>Esimple</i>	→	<i>Term</i>
	<i>Esimple</i>	→	addop Term
	<i>Term</i>	→	<i>Term mulop Factor</i>
	<i>Term</i>	→	<i>Term ybool Factor</i>
	<i>Term</i>	→	<i>Factor</i>
	<i>Factor</i>	→	<i>Ref</i>
	<i>Factor</i>	→	nint
	<i>Factor</i>	→	ctebool
	<i>Factor</i>	→	nobool Factor
	<i>Factor</i>	→	lpar Expr rpar
	<i>Ref</i>	→	id
→	<i>Ref</i>	→	id lpar LisExpr rpar
→	<i>LisExpr</i>	→	<i>LisExpr coma Expr</i>
→	<i>LisExpr</i>	→	<i>Expr</i>

Figura B.4: Gramática para un compilador con funciones.

Apéndice C

El lenguaje intermedio m2r

El lenguaje es un ensamblador para una máquina virtual (m2r), que sólo tiene dos registros, A y B, que actúan como acumulador y registro base respectivamente. Tiene un espacio de memoria para el código lo suficientemente grande como para almacenar un programa típico, y otro espacio de 16384 posiciones de memoria para datos (de la 0 a la 16383). Este espacio de datos debe ser gestionado por el compilador según el criterio de quien lo diseñe. Los tipos de datos básicos, entero y real, se pueden almacenar en *una* posición de memoria (aunque internamente se representan de distinta manera y por tanto un valor entero no puede utilizarse como si fuera real sin una conversión explícita, por ejemplo).

El intérprete de la máquina objeto entiende texto ASCII compuesto por instrucciones, una por línea, tomadas del conjunto que se describe seguidamente. Las instrucciones pueden ir precedidas de un número de línea, de una etiqueta o de nada. El intérprete admite comentarios que empiezan por ‘;’ y terminan con el final de la línea.

El repertorio de instrucciones de m2r es:

<code>mov <i>fuelle</i> <i>destino</i></code>	Copia en <i>destino</i> el valor <i>fuelle</i> .
<code>addi <i>fuelle</i></code>	Suma a A el valor <i>fuelle</i> . Se supone que tanto el acumulador como el valor <i>fuelle</i> son enteros, y el resultado es también entero.
<code>addr <i>fuelle</i></code>	Igual que <code>addi</code> , pero todos los operandos y el resultado son reales.
<code>subi <i>fuelle</i></code>	Resta de A (entero) el valor <i>fuelle</i> (entero), y deja el resultado (entero) en A.
<code>subr <i>fuelle</i></code>	Igual que <code>subi</code> pero con reales.
<code>muli <i>fuelle</i></code>	Multiplica A (entero) por el valor <i>fuelle</i> (entero) y guarda el resultado (entero) en A.
<code>mulr <i>fuelle</i></code>	Igual que <code>muli</code> pero con reales.
<code>divi <i>fuelle</i></code>	Divide A (entero) por el valor <i>fuelle</i> (entero) y guarda el cociente de la división entera en A (que es entero).

<code>divr fuente</code>	Igual que <code>divi</code> pero con reales.
<code>modi fuente</code>	Igual que <code>divi</code> , pero en lugar de guardar el cociente guarda el resto de la división (también entero).
<code>andi fuente</code>	Deja en <code>A</code> un 1 si el valor (entero) de <code>A</code> y el valor (entero) de <code>fuentes</code> son iguales a 1, y deja un 0 en otro caso. El resultado, que queda en <code>A</code> , es entero.
<code>andr fuente</code>	Igual que <code>andi</code> pero se supone que <code>A</code> y <code>fuentes</code> son reales, aunque el resultado es entero.
<code>ori fuente</code>	Deja en <code>A</code> un 0 si el valor (entero) de <code>A</code> y el valor (entero) de <code>fuentes</code> son iguales a 0, y deja un 1 en otro caso. Como con <code>andi</code> , el resultado es entero.
<code>orr fuente</code>	Igual que <code>ori</code> pero se supone que <code>A</code> y <code>fuentes</code> son reales, aunque el resultado es entero.
<code>noti</code>	Deja en <code>A</code> un 1 si el valor (entero) de <code>A</code> es cero, y deja un 0 en otro caso. El resultado es entero.
<code>notr</code>	Igual que <code>noti</code> , pero el valor de <code>A</code> debe ser real y el resultado es entero.
<code>itor</code>	Convierte el valor entero de <code>A</code> en un valor real y lo deja en el acumulador.
<code>rtoi</code>	Toma la parte entera del valor real de <code>A</code> y la deja como entero en el acumulador. Es equivalente a la sentencia en C “ <code>A = (int)A</code> ”.
<code>halt</code>	Detiene la máquina (el intérprete).
<code>wri fuente</code>	Imprime el valor (entero) de <code>fuentes</code> .
<code>wrr fuente</code>	Imprime el valor (real) de <code>fuentes</code> .
<code>wrc fuente</code>	Imprime el carácter representado por los 8 bits más bajos del valor entero <code>fuentes</code> .
<code>wrl</code>	Imprime un salto de línea.
<code>rdi destino</code>	Lee un entero de la consola y lo carga en <code>destino</code> .
<code>rdr destino</code>	Lee un real de la consola y lo carga en <code>destino</code> .
<code>rdc destino</code>	Lee un carácter de la consola y carga su código ASCII en <code>destino</code> .

eqli <i>fuelle</i>	Deja un 1 en A si el valor (entero) de A es igual que el valor (entero) <i>fuelle</i> , y deja un 0 en otro caso. El resultado es entero.
eqlr <i>fuelle</i>	Igual que eqli , pero los operandos son reales. El resultado también es entero.
neqi <i>fuelle</i>	Deja un 1 en A si el valor (entero) de A es distinto del valor (entero) <i>fuelle</i> , y deja un 0 en otro caso. El resultado es entero.
neqr <i>fuelle</i>	Igual que neqi , pero los operandos son reales. El resultado también es entero.
gtri <i>fuelle</i>	Deja un 1 en A si el valor (entero) de A es mayor que el valor (entero) <i>fuelle</i> , y deja un 0 en otro caso. El resultado es entero.
gtrr <i>fuelle</i>	Igual que gtri , pero los operandos son reales. El resultado también es entero.
geqi <i>fuelle</i>	Deja un 1 en A si el valor (entero) de A es mayor o igual que el valor (entero) <i>fuelle</i> , y deja un 0 en otro caso. El resultado es entero.
geqr <i>fuelle</i>	Igual que geqi , pero los operandos son reales. El resultado también es entero.
lssi <i>fuelle</i>	Deja un 1 en A si el valor (entero) de A es menor que el valor (entero) <i>fuelle</i> , y deja un 0 en otro caso. El resultado es entero.
lssr <i>fuelle</i>	Igual que lssi , pero los operandos son reales. El resultado también es entero.
leqi <i>fuelle</i>	Deja un 1 en A si el valor (entero) de A es menor o igual que el valor (entero) <i>fuelle</i> , y deja un 0 en otro caso. El resultado es entero.
leqr <i>fuelle</i>	Igual que leqi , pero los operandos son reales. El resultado también es entero.
jmp <i>posprog</i>	Salta a la posición de programa indicada por el valor <i>posprog</i> .
jz <i>posprog</i>	Salta a la posición de programa indicada por el valor <i>posprog</i> si en A hay un cero. El valor de A debe ser entero.
jnz <i>posprog</i>	Salta a la posición de programa indicada por el valor <i>posprog</i> si en A hay un número entero distinto de cero.

mv_{etq} *etiqueta destino* Copia en *destino* la posición de programa asociada a la *etiqueta*. Esta instrucción sirve para almacenar la posición de programa a la que hay que volver después de una llamada a una función.

Las clases posibles de *destino* son:

- n** la dirección de memoria **n**.
- A** el acumulador.
- @A** la dirección de memoria que representa el valor que hay en **A**.
- @B+n** la dirección que se obtiene de sumar **n** al contenido de **B**.
- @B-n** la dirección que se obtiene de restar **n** al contenido de **B**.
- B** el registro base.

Las clases posibles de *fuentes* son:

- #i el valor numérico entero *i*.
- \$r el valor numérico real *r*.
- n el valor almacenado en la dirección de memoria *n*.
- A el valor almacenado en el acumulador.
- @A el valor almacenado en la dirección que representa el valor que hay en A.
- @B+n el valor almacenado en la dirección que se obtiene de sumar *n* al contenido de B.
- @B-n el valor almacenado en la dirección que se obtiene de restar *n* al contenido de B.
- B el valor almacenado en el registro base.

Las clases posibles de *posprog* son:

- n la posición de programa *n*.
- Ln una etiqueta, compuesta por la letra L seguida de un número: por ejemplo, L25.
- @A la posición de programa contenida en A.
- @B+n la posición de programa contenida en la dirección que se obtiene de sumar *n* al contenido de B.
- @B-n la posición de programa contenida en la dirección que se obtiene de restar *n* al contenido de B.

Un ejemplo de programa objeto correcto (aunque no necesariamente traducción de un programa fuente determinado) sería el siguiente:

```

mov #2 A      ; guarda 2 en A
addi #3       ; suma 3 a A
mov A 23      ; guarda el valor de A (5) en la direccion 23
mov 23 A      ; guarda el contenido de la direccion 23 (5) en A
subi #3       ; resta 3 de A
wri A         ; imprime el valor almacenado en A (2)
wrl           ; imprime un salto de linea
mov #7 A      ; guarda 7 en A
itor          ; convierte el 7 que en A en real (7.0)
divr $3.5     ; divide el valor que hay en A (7.0) por 3.5
wrr A         ; imprime el valor real que hay en A (2.0)
wrl           ; imprime un salto de linea
halt          ; termina el programa

```

El texto que sigue a ‘;’ en cada línea es un comentario.

Bibliografía

En este epígrafe vamos a citar aquéllos libros de texto sobre diseño de compiladores que, a nuestro juicio, son más adecuados para un curso como aquellos hacia los que está orientado nuestro libro. Además, citaremos otros libros interesantes, pero que no nos parecen tan importantes. Las referencias aparecen en orden de importancia (siempre según nuestro criterio).

[Louden, 1997] Louden, K. C. (1997). *Compiler construction: principles and practice*. PWS Publishing Company, Boston, Massachusetts.

Probablemente el mejor libro que se ha escrito en los últimos años. Aunque es muy completo, no llega a ser tan exhaustivo como el libro de [Aho, Sethi y Ullman, 1990], pero tiene un enfoque mucho más pedagógico y es un libro muy apropiado para la mayoría de los cursos sobre diseño de compiladores. No solamente expone bien todas las posibilidades que pueden plantearse en los distintos temas, sino que además el enfoque y la estructura de cada tema son los más adecuados.

[Aho, Sethi y Ullman, 1990] Aho, A. V., Sethi, R. y Ullman, J. D. (1990). *Compiladores: principios, técnicas y herramientas*. Addison-Wesley Iberoamericana, Madrid.

Este libro es la traducción al castellano de *Compilers: Principles, Techniques and Tools* (1986), Addison-Wesley, Reading, MA. Aunque la calidad de la traducción no es excesivamente buena, es el libro más importante y conocido¹ en esta materia. Es un libro muy completo, que trata en profundidad todos los aspectos del diseño de compiladores. Sin embargo, empieza a quedarse obsoleto en algunos aspectos. El contenido del libro se presta a ser utilizado para cursos de distintos niveles y puede llegar a ser excesivo para cursos introductorios.

[Bennett, 1990] Bennett, J. P. (1990). *Introduction to compiling techniques: a first course using ANSI C, LEX and YACC*. McGraw-Hill International (UK).

Un buen libro para cursos introductorios al diseño de compiladores. Aunque no es tan completo como los otros libros (ni pretende serlo), tiene excelentes

¹También se le conoce coloquialmente como el “libro del dragón” (the “dragon book”) porque en la cubierta aparece un dragón.

planteamientos pedagógicos en muchos aspectos. Además, está orientado a la realización práctica de compiladores con LEX y YACC, lo cual lo hace muy adecuado para cursos orientados a los trabajos prácticos.

[Fischer y LeBlanc, 1991] Fischer, C. N., LeBlanc, R. J., Jr. (1991). *Crafting a Compiler with C*. Benjamin/Cummings, Menlo Park, California.

Un libro que trata especialmente bien el capítulo de análisis sintáctico descendente predictivo, que es la especialidad de los autores. También plantea bien los capítulos de análisis léxico y análisis sintáctico ascendente, pero el resto del libro está basado en ejemplos en C que dificultan la comprensión del contenido.

Otras referencias interesantes

[Terry, 1997] Terry, P. D. (1997). *Compilers & compiler generators: an introduction with C++*. International Thomson Computer Press, Boston, Massachusetts.

[Aho y Ullman, 1972] Aho, A. V., Ullman, J. D. (1972). *The Theory of Parsing, Translation and Compiling, vol. 1: Parsing*. Prentice-Hall, Englewood Cliffs, N.J.

[Teufel, Schmidt y Teufel, 1995] Teufel, B., Schmidt, S., Teufel, T. (1995). *Compiladores: conceptos fundamentales*. Addison-Wesley Iberoamericana, Wilmington, Delaware.

[Tremblay y Sorenson, 1985] Tremblay, J. P., Sorenson, P. G. (1985). *The theory and practice of compiler writing*. McGraw-Hill, New York.

[Appel, 1998] Appel, A. W. (1998) *Modern compiler implementation in Java*. Cambridge University Press, Cambridge, UK.

[Holub, 1990] Holub, A. I. (1990) *Compiler Design in C*. Prentice-Hall, Englewood Cliffs, N.J.

[Hopcroft y Ullman, 1979] Hopcroft, J. E., Ullman, J. D. (1979). *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, Massachusetts.