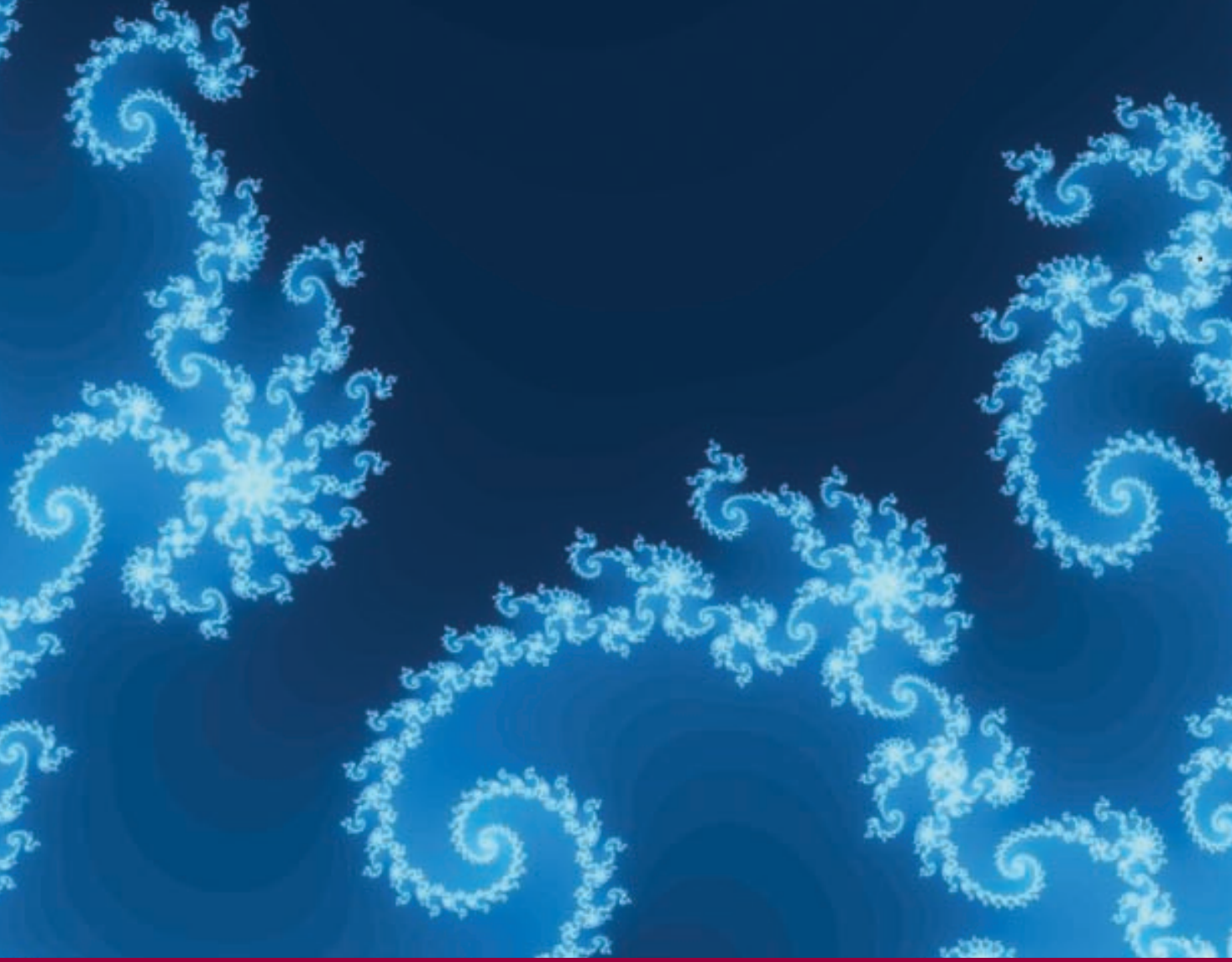


Compiladores e interpretes:
teoría y práctica

www.librosite.net/pulido

Manuel Alfonseca Moreno
Marina de la Cruz Echeandía
Alfonso Ortega de la Puente
Estrella Pulido Cañabate

PEARSON
Prentice Hall

Compiladores e intérpretes: teoría y práctica

Compiladores e intérpretes: teoría y práctica

Manuel Alfonseca Moreno
Marina de la Cruz Echeandía
Alfonso Ortega de la Puente
Estrella Pulido Cañabate
Departamento de Ingeniería Informática
Universidad Autónoma de Madrid

Madrid • México • Santafé de Bogotá • Buenos Aires • Caracas • Lima
Montevideo • San Juan • San José • Santiago • Sâo Paulo • Reading, Massachusetts • Harlow, England

Datos de catalogación bibliográfica

ALFONSECA MORENO, M.; DE LA CRUZ ECHEANDÍA, M.; ORTEGA DE LA PUENTE, A.; PULIDO CAÑABATE, E.
Compiladores e intérpretes: teoría y práctica
PEARSON EDUCACIÓN, S.A., Madrid, 2006

ISBN 10: 84-205-5031-0
ISBN 13: 978-84-205-5031-2
MATERIA: Informática, 004.4

Formato: 195 × 250 mm Páginas: 376

Alfonseca Moreno, M.; de la Cruz Echeandía, M.; Ortega de la Puente, A.; Pulido Cañabate, E.
Compiladores e intérpretes: teoría y práctica

ISBN: 84-205-5031-0
ISBN 13: 978-84-205-5031-2
Depósito Legal: M.

PEARSON PRENTICE HALL es un sello editorial autorizado de PEARSON EDUCACIÓN, S.A.

Equipo editorial
Editor: Miguel Martín-Romo
Técnico editorial: Marta Caicoya
Equipo de producción:
Director: José A. Clares
Técnico: José A. Hernán
Diseño de cubierta: Equipo de diseño de PEARSON EDUCACIÓN, S. A.
Composición: JOSUR TRATAMIENTOS DE TEXTOS, S.L.
Impreso por:

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Este libro ha sido impreso con papel y tintas ecológico

Contenido 1

Capítulo 1

Lenguajes, gramáticas y procesadores

1.1 Gödel y Turing

En el año 1931 se produjo una revolución en las ciencias matemáticas, con el descubrimiento realizado por Kurt Gödel (1906-1978) y publicado en su famoso artículo [1], que quizá deba considerarse el avance matemático más importante del siglo XX.

En síntesis, el teorema de Gödel dice lo siguiente: *Toda formulación axiomática consistente de la teoría de números contiene proposiciones indecidibles*. Es decir, cualquier teoría matemática ha de ser incompleta. Siempre habrá en ella afirmaciones que no se podrán demostrar ni negar.

El teorema de Gödel puso punto final a las esperanzas de los matemáticos de construir un sistema completo y consistente, en el que fuese posible demostrar cualquier teorema. Estas esperanzas habían sido expresadas en 1900 por David Hilbert (1862-1943), quien generalizó sus puntos de vista proponiendo el *problema de la decisión* (*Entscheidungsproblem*), cuyo objetivo era descubrir un método general para decidir si una fórmula lógica es verdadera o falsa.

En 1937, el matemático inglés Alan Mathison Turing (1912-1953) publicó otro artículo famoso sobre los *números calculables*, que desarrolló el teorema de Gödel y puede considerarse el origen oficial de la informática teórica. En este artículo introdujo la *máquina de Turing*, una entidad matemática abstracta que formalizó por primera vez el concepto de *algoritmo*[1] y resultó ser precursora de las máquinas de calcular automáticas, que comenzaron a extenderse a partir de la década siguiente. Además, el teorema de Turing demostraba que existen problemas irresolubles, es decir, que ninguna máquina de Turing (y, por ende, ninguna computadora) será

[1] Recuérdese que se llama *algoritmo* a un conjunto de reglas que permite obtener un resultado determinado a partir de ciertos datos de partida. El nombre procede del matemático persa Abu Ja'far Mohammed ibn Musa al-Jowârizmî, autor de un tratado de aritmética que se publicó hacia el año 825 y que fue muy conocido durante la Edad Media.

capaz de obtener su solución. Por ello se considera a Turing el padre de la *teoría de la computabilidad*.

El teorema de Turing es, en el fondo, equivalente al teorema de Gödel. Si el segundo demuestra que no todos los teoremas pueden demostrarse, el primero dice que no todos los problemas pueden resolverse. Además, la demostración de ambos teoremas es muy parecida. Uno de esos problemas que no se puede resolver es el denominadol *problema de la parada de la máquina de Turing*. Puede demostrarse que la suposición de que es posible predecir, dada la descripción de una máquina de Turing y la entrada que recibe, si llegará a pararse o si continuará procesando información indefinidamente, lleva a una contradicción. Esta forma de demostración, muy utilizada en las ciencias matemáticas, se llama *reducción al absurdo*.

1.2 Autómatas

El segundo eslabón en la cadena vino de un campo completamente diferente: la ingeniería eléctrica. En 1938, otro artículo famoso [2] del matemático norteamericano Claude Elwood Shannon (1916-2001), quien más tarde sería más conocido por su teoría matemática de la comunicación, vino a establecer las bases para la aplicación de la lógica matemática a los circuitos combinatorios y secuenciales, construidos al principio con relés y luego con dispositivos electrónicos de vacío y de estado sólido. A lo largo de las décadas siguientes, las ideas de Shannon se convirtieron en la teoría de las máquinas secuenciales y de los autómatas finitos.

Los autómatas son sistemas capaces de transmitir información. En sentido amplio, todo sistema que acepta señales de su entorno y, como resultado, cambia de estado y transmite otras señales al medio, puede considerarse como un autómata. Con esta definición, cualquier máquina, una central telefónica, una computadora, e incluso los seres vivos, los seres humanos y las sociedades se comportarían como autómatas. Este concepto de autómata es demasiado general para su estudio teórico, por lo que se hace necesario introducir limitaciones en su definición.

Desde su nacimiento, la teoría de autómatas encontró aplicación en campos muy diversos, pero que tienen en común el manejo de conceptos como el *control*, la *acción*, la *memoria*. A menudo, los objetos que se controlan, o se recuerdan, son símbolos, palabras o frases de algún tipo.

Estos son algunos de los campos en los que ha encontrado aplicación la Teoría de Autómatas:

- Teoría de la comunicación.
- Teoría del control.
- Lógica de los circuitos secuenciales.
- Computadoras.
- Redes conmutadoras y codificadoras.
- Reconocimiento de patrones.
- Fisiología del sistema nervioso.
- Estructura y análisis de los lenguajes de programación para computadoras.
- Traducción automática de lenguajes.
- Teoría algebraica de lenguajes.

Se sabe que un autómata (o una máquina secuencial) recibe información de su entorno (*entrada* o *estímulo*), la transforma y genera nueva información, que puede transmitirse al entorno (*salida* o *respuesta*). Puede darse el caso de que la información que devuelve el autómata sea muy reducida: podría ser una señal binaria (como el encendido o apagado de una lámpara), que indica si la entrada recibida por el autómata es aceptada o rechazada por éste. Tendríamos, en este caso, un *autómata aceptador*.

1.3 Lenguajes y gramáticas

El tercer eslabón del proceso surgió de un campo que tradicionalmente no había recibido consideración de científico: la lingüística, la teoría de los lenguajes y las gramáticas. En la década de 1950, el lingüista norteamericano Avram Noam Chomsky (1928-) revolucionó su campo de actividad con la *teoría de las gramáticas transformacionales* [3, 4], que estableció las bases de la lingüística matemática y proporcionó una herramienta que, aunque Chomsky la desarrolló para aplicarla a los lenguajes naturales, facilitó considerablemente el estudio y la formalización de los lenguajes de computadora, que comenzaban a aparecer precisamente en aquella época.

El estudio de los lenguajes se divide en el análisis de la estructura de las frases (gramática) y de su significado (semántica). A su vez, la gramática puede analizar las formas que toman las palabras (morfología), su combinación para formar frases correctas (sintaxis) y las propiedades del lenguaje hablado (fonética). Por el momento, tan sólo esta última no se aplica a los lenguajes de computadora.

Aunque desde el punto de vista teórico la distinción entre sintaxis y semántica es un poco artificial, tiene una enorme trascendencia desde el punto de vista práctico, especialmente para el diseño y construcción de compiladores, objeto de este libro.

1.4 Máquinas abstractas y lenguajes formales

La teoría de lenguajes formales resultó tener una relación sorprendente con la teoría de máquinas abstractas. Los mismos fenómenos aparecen independientemente en ambas disciplinas y es posible establecer correspondencias entre ellas (lo que los matemáticos llamarían un *isomorfismo*).

Chomsky clasificó las gramáticas y los lenguajes formales de acuerdo con una jerarquía de cuatro grados, cada uno de los cuales contiene a todos los siguientes. El más general se llama *gramáticas del tipo 0 de Chomsky*. A estas gramáticas no se les impone restricción alguna. En consecuencia, el conjunto de los lenguajes que representan coincide con el de todos los lenguajes posibles.

El segundo grado es el de las *gramáticas del tipo 1*, que introducen algunas limitaciones en la estructura de las frases, aunque se permite que el valor sintáctico de las palabras dependa de su

posición en la frase, es decir, de su contexto. Por ello, los lenguajes representados por estas gramáticas se llaman *lenguajes sensibles al contexto*.

Las gramáticas del tercer nivel son las del *tipo 2 de Chomsky*, que restringen más la libertad de formación de las reglas gramaticales: en las gramáticas de este tipo, el valor sintáctico de una palabra es independiente de su posición en la frase. Por ello, los lenguajes representados por estas gramáticas se denominan *lenguajes independientes del contexto*.

Por último, las gramáticas del *tipo 3 de Chomsky* tienen la estructura más sencilla y corresponden a los *lenguajes regulares*. En la práctica, todos los lenguajes de computadora quedan por encima de este nivel, pero los lenguajes regulares no dejan por eso de tener aplicación.

Pues bien: paralelamente a esta jerarquía de gramáticas y lenguajes, existe otra de máquinas abstractas equivalentes. A las gramáticas del tipo 0 les corresponden las máquinas de Turing; a las del tipo 1, los autómatas acotados linealmente; a las del tipo 2, los autómatas a pila; finalmente, a las del tipo 3, corresponden los autómatas finitos. Cada uno de estos tipos de máquinas es capaz de resolver problemas cada vez más complicados: los más sencillos, que corresponden a los autómatas finitos, se engloban en el *álgebra de las expresiones regulares*. Los más complejos precisan de la capacidad de una máquina de Turing (o de cualquier otro dispositivo equivalente, *computacionalmente completo*, como una computadora digital) y se denominan *problemas recursivamente enumerables*. Y, por supuesto, según descubrió Turing, existen aún otros problemas que no tienen solución: los *problemas no computables*.

La Figura 1.1 resume la relación entre las cuatro jerarquías de las gramáticas, los lenguajes, las máquinas abstractas y los problemas que son capaces de resolver.

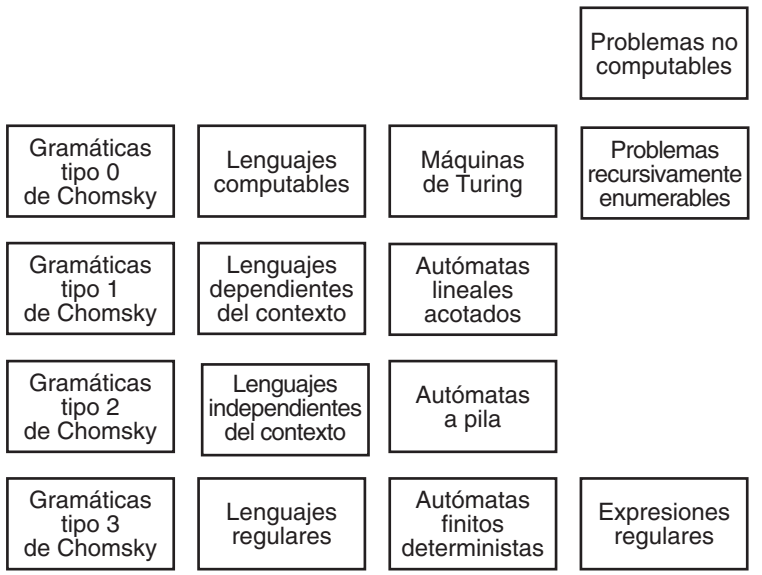


Figura 1.1. Relación jerárquica de las máquinas abstractas y los lenguajes formales.

1.5 Alfabetos, símbolos y palabras

Se llama *alfabeto* a un conjunto finito, no vacío. Los elementos de un alfabeto se llaman *símbolos*. Un alfabeto se define por enumeración de los símbolos que contiene. Por ejemplo:

Σ_1 = {A,B,C,D,E,...,Z}
Σ_2 = {0,1}
Σ_3 = {0,1,2,3,4,5,6,7,8,9,.}
Σ_4 = {/,\}

Se llama *palabra*, formada con los símbolos de un alfabeto, a una secuencia finita de los símbolos de ese alfabeto. Se utilizarán letras minúsculas como x o y para representar las palabras de un alfabeto:

x = JUAN (palabra sobre Σ_1)
y = 1234 (palabra sobre Σ_3)

Se llama longitud de una palabra al número de letras que la componen. La longitud de la palabra x se representa con la notación $|x|$. La palabra cuya longitud es cero se llama *palabra vacía* y se representa con la letra griega lambda (λ). Evidentemente, cualquiera que sea el alfabeto considerado, siempre puede formarse con sus símbolos la palabra vacía.

El conjunto de todas las palabras que se pueden formar con las letras de un alfabeto se llama *lenguaje universal* de Σ. De momento se utilizará la notación W(Σ) para representarlo. Es evidente que W(Σ) es un conjunto infinito. Incluso en el peor caso, si el alfabeto sólo tiene una letra (por ejemplo, Σ = {a}), las palabras que podremos formar son:

W(Σ) = {λ, a, aa, aaa, ...}

Es obvio que este conjunto tiene infinitos elementos. Obsérvese que la palabra vacía pertenece a los lenguajes universales de todos los alfabetos posibles.

1.6 Operaciones con palabras

Esta sección define algunas operaciones sobre el conjunto W(Σ) de todas las palabras que se pueden construir con las letras de un alfabeto $\Sigma = \{a_1, a_2, a_3, ...\}$.

1.6.1. Concatenación de dos palabras

Sean dos palabras x e y tales que $x \in W(\Sigma)$, $y \in W(\Sigma)$. Suponiendo que x tiene i letras, e y tiene j letras:

$x = a_1a_2...a_i$
$y = b_1b_2...b_j$

Donde todas las letras a_p, b_q son símbolos del alfabeto Σ. Se llama *concatenación de las palabras x e y* (y se representa xy) a otra palabra z, que se obtiene poniendo las letras de y a continuación de las letras de x:

$$z = xy = a_1 \ldots a_i b_1 \ldots b_j$$

La concatenación se representa a veces también x.y. Esta operación tiene las siguientes propiedades:

1. Operación cerrada: la concatenación de dos palabras de W(Σ) es una palabra de W(Σ).

   $$x \in W(\Sigma) \wedge y \in W(\Sigma) \Rightarrow xy \in W(\Sigma)$$

2. Propiedad asociativa:

   $$x(yz) = (xy)z$$

Por cumplir las dos propiedades anteriores, la operación de concatenación de las palabras de un alfabeto es un *semigrupo*.

3. Existencia de elemento neutro. La palabra vacía (λ) es el elemento neutro de la concatenación de palabras, tanto por la derecha, como por la izquierda. En efecto, sea x una palabra cualquiera. Se cumple que:

   $$\lambda x = x\lambda = x$$

Por cumplir las tres propiedades anteriores, la operación de concatenación de las palabras de un alfabeto es un *monoide* (semigrupo con elemento neutro).

4. La concatenación de palabras no tiene la propiedad conmutativa, como demuestra un contraejemplo. Sean las palabras x=abc, y=ad. Se verifica que

   xy=abcad
   yx=adabc

   Es evidente que xy no es igual a yx.

Sea z = xy. Se dice que x es *cabeza* de z y que y es *cola* de z. Además, x es *cabeza propia* de z si y no es la palabra vacía. De igual manera, y es *cola propia* de z si x no es la palabra vacía.

Se observará que la función *longitud de una palabra* tiene, respecto a la concatenación, propiedades semejantes a las de la función *logaritmo* respecto a la multiplicación de números reales:

$$|xy| = |x| + |y|$$

1.6.2. Monoide libre

Sea un alfabeto Σ. Cada una de sus letras puede considerarse como una palabra de longitud igual a 1, perteneciente a W(Σ). Aplicando a estas palabras elementales la operación concatenación, puede formarse cualquier palabra de W(Σ) excepto λ, la palabra vacía. Se dice entonces que Σ es un *conjunto de generadores* de W(Σ)-{λ}. Este conjunto, junto con la operación concatenación, es un semigrupo, pero no un monoide (pues carece de elemento neutro). Se dice que W(Σ)-{λ}

es el *semigrupo libre* engendrado por Σ. Añadiendo ahora la palabra vacía, diremos que W(Σ) es el *monoide libre* generado por Σ.

1.6.3. Potencia de una palabra

Estrictamente hablando, ésta no es una operación nueva, sino una notación que reduce algunos casos de la operación anterior.

Se llama *potencia i-ésima* de una palabra a la operación que consiste en concatenarla consigo misma i veces. Como la concatenación tiene la propiedad asociativa, no es preciso especificar el orden en que tienen que efectuarse las operaciones.

$x^i = xxx...x$ (i veces)

Definiremos también $x^1 = x$.

Es evidente que se verifica que:

$x^{i+1} = x^i x = xx^i \quad (i>0)$
$x^i x^j = x^{i+j} \quad (i,j>0)$

Para que las dos relaciones anteriores se cumplan también para $i, j = 0$ bastará con definir, para todo x,

$x^0 = \lambda$

También en este caso, las propiedades de la función *longitud* son semejantes a las del logaritmo.

$|x^i| = i.|x|$

1.6.4. Reflexión de una palabra

Sea $x = a_1 a_2 ... a_n$. Se llama *palabra refleja* o *inversa* de x, y se representa x^{-1}:

$x^{-1} = a_n ... a_2 a_1$

Es decir, a la que está formada por las mismas letras en orden inverso. La función longitud es invariante respecto a la reflexión de palabras:

$|x^{-1}| = |x|$

1.7 Lenguajes

Se llama *lenguaje* sobre el alfabeto a todo subconjunto del lenguaje universal de Σ.

$L \subset W(\Sigma)$

En particular, el conjunto vacío Φ es un subconjunto de W(Σ) y se llama por ello *lenguaje vacío*. Este lenguaje no debe confundirse con el que contiene como único elemento la palabra

vacía, $\{\lambda\}$, que también es un subconjunto (diferente) de $W(\Sigma)$. Para distinguirlos, hay que fijarse en que el *cardinal* (el número de elementos) de estos dos conjuntos es distinto.

$$c(\Phi) = 0$$
$$c(\{\lambda\}) = 1$$

Obsérvese que tanto Φ como $\{\lambda\}$ son lenguajes sobre cualquier alfabeto. Por otra parte, un alfabeto puede considerarse también como uno de los lenguajes generados por él mismo: el que contiene todas las palabras de una sola letra.

1.7.1. Unión de lenguajes

Sean dos lenguajes definidos sobre el mismo alfabeto, $L_1 \subset W(\Sigma)$, $L_2 \subset W(\Sigma)$. Llamamos *unión* de los dos lenguajes, $L_1 \cup L_2$, al lenguaje definido así:

$$\{x \mid x \in L_1 \vee x \in L_2\}$$

Es decir, al conjunto formado por las palabras que pertenezcan indistintamente a uno u otro de los dos lenguajes. La unión de lenguajes tiene las siguientes propiedades:

1. Operación cerrada: la unión de dos lenguajes sobre el mismo alfabeto es también un lenguaje sobre dicho alfabeto.
2. Propiedad asociativa: $(L_1 \cup L_2) \cup L_3 = L_1 \cup (L_2 \cup L_3)$.
3. Existencia de elemento neutro: cualquiera que sea el lenguaje L, el lenguaje vacío Φ cumple que

   $$\Phi \cup L = L \cup \Phi = L$$

Por cumplir las tres propiedades anteriores, la unión de lenguajes es un monoide.

4. Propiedad conmutativa: cualesquiera que sean L_1 y L_2, se verifica que $L_1 \cup L_2 = L_2 \cup L_1$.

Por tener las cuatro propiedades anteriores, la unión de lenguajes es un monoide abeliano.

5. Propiedad idempotente: cualquiera que sea L, se verifica que

   $$L \cup L = L$$

1.7.2. Concatenación de lenguajes

Sean dos lenguajes definidos sobre el mismo alfabeto, $L_1 \subset W(\Sigma)$, $L_2 \subset W(\Sigma)$. Llamamos *concatenación* de los dos lenguajes, L_1L_2, al lenguaje definido así:

$$\{xy \mid x \in L_1 \wedge y \in L_2\}$$

Es decir: todas las palabras del lenguaje concatenación se forman concatenando una palabra del primer lenguaje con otra del segundo.

La definición anterior sólo es válida si L_1 y L_2 contienen al menos un elemento. Extenderemos la operación concatenación al lenguaje vacío de la siguiente manera:

$\Phi L = L\Phi = \Phi$

La concatenación de lenguajes tiene las siguientes propiedades:

1. Operación cerrada: la concatenación de dos lenguajes sobre el mismo alfabeto es otro lenguaje sobre el mismo alfabeto.
2. Propiedad asociativa: $(L_1L_2)L_3 = L_1(L_2L_3)$.
3. Existencia de elemento neutro: cualquiera que sea el lenguaje L, el lenguaje de la palabra vacía cumple que

   $\{\lambda\}L = L\{\lambda\} = L$

Por cumplir las tres propiedades anteriores, la concatenación de lenguajes es un monoide.

1.7.3. Binoide libre

Acabamos de ver que existen dos monoides (la unión y la concatenación de lenguajes) sobre el conjunto L de todos los lenguajes que pueden definirse con un alfabeto dado Σ. Se dice que estas dos operaciones constituyen un *binoide*. Además, las letras del alfabeto pueden considerarse como lenguajes de una sola palabra. A partir de ellas, y mediante las operaciones de unión y concatenación de lenguajes, puede generarse cualquier lenguaje sobre dicho alfabeto (excepto Φ y $\{\lambda\}$). Por lo tanto, el alfabeto es un conjunto de generadores para el conjunto L, por lo que L se denomina *binoide libre* generado por Σ.

1.7.4. Potencia de un lenguaje

Estrictamente hablando, ésta no es una operación nueva, sino una notación que reduce algunos casos de la operación anterior.

Se llama *potencia i-ésima* de un lenguaje a la operación que consiste en concatenarlo consigo mismo i veces. Como la concatenación tiene la propiedad asociativa, no es preciso especificar el orden en que tienen que efectuarse las i operaciones.

$L^i = LLL...L$ (i veces)

Definiremos también $L^1 = L$.

Es evidente que se verifica que:

$L^{i+1} = L^iL = LL^i \quad (i>0)$
$L^iL^j = L^{i+j} \quad (i,j>0)$

Para que las dos relaciones anteriores se cumplan también para i,j = 0 bastará con definir, para todo L:

$L^0 = \{\lambda\}$

1.7.5. Clausura positiva de un lenguaje

La clausura positiva de un lenguaje L se define así:

$$L^{+} = \bigcup_{i=1}^{\infty} L^{i}$$

Es decir, el lenguaje obtenido uniendo el lenguaje L con todas sus potencias posibles, excepto L^0. Obviamente, ninguna clausura positiva contiene la palabra vacía, a menos que dicha palabra esté en L.

Puesto que el alfabeto Σ es también un lenguaje sobre Σ, puede aplicársele esta operación. Se verá entonces que

$$\Sigma^{+} = W(\Sigma) - \{\lambda\}$$

1.7.6. Iteración, cierre o clausura de un lenguaje

La iteración, cierre o clausura de un lenguaje L se define así:

$$L^{*} = \bigcup_{i=0}^{\infty} L^{i}$$

Es decir, el lenguaje obtenido uniendo el lenguaje L con todas sus potencias posibles, incluso L^0. Obviamente, todas las clausuras contienen la palabra vacía.

Son evidentes las siguientes identidades:

$$L^{*} = L^{+} \cup \{\lambda\}$$
$$L^{+} = LL^{*} = L^{*}L$$

Puesto que el alfabeto Σ es también un lenguaje sobre Σ, puede aplicársele esta operación. Se verá entonces que

$$\Sigma^{*} = W(\Sigma)$$

A partir de este momento, representaremos al lenguaje universal sobre el alfabeto Σ con el símbolo Σ^*.

1.7.7. Reflexión de lenguajes

Sea L un lenguaje cualquiera. Se llama *lenguaje reflejo* o *inverso* de L, y se representa con L^{-1}:

$$\{x^{-1} \mid x \in L\}$$

Es decir, al que contiene las palabras inversas a las de L.

1.7.8. Otras operaciones

Pueden definirse también para los lenguajes las operaciones *intersección* y *complementación* (con respecto al lenguaje universal). Dado que éstas son operaciones clásicas de teoría de conjuntos, no es preciso detallarlas aquí.

1.8 Ejercicios

1. Sea $=\{!\}$ y $x=!$. Definir las siguientes palabras: xx, xxx, x^3, x^8, x^0. ¿Cuáles son sus longitudes? Definir Σ^*.
2. Sea $\Sigma = \{0,1,2\}$, $x=00$, $y=1$, $z=210$. Definir las siguientes palabras: xy, xz, yz, xyz, x^3, x^2y^2, $(xy)^2$, $(zxx)^3$. ¿Cuáles son sus longitudes, cabezas y colas?
3. Sea $\Sigma = \{0,1,2\}$. Escribir seis de las cadenas más cortas de Σ^+ y de Σ^*.

1.9 Conceptos básicos sobre gramáticas

Como se ha dicho anteriormente, una gramática describe la estructura de las frases y de las palabras de un lenguaje. Aplicada a los lenguajes naturales, esta ciencia es muy antigua: los primeros trabajos aparecieron en la India durante la primera mitad del primer milenio antes de Cristo, alcanzándose el máximo apogeo con Panini, que vivió quizá entre los siglos VII y IV antes de Cristo y desarrolló la primera gramática conocida, aplicada al lenguaje sánscrito. Casi al mismo tiempo, puede que independientemente, el sofista griego Protágoras (h. 485-ca. 411 a. de J.C.) fundó una escuela gramatical, que alcanzó su máximo esplendor en el siglo II antes de Cristo.

Se llama *gramática formal* a la cuádrupla

$$G = (\Sigma_T, \Sigma_N, S, P)$$

donde Σ_T es el alfabeto de *símbolos terminales*, y Σ_N es el alfabeto de *símbolos no terminales*. Se verifica que

$$\Sigma_T \cap \Sigma_N = \Phi$$

y $\Sigma = \Sigma_T \cup \Sigma_N$.

$\Sigma_N \in \Sigma_N$ es el *axioma*, *símbolo inicial*, o *símbolo distinguido*. Finalmente, P es un conjunto finito de reglas de producción de la forma $u ::= v$, donde se verifica que:

$u \in \Sigma^+$
$u = xAy$
$x, y \in \Sigma^*$
$A \in \Sigma_N$
$v \in \Sigma^*$

Es decir, u es una palabra no vacía del lenguaje universal del alfabeto Σ que contiene al menos un símbolo no terminal y v es una palabra, posiblemente vacía, del mismo lenguaje universal.

Veamos un ejemplo de gramática:

```
Σ_T = {0,1,2,3,4,5,6,7,8,9}
Σ_N = {N,C}
S = N
P = {
    N ::= CN
    N ::= C
    C ::= 0
    C ::= 1
    C ::= 2
    C ::= 3
    C ::= 4
    C ::= 5
    C ::= 6
    C ::= 7
    C ::= 8
    C ::= 9
    }
```

1.9.1. Notación de Backus

Notación abreviada: si el conjunto de producciones contiene dos reglas de la forma

```
u ::= v
u ::= w
```

pueden representarse abreviadamente con la notación

```
u ::= v | w
```

La notación u::=v de las reglas de producción, junto con la regla de abreviación indicada, se denomina *Forma Normal de Backus*, o BNF, de las iniciales de su forma inglesa *Backus Normal Form*, o también *Backus-Naur Form*.

La gramática del ejemplo anterior puede representarse en BNF de la manera siguiente:

```
Σ_T = {0,1,2,3,4,5,6,7,8,9}
Σ_N = {N,C}
S = N
P = {
    N ::= CN | C
    C ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    }
```

1.9.2. Derivación directa

Sea Σ un alfabeto y $x::=y$ una producción sobre las palabras de ese alfabeto. Sean v y w dos palabras del mismo alfabeto ($v, w \in \Sigma^*$). Se dice que *w es derivación directa de v*, o que *v produce directamente w*, o que *w se reduce directamente a v*, si existen dos palabras $z, u \in \Sigma^*$, tales que:

$v = zxu$
$w = zyu$

Es decir, si v contiene la palabra x y, al sustituir x por y, v se transforma en w.

Indicamos esta relación con el símbolo $v \rightarrow w$.

COROLARIO: Si $x::=y$ es una producción sobre Σ, se sigue que $x \rightarrow y$.

Ejemplos:

- Sea Σ el alfabeto castellano de las letras mayúsculas, y ME ::= BA una producción sobre Σ. Es fácil ver que CAMELLO→CABALLO.
- Sea el alfabeto Σ={0,1,2,N,C}, y el conjunto de producciones

  N ::= CN
  N ::= C
  C ::= 0
  C ::= 1
  C ::= 2

  Pueden demostrarse fácilmente las siguientes derivaciones directas:

  $N \rightarrow CN \rightarrow CCN \rightarrow CCC \rightarrow 2CC \rightarrow 21C \rightarrow 210$

1.9.3. Derivación

Sea Σ un alfabeto y P un conjunto de producciones sobre las palabras de ese alfabeto. Sean v y w dos palabras del mismo alfabeto ($v, w \in \Sigma^*$). Se dice que *w es derivación de v*, o que *v produce w*, o que *w se reduce a v*, si existe una secuencia finita de palabras $u_0, u_1, \ldots, u_n$ $(n>0)$, tales que

$$v = u_0 \rightarrow u_1 \rightarrow u_2 \ldots u_{n-1} \rightarrow u_n = w$$

Indicamos esta relación con el símbolo $v \rightarrow^+ w$. La secuencia anterior se llama *derivación de longitud n*.

En el ejemplo anterior, puede verse que $N \rightarrow^+ 210$ mediante una secuencia de longitud 6.

COROLARIO: Si $v \rightarrow w$, entonces $v \rightarrow^+ w$ mediante una secuencia de longitud 1.

1.9.4. Relación de Thue

Sea Σ un alfabeto y P un conjunto de producciones sobre las palabras de ese alfabeto. Sean v y w dos palabras del mismo alfabeto. Se dice que existe una *relación de Thue* entre v y w si se verifica que $v \rightarrow+ w$ o $v = w$. Expresaremos esta relación con el símbolo $v \rightarrow* w$.

1.9.5. Formas sentenciales y sentencias

Sea una gramática $G = (\Sigma_T, \Sigma_N, S, P)$. Una palabra $x \in \Sigma^*$ se denomina *forma sentencial* de G si se verifica que $S \rightarrow* x$, es decir, si existe una relación de Thue entre el axioma de la gramática y x. Dicho de otro modo: si x=S o x deriva de S.

Ejercicio: En el ejemplo anterior, comprobar que CCN, CN2 y 123 son formas sentenciales, pero no lo es NCN.

Si una forma sentencial x cumple que $x \in \Sigma_T^*$ (es decir, está formada únicamente por símbolos terminales), se dice que x es una *sentencia* o *instrucción* generada por la gramática G.

Ejercicio: ¿Cuáles de las formas sentenciales anteriores son sentencias?

1.9.6. Lenguaje asociado a una gramática

Sea una gramática $G = (\Sigma_T, \Sigma_N, S, P)$. Se llama *lenguaje asociado a G*, o *lenguaje generado por G*, o *lenguaje descrito por G*, al conjunto $L(G) = \{x \mid S \rightarrow* x \wedge x \in \Sigma_T^*\}$. Es decir, el conjunto de todas las sentencias de G (todas las cadenas de símbolos terminales que derivan del axioma de G).

Ejemplo: El lenguaje asociado a la gramática de los ejemplos anteriores es el conjunto de todos los números naturales más el cero.

1.9.7. Frases y asideros

Sea G una gramática y v=xuy una de sus formas sentenciales. Se dice que u es una *frase* de la forma sentencial v respecto del símbolo no terminal $U \in \Sigma_N$ si

$S \rightarrow* xUy$
$U \rightarrow+ u$

Es decir, si en la derivación que transforma S en xuy se pasa por una situación intermedia en la que x e y ya han sido generados, y sólo falta transformar el símbolo no terminal U en la frase u.

Si U es una forma sentencial de G, entonces todas las frases que derivan de U serán, a su vez, formas sentenciales de G.

Una frase de v=xuy se llama *frase simple* si

$S \rightarrow^* xUy$
$U \rightarrow u$

Es decir, si la derivación de U a u se realiza en un solo paso.

Se llama *asidero* de una forma sentencial v a la frase simple situada más a la izquierda en v.

Ejercicio: En la gramática que define los números enteros positivos, demostrar que N no es una frase de 1N. Encontrar todas las frases de 1N. ¿Cuáles son frases simples? ¿Cuál es el asidero?

1.9.8. Recursividad

Una gramática G se llama *recursiva en U*, $U \in \Sigma_N$, si $U \rightarrow^+ xUy$. Si x es la palabra vacía ($x=\lambda$) se dice que la gramática es *recursiva a izquierdas.* Si $y=\lambda$, se dice que G es *recursiva a derechas* en U. Si un lenguaje es infinito, la gramática que lo representa tiene que ser recursiva.

Una regla de producción es *recursiva* si tiene la forma U ::= xUy. Se dice que es *recursiva a izquierdas* si $x=\lambda$, y *recursiva a derechas* si $y=\lambda$.

1.9.9. Ejercicios

1. Sea la gramática G = ({a,b,c,0,1}, {I}, I, {I::=a|b|c|Ia|Ib|Ic|I0|I1}). ¿Cuál es el lenguaje descrito por esta gramática? Encontrar, si es posible, derivaciones de a, ab0, a0c01, 0a, 11, aaa.
2. Construir una gramática para el lenguaje $\{ab^n a \mid n=0,1,...\}$.
3. Sea la gramática G = ({+,-,*,/,(,),i}, {E,T,F}, E, P), donde P contiene las producciones:

   E ::= T | E+T | E-T

   T ::= F | T*F | T/F

   F ::= (E) | i

   Obtener derivaciones de las siguientes sentencias: i, (i), i*i, i*i+i, i*(i+i).

1.10 Tipos de gramáticas

Chomsky clasificó las gramáticas en cuatro grandes grupos (G_0, G_1, G_2, G_3), cada uno de los cuales incluye a los siguientes, de acuerdo con el siguiente esquema:

$G_3 \subset G_2 \subset G_1 \subset G_0$

1.10.1. Gramáticas de tipo 0

Son las gramáticas más generales. Las reglas de producción tienen la forma u::=v, donde $u \in \Sigma^+$, $v \in \Sigma^*$, u=xAy, $x, y \in \Sigma^*$, $A \in \Sigma_N$, sin ninguna restricción adicional. Los lenguajes representados por estas gramáticas se llaman *lenguajes sin restricciones*.

Puede demostrarse que todo lenguaje representado por una gramática de tipo 0 de Chomsky puede describirse también por una gramática perteneciente a un grupo un poco más restringido (*gramáticas de estructura de frases*), cuyas producciones tienen la forma xAy::=xvy, donde $x, y, \in \Sigma v^*$, $A \in \Sigma_N$. Puesto que v puede ser igual a λ, se sigue que algunas de las reglas de estas gramáticas pueden tener una parte derecha más corta que su parte izquierda. Si tal ocurre, se dice que la regla es *compresora*. Una gramática que contenga al menos una regla compresora se llama *gramática compresora*. En las gramáticas compresoras, las derivaciones pueden ser decrecientes, pues la longitud de las palabras puede disminuir en cada uno de los pasos de la derivación.

Veamos un ejemplo: Sea la gramática G = ({a,b}, {A,B,C}, A, P), donde P contiene las producciones

```
A ::= aABC | abC
CB ::= BC
bB ::= bb
bC ::= b
```

Esta es una gramática de tipo 0, pero no de estructura de frases, pues la regla CB::=BC no cumple las condiciones requeridas. Sin embargo, esta regla podría sustituirse por las cuatro siguientes:

```
CB ::= XB
XB ::= XY
XY ::= BY
BY ::= BC
```

Con este cambio, se pueden obtener las mismas derivaciones en más pasos, pero ahora sí se cumplen las condiciones para que la gramática sea de estructura de frases. Por tanto, el lenguaje descrito por la primera gramática es el mismo que el de la segunda. Obsérvese que esta gramática de estructura de frases equivalente a la gramática de tipo 0 tiene tres reglas de producción más y dos símbolos adicionales (X, Y) en el alfabeto de símbolos no terminales.

Esta es la derivación de la sentencia aaabbb (a^3b^3) en la gramática de tipo 0 dada más arriba:

$$A \rightarrow aABC \rightarrow aaABCBC \rightarrow aaabCBCBC \rightarrow aaabBCBC \rightarrow$$
$$\rightarrow aaabbCBC \rightarrow aaabbBC \rightarrow aaabbbC \rightarrow aaabbb$$

Obsérvese que esta gramática es compresora, por contener la regla bC::=b. Puede comprobarse que el lenguaje representado por esta gramática es $\{a^n b^n \mid n=1,2,...\}$.

1.10.2. Gramáticas de tipo 1

Las reglas de producción de estas gramáticas tienen la forma $xAy::=xvy$, donde $x, y \in \Sigma^*$, $v \in \Sigma^+$, $A \in \Sigma_N$, que se interpreta así: A puede transformarse en v cuando se encuentra entre el contexto izquierdo x y el contexto derecho y.

Como v no puede ser igual a λ, se sigue que estas gramáticas no pueden contener reglas compresoras. Se admite una excepción en la regla $S::=\lambda$, que sí puede pertenecer al conjunto de producciones de una gramática de tipo 1. Aparte de esta regla, que lleva a la derivación trivial $S \rightarrow \lambda$, ninguna derivación obtenida por aplicación de las reglas de las gramáticas de tipo 1 puede ser decreciente. Es decir: si $u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_n$ es una derivación correcta, se verifica que $|u_1| \leq |u_2| \leq \ldots \leq |u_n|$

En consecuencia, en una gramática tipo 1, la palabra vacía λ pertenece al lenguaje representado por la gramática si y sólo si la regla $S::=\lambda$ pertenece al conjunto de producciones de la gramática.

Los lenguajes representados por las gramáticas tipo 1 se llaman *lenguajes dependientes del contexto* (*context-sensitive*, en inglés).

Es evidente que toda gramática de tipo 1 es también una gramática de tipo 0. En consecuencia, todo lenguaje dependiente del contexto es también un lenguaje sin restricciones.

1.10.3. Gramáticas de tipo 2

Las reglas de producción de estas gramáticas tienen la forma $A::=v$, donde $v \in \Sigma^*$, $A \in \Sigma_N$. En particular, v puede ser igual a λ. Sin embargo, para toda gramática de tipo 2 que represente un lenguaje L, existe otra equivalente, desprovista de reglas de la forma $A::=\lambda$, que representa al lenguaje $L-\{\lambda\}$. Si ahora se añade a esta segunda gramática la regla $S::=\lambda$, el lenguaje representado volverá a ser L. Por lo tanto, las gramáticas de tipo 2 pueden definirse también de esta forma más restringida: las reglas de producción tendrán la forma $A::=v$, donde $v \in \Sigma^+$, $A \in \Sigma_N$. Además, pueden contener la regla $S::=\lambda$.

Los lenguajes descritos por gramáticas de tipo 2 se llaman *lenguajes independientes del contexto* (*context-free*, en inglés), pues la conversión de A en v puede aplicarse cualquiera que sea el contexto donde se encuentre A. La sintaxis de casi todos los lenguajes humanos y todos los de programación de computadoras puede describirse mediante gramáticas de este tipo.

Es evidente que toda gramática de tipo 2 cumple también los requisitos para ser una gramática de tipo 1. En consecuencia, todo lenguaje independiente del contexto pertenecerá también a la clase de los lenguajes dependientes del contexto.

Veamos un ejemplo: Sea la gramática $G = (\{a,b\}, \{S\}, S, \{S::=aSb|ab\})$. Se trata, evidentemente, de una gramática de tipo 2. Esta es la derivación de la sentencia $aaabbb$ (a^3b^3):

$$S \rightarrow aSb \rightarrow aaSbb \rightarrow aaabbb$$

Puede comprobarse que el lenguaje representado por esta gramática es $\{a^n b^n \mid n=1,2,...\}$, es decir, el mismo que se vio anteriormente con un ejemplo de gramática de tipo 0. En general, un mismo lenguaje puede describirse mediante muchas gramáticas diferentes, no siempre del mismo tipo. En cambio, una gramática determinada describe siempre un lenguaje único.

1.10.4. Gramáticas de tipo 3

Estas gramáticas se clasifican en los dos grupos siguientes:

1. Gramáticas *lineales por la izquierda*, cuyas reglas de producción pueden tener una de las formas siguientes:

   A ::= a
   A ::= Va
   S ::= λ

2. Gramáticas *lineales por la derecha*, cuyas reglas de producción pueden tomar una de las formas siguientes:

   A ::= a
   A ::= aV
   S ::= λ

En ambos casos, $a \in \Sigma_T$, $A, V \in \Sigma_N$ y S es el axioma de la gramática.

Los lenguajes que pueden representarse mediante gramáticas de tipo 3 se llaman *lenguajes regulares*. Es fácil ver que toda gramática de tipo 3 cumple también los requisitos para ser gramática de tipo 2. Por lo tanto, todo lenguaje regular pertenecerá también a la clase de los lenguajes independientes del contexto.

Veamos un ejemplo: Sea la gramática G = ({0,1}, {A,B}, A, P), donde P contiene las producciones

A ::= 1B | 1
B ::= 0A

Se trata de una gramática lineal por la derecha. Es fácil ver que el lenguaje descrito por la gramática es

$L_2 = \{1, 101, 10101, ...\} = \{1(01)^n \mid n=0,1,...\}$

1.10.5. Gramáticas equivalentes

Se dice que dos gramáticas son *equivalentes* cuando describen el mismo lenguaje.

1.10.6. Ejercicios

1. Sean las gramáticas siguientes:

   - G_1 = ({c}, {S,A}, S, {S::=λ|A, A::=AA|c})
   - G_2 = ({c,d}, {S,A}, S, {S::=λ|A, A::=cAd|cd})
   - G_3 = ({c,d}, {S,A}, S, {S::=λ|A, A::=Ad|cA|c|d})
   - G_4 = ({c,d}, {S,A,B}, S, {S::=cA, A::=d|cA|Bd, B::=d|Bd})
   - G_5 = ({c}, {S,A}, S, {S::=λ|A, A::=AcA|c})
   - G_6 = ({0,c}, {S,A,B}, S, {S::=AcA, A::=0, Ac::=AAcA|ABc|AcB, B::=A|AB})

   Definir el lenguaje descrito por cada una de ellas, así como las relaciones de inclusión entre los seis lenguajes, si las hay.

   Encontrar una gramática de tipo 2 equivalente a G_6.

2. Sean los lenguajes siguientes:

   - L_1 = {0^m1^n | m n 0}
   - L_2 = {$0^k1^m0^n$ | n=k+m}
   - L_3 = {wcw | w{0,1}*}
   - L_4 = {wcw^{-1} | w{0,1}*}
   - L_5 = {10^n | n=0,1,2,...}

   Construir una gramática que describa cada uno de los lenguajes anteriores.

3. Se llama *palíndromo* a toda palabra x que cumpla x=x^{-1}. Se llama *lenguaje palindrómico* a todo lenguaje cuyas palabras sean todas palíndromos. Sean las gramáticas

   - G_1 = ({a,b}, {S}, S, {S::=aSa|aSb|bSb|bSa|aa|bb})
   - G_2 = ({a,b}, {S}, S, {S::=aS|Sa|bS|Sb|a|b})

   ¿Alguna de ellas describe un lenguaje palindrómico?

4. Sea L un lenguaje palindrómico. ¿Es L^{-1} un lenguaje palindrómico? ¿Lo es L L^{-1}?

5. Sea x un palíndromo. ¿Es L=x* un lenguaje palindrómico?

1.11 Árboles de derivación

Toda derivación de una gramática de tipo 1, 2 o 3 puede representarse mediante un árbol, que se construye de la siguiente manera:

1. La raíz del árbol se denota por el axioma de la gramática.

2. Una derivación directa se representa por un conjunto de ramas que salen de un nodo. Al aplicar una regla, un símbolo de la parte izquierda queda sustituido por la palabra x de la

parte derecha. Por cada uno de los símbolos de x se dibuja una rama que parte del nodo dado y termina en otro, denotado por dicho símbolo.

Sean dos símbolos A y B en la palabra x. Si A está a la izquierda de B en x, entonces la rama que termina en A se dibujará a la izquierda de la rama que termina en B.

Para cada rama, el nodo de partida se llama *padre* del nodo final. Este último es el *hijo* del primero. Dos nodos hijos del mismo padre se llaman *hermanos*. Un nodo es *ascendiente* de otro si es su padre o es ascendiente de su padre. Un nodo es *descendiente* de otro si es su hijo o es descendiente de uno de sus hijos.

A lo largo del proceso de construcción del árbol, los nodos finales de cada paso sucesivo, leídos de izquierda a derecha, dan la forma sentencial obtenida por la derivación representada por el árbol.

Se llama *rama terminal* aquella que se dirige hacia un nodo denotado por un símbolo terminal de la gramática. Este nodo se denomina *hoja* o *nodo terminal* del árbol. El conjunto de las hojas del árbol, leído de izquierda a derecha, da la sentencia generada por la derivación.

Ejemplo: Sea la gramática G = ({0,1,2,3,4,5,6,7,8,9}, {N,C}, N, {N::=C|CN, C::=0|1|2|3|4|5|6|7|8|9}). Sea la derivación:

$$N \rightarrow CN \rightarrow CCN \rightarrow CCC \rightarrow 2CC \rightarrow 23C \rightarrow 235$$

La Figura 1.2 representa el árbol correspondiente a esta derivación.

A veces, un árbol puede representar varias derivaciones diferentes. Por ejemplo, el árbol de la Figura 1.2 representa también, entre otras, a las siguientes derivaciones:

$$N \rightarrow CN \rightarrow CCN \rightarrow CCC \rightarrow CC5 \rightarrow 2C5 \rightarrow 235$$
$$N \rightarrow CN \rightarrow 2N \rightarrow 2CN \rightarrow 23N \rightarrow 23C \rightarrow 235$$

Sea $S \rightarrow w1 \rightarrow w2 \ldots x$ una derivación de la palabra x en la gramática G. Se dice que ésta es *la derivación más a la izquierda* de x en G, si en cada uno de los pasos o derivaciones directas se ha aplicado una producción cuya parte izquierda modifica el símbolo no terminal situado

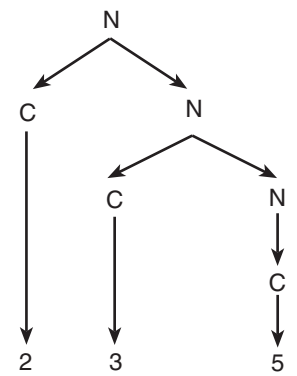


Figura 1.2. Árbol equivalente a una derivación.

más a la izquierda en la forma sentencial anterior. Dicho de otro modo: en cada derivación directa $u \rightarrow v$ se ha generado el asidero de v.

En las derivaciones anteriores, correspondientes al árbol de la Figura 1.2, la derivación más a la izquierda es la última.

1.11.1. Subárbol

Dado un árbol A correspondiente a una derivación, se llama *subárbol* de A al árbol cuya raíz es un nodo de A, cuyos nodos son todos los descendientes de la raíz del subárbol en A, y cuyas ramas son todas las que unen dichos nodos entre sí en A.

Los nodos terminales de un subárbol, leídos de izquierda a derecha, forman una frase respecto a la raíz del subárbol. Si todos los nodos terminales del subárbol son hijos de la raíz, entonces la frase es simple.

1.11.2. Ambigüedad

A veces, una sentencia puede obtenerse en una gramática por medio de dos o más árboles de derivación diferentes. En este caso, se dice que la sentencia es *ambigua*. Una gramática es ambigua si contiene al menos una sentencia ambigua.

Aunque la gramática sea ambigua, es posible que el lenguaje descrito por ella no lo sea. Puesto que a un mismo lenguaje pueden corresponderle numerosas gramáticas, que una de éstas sea ambigua no implica que lo sean las demás. Sin embargo, existen lenguajes para los que es imposible encontrar gramáticas no ambiguas que los describan. En tal caso, se dice que estos lenguajes son *inherentemente ambiguos*.

Ejemplo: Sea la gramática G_1 = ({i,+,*,(,)}, {E}, E, {E::=E+E|E*E|(E)|i}) y la sentencia i+i*i. La Figura 1.3 representa los dos árboles que generan esta sentencia en dicha gramática.

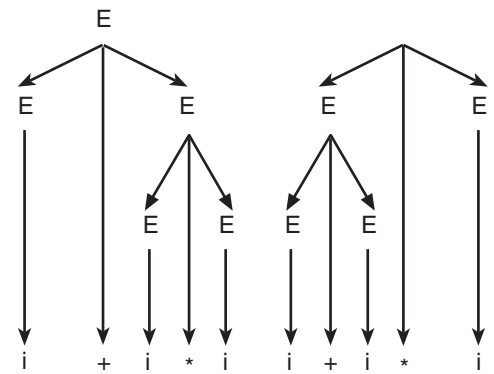


Figura 1.3. Dos árboles que producen la sentencia i+i*i en G_1.

Sin embargo, la gramática G_2 = ({i,+,*,(,)}, {E,T,F}, E, {E::=T|E+T, T::=F|T*F, F::=(E)|i}) es equivalente a la anterior (genera el mismo lenguaje) pero no es ambigua. En efecto, ahora existe un solo árbol de derivación de la sentencia i+i*i: el de la Figura 1.4.

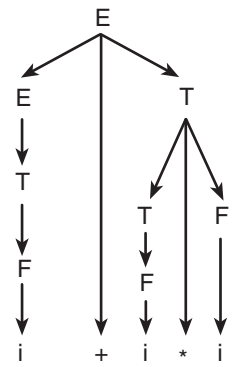


Figura 1.4. Árbol de la sentencia i+i*i en G_2.

La ambigüedad puede definirse también así:

Una gramática es ambigua si existe en ella una sentencia que pueda obtenerse a partir del axioma mediante dos derivaciones más a la izquierda distintas.

1.11.3. Ejercicios

1. En la gramática G_2 del apartado anterior, dibujar árboles sintácticos para las derivaciones siguientes:
   - E → T → F → i
   - E → T → F → (E) → (T) → (F) → (i)
   - E → +T → +F → +i
2. En la misma gramática G_2, demostrar que las sentencias i+i*i y i*i*i no son ambiguas. ¿Qué operador tiene precedencia en cada una de esas dos sentencias?
3. Demostrar que la siguiente gramática es ambigua, construyendo dos árboles para cada una de las sentencias i+i*i y i+i+i: ({i,+,-,*,/,(,)}, {E,O}, E, {E::=i|(E)|EOE, O::=+|-|*|/}).

1.12 Gramáticas limpias y bien formadas

Una gramática se llama *reducida* si no contiene símbolos inaccesibles ni reglas superfluas. Se llama *limpia* si tampoco contiene reglas innecesarias.

Se llama *gramática bien formada* a una gramática independiente del contexto que sea limpia y que carezca de símbolos y reglas no generativos y de reglas de redenominación.

1.12.1. Reglas innecesarias

En una gramática, las reglas de la forma `U::=U` son innecesarias y la hacen ambigua. A partir de ahora se supondrá que una gramática no tiene tales reglas o, si las tiene, serán eliminadas.

1.12.2. Símbolos inaccesibles

Supóngase que una gramática contiene una regla de la forma `U::=x`, donde `U` es un símbolo no terminal, distinto del axioma, que no aparece en la parte derecha de ninguna otra regla. Se dice que `U` es un *símbolo inaccesible* desde el axioma.

Si un símbolo `V` es accesible desde el axioma `S`, debe cumplir que

$S \rightarrow^* xVy,\ x,y \in \Sigma^*$

Para eliminar los símbolos inaccesibles, se hace una lista de todos los símbolos de la gramática y se marca el axioma `S`. A continuación, se marcan todos los símbolos que aparezcan en la parte derecha de cualquier regla cuya parte izquierda sea un símbolo marcado. El proceso continúa hasta que no se marque ningún símbolo nuevo. Los símbolos que se queden sin marcar, son inaccesibles.

1.12.3. Reglas superfluas

El concepto de *regla superflua* se explicará con un ejemplo. Sea la gramática `G = ({e,f}, {S,A,B,C,D}, S, {S::=Be, A::=Ae|e, B::=Ce|Af, C::=Cf, D::=f})`. La regla `C::=Cf` es superflua, pues a partir de `C` no se podrá llegar nunca a una cadena que sólo contenga símbolos terminales. Para no ser superfluo, un símbolo no terminal `U` debe cumplir:

$U \rightarrow^+ t,\ t\Sigma_T^*$

El siguiente algoritmo elimina los símbolos superfluos:

1. Marcar los símbolos no terminales para los que exista una regla `U::=x`, donde `x` sea una cadena de símbolos terminales, o de no terminales marcados.

2. Si todos los símbolos no terminales han quedado marcados, no existen símbolos superfluos en la gramática. Fin del proceso.
3. Si la última vez que se pasó por el paso 1 se marcó algún símbolo no terminal, volver al paso 1.
4. Si se llega a este punto, todos los símbolos no terminales no marcados son superfluos.

1.12.4. Eliminación de símbolos no generativos

Sea la gramática independiente del contexto G=(Σ_T, Σ_N, S, P). Para cada símbolo A$\in\Sigma_N$ se construye la gramática G(A)=(Σ_T, Σ_N, A, P). Si L(G(A)) es vacío, se dice que A es un símbolo no generativo. Entonces se puede suprimir A en Σ_N, así como todas las reglas que contengan A en P, obteniendo otra gramática más sencilla, que representa el mismo lenguaje.

1.12.5. Eliminación de reglas no generativas

Se llaman reglas no generativas las que tienen la forma A::=λ. Si el lenguaje representado por una gramática no contiene la palabra vacía, es posible eliminarlas todas. En caso contrario, se pueden eliminar todas menos una: la regla S::=λ, donde S es el axioma de la gramática. Para compensar su eliminación, por cada símbolo A de Σ_N (A distinto de S) tal que A$\rightarrow$*λ en G, y por cada regla de la forma B::=xAy, añadiremos una regla de la forma B::=xy, excepto en el caso de que x=y=λ. Es fácil demostrar que las dos gramáticas (la inicial y la corregida) representan el mismo lenguaje.

1.12.6. Eliminación de reglas de redenominación

Se llama *regla de redenominación* a toda regla de la forma A::=B. Para compensar su eliminación, basta añadir el siguiente conjunto de reglas:

Para cada símbolo A de Σ_N tal que A$\rightarrow$*B en G, y para cada regla de la forma B::=x, donde x no es un símbolo no terminal, añadiremos una regla de la forma A::=x.

Es fácil demostrar que las dos gramáticas (la inicial y la corregida) representan el mismo lenguaje.

1.12.7. Ejemplo

Sea G={{0,1},{S,A,B,C},S,P), donde P es el siguiente conjunto de producciones:

```
S ::= AB | 0S1 | A | C
A ::= 0AB | λ
B ::= B1 | λ
```

Es evidente que C es un símbolo no generativo, por lo que la regla S::=C es superflua y podemos eliminarla, quedando:

```
S ::= AB | 0S1 | A
A ::= 0AB | λ
B ::= B1 | λ
```

Eliminemos ahora las reglas de la forma X::= λ:

```
S ::= AB | 0S1 | A | B | λ
A ::= 0AB | 0B | 0A | 0
B ::= B1 | 1
```

Ahora eliminamos las reglas de redenominación S::=A|B:

```
S ::= AB | 0S1 | 0AB | 0B | 0A | 0 | B1 | 1 | λ
A ::= 0AB | 0B | 0A | 0
B ::= B1 | 1
```

Hemos obtenido una gramática bien formada.

1.12.8. Ejercicio

1. Limpiar la gramática G = ({i,+}, {Z,E,F,G,P,Q,S,T}, Z, {Z::=E+T, E::=E|S+F|T, F::=F|FP|P, P::=G, G::=G|GG|F, T::=T*i|i, Q::=E|E+F|T|S, S::=i})

1.13 Lenguajes naturales y artificiales

La teoría de gramáticas transformacionales de Chomsky se aplica por igual a los lenguajes naturales (los que hablamos los seres humanos) y los lenguajes de programación de computadoras. Con muy pocas excepciones, todos estos lenguajes tienen una sintaxis que se puede expresar con gramáticas del tipo 2 de Chomsky; es decir, se trata de lenguajes independientes del contexto. Las dos excepciones conocidas son el alemán suizo y el bambara.

El alemán suizo, para expresar una frase parecida a ésta:

Juan vio a Luis dejar que María ayudara a Pedro a hacer que Felipe trabajara

admite una construcción con sintaxis parecida a la siguiente:

Juan Luis María Pedro Felipe vio dejar ayudar hacer trabajar

Algunos de los verbos exigen acusativo, otros dativo. Supongamos que los que exigen acusativo estuviesen todos delante, y que después viniesen los que exigen dativo. Tendríamos una construcción sintáctica de la forma:

$A^nB^mC^nD^m$

donde A = frase nominal en acusativo, B = frase nominal en dativo, C = verbo que exige acusativo, D = verbo que exige dativo. Esta construcción hace que la sintaxis del lenguaje no sea independiente del contexto (es fácil demostrarlo mediante técnicas como el *lema de bombeo* [5]).

El bambara es una lengua africana que, para formar el plural de una palabra o de una frase, simplemente la repite. Por lo tanto, en esta lengua es posible construir frases con sintaxis parecida a las siguientes:

- Para decir *cazador de perros* diríamos *cazador de perro perro*.
- Para decir *cazadores de perros* diríamos *cazador de perro perro cazador de perro perro*.

Y así sucesivamente. Obsérvese que esto hace posible generar frases con una construcción sintáctica muy parecida a la que hace que el alemán suizo sea dependiente del contexto:

$A^nB^mA^nB^m$

donde A sería *cazador* y B *perro*.

1.13.1. Lenguajes de programación de computadoras

A lo largo de la historia de la Informática, han surgido varias generaciones de lenguajes artificiales, progresivamente más complejas:

Primera generación: lenguajes de la máquina. Los programas se escriben en código binario. Por ejemplo:

```
00000101101000000000000000
```

Segunda generación: lenguajes simbólicos. Cada instrucción de la máquina se representa mediante símbolos. Por ejemplo:

```
ADD AX,P1
```

Tercera generación: lenguajes de alto nivel. Una sola instrucción de este tipo representa usualmente varias instrucciones de la máquina. Por ejemplo:

```
P1 = P2 + P3;
```

Son lenguajes de alto nivel, FORTRAN, COBOL, LISP, BASIC, C, C++, APL, PASCAL, SMALLTALK, JAVA, ADA, PROLOG, y otros muchos.

1.13.2. Procesadores de lenguaje

Los programas escritos en lenguaje de la máquina son los únicos que se pueden ejecutar directamente en una computadora. Los restantes hay que traducirlos.

- Los lenguajes simbólicos se traducen mediante programas llamados *ensambladores*, que convierten cada instrucción simbólica en la instrucción máquina equivalente. Estos programas suelen ser relativamente sencillos y no se van a considerar aquí.
- Los programas escritos en lenguajes de alto nivel se traducen mediante programas llamados, en general, *traductores* o *procesadores de lenguaje*. Existen tres tipos de estos traductores:

  — **Compilador:** analiza un programa escrito en un lenguaje de alto nivel (programa fuente) y, si es correcto, genera un código equivalente (programa objeto) escrito en otro lenguaje, que puede ser de primera generación (de la máquina), de segunda generación (simbólico) o de tercera generación. El programa objeto puede guardarse y ejecutarse tantas veces como se quiera, sin necesidad de traducirlo de nuevo.

  Un compilador se representa con el símbolo de la Figura 1.5, donde `A` es el *lenguaje fuente*, `B` es el *lenguaje objeto* y `C` es el lenguaje en que está escrito el propio compilador, que al ser un programa que debe ejecutarse en una computadora, también habrá tenido que ser escrito en algún lenguaje, no necesariamente el mismo que el lenguaje fuente o el lenguaje objeto.

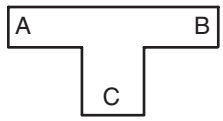


Figura 1.5. Representación simbólica de un compilador.

Entre los lenguajes que usualmente se compilan podemos citar FORTRAN, COBOL, C, C++, PASCAL y ADA.

— **Intérprete:** analiza un programa escrito en un lenguaje de alto nivel y, si es correcto, lo ejecuta directamente en el lenguaje de la máquina en que se está ejecutando el intérprete. Cada vez que se desea ejecutar el programa, es preciso interpretarlo de nuevo.

Un intérprete se representa con el símbolo de la Figura 1.6, donde `A` es el *lenguaje fuente* y `C` es el lenguaje en que está escrito el propio intérprete, que también debe ejecutarse y habrá sido escrito en algún lenguaje, usualmente distinto del lenguaje fuente.

Figura 1.6. Representación simbólica de un intérprete.

Entre los lenguajes que usualmente se interpretan citaremos LISP, APL, SMALLTALK, JAVA y PROLOG. De algún lenguaje, como BASIC, existen a la vez compiladores e intérpretes.

— **Compilador-intérprete:** traduce el programa fuente a un formato o lenguaje intermedio, que después se interpreta.

Un compilador-intérprete se representa con los símbolos de la Figura 1.7, donde `A` es el *lenguaje fuente*, `B` es el *lenguaje intermedio*, `C` es el lenguaje en que está escrito el compilador y `D` es el lenguaje en que está escrito el intérprete, no necesariamente el mismo que `A`, `B` o `C`.

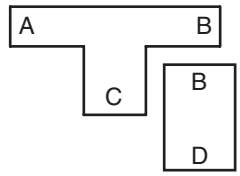


Figura 1.7. Representación simbólica de un compilador-intérprete.

JAVA es un ejemplo típico de lenguaje traducido mediante un compilador-intérprete, pues primero se compila a BYTECODE, y posteriormente éste se interpreta mediante una máquina virtual de JAVA, que no es otra cosa que un intérprete de BYTECODE. En este caso, `A` es JAVA, `B` es BYTECODE, `C` es el lenguaje en que esté escrito el compilador de JAVA a BYTECODE, y `D` es el lenguaje en que esté escrita la máquina virtual de JAVA.

Los compiladores generan código más rápido que los intérpretes, pues éstos tienen que analizar el código cada vez que lo ejecutan. Sin embargo, los intérpretes proporcionan ciertas ventajas, que en algunos compensan dicha pérdida de eficiencia, como la protección contra virus, la independencia de la máquina y la posibilidad de ejecutar instrucciones de alto nivel generadas durante la ejecución del programa. Los compiladores-intérpretes tratan de obtener estas ventajas con una pérdida menor de tiempo de ejecución.

1.13.3. Partes de un procesador de lenguaje

Un compilador se compone de las siguientes partes (*véase* la Figura 1.8):

- **Tabla de símbolos o identificadores.**
- **Analizador morfológico,** también llamado analizador lexical, preprocesador o *scanner*, en inglés. Realiza la primera fase de la compilación. Convierte el programa que va a ser compilado en una serie de unidades más complejas (unidades sintácticas) que desempeñan el papel de símbolos terminales para el analizador sintáctico. Esto puede hacerse generando dichas

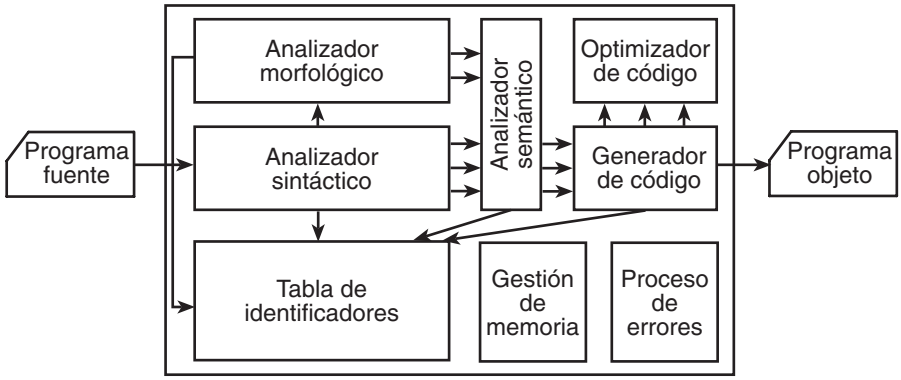


Figura 1.8. Estructura de un compilador.

unidades de una en una o línea a línea. Elimina espacios en blanco y comentarios, y detecta errores morfológicos. Usualmente se implementa mediante un autómata finito determinista.

- **Analizador sintáctico**, también llamado *parser*, en inglés. Es el elemento fundamental del procesador, pues lleva el control del proceso e invoca como subrutinas a los restantes elementos del compilador. Realiza el resto de la reducción al axioma de la gramática para comprobar que la instrucción es correcta. Usualmente se implementa mediante un autómata a pila o una construcción equivalente.
- **Analizador semántico**. Comprueba la corrección semántica de la instrucción, por ejemplo, la compatibilidad del tipo de las variables en una expresión.
- **Generador de código**. Traduce el programa fuente al lenguaje objeto utilizando toda la información proporcionada por las restantes partes del compilador.
- **Optimizador de código**. Mejora la eficiencia del programa objeto en ocupación de memoria o en tiempo de ejecución.
- **Gestión de memoria**, tanto en el procesador de lenguaje como en el programa objeto.
- **Recuperación de errores detectados**.

En los compiladores de un solo paso o etapa, suele fundirse el analizador semántico con el generador de código. Otros compiladores pueden ejecutarse en varios pasos. Por ejemplo, en el primero se puede generar un código intermedio en el que ya se han realizado los análisis morfológico, sintáctico y semántico. El segundo paso es otro programa que parte de ese código intermedio y, a partir de él, genera el código objeto. Todavía es posible que un compilador se ejecute en tres pasos, dedicándose el tercero a la optimización del código generado por la segunda fase.

En un intérprete no existen las fases de generación y optimización de código, que se sustituyen por una fase de ejecución de código.

1.13.4. Nota sobre sintaxis y semántica

Aunque la distinción entre sintaxis y semántica, aplicada a los lenguajes humanos, es muy antigua, el estudio de los lenguajes de computadora ha hecho pensar que, en el fondo, se trata de una

distinción artificial. Dado que un lenguaje de computadora permite escribir programas capaces de resolver (en principio) cualquier problema computable, es obvio que el lenguaje completo (sintaxis + semántica) se encuentra al nivel de una máquina de Turing o de una gramática de tipo 0 de Chomsky. Sin embargo, el tratamiento formal de este tipo de gramáticas es complicado: su diseño resulta oscuro y su análisis muy costoso. Estas dificultades desaconsejan su uso en el diseño de compiladores e intérpretes. Para simplificar, se ha optado por separar todas aquellas componentes del lenguaje que se pueden tratar mediante gramáticas independientes del contexto (o de tipo 2 de Chomsky), que vienen a coincidir con lo que, en los lenguajes naturales, se venía llamando *sintaxis*. Su diseño resulta más natural al ingeniero informático y existen muchos algoritmos eficientes para su análisis. La máquina abstracta necesaria para tratar estos lenguajes es el autómata a pila.

Por otra parte, podríamos llamar *semántica* del lenguaje de programación todo aquello que habría que añadir a la parte independiente del contexto del lenguaje (la sintaxis) para hacerla computacionalmente completa. Por ejemplo, con reglas independientes del contexto, no es posible expresar la condición de que un identificador debe ser declarado antes de su uso, ni comprobar la coincidencia en número, tipo y orden entre los parámetros que se pasan en una llamada a una función y los de su declaración. Para describir formalmente la semántica de los lenguajes de programación, se han propuesto diferentes modelos[2], la mayoría de los cuales parte de una gramática independiente del contexto que describe la sintaxis y la extiende con elementos capaces de expresar la semántica. Para la implementación de estos modelos, los compiladores suelen utilizar un autómata a pila, un diccionario (o tabla de símbolos) y un conjunto de algoritmos. El autómata analiza los aspectos independientes del contexto (la sintaxis), mientras que las restantes componentes resuelven los aspectos dependientes (la semántica).

1.14 Resumen

Este capítulo ha revisado la historia de la Informática, señalando los paralelos sorprendentes que existen entre disciplinas tan aparentemente distintas como la Computabilidad, la Teoría de autómatas y máquinas secuenciales, y la Teoría de gramáticas transformacionales.

Se recuerdan y resumen las definiciones de alfabeto, palabra, lenguaje y gramática; las operaciones con palabras y lenguajes; los conceptos de derivación, forma sentencial, sentencia, frase y asidero; la idea de recursividad; los diversos tipos de gramáticas; la representación de las derivaciones por medio de árboles sintácticos; el concepto de ambigüedad sintáctica y la forma de obtener gramáticas limpias.

Finalmente, la última parte del capítulo clasifica los lenguajes, tanto naturales como artificiales o de programación, introduce el concepto de procesador de lenguaje y sus diversos tipos (compiladores, intérpretes y compiladores-intérpretes), y da paso al resto del libro, especificando cuáles son las partes en que se divide usualmente un compilador o un intérprete.

---

[2] En este libro sólo será objeto de estudio el modelo de especificación formal de la semántica de los lenguajes de programación basado en las gramáticas de atributos [6, 7].

1.15 Bibliografía

[1] Gödel, K. (1931): «Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme», I. *Monatshefte für Mathematik und Physik*, 38, pp. 173-198.

[2] Shannon, C. (1938): «A symbolic analysis of relay and switching circuits», *Transactions American Institute of Electrical Engineers*, vol. 57, pp. 713-723.

[3] Chomsky, N. (1956): «Three models for the description of language», *IRE Transactions on Information Theory*, 2, pp. 113-124.

[4] Chomsky, N. (1959): «On certain formal properties of grammars», *Information and Control*, 1, pp. 91-112.

[5] Alfonseca, M.; Sancho, J., y Martínez Orga, M. (1997): *Teoría de lenguajes, gramáticas y autómatas.* Madrid. Promo-Soft. Publicaciones R.A.E.C.

[6] Knuth, D. E. (1971): «Semantics of context-free languages», *Mathematical Systems Theory*, 2(2), pp.127-145, junio 1968. Corregido en *Mathematical Systems Theory*, 5(1), pp. 95-96, marzo 1971.

[7] Knuth, D. E. (1990): «The genesis of attribute grammars», en Pierre Deransart & Martin Jourdan, editors, *Attribute grammars and their applications* (WAGA), vol. 461 de *Lecture Notes in Computer Science*, pp. 1-12, Springer-Verlag, New York-Heidelberg-Berlín, septiembre 1990.

Capítulo 2

Tabla de símbolos

La tabla de símbolos es la componente del compilador que se encarga de todos los aspectos dependientes del contexto relacionados con las restricciones impuestas a los *nombres* que puedan aparecer en los programas (nombres de variables, constantes, funciones, palabras reservadas…). Estas restricciones obligan a llevar la cuenta, durante todo el proceso de la compilación, de los nombres utilizados (junto con toda la información relevante que se deduzca de la definición del lenguaje de programación), para poder realizar las comprobaciones e imponer las restricciones necesarias.

Por otro lado, una preocupación muy importante en el diseño de algoritmos para la solución de problemas computables es obtener el mejor rendimiento posible. Para ello, es esencial la elección correcta de las estructuras de datos. El tiempo que necesitan los algoritmos para procesar sus entradas suele depender del tamaño de éstas y difiere de unas estructuras de datos a otras. Los párrafos siguientes contienen reflexiones que justifican la elección de las estructuras de datos y algoritmos más utilizados para las tablas de símbolos de los compiladores.

2.1 Complejidad temporal de los algoritmos de búsqueda

En informática, la complejidad de los algoritmos se puede estudiar estimando la dependencia entre el tiempo que necesitan para procesar su entrada y el tamaño de ésta. El mejor rendimiento se obtiene si ambos son independientes: el tiempo es constante. En orden decreciente de eficiencia, otros algoritmos presentan dependencia logarítmica, polinómica (lineal, cuadrática, etc.) o exponencial. Para clasificar un algoritmo de esta forma, se elige una de sus instrucciones y se estima el tiempo empleado en ejecutarla mediante el número de veces que el algoritmo tiene que ejecutar dicha instrucción, se calcula el tiempo en función del tamaño de la entrada y se estudia su *orden* cuando la entrada se hace arbitrariamente grande.

A lo largo de este capítulo se usará `n` para representar el tamaño de la entrada. Para la estimación de los órdenes de eficiencia, los valores concretos de las constantes que aparecen en las funciones no son relevantes, por lo que la dependencia constante se representa como `1`, la logarítmica como `log(n)`, la lineal como `n`, la cuadrática como n^2 y la exponencial como e^n. Resulta útil considerar el peor tiempo posible (el tiempo utilizado en tratar la entrada que más dificultades plantea) y el tiempo medio (calculado sobre todas las entradas o sobre una muestra suficientemente representativa de ellas).

Buscar un dato en una estructura implica compararlo con alguno de los datos contenidos en ella. La instrucción seleccionada para medir el rendimiento de los algoritmos de búsqueda suele ser esta comparación, que recibe el nombre de *comparación de claves*.

Una explicación más detallada de esta materia queda fuera del objetivo de este libro. El lector interesado puede consultar [1, 2].

2.1.1. Búsqueda lineal

La búsqueda lineal supone que los datos entre los que puede estar el buscado se guardan en una lista o vector, no necesariamente ordenados.

Este algoritmo (*véase* la Figura 2.1) recorre la estructura desde la primera posición, comparando (comparación de clave) cada uno de los elementos que encuentra con el buscado. Termina por una de las dos situaciones siguientes: o se llega al final de la estructura o se encuentra el dato buscado. En la primera situación, el dato no se ha encontrado y el algoritmo no termina con éxito.

```
ind BúsquedaLineal (tabla T, ind P, ind U, clave k)
     Para i de P a U:
          Si T [i] == k:
                devolver i;
     devolver "error";
```

Figura 2.1. Pseudocódigo del algoritmo de búsqueda lineal del elemento `k` en la tabla no necesariamente ordenada `T`, entre las posiciones `P` y `U`.

La situación más costosa es la que obliga a recorrer la estructura de datos completa. Esto puede ocurrir si la búsqueda termina sin éxito o, en caso contrario, si el elemento buscado es el último de la estructura. En estos casos (tiempo peor) el orden coincide con el tamaño de la entrada (`n`). La dependencia es lineal.

2.1.2. Búsqueda binaria

La búsqueda binaria supone que los datos, entre los que puede estar el buscado, se guardan en una lista o vector ordenados.

Este algoritmo (*véase* la Figura 2.2) aprovecha el orden propio de la estructura para descartar, en cada iteración, la mitad de la tabla donde, con seguridad, no se encuentra el dato buscado. En la siguiente iteración sólo queda por estudiar la otra mitad, en la que sí puede encontrarse. Para ello se compara el elemento buscado con el que ocupa *la posición central* de la estructura (comparación de clave). Si éste es posterior (anterior) al buscado, puede descartarse la segunda (primera) mitad de la estructura. El algoritmo termina por una de las dos razones siguientes: alguna de las comparaciones encuentra el elemento buscado o la última tabla analizada tiene sólo un elemento, que no es el buscado. La última circunstancia significa que el dato no ha sido encontrado y la búsqueda termina sin éxito.

```
ind BusquedaBinaria
 (tabla T, ind P, ind U, clave K)
     mientras P≤U
          M=⌊(P+U) /2⌋
          Si T[M] < K
               P=M+1;
          else Si T[M]>K
               U=M-1;
          else devolver M;
       devolver Error;
```

Figura 2.2. Pseudocódigo del algoritmo de búsqueda binaria del elemento `k` en la tabla ordenada `T` entre las posiciones `P` y `U`.

Como mucho, este algoritmo realiza las iteraciones necesarias para reducir la búsqueda a una tabla con un solo elemento. Se puede comprobar que el tamaño de la tabla pendiente en la iteración i-ésima es igual a $n/2^i$. Al *despejar* de esta ecuación el número de iteraciones, se obtendrá una función de $\log_2(n)$, que es la expresión que determina, tanto el tiempo peor, como el tiempo medio del algoritmo.

2.1.3. Búsqueda con árboles binarios ordenados

Los árboles binarios ordenados se pueden definir de la siguiente manera:

Si se llama `T` al árbol, `clave(T)` al elemento contenido en su raíz, `izquierdo(T)` y `derecho(T)` a sus hijos izquierdo y derecho, respectivamente, y `nodos(T)` al conjunto de sus nodos, `T` es un árbol binario ordenado si y sólo si cumple que:

$\forall$ `T'`$\in$ `nodos(T)` `clave(izquierdo(T'))` $\leq$ `clave(T')` $\leq$ `clave(derecho(T'))`

El algoritmo de búsqueda (*véase* la Figura 2.3) consulta la raíz del árbol para decidir si la búsqueda ha terminado con éxito (en el caso en el que el elemento buscado coincida con la clave del árbol) o, en caso contrario, en qué subárbol debe seguir buscando: si la clave del árbol es posterior (anterior) al elemento buscado, la búsqueda continúa por el árbol `izquierdo(T)` (`derecho(T)`). Si en cualquier momento se encuentra un subárbol vacío, la búsqueda termina sin éxito. Se suelen utilizar distintas variantes de este algoritmo para que el valor devuelto resulte de la máxima utilidad: en ocasiones basta con el dato buscado, o con una indicación de que se ha terminado sin éxito; en otras, el retorno de la función apunta al subárbol donde está el elemento buscado o donde debería estar.

La Figura 2.3 resalta la comparación de claves. En el peor de los casos (que la búsqueda termine con fracaso, tras haber recorrido los subárboles más profundos, o que el elemento buscado esté precisamente en el nivel más profundo del árbol), el número de comparaciones de clave coincidirá con la profundidad del árbol. Es decir, los tiempos peor y medio dependen de la profundidad del árbol. Se escribirá `prof(T)` para representar la profundidad del árbol `T`.

```
ArbolBin Buscar(clave K, ArbolBin T)
     Si vacío(T) «devolver árbol_vacío;»
     Si k == clave(T) «devolver T»
     Si k < clave(T)
          «devolver(Buscar(k,izquierdo(T));»
     Si k > clave(T)
          «devolver(Buscar(k,derecho(T));»
```

Figura 2.3. Pseudocódigo recursivo del algoritmo de búsqueda del elemento `k` en el árbol binario ordenado `T`.

Es interesante observar que este razonamiento no expresa una dependencia directa de `n`, sino de un parámetro del árbol binario que depende tanto de `n` como de la manera en la que se creó el árbol binario en el que se busca. La Figura 2.4 muestra dos posibles árboles binarios ordenados y correctos formados con el conjunto de datos `{0,1,2,3,4}`

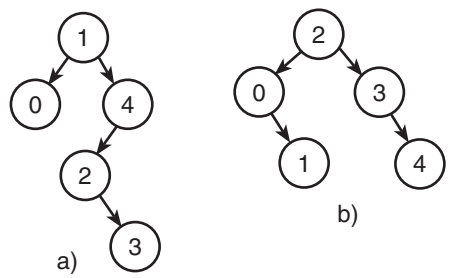


Figura 2.4. Dos árboles binarios distintos para el conjunto de datos `{0,1,2,3,4}`: a) con profundidad 3, b) con profundidad 2.

Posteriormente se profundizará más en esta reflexión, para comprender cómo depende `prof(T)` de `n`.

2.1.4. Búsqueda con árboles AVL

Los árboles de Adelson-Velskii y Landis (AVL) son un subconjunto de los árboles binarios ordenados. Un árbol binario ordenado es un árbol AVL si y sólo si las profundidades de los hijos de cualquier nodo no difieren en más de una unidad. Por tanto, aunque pueda haber muchos árboles AVL para el mismo conjunto de datos, se tiene la garantía de que la profundidad es siempre *más o menos* la del árbol binario menos profundo posible.

El árbol binario menos profundo que se puede formar con `n` nodos es el que tiene todos sus niveles completos, es decir, cada nodo que no sea una hoja tiene exactamente `2` hijos. Es fácil comprobar que el número de nodos que hay en el nivel i-ésimo de un árbol con estas características es igual a 2^i, y también que el total de nodos en un árbol de este tipo, de profundidad `k`, es igual a $2^{k+1}-1$. Si se despeja la profundidad `k` necesaria para que el número de nodos sea igual a `n`, quedará en función de $\log_2(n)$.

Se ha dicho que la profundidad es *más o menos* la del árbol binario menos profundo posible, porque se permite una diferencia en la profundidad de como mucho una unidad, que se puede despreciar para valores grandes de `n`. Por tanto, los árboles AVL garantizan que la profundidad es del *orden* de `log(n)`, mientras que en la sección anterior se vio que el tiempo peor y medio de la búsqueda en un árbol binario `T` es del orden de `prof(T)`.

2.1.5. Resumen de rendimientos

El estudio del mejor tiempo posible para cualquier algoritmo de búsqueda que se base en la comparación de claves se parece al razonamiento informal de las secciones anteriores respecto a los árboles binarios. Intuitivamente, se puede imaginar que el mejor tiempo posible estará asociado a la profundidad del árbol binario menos profundo que se puede formar con `n` nodos: $\log_2(n)$. Se puede concluir, por tanto, que éste es el rendimiento mejor posible para los algoritmos de ordenación basados en comparaciones de clave. La Tabla 2.1 muestra un resumen de los rendimientos observados.

Tabla 2.1. Resumen de los rendimientos de los algoritmos de búsqueda con comparación de clave.

Algoritmo	Orden del tiempo peor	Orden del tiempo medio
Búsqueda lineal	n	—
Búsqueda binaria	log(n)	log(n)
Cota inferior	log(n)	log(n)

2.2 El tipo de datos diccionario

2.2.1. Estructura de datos y operaciones

La teoría de estructuras de datos define el *diccionario* como una colección ordenada de información organizada de forma que una parte de ella se considera su *clave*. La clave se utiliza para localizar la información en el diccionario, de la misma manera en que, en un diccionario de la lengua, las palabras se utilizan como clave para encontrar su significado.

En un diccionario se dispondrá, al menos, de las operaciones de búsqueda, inserción y borrado:

- `Posicion Buscar(clave k, diccionario D)`:
  - — Busca el dato `k` en el diccionario `D`.
  - — La función devuelve la posición ocupada por el dato (en caso de acabar con éxito) o una indicación de que la búsqueda ha terminado sin encontrarlo.
- `Estado Insertar(clave k, diccionario D)`:
  - — Añade al diccionario `D` la información `k`.
  - — El retorno de la función informa sobre el éxito de la inserción.
- `Estado Borrar(clave k, diccionario D)`:
  - — Elimina del diccionario `D` la información `k`.

Aunque la implementación de estas funciones admita variaciones, se puede considerar general el pseudocódigo de las Figuras 2.5 y 2.6.

Puede observarse en ellas que el trabajo más importante de la inserción y el borrado es la búsqueda inicial de la clave tratada. Por tanto, la complejidad temporal de las tres operaciones

```
Estado Insertar (clave k, diccionario D)
      Posicion = Buscar(k,D);
      Si «Posicion indica que no está»
         «Modificar D para que incluya k»
         devolver «inserción correcta»
      en otro caso
         devolver «error»
```

Figura 2.5. Pseudocódigo del algoritmo de inserción de la clave `k` en el diccionario `D`.

```
Estado Borrar (clave k, diccionario D)
     Posicion = Buscar(k,D);
     Si «Posicion indica que no está»
        devolver «error»
     en otro caso
        «Modificar D para eliminar k»
        devolver «borrado correcto»
```

Figura 2.6. Pseudocódigo del algoritmo de borrado de la clave `k` del diccionario `D`.

queda determinada por la de la búsqueda. En las próximas secciones se elegirá razonadamente una implementación adecuada, en cuanto a rendimiento temporal, de esta estructura de datos.

2.2.2. Implementación con vectores ordenados

Cuando las claves del diccionario se almacenan en un vector ordenado, la operación de búsqueda puede realizarse con el algoritmo de búsqueda binaria descrito en secciones anteriores. Tanto su tiempo medio como su tiempo peor suponen un rendimiento aceptable (*véase* la Tabla 2.1).

Se estudiará a continuación si el trabajo extra añadido a la búsqueda en el resto de las operaciones empeora su rendimiento. En el caso de la inserción, para que el vector siga ordenado después de realizarla, es necesario, tras localizar la posición que la nueva clave debería ocupar en el vector, desplazar los elementos siguientes para dejar una posición libre (*véase* la Figura 2.7) En el peor de los casos (si la nueva clave debe ocupar la primera posición del vector) sería necesario mover todos los elementos, con un rendimiento temporal de orden lineal (`n`). La colocación de la información en su ubicación final se realiza en tiempo constante.

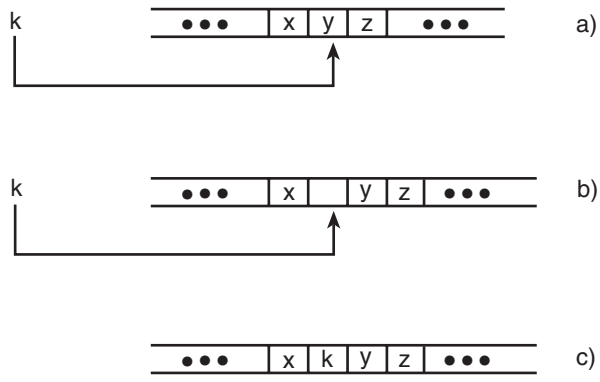


Figura 2.7. Inserción de la clave `k` en el diccionario `D` (vector ordenado): a) se busca la clave `k` en `D`; b) tras comprobar que no está se *hace hueco* para `k`; c) `k` ocupa su posición en `D`.

El rendimiento de la inserción es la suma del de la búsqueda (`log(n)`), más el del desplazamiento (`n`) y el de la asignación (`1`) y, por lo tanto, está determinado por el *peor* de ellos: `n`.

El rendimiento lineal no es aceptable, por lo que no es necesario estudiar el borrado para rechazar el uso de vectores ordenados en la implementación del diccionario.

2.2.3. Implementación con árboles binarios ordenados

En la Sección 2.1.3 se ha mostrado que el rendimiento temporal de la búsqueda depende de la profundidad del árbol y que existen diferentes árboles binarios ordenados para el mismo conjunto de datos con distintas profundidades. Si no se puede elegir *a priori* el árbol en el que se va a buscar, lo que suele ocurrir casi siempre, ya que el uso de un diccionario suele comenzar cuando está vacío, y se va rellenando a medida que se utiliza, tampoco se puede asegurar que no se termine utilizando el *peor árbol binario posible* que muestra la Figura 2.8, que, como se ve, no se puede distinguir de una *simple lista*.

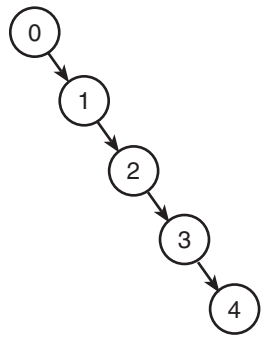


Figura 2.8. Uno de los *peores* árboles binarios posibles respecto al rendimiento temporal de la búsqueda con los datos `{0,1,2,3,4}`.

En este caso (*véase* la Sección 2.1.1) el rendimiento temporal de la búsqueda depende linealmente del tamaño de la entrada (es *de orden* `n`). Este rendimiento no es aceptable, lo que basta para rechazar los árboles binarios ordenados para implementar el diccionario. A pesar de esto, se realizará el estudio de las operaciones de inserción y borrado, para facilitar la comprensión de la siguiente sección.

• Inserción

La inserción de la clave `k` en el árbol binario ordenado `T` comienza con su búsqueda. Si suponemos que el algoritmo de búsqueda devuelve un puntero al nodo padre donde debería insertarse el nuevo elemento, lo único que quedaría por hacer es crear un nodo nuevo y asignarlo como hijo izquierdo (derecho) al retorno de la búsqueda, si `k` es anterior (posterior) a la clave del árbol. La Figura 2.9 muestra gráficamente esta situación y la Figura 2.10 el pseudocódigo correspondiente.

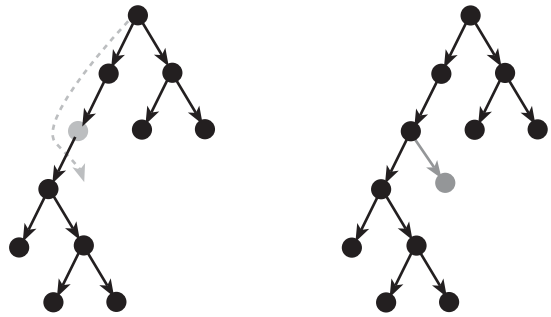


Figura 2.9. Representación gráfica de la inserción en árboles binarios ordenados: a) la flecha discontinua y el nodo claro indican la posición donde debería insertarse la nueva clave; b) resultado de la inserción; se resaltan las modificaciones en el árbol de partida.

Puede observarse que el trabajo añadido a la búsqueda consiste en una comparación de clave y una modificación del valor de una variable. Este trabajo es el mismo para cualquier tamaño de la entrada (`n`), por lo que supondrá un incremento de tiempo constante que se podrá despreciar para valores grandes de `n`, por lo que el rendimiento de la inserción es el mismo que el de la búsqueda.

```
estado Insertar(clave k, ArbolBin T)
  ArbolBin arbol_auxiliar T', T'';
  T'=Buscar(k,T);
  T''=nuevo_nodo(k);
  Si «no es posible crear el nodo» «devolver error»
  Si k < clave(T') izquierdo(T')=T'';
  else derecho(T')=T'';
  «devolver "ok"»
```

Figura 2.10. Pseudocódigo del algoritmo de inserción de la clave `k` en el árbol binario ordenado `T`.

• Borrado

El borrado de la clave `k` del árbol binario ordenado `T` presenta la dificultad de asegurar que el árbol sigue ordenado tras eliminar el nodo que contiene a `k`. La disposición de los nodos en estos árboles permite, sin embargo, simplificar el proceso gracias a los dos resultados siguientes:

1. (*Véase* la Figura 2.11) Para cualquier árbol binario ordenado `T` y cualquier nodo `b` del mismo, se pueden demostrar las siguientes afirmaciones relacionadas con el nodo `b'`, que contiene el antecesor inmediato del nodo `b` en el árbol:

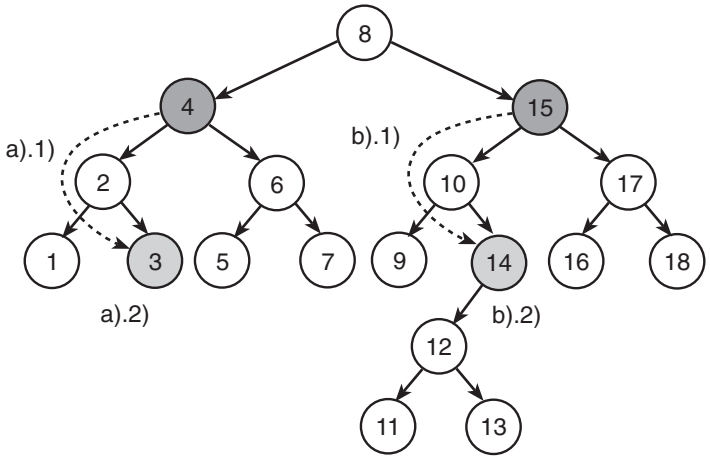


Figura 2.11. Dos ejemplos de localización del nodo con el antecesor inmediato de otro dado en un árbol binario ordenado. a) El antecesor inmediato de 4 es 3: 1) el nodo raíz del subárbol de los elementos menores que 4 contiene el 2; 2) al descender siguiendo los hijos derechos a partir del nodo que contiene al 2, se termina en el nodo que contiene al 3. b) El antecesor inmediato de 15 es 14: 1) los elementos menores que 15 están en el subárbol de raíz 10; 2) al descender por los hijos derechos se termina en el nodo que contiene el 14. Obsérvese que en este caso existe hijo izquierdo (el subárbol que comienza en 12), pero no hijo derecho.

- Por definición de árbol binario ordenado, `b'` tendrá que estar en el subárbol `izquierdo(b)` y debe ser el nodo que esté más a la derecha en dicho subárbol.
- Por lo tanto, se puede localizar `b'` realizando los siguientes pasos:
  1. Se llamará `ib` a `izquierdo(b)` en `T`.
  2. A partir de `derecho(ib)`, y mientras exista el subárbol hijo derecho, se avanza hacia los niveles más profundos del árbol por el subárbol hijo derecho del anterior.
  3. `b'` es la raíz del árbol encontrado mediante los pasos 1 y 2.
- Puede observarse que `b'` necesariamente debe carecer de hijo derecho, pues en otro caso no se habría terminado aún el paso 2.

2. (*Véase* la Figura 2.12) Se puede demostrar que, para cualquier árbol binario ordenado `T` y cualquier nodo `b` del mismo, el árbol `T'` construido mediante el siguiente proceso corresponde al resultado de eliminar el nodo `b` de `T`:
   - Inicialmente `T'` es una copia de `T`.
   - Sea `b'` el nodo que contiene el antecesor inmediato al nodo `b` en `T`.
   - En `T'` se sustituye el contenido del nodo `b` por el de su antecesor inmediato en el árbol (`b'`).

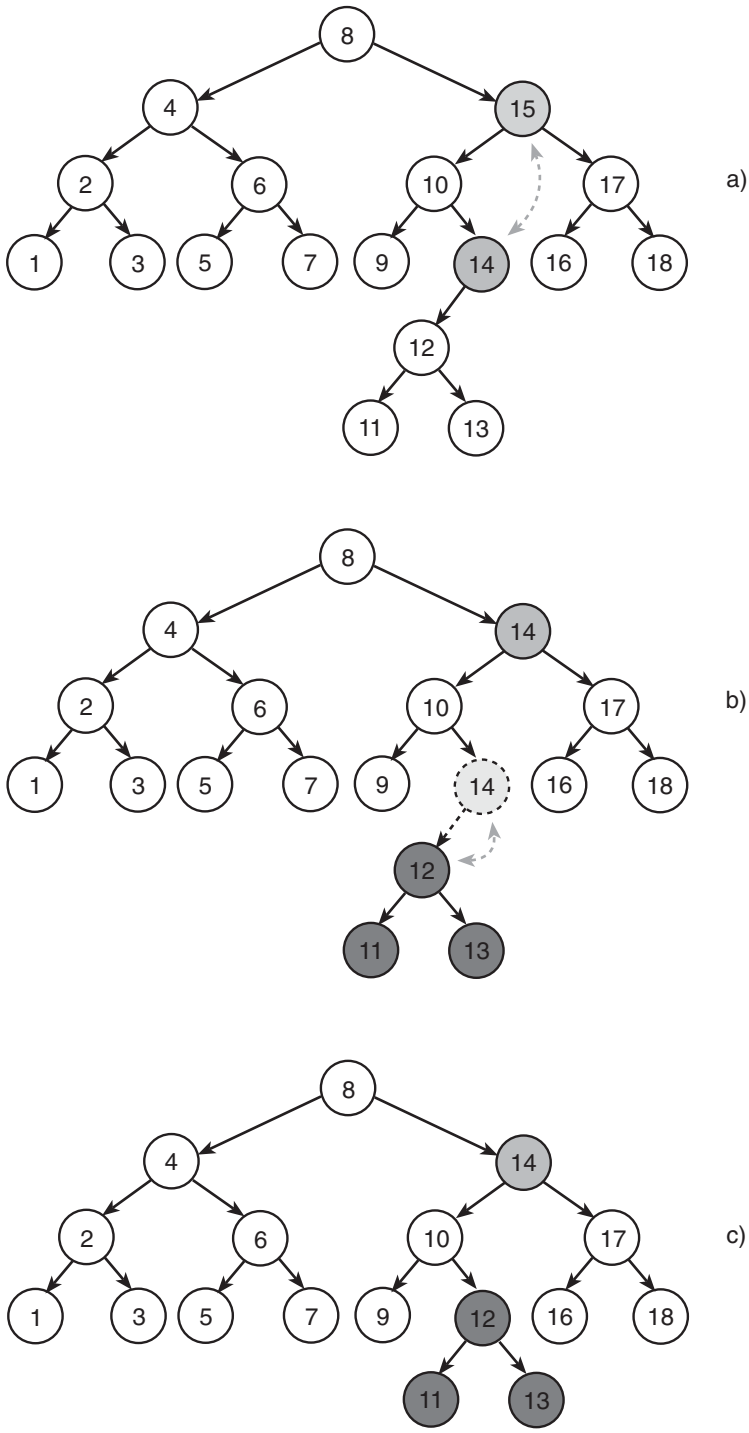


Figura 2.12. Borrado del nodo 15: a) localización y sustitución de 15 por su antecesor inmediato en `T`, b) sustitución del antecesor por su hijo izquierdo, c) árbol final.

- En `T'` se sustituye el subárbol cuya raíz es `b'` por el subárbol `izquierdo(b')`, si éste existe. Si no existe, se elimina el nodo `b'`.

Por lo tanto, el borrado conlleva dos búsquedas (la del elemento que se va a borrar y la de su antecesor inmediato en el árbol) y el cambio de valor de dos variables del árbol (el contenido del nodo del elemento borrado y el subárbol donde estaba el antecesor inmediato del elemento borrado). El rendimiento temporal del proceso completo sigue siendo del orden de la profundidad del árbol (`prof(T)`). Realmente se necesitará el doble de la profundidad del árbol (por las dos búsquedas) más un tiempo constante, que no depende del tamaño de la entrada (por las dos asignaciones).

2.2.4. Implementación con AVL

Cuando las claves del diccionario se almacenan en un árbol AVL, el tiempo medio y el tiempo peor de la búsqueda suponen un rendimiento aceptable (*véase* la Tabla 2.1). Se estudiará a continuación si el trabajo *extra* añadido a la búsqueda en el resto de las operaciones empeora su rendimiento.

• Inserción

Para asegurarse de que los subárboles de un árbol AVL están balanceados, es preciso realizar algunas operaciones adicionales en la inserción, que suelen llamarse *rotaciones*, respecto al algoritmo descrito para árboles binarios ordenados. En concreto, cada vez que se inserta una clave, se necesitará una o dos *rotaciones* (con una o dos modificaciones de valor) para asegurarse de que las profundidades de los subárboles de todo el árbol siguen difiriendo, a lo más, en una unidad. Por tanto, la inserción en árboles AVL añade un trabajo que no depende del tamaño de la entrada y que requerirá un tiempo de orden constante (`1`), que puede despreciarse para entradas grandes (valores grandes de `n`) frente a `prof(T)`, que para árboles AVL es del orden de `log(n)`, por lo que el orden de la complejidad temporal de la inserción sigue siendo `log(n)`.

• Borrado

Se puede comprobar que el borrado de una clave en árboles binarios ordenados modifica, como mucho, en una unidad la profundidad de uno de los subárboles. Por lo tanto, se puede repetir la reflexión del apartado anterior para afirmar que el borrado en árboles AVL sólo requiere un trabajo adicional constante (de orden `1`), respecto al borrado en árboles binarios ordenados, que se puede despreciar para entradas grandes frente a `prof(T)`, por lo que `log(n)` es también el orden de complejidad del borrado. Por lo tanto, los árboles AVL sería una buena opción para implementar el diccionario, si la técnica más eficiente fuese la comparación de claves.

2.3 Implementación del tipo de dato diccionario con tablas *hash*

En esta sección se intentará responder a la pregunta de si existe alguna técnica más eficiente que la comparación de claves. Para ello se analizarán alternativas mejores que la dependencia logarítmica entre el tiempo de ejecución de la búsqueda y el tamaño de la entrada.

2.3.1. Conclusiones sobre rendimiento

En las secciones anteriores se ha reflexionado sobre la implementación de las tablas de símbolos mediante algoritmos basados en comparaciones de clave. Se ha llegado a la conclusión (*véase* la Tabla 2.1) de que el rendimiento temporal de esta técnica está acotado inferiormente por la dependencia logarítmica del tamaño de la entrada, del orden de `log(n)`. Aunque desde el punto de vista de la complejidad de algoritmos este rendimiento es aceptable, para el problema tratado en este libro, supone que el tiempo necesario para compilar un programa depende logarítmicamente del número de identificadores que contenga. Resulta claro que el diseñador del compilador no puede predecir el valor de este número.

La teoría de la complejidad de algoritmos suele considerar que, si se quiere mejorar el rendimiento logarítmico `log(n)`, se necesita conseguir un rendimiento constante, que no dependa del tamaño de la entrada (del orden de `1`).

La conclusión de todas estas reflexiones es que resulta imposible obtener un tiempo de compilación independiente del número de identificadores que contengan los programas, mediante estructuras de datos y algoritmos que sólo utilicen comparaciones entre ellos para insertarlos, buscarlos o borrarlos de la tabla de símbolos. Para conseguir ese rendimiento, es necesario recurrir a estructuras de datos más complejas. En los próximos apartados se analizará el uso de funciones y tablas *hash* o de dispersión para lograr ese rendimiento.

2.3.2. Conceptos relacionados con tablas *hash*

Intuitivamente, una tabla *hash* es un tipo de datos que consta de un vector (para almacenar los datos) y una función (*hash* o de dispersión, que es su significado en inglés), que garantiza (idealmente) que a cada dato se le asocie una posición única en el vector. La tabla hash se usa de la siguiente manera:

- Se elige una parte de los datos para considerarla clave de los mismos (por ejemplo, si se está almacenando información sobre personas, la clave podría ser su DNI).
- Se diseña una función hash que asocie (idealmente) a cada valor de la clave una posición única en el vector.
- Cuando se desea localizar un dato en el vector (ya sea para añadirlo a la tabla, para recuperarlo o para eliminarlo) se aplica a su clave la función hash, que proporciona el índice en el vector que corresponde al dato.

La mejora consiste en que el tiempo necesario para buscar un elemento ya no depende del número de elementos contenidos en la tabla. El rendimiento temporal será la suma del cálculo de la función hash y del tiempo necesario para realizar la asignación, que no depende del tamaño de la tabla. Por lo tanto, es esencial que el diseño de la función hash tampoco dependa del tamaño de la tabla. En ese caso, el rendimiento de la búsqueda de una clave en una tabla hash será constante (del orden de `1`) y no dependerá del tamaño de la entrada. Formalmente, la función hash toma valores en el conjunto de claves y recorre el conjunto de índices o posiciones en el vector,

por lo que sólo depende de la clave, lo que garantizaría la independencia entre su rendimiento y el tamaño de la tabla.

El nombre de la estructura (hash o dispersión) hace referencia a otro aspecto importante que será analizado con detalle en las próximas secciones: la función debería *dispersar* las claves adecuadamente por el vector; es decir, en teoría, a cada clave se le debería hacer corresponder de forma biunívoca una posición del vector. Ésta es una situación ideal prácticamente inalcanzable. En realidad, es inevitable que las funciones hash asignen la misma posición en el vector a más de una clave. Las diferentes técnicas para solventar esta circunstancia originan distintos tipos de tablas de dispersión con diferentes rendimientos, que serán objeto de las próximas secciones.

Las tablas hash también se llaman *tablas de entrada calculada*, ya que la función hash se usa para calcular la posición de cada entrada en la tabla.

2.3.3. Funciones hash

Se usará la siguiente notación para las funciones hash: sea `K` el conjunto de claves, sean `1`[1] y `m` respectivamente las posiciones mínima y máxima de la tabla, y sea `N` el conjunto de los números naturales. Cualquier función hash `h` se define de la siguiente forma:

$$h:K \rightarrow [1,m]$$

Funciones hash inyectivas

Lo ideal es que `h` fuese al menos *inyectiva*, ya que de esta forma se garantiza que a dos claves distintas les corresponden siempre posiciones distintas del vector. Formalmente

$$\forall k,k' \in K;\ k \neq k' \Rightarrow h(k) \neq h(k')$$

Lo que se pierde al no obligar a `h` a ser biyectiva es que no se garantiza que se ocupen todas las posiciones del vector. En la práctica, es muy difícil diseñar funciones hash inyectivas, por lo que hay que conformarse con *funciones no inyectivas razonablemente buenas*.

La no inyectividad causa un problema importante: las *colisiones*. Se llama *colisión* a la situación en la que `h` asigna la misma posición en el vector a dos o más claves distintas. Formalmente: $\exists k,k' \in K \mid k \neq k' \wedge h(k)=h(k')$.

¿Cómo implementar tablas hash a pesar de las colisiones? Ya que se permiten, al menos, se intentará minimizar su aparición. Informalmente, se pretende que sea pequeña la probabilidad de que se produzca una colisión y, por tanto, grande la de que no se produzca.

Formalmente, en una situación ideal, sería deseable que la probabilidad de colisión fuese igual a $1/m$, y la de que no haya colisión, $(m-1)/m$. La Figura 2.13 muestra gráficamente esta circunstancia al insertar la segunda clave `k'` en el supuesto de haber insertado ya la clave `k` (con $k \neq k'$).

[1] El valor mínimo para las posiciones en la tabla puede ser 0 o 1, como el origen de los índices en los distintos lenguajes de programación. En este capítulo se usará indistintamente, y según convenga, un valor u otro.

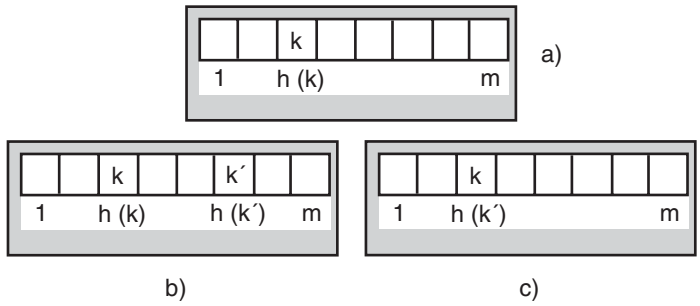


Figura 2.13. Justificación intuitiva de la probabilidad de colisión. a) Situación inicial, tras insertar la clave `k`. b) Al insertar `k'` no se produce colisión, `h(k)≠h(k')`, el número de casos favorables es `m-1`, ya que sólo la posición `h(k)` es un caso desfavorable; el número de casos posibles es `m`. c) Se produce colisión; el número de casos favorables es 1 y el número de casos posibles es `m`.

Funciones hash pseudoaleatorias

El siguiente mecanismo para la inserción de la clave `k`, que recibe el nombre de *aleatorio* o *pseudoaleatorio*, permitiría alcanzar este objetivo:

1. Se *tira un dado* con `m` caras.
2. Se anota el valor de la cara superior (`i`).
3. Se toma `i` como el valor hash para la clave `k`: `h(k)=i`.
3. Se accede a la posición `i` de la tabla y se le asigna la información de la clave `k`.

Resulta claro que este esquema no es válido para implementar las tablas hash, ya que el mecanismo de recuperación de la información asociada a la clave `k'` tendría los siguientes pasos:

1. Se *tira un dado* con `m` caras.
2. Se anota el valor de la cara superior (`j`)
3. Se define `j` como el valor hash para la clave `k`: `h(k)=j`.
4. Se accede a la posición `j` de la tabla y se recupera la información almacenada en ella.

El método anterior sólo funcionará si los resultados del experimento aleatorio se repiten siempre que se aplique a la misma clave. Pero eso entra en contradicción con la definición del experimento aleatorio. En particular, las funciones hash pseudoaleatorias no garantizan en modo alguno que, una vez que se ha insertado la información de una clave, se pueda recuperar.

En las próximas secciones se analizará la pérdida de eficiencia asociada a las colisiones. El objetivo será comprobar que su gestión, aunque implique la pérdida del rendimiento temporal constante, no empeora la dependencia logarítmica. La utilidad de las funciones hash aleatorias consiste en su uso en el estudio teórico de los rendimientos. La gestión de las colisiones dificulta el análisis con funciones hash que no sean pseudoaleatorias.

Funciones hash uniformes

Se pedirá a las funciones hash que sean *relativamente uniformes*, es decir, que distribuyan las claves de manera uniforme por la tabla, sin que queden muchas posiciones libres cuando comiencen a aparecer colisiones. En este contexto, resulta esencial encontrar un mecanismo que genere elementos de un subconjunto de los números naturales `[1,m]` sin repeticiones. El álgebra y la teoría de números definen la operación módulo para el cálculo del resto de la división entre dos números enteros, los grupos cíclicos, su estructura y las condiciones para su existencia. Estos resultados, que quedan fuera del ámbito de este libro, pueden utilizarse para definir funciones *hash relativamente uniformes*. A continuación se muestran, sin justificar, algunos ejemplos.

Funciones hash de multiplicación

Se utiliza una función auxiliar uniforme, con imagen en el intervalo real `[0,1]`. Su producto por el tamaño de la tabla (`m`) nos lleva a una función real uniforme con imagen en el intervalo `[0,m]`. Formalmente $h(k)=\lfloor m(k\phi)\rfloor$, donde

- `k` es un valor numérico asociado con la clave. Si la clave no es numérica, se supondrá la existencia de una función que calcule un valor numérico a partir de la clave. Por simplificar la notación, se omitirá esta función. En próximas secciones se describirán con más detalle algunas técnicas para obtener valores numéricos a partir de claves no numéricas.
- `m` es la posición máxima dentro de la tabla, que debe cumplir la siguiente condición:

$$\exists p \in Z \mid p \text{ es primo} \wedge m=2^p$$

- $\phi \in R-Q$ es un número irracional. Es frecuente utilizar el valor

$$\phi = \frac{\sqrt{5}-1}{2}$$

- $\lfloor x \rfloor$ es la función *suelo*, que calcula el entero más próximo *por debajo* de su argumento.
- (x) es la función *parte fraccionaria*, definida así: $(x) = x-\lfloor x \rfloor$

La Tabla 2.2 muestra un ejemplo de otra función hash de multiplicación.

Tabla 2.2. Algunos valores de la función hash de multiplicación que utiliza $\phi=\pi$ y $m=25$. Se resaltan las colisiones.

k	kx	h(k)	k	kx	h(k)
1	3.141592654	**3**	7	21.99114858	24
2	6.283185307	7	8	25.13274123	**3**
3	9.424777961	**10**	9	28.27433388	6
4	12.56637061	14	10	31.41592654	**10**
5	15.70796327	17	1	34.55751919	13
6	18.84955592	21	12	37.69911184	17

Funciones hash de división

Se utiliza la función módulo `h(k)=k%m`, donde

- `k` es, como en el caso anterior, un valor numérico asociado a la clave.
- `m` es el valor máximo de posición dentro de la tabla. Se le exige que sea primo.
- `%` es la función *módulo*, definida como el resto de la división de `x` entre `m`.

La Tabla 2.3 muestra ejemplos de esta función hash.

Tabla 2.3. Algunos valores de la función hash de división, para `m=7`. Se resaltan las colisiones.

k	h (k)
1	1
2	2
3	3
4	4
5	5
6	6

k	h (k)
7	0
8	1
9	2
10	3
11	4
12	5

Otras funciones hash

A pesar de los argumentos teóricos anteriores, el diseño de funciones hash tiene mucho de trabajo *artesanal* y es una tarea complicada. Por eso, a continuación, se describe una función hash bien documentada en la literatura, que en la práctica ha mostrado ser buena. Puede encontrarse una exposición detallada en [3].

Dicha función hash es un algoritmo iterativo que calcula un valor auxiliar (h_i) entre 0 y la longitud `m` de la clave `id`. El valor final `h(id)` se obtiene a partir de alguno de los bits del valor m-ésimo (h_m).

$$\begin{cases} h_0=0 \\ h_i=k\times h_{i-1}+c_i \quad \forall i\ 1\leq i\leq m \end{cases}$$
$$h(k)=bits(h_m,30)\%n$$

donde

- `k` es una constante deducida experimentalmente.
- `n` es el tamaño de la tabla, deducido con `k` experimentalmente.
- `bits(x,j)` es una función que obtiene los `j` bits menos significativos del entero `x`.
- c_i es el código ASCII del carácter i-ésimo de `id`

2.3.4. Factor de carga

Un concepto muy importante en el estudio de la eficiencia es el *factor de carga*. Dada una tabla hash con espacio para `m` claves, en la que ya se han insertado `n`, se llama *factor de carga* y se representa mediante la letra λ, al cociente entre `n` y `m`.

$$\lambda = \frac{n}{m}$$

2.3.5. Solución de las colisiones

Puesto que se van a usar funciones hash que permiten colisiones, es necesario articular mecanismos para reaccionar frente a éstas. Aunque son muchas las alternativas posibles, en las siguientes secciones se explicarán algunas de ellas con detalle.

2.3.6. Hash con direccionamiento abierto

Esta técnica recibe su nombre del hecho de que la posición final que se asigna a una clave no está totalmente determinada por la función hash. Lo más característico de este método es que las colisiones se solucionan dentro del mismo espacio utilizado por la tabla, es decir, no se usa ninguna estructura de datos auxiliar para ello. De aquí se deduce la necesidad de que haya siempre posiciones libres en la tabla, es decir, que el tamaño reservado para ella sea siempre mayor que el número de claves que se va a insertar.

La inserción de una clave `k` mediante direccionamiento abierto funciona de la siguiente manera (la Figura 2.14 muestra el pseudocódigo para la inserción, común a todas las variantes de encadenamiento abierto):

1. Se estima el número de claves que se va insertar en la tabla.
2. Se dimensiona la tabla para que *siempre haya posiciones libres*. El tamaño necesario depende de otros aspectos de la técnica que se explicarán a continuación.
3. Cuando se va a insertar la clave, se calcula la posición que le asignaría la función hash `h(k)`. Si dicha posición está libre, no hay colisión y la información se guarda en esa posición. En otro caso hay colisión: se recorre la tabla buscando la primera posición libre (`j`). La información se guarda en dicha posición j-ésima.

La manera de encontrar la primera posición libre se llama *sondeo* o *rehash*. Como se verá a continuación, el sondeo no implica necesariamente que las claves que colisionan en la misma posición de la tabla ocupen finalmente posiciones contiguas. Por esta causa, también se conoce al direccionamiento abierto como *espaciado*. El objetivo del sondeo es ocupar la mayor parte de la tabla con el mejor rendimiento posible.

La recuperación de información de la tabla tiene que tener en cuenta que ya no se garantiza que la información de la clave `k` esté en la posición `h(k)`. Hay que utilizar el mismo mecanismo de sondeo empleado en la inserción para recorrer la tabla, hasta encontrar la clave buscada. La Figura 2.15 muestra el pseudocódigo de la búsqueda, común a todas las variantes del encadenamiento abierto.

```
indice Insertar(clave k, TablaHash T)
   indice posicion=funcion_hash(k,T);
   int i=0; /*Numero de reintentos*/
   Si k == T.datos[posicion].clave devolver posicion;
/*Ya estaba*/
   else{
     Mientras no vacia(T.datos[posicion])
          y no posicion == funcion_hash(k,T)
          y no k == T.datos[posicion].clave
     {posicion = (posicion + delta(i++))mod tamaño(T);}
     if vacia(T.datos[posicion]) {/*No estaba y se inserta*/
          T.datos[posicion].clave = k;
          devolver posición;}
     if posicion == funcion_hash(k,T) devolver -1;
          /* T no tiene espacio para ese valor de hash */
     if k == T.datos[posicion].clave devolver posicion;
     /*Ya estaba*/
}
```

Figura 2.14. Pseudocódigo del algoritmo de inserción de la clave `k` en la tabla hash `T`, común a todas las técnicas con direccionamiento abierto. Se resalta el *sondeo*.

```
indice Buscar(clave k, TablaHash T)
   indice posicion=funcion_hash(k,T);

   Si k == T.datos[posicion].clave devolver posicion;
   else
   {
     Mientras no vacia(T.datos[posicion])
          y no posicion == funcion_hash(k,T)
          y no k == T.datos[posicion].clave
     {posicion = (posicion + delta(i++))mod tamaño(T);}
     if vacia(T.datos[posicion]) devolver -1; /*No está*/
     if posicion == funcion_hash(k,T) devolver -1;
          /* Además esto significa que la tabla no tiene
          espacio disponible para ese valor de hash */
     if k == T.datos[posicion].clave devolver posicion;
/*Está*/
   }
```

Figura 2.15. Pseudocódigo del algoritmo de búsqueda de la clave `k` en la tabla hash `T`, común a todas las técnicas con direccionamiento abierto. Se resalta el *sondeo*.

Los algoritmos de las Figuras 2.14 y 2.15 muestran el sondeo como un desplazamiento representado por la función `delta`, que se suma a la posición devuelta por la función hash, en un bucle que recorre la tabla buscando la clave, cuando es necesario. En los próximos párrafos se analizarán diferentes tipos de sondeo, es decir, distintas implementaciones de la función `delta`.

Obsérvese que de la Figura 2.14 pueden deducirse distintas condiciones para concluir que no hay sitio en la tabla:

- Cuando la tabla está totalmente llena. Ya se ha advertido de la necesidad de que la tabla sea lo suficientemente grande para que esta situación no se produzca nunca.
- Cuando, durante la repetición del sondeo, independientemente de que haya posiciones libres en la tabla, se llega a una posición previamente visitada. En este caso, aunque la tabla tenga sitio, no se va a poder llegar a él.

La segunda condición es muy importante para el diseño del sondeo. Hasta ahora se podía pensar que la gestión correcta de todas las claves se garantizaba con una tabla suficientemente grande. Sin embargo, un *sondeo deficiente*, aunque se realice en una tabla muy grande, puede dar lugar a un rendimiento similar al conseguido con una tabla demasiado pequeña. Un ejemplo trivial de *sondeo deficiente* es el que, tras una colisión, sólo visita una única posición más, que es siempre la primera de la tabla.

Como se verá a continuación, esta segunda condición es la que más determina el diseño de los sondeos y el rendimiento del direccionamiento abierto.

Sondeo lineal

El sondeo lineal busca sitio en las posiciones siguientes, en la misma secuencia en que están en la tabla. Si se supone que `posición=h(k)` y que no está libre, el sondeo lineal mirará en la secuencia de posiciones $\{posición+1, posición+2, ...\} = \{posición+i\}_{1 \le i \le m-posición}$. Por lo tanto, todas las claves que colisionen estarán agrupadas en posiciones contiguas a la que les asigna la función hash. La Figura 2.16 muestra gráficamente esta circunstancia.

Hay diferentes métodos para estimar el rendimiento del direccionamiento abierto con sondeo lineal. En la literatura se pueden encontrar justificaciones, tanto analíticas como basadas en simulaciones [1, 2]. Todas las justificaciones coinciden en que la dependencia del rendimiento

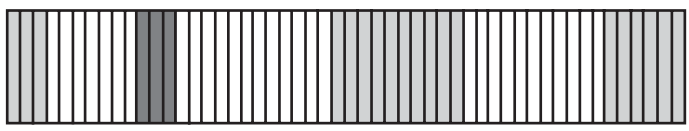


Figura 2.16. Posible estado de una tabla hash con direccionamiento abierto y sondeo lineal. Hay tres grupos de claves que colisionan, con tres, diez y nueve claves, respectivamente. La primera posición de cada grupo (por la izquierda) es la asignada por la función hash. Obsérvese que el último grupo continúa en las primeras posiciones de la tabla.

temporal respecto al factor de carga, cuando se buscan claves que no están en la tabla hash, se puede aproximar mediante la siguiente expresión:

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

También coinciden en que, cuando se buscan claves que sí están en la tabla, el rendimiento se puede aproximar mediante la expresión

$$\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$$

En la práctica es poco conveniente que las claves que colisionan formen bandas contiguas.

Sondeo multiplicativo

El sondeo multiplicativo intenta superar el inconveniente que suponen las bandas de claves que colisionan en la tabla hash. Para ello se articula un mecanismo poco costoso para dispersar los reintentos por la tabla, impidiendo la formación de bandas al espaciarlos uniformemente. Intuitivamente, se usa como incremento el valor devuelvo por la función hash, de forma que, en el primer reintento, se *saltan* `h(k)` posiciones; en el segundo, `2*h(k)` posiciones, etc. La Figura 2.17 muestra el pseudocódigo del sondeo multiplicativo.

```
int delta(int numero_reintento, indice posicion_inicial)
{
     return (posicion_inicial*numero_reintento);
}
```

Figura 2.17. Pseudocódigo del algoritmo sondeo multiplicativo. Se necesitan dos argumentos, el número de reintentos y la posición inicial.

Obsérvese que la posición `0` de la tabla no se debe utilizar, pues el sondeo multiplicativo sólo visitaría ésta posición en todos los reintentos.

La Figura 2.18 muestra gráficamente un ejemplo de la gestión de una tabla hash con este método.

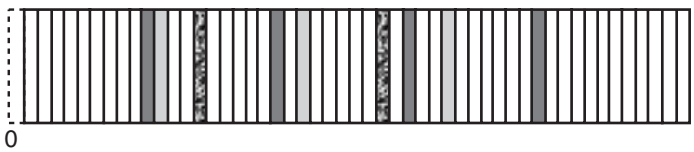


Figura 2.18. Posible estado de una tabla hash con direccionamiento abierto y sondeo multiplicativo. Hay tres grupos de claves que colisionan, con cuatro, tres y dos claves, respectivamente. El primer grupo corresponde a la posición inicial 10, el segundo a la 14 y el tercero a la 11. Obsérvese que la posición 0 no se usa y que las posiciones visitadas por cada grupo de sondeos se entremezclan.

El sondeo multiplicativo tiene una propiedad interesante: cuando el número de reintentos es suficientemente grande, al sumar el desplazamiento proporcionado por el sondeo multiplicativo se obtiene una posición fuera de la tabla. Las Figuras 2.14 y 2.15 muestran que se utiliza la operación `mod tamaño(T)` para seguir recorriendo la tabla circularmente en estos casos. Es fácil comprender que, si la tabla tiene un tamaño primo, los sondeos la cubrirán por completo y no se formarán bandas contiguas.

Otros sondeos

En general, podría utilizarse cualquier algoritmo para el código de la función `delta`. Se pueden obtener así diferentes tipos de sondeo. Una variante es el *sondeo cuadrático*, que generaliza el sondeo multiplicativo de la siguiente manera: el sondeo multiplicativo realmente evalúa una función lineal, `f(x)=posición_inicial*x` (donde `x` es el número de reintentos). El *sondeo cuadrático* utiliza un polinomio de segundo grado $g(x)=a*x^2+b*x+c$, en el que hay que determinar las constantes `a`, `b` y `c`. La Figura 2.19 muestra el pseudocódigo del sondeo cuadrático.

```
int delta(int numero_reintento)
{
     return (a*numero_reintento²+b*numero_reintento+c);
}
```

Figura 2.19. Pseudocódigo del algoritmo del sondeo cuadrático. Queda pendiente determinar las constantes del polinomio de segundo grado.

Otra variante, que sólo tiene interés teórico, consiste en generar el incremento de la función de manera pseudoaleatoria. La Figura 2.20 muestra el pseudocódigo del sondeo aleatorio.

```
int delta( )
{
     return ( random() );
}
```

Figura 2.20. Pseudocódigo del algoritmo del sondeo pseudoaleatorio.

Este método sólo tiene interés para el estudio analítico del rendimiento temporal. Se puede demostrar, aunque queda fuera del objetivo de este libro, que la dependencia del factor de carga del rendimiento temporal en la búsqueda de una clave que no se encuentra en la tabla hash, puede aproximarse mediante la siguiente expresión:

$$\frac{1}{1-\lambda}$$

La búsqueda de claves que sí están en la tabla se puede aproximar mediante esta otra:

$$\frac{1}{\lambda}\log\left(\frac{1}{1-\lambda}\right)$$

Redimensionamiento de la tabla hash

A lo largo de la sección anterior, se han mencionado diferentes circunstancias por las que las tablas hash, gestionadas con direccionamiento abierto, pueden quedarse sin sitio para insertar claves nuevas:

- Cuando *toda la tabla* está llena.
- Cuando, aunque exista espacio en la tabla, el mecanismo de sondeo no es capaz de encontrarlo para la clave estudiada.

El conocimiento del rendimiento de una técnica concreta permite añadir otra causa para la redimensión de la tabla: que el rendimiento caiga por debajo de un umbral. Todas las fórmulas de estimación del rendimiento dependen del factor de carga, y éste del tamaño de la tabla y del número de datos que contenga. Es fácil tener en cuenta el número de datos almacenado en la tabla (que se incrementa cada vez que se inserta una nueva clave) y, por tanto, estimar el rendimiento en cada inserción. En cualquiera de los casos, el redimensionamiento de la tabla consta de los siguientes pasos:

1. Crear una nueva tabla mayor. El nuevo tamaño tiene que seguir manteniendo las restricciones de tamaño de los algoritmos utilizados (ser primo, potencia con exponente primo, etc.).
2. Obtener de la tabla original toda la información que contenga e insertarla en la tabla nueva.

2.3.7. Hash con encadenamiento

Esta técnica se diferencia de la anterior en el uso de listas para contener las claves que colisionan en la misma posición de la tabla hash. De esta forma, la tabla está formada por un vector de listas de claves. La función hash proporciona acceso a la lista en la que se encuentran todas las claves a las que les corresponde el mismo valor de función hash.

La Figura 2.21 muestra un ejemplo de estas tablas.

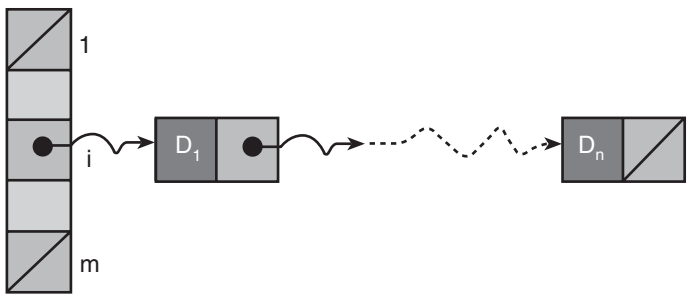


Figura 2.21. Representación gráfica de una tabla hash con listas de desbordamiento. Se resalta la lista de la posición `h(k)=i`.

Es frecuente que las listas se implementen utilizando memoria dinámica, es decir, solicitando espacio al sistema operativo cuando se necesite, sin más limitación que la propia de la computadora.

Aunque se puede utilizar cualquier algoritmo de inserción y búsqueda en listas ordenadas, se supondrá que las listas no están necesariamente ordenadas, por lo que se utilizará la búsqueda lineal. Se puede demostrar, aunque queda fuera de los objetivos de este libro, que el rendimiento temporal de la búsqueda de una clave que no está en la tabla puede aproximarse precisamente mediante el factor de carga. Esta afirmación puede comprenderse intuitivamente. La búsqueda lineal de claves que no están en la lista tiene un rendimiento temporal del orden del tamaño de la lista. Si se supone, como se está haciendo, que la función hash es uniforme, podemos suponer que en una tabla de `m` listas en las que hay `n` elementos en total (con `n` posiblemente mayor que `m`) cada lista tendrá aproximadamente `n/m` elementos. Éste es, precisamente, el valor de λ. También se puede demostrar, aunque no se va a justificar ni siquiera intuitivamente, que la dependencia, en el caso de que las claves buscadas estén en la tabla, puede aproximarse mediante la siguiente expresión:

$$1 + \frac{1}{2}\lambda - \frac{1}{2m}$$

2.4 Tablas de símbolos para lenguajes con estructuras de bloques

2.4.1. Conceptos

Los lenguajes de programación con estructura de bloques tienen mecanismos para definir el alcance de los nombres e identificadores (las secciones del código donde estarán definidos). Es decir, en los lenguajes de programación con estructura de bloques, en cada bloque sólo están definidos algunos identificadores. Los bloques más frecuentes son las subrutinas (funciones o procedimientos), aunque también hay lenguajes de programación que permiten definir bloques que no corresponden a subrutinas.

La mayoría de los lenguajes de programación de alto nivel (Algol, PL/I, Pascal, C, C++, Prolog, LISP, Java, etc.) tienen estructura de bloques. La Figura 2.22 muestra un ejemplo de un programa escrito con un lenguaje ficticio, con estructura de bloques similares a las de C.

A lo largo de esta sección se utilizarán los siguientes conceptos:

- **Ámbito.** Sinónimo de bloque. Se utilizará indistintamente.
- **Ámbitos asociados a una línea de código.** Toda línea de código de un programa escrito con un lenguaje de estructura de bloques está incluida directamente en un bloque. A su vez, cada bloque puede estar incluido en otro, y así sucesivamente. Los *ámbitos asociados*

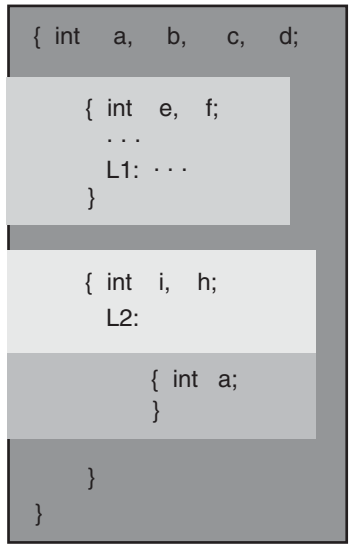


Figura 2.22. Bloques en un programa escrito con un lenguaje ficticio con estructura de bloques similar a la de C. Los bloques se inician y terminan, respectivamente, con los símbolos `{` y `}`. Sólo se declaran identificadores de tipo entero y etiquetas (cuando tras el nombre del identificador se escribe el símbolo :). En el bloque más externo, están declaradas las variables `a`, `b`, `c` y `d`. En el segundo bloque, en orden de apertura, se declara la variable `e`, la variable `f` y la etiqueta `L1`. En el tercero, las variables `i` y `h` y la etiqueta `L2`, y en el cuarto la variable `a`. El comportamiento, cuando colisionan identificadores con el mismo nombre, depende del diseñador del lenguaje de programación.

a una línea de código son todos aquellos que directa o indirectamente incluyen a la línea de código.

- **Bloque abierto.** Dada una línea de código, todos los ámbitos asociados a ella están abiertos para ella.
- **Bloque cerrado.** Dada una línea de código, todos los bloques no abiertos para esa línea se consideran cerrados para ella.
- **Profundidad de un bloque.** La profundidad de un bloque se define de la siguiente manera recursiva:
  - — El bloque más externo tiene profundidad 0.
  - — Al abrir un bloque, su profundidad es igual a uno más la profundidad del bloque en el que se abre.
- **Bloque actual.** En cada situación concreta, el ámbito actual es el más profundo de los abiertos.
- **Identificadores activos** en un ámbito concreto. Se entenderá por identificador activo el que está definido y es accesible en un bloque.

- **Identificador global o local.** Estos dos términos se utilizan cuando hay al menos dos bloques, uno incluido en el otro. El término local se refiere a los identificadores definidos sólo en el bloque más interno y, por tanto, inaccesibles desde el bloque que lo incluye. El término global se aplica a los identificadores activos en el bloque externo, que desde el punto de vista del bloque interno estaban ya definidos cuando dicho bloque se abrió. También se usará el término global para situaciones similares a ésta.

Aunque los diferentes lenguajes de programación pueden seguir criterios distintos, es frecuente usar las siguientes reglas:

- En un punto concreto de un programa sólo están activos los identificadores definidos en el ámbito actual y los definidos en los ámbitos abiertos en ese punto del programa.
- En general, las coincidencias de nombres (cuando el nombre de un identificador del bloque actual coincide con el de otro u otros de algún bloque abierto) se resuelven a favor del bloque actual; es decir, prevalece la definición del bloque actual, que *oculta* las definiciones anteriores, haciendo inaccesibles los demás identificadores que tienen el mismo nombre.
- En las subrutinas, los nombres de sus argumentos son locales a ella, es decir, no son accesibles fuera de la misma. El nombre de la subrutina es local al bloque en que se definió y global para la subrutina.

Hay muchas maneras de organizar la tabla de símbolos para gestionar programas escritos en lenguajes con estructura de bloques. A continuación se describirán las dos posibilidades que podrían considerarse extremas:

- Uso de una tabla de símbolos distinta para cada ámbito.
- Uso de una tabla de símbolos para todos los ámbitos.

Los algoritmos de la tabla también dependen de otros factores de diseño del compilador: por ejemplo, si basta realizar una pasada, o si el compilador necesitará más de un paso por el programa fuente.

2.4.2. Uso de una tabla por ámbito

En este caso, para gestionar correctamente los identificadores es necesaria una colección de tablas hash, una para cada ámbito. Lo importante es que se mantenga el orden de apertura de los ámbitos abiertos.

Compiladores de un paso

Es la situación más sencilla. En este caso, los ámbitos no se consultan una vez que se cierran, por lo que pueden descartarse sus tablas hash. En esta circunstancia, se puede utilizar una pila de ámbitos abiertos. Esta estructura de datos devuelve primero los elementos insertados en ella más recientemente, por lo que se mantiene automáticamente el orden de apertura de los ámbitos.

La Figura 2.23 muestra un ejemplo del uso de esta técnica con un programa.

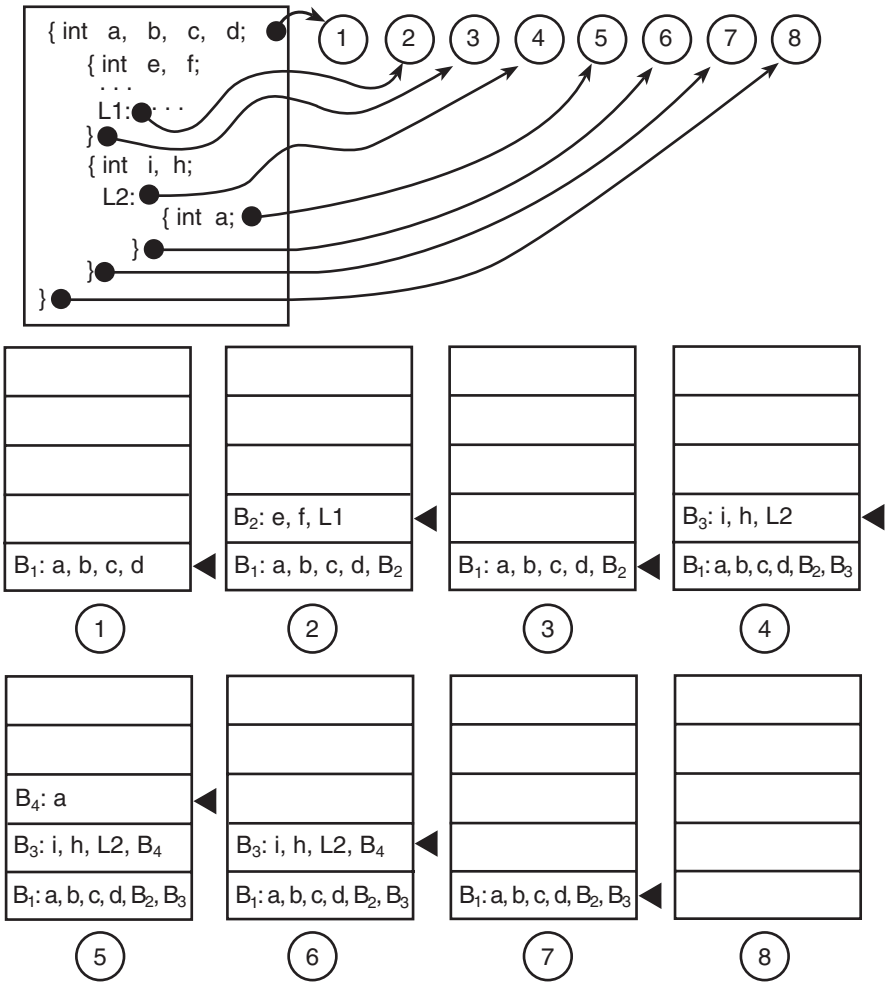


Figura 2.23. Ejemplo de tabla de símbolos de un programa escrito con un lenguaje con estructura de bloques. La tabla de símbolos utiliza una tabla hash para cada ámbito; el compilador sólo realiza un paso. En la pila de tablas hash se señala la cima.

La **inserción** de una nueva clave se realiza en la tabla correspondiente al ámbito actual (el que ocupa la cima de la pila).

La **búsqueda** de una clave es la operación que más se complica, ya que, si el identificador no ha sido declarado en el ámbito actual (no pertenece a su tabla hash), es necesario recorrer la pila completa, hasta el ámbito exterior, para asegurar que dicho identificador no ha sido declarado en el programa y, por tanto, no puede ser utilizado.

La **gestión de los bloques** se realiza así:

1. Cuando se **abre un nuevo bloque:**

- Se añade su nombre, si lo tiene, como identificador en el bloque actual, antes de abrir el nuevo bloque, ya que los nombres de las subrutinas tienen que ser locales al bloque donde se declaran.
- Se crea una nueva tabla hash para el bloque nuevo.
- Se inserta en la pila (*push*) la nueva tabla hash, que pasa a ser la del ámbito actual.
- Se inserta en el ámbito actual el nombre del nuevo bloque, ya que los nombres de las subrutinas son globales a la propia subrutina.

2. Cuando se **cierra un bloque:**

   - Se saca de la pila (*pop*) la tabla hash del ámbito actual y se elimina dicha tabla.

Compiladores de más de un paso

El criterio general es el mismo que en el caso anterior, pero se necesita conservar las tablas hash de los bloques cerrados, por si se requiere su información en pasos posteriores.

Un esquema fácil de describir consiste en modificar la pila del apartado anterior para convertirla en una lista, que conserve juntas, por encima de los ámbitos abiertos, las tablas hash de los ámbitos cerrados. De esta manera, el ámbito actual estará siempre por debajo de los cerrados. Por debajo de él, se encontrará la misma pila descrita anteriormente. Es necesario añadir la información necesaria para marcar los bloques como abiertos o cerrados.

La gestión descrita en el apartado anterior sólo cambia en lo relativo a los ámbitos cerrados: cuando **un ámbito se cierra**, se marca como cerrado. Es fácil imaginar que la pila necesitará de ciertos datos adicionales (al menos, un apuntador al ámbito actual) para su gestión eficiente. La Figura 2.24 muestra una tabla hash de este tipo, para el mismo programa fuente del ejemplo de la Figura 2.23.

2.4.3. Evaluación de estas técnicas

Entre los inconvenientes de estas técnicas se pueden mencionar los siguientes:

- Se puede fragmentar en exceso el espacio destinado en el compilador a la tabla de símbolos, lo que origina cierta ineficiencia en cuanto al espacio utilizado.
- La búsqueda, que implica la consulta de varias tablas, puede resultar ineficiente en cuanto al tiempo utilizado.

2.4.4. Uso de una sola tabla para todos los ámbitos

Este enfoque pretende minimizar el efecto de los inconvenientes detectados en el apartado anterior. Es evidente que, una vez que se conoce qué tratamiento tiene que darse a los identificadores de los programas escritos con lenguajes con estructura de bloques, es posible implementar sus

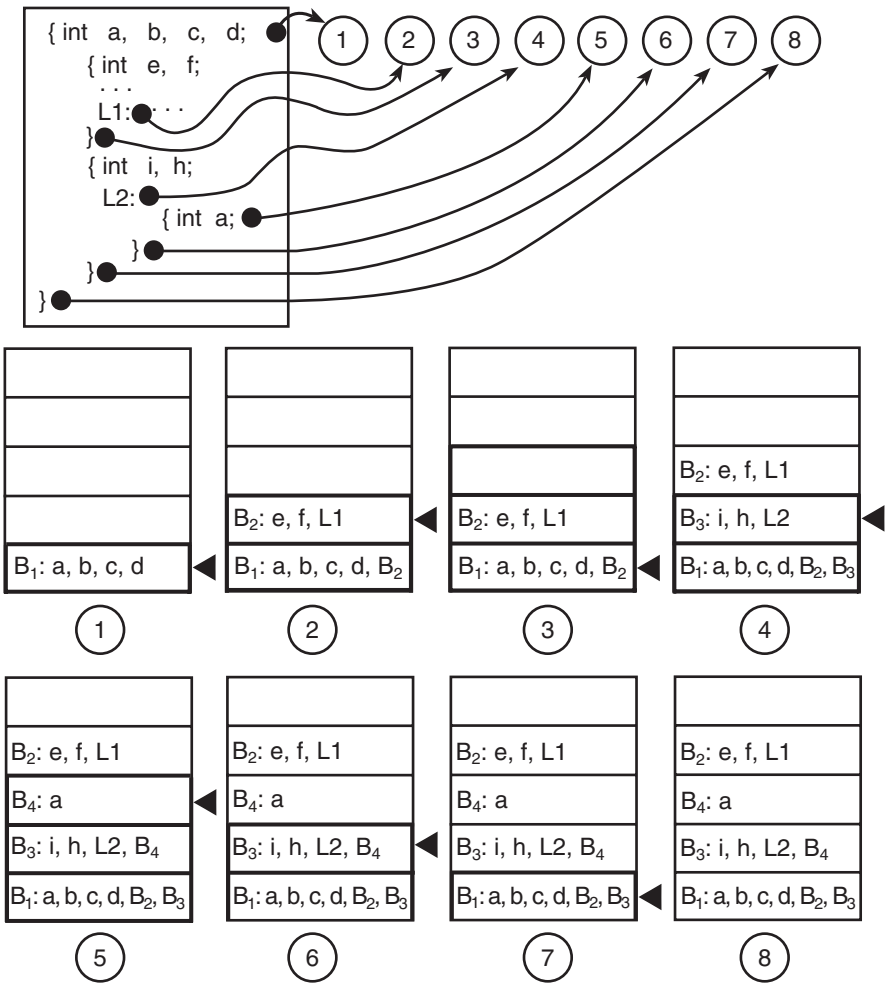


Figura 2.24. Ejemplo de tabla de símbolos de un programa escrito con un lenguaje con estructura de bloques. La tabla de símbolos utiliza una tabla hash para cada ámbito; el compilador realiza más de un paso. En la pila de tablas hash se señala la cima. Los ámbitos abiertos están rodeados por un recuadro más grueso que los cerrados.

tablas de símbolos utilizando una sola tabla para todos los bloques. A continuación se mencionan los aspectos más relevantes que hay que tener en cuenta:

1. Habrá que mantener información, por un lado sobre los bloques, y por otro sobre los identificadores.
2. De cada bloque se tiene que guardar, al menos, la siguiente información:
   - Identificación del bloque.
   - Apuntador al bloque en el que se declaró.
   - Apuntador al espacio donde se guardan sus identificadores.

No se describirán más detalles de esta técnica, ya que su implementación es sólo un problema de programación.

2.5 Información adicional sobre los identificadores en las tablas de símbolos

De la definición del lenguaje de programación utilizado depende la información que hay que almacenar en la tabla de símbolos para el tratamiento correcto del programa:

- Clase del identificador, para indicar a qué tipo de objeto se refiere el identificador. Por ejemplo, podría ser una variable, función o procedimiento, una etiqueta, la definición de un tipo de dato, el valor concreto de una enumeración, etc.
- Tipo, para indicar el tipo de dato. Por ejemplo: entero, real, lógico, complejo, carácter, cadena de caracteres, tipo estructurado, tipo declarado por el programador, subrutina que devuelve un dato, subrutina que no devuelve dato alguno, operador, etc.

2.6 Resumen

Uno de los objetivos de este capítulo es la justificación de la elección de las tablas de dispersión o hash para la implementación de la tabla de símbolos de los compiladores e intérpretes. En primer lugar se repasa, de manera informal e intuitiva, la complejidad temporal de los algoritmos de búsqueda más utilizados (lineal y binaria sobre vectores de datos y los específicos de árboles binarios ordenados y árboles AVL). Se muestra cómo la comparación de claves limita el rendimiento de una manera inaceptable y se justifica el uso de las tablas hash, de las que se describen con más detalle diferentes variantes. Debido a la complejidad de la teoría en la que se basan estos resultados y que el ámbito de este libro no presupone al lector ningún conocimiento específico de la materia, se ha pretendido, siempre que ha sido posible, acompañar cada resultado con una justificación intuitiva y convincente que supla la ausencia de la demostración formal.

El capítulo termina con la descripción de dos aspectos prácticos propios del uso que los compiladores e intérpretes dan a la tabla hash: las tablas de símbolos para los lenguajes de programación que tienen estructura de bloques y la información adicional que se necesita conservar en la tabla de símbolos sobre los identificadores.

2.7 Ejercicios y otro material práctico

El lector encontrará en http://www.librosite.net/pulido abundante material práctico sobre el contenido de este capítulo con ejercicios resueltos y versiones ejecutables de los algoritmos descritos.

2.8 Bibliografía

[1] Knuth, D. E. (1997): *The art of computer programming,* Addison Wesley Longman.

[2] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L., y Stein, C. (2001): *Introduction to algorithms,* The MIT Press, McGraw-Hill Book Company.

[3] McKenzie, B. J.; Harries R. y Bell, T. C. (1990): "Selecting a hashing algorithm", *Software - Practice and Experience*, 20(2), 209-224.

Capítulo 3

Análisis morfológico

3.1 Introducción

El analizador morfológico, también conocido como analizador léxico (*scanner,* en inglés) se encarga de dividir el programa fuente en un conjunto de unidades sintácticas (*tokens,* en inglés). Una unidad sintáctica es una secuencia de caracteres con cohesión lógica. Ejemplos de unidades sintácticas son los identificadores, las palabras reservadas, los símbolos simples o múltiples y las constantes (numéricas o literales).

Para llevar a cabo esta división del programa en unidades sintácticas, el analizador morfológico utiliza un subconjunto de las reglas de la gramática del lenguaje en el que está escrito el programa que se va a compilar. Este subconjunto de reglas corresponde a un lenguaje regular, es decir, un lenguaje definido por expresiones regulares.

El analizador morfológico lleva a cabo también otra serie de tareas auxiliares como el tratamiento de los comentarios y la eliminación de blancos y símbolos especiales (caracteres de tabulación y saltos de línea, entre otros).

La Tabla 3.1 muestra las unidades sintácticas de un lenguaje ejemplo y la Figura 3.1 muestra la gramática independiente del contexto que usará el analizador morfológico para identificar las unidades sintácticas de dicho lenguaje.

Un analizador morfológico es un autómata finito determinista que reconoce el lenguaje generado por las expresiones regulares correspondientes a las unidades sintácticas del lenguaje fuente. En las secciones siguientes se describe cómo programar manualmente dicho autómata mediante un proceso que comprende los siguientes pasos:

Tabla 3.1. Unidades sintácticas de un lenguaje ejemplo.

Palabras reservadas	Símbolos simples	Símbolos dobles	Otros

begin
end
bool
int
ref
function
if
then
fi
else
while
do
input
output
deref
true
false
	;
	,
	+
	–
	*
	(
	=
	>
		:=
		<=
			número (uno o más dígitos)
			
			identificador (una letra seguida de 0 o más letras y/o dígitos)

1. Construir el Autómata Finito No Determinista (AFND) correspondiente a una expresión regular.
2. Transformar el AFND obtenido en el paso 1 en un Autómata Finito Determinista (AFD).
3. Minimizar el número de estados del AFD obtenido en el paso 2.
4. Implementar en forma de código el AFD obtenido en el paso 3.

```
<US>          ::= <p_reservada> | <s_simple> | <s_doble> | <id> | <cte_num>
<p_reservada> ::= begin | end | bool | int | ref | function | if | then | fi | else | while
                  | do | repeat | input | output | deref | true | false
<s_simple>    ::= ; | , | + | – | * | ( | ) | = | >
<s_doble>     ::= := | <=
<cte_num>     ::= <digito> | <cte_num> <digito>
<id>          ::= <letra> | <letra><resto_id>
<resto_id>    ::= <alfanumerico> | <alfanumerico><resto_id>
<alfanumerico> ::= <digito> | <letra>
<digito>      ::= 0 | 1 | ... | 9
<letra>       ::= a | b | ... | z | A | B | ... | Z
```

Figura 3.1. Gramática para las unidades sintácticas del lenguaje ejemplo.

También es posible implementar el autómata correspondiente al analizador morfológico utilizando una herramienta de generación automática como la que se describe en la última sección del capítulo.

3.2 Expresiones regulares

Una expresión regular es una forma abreviada de representar cadenas de caracteres que se ajustan a un determinado patrón. Al conjunto de cadenas representado por la expresión `r` se lo llama *lenguaje generado por la expresión regular* `r` y se escribe `L(r)`.

Una expresión regular se define sobre un alfabeto Σ y es una cadena formada por caracteres de dicho alfabeto y por una serie de operadores también llamados metacaracteres.

Las expresiones regulares básicas se definen de la siguiente forma:

1. El símbolo Φ (conjunto vacío) es una expresión regular y $L(\Phi) = \{\}$
2. El símbolo λ (palabra vacía) es una expresión regular y $L(\lambda) = \{\lambda\}$
3. Cualquier símbolo $a \in \Sigma$ es una expresión regular y $L(a) = \{a\}$

A partir de estas expresiones regulares básicas pueden construirse expresiones regulares más complejas aplicando las siguientes operaciones:

1. *Concatenación* (se representa con el metacarácter `.`)

   Si `r` y `s` son expresiones regulares, entonces `r.s` también es una expresión regular y `L(r.s)=L(r).L(s)`. El operador `.` puede omitirse de modo que `rs` también representa la concatenación.
   La concatenación de dos lenguajes L_1 y L_2 se obtiene concatenando cada cadena de L_1 con todas las cadenas de L_2. Por ejemplo, si $L_1 = \{00 , 1\}$ y $L_2 = \{11 , 0 , 10\}$, entonces $L_1L_2 = \{0011,000,0010,111,10,110\}$.

2. *Unión* (se representa con el metacarácter `|`)

   Si `r` y `s` son expresiones regulares, entonces `r | s` también es una expresión regular y `L(r | s) = L(r)` $\cup$ `L(s)`. Por ejemplo, el lenguaje generado por la expresión regular `ab | c` es `L(ab | c) = {ab , c}`.

3. *Cierre* o *clausura* (se representa con el metacarácter `*`)

   Si `r` es una expresión regular, entonces `r*` también es una expresión regular y `L(r*) = L(r)*`.

   La operación de cierre aplicada a un lenguaje `L` se define así:

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

donde L^i es igual a la concatenación de L consigo mismo `i` veces y $L^0 = \lambda$. Por ejemplo, el lenguaje generado por la expresión regular `a*ba*` es `L(a*ba*) = {b , ab , ba , aba , aab , ...}`, es decir, el lenguaje formado por todas las cadenas de `a`'s y `b`'s que contienen una única `b`.

Cuando aparecen varias operaciones en una expresión regular, el orden de precedencia es el siguiente: cierre, concatenación y unión. Este orden puede modificarse mediante el uso de paréntesis.

3.3 Autómata Finito No Determinista (AFND) para una expresión regular

Intuitivamente un autómata finito consta de un conjunto de estados y, partiendo de un estado inicial, realiza transiciones de un estado a otro en respuesta a los símbolos de entrada que procesa. Cuando el autómata alcanza un estado de los que se denominan finales, se dice que ha reconocido la palabra formada por concatenación de los símbolos de entrada procesados.

Un autómata finito puede ser determinista o no determinista. La expresión «no determinista» significa que desde un mismo estado puede haber más de una transición etiquetada con el mismo símbolo de entrada.

Un autómata finito no determinista es una quíntupla $(\Sigma, Q, \delta, q0, F)$, donde

1. Σ es un conjunto finito de símbolos de entrada o *alfabeto.*
2. `Q` es un conjunto finito de estados.
3. δ es la *función de transición* que recibe como argumentos un estado y un símbolo de entrada o el símbolo λ y devuelve un subconjunto de `Q`.
4. $q0 \in Q$ es el *estado inicial.*
5. $F \subseteq Q$ es el conjunto de *estados finales.*

Un autómata finito puede representarse mediante lo que se conoce como *diagrama de transición,* que es un grafo dirigido construido de la siguiente forma:

- Cada nodo está etiquetado con un elemento de `Q`.
- Si $\delta(p,a) = q$, se dibuja un arco del nodo con etiqueta `p` al nodo con etiqueta `q` etiquetado con el símbolo `a`.
- El estado inicial aparece señalado con una flecha sin origen.
- Los estados finales aparecen marcados con un doble círculo.

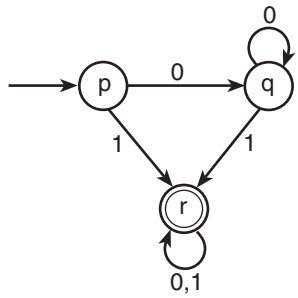


Figura 3.2. Diagrama de transición para un autómata.

La Figura 3.2 muestra el diagrama de transición para el autómata finito determinista ({0,1},{p,q,r},δ,p,{q}), donde δ está definida de la siguiente forma:

δ(p,0)=q	δ(p,1)=r	δ(q,0)=q
δ(q,1)=r	δ(r,0)=r	δ(r,1)=r

Para toda expresión regular `e` es posible construir un autómata finito no determinista que acepte el lenguaje generado por dicha expresión regular. El algoritmo es recursivo y consta de los siguientes pasos:

1. Si `e` = Φ, el autómata correspondiente es el que aparece en la Figura 3.3(a).
2. Si `e` = λ, el autómata correspondiente es el que aparece en la Figura 3.3(b).
3. Si `e` = `a`, `a` ∈ Σ, el autómata correspondiente es el que aparece en la Figura 3.3(c).

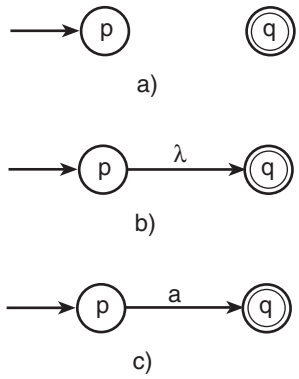


Figura 3.3. Autómatas para expresiones regulares básicas.

4. Si `e` = `r` | `s` y tenemos los autómatas correspondientes a `r` y `s`, que representaremos como aparecen en la Figura 3.4, el autómata correspondiente a la expresión `r` | `s` es el que aparece en la Figura 3.5(a).

Figura 3.4. Autómatas correspondientes a las expresiones `r` y `s`.

5. Si `e = rs` y tenemos los autómatas correspondientes a `r` y `s`, que representaremos como aparecen en la Figura 3.4, el autómata correspondiente a la expresión `rs` es el que aparece en la Figura 3.5(b).

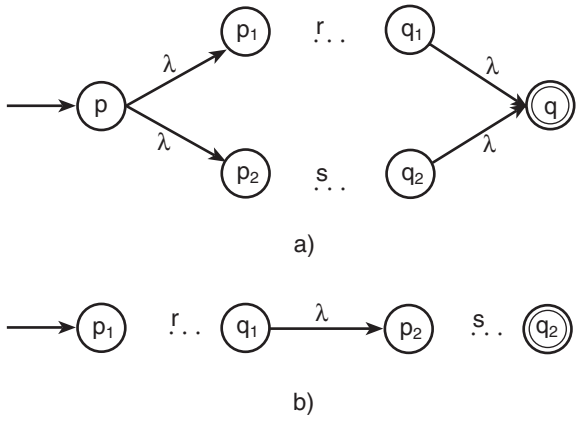


Figura 3.5. Autómatas correspondientes a las expresiones `r|s` y `rs`.

6. Si `e = r*` y tenemos el autómata correspondiente a `r`, que representaremos como aparece en la Figura 3.6(a), el autómata correspondiente a la expresión `r*` es el que aparece en la Figura 3.6(b).

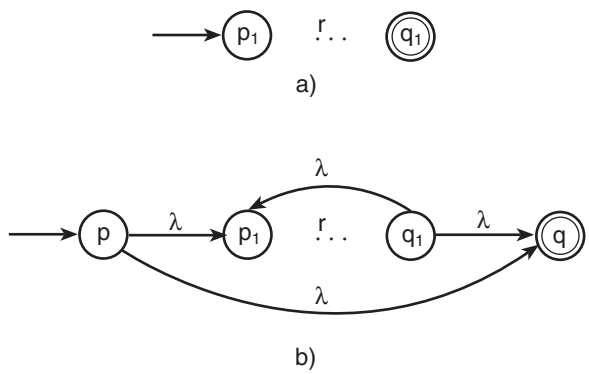


Figura 3.6. Autómatas correspondientes a las expresiones `r` y `r*`.

Como ejemplo, consideremos las reglas que definen las constantes numéricas en la gramática de la Figura 3.1, que son las siguientes:

```
<cte_num> ::= <digito> | <cte_num> <digito>
```

Para obtener la expresión regular correspondiente a estas reglas hay que construir primero el autómata que reconoce el lenguaje generado por la gramática y, en un segundo paso, obtener la expresión regular equivalente al autómata. En [1] se describen en detalle los algoritmos necesarios, el primero de los cuales sólo es aplicable a gramáticas tipo 3.

Aplicando este proceso a las reglas que definen las constantes numéricas se obtiene la expresión regular `digito.digito*`. Esta expresión es el resultado de concatenar dos expresiones regulares: `digito` y `digito*`. Aplicando a esta expresión el paso 3 del algoritmo recursivo descrito anteriormente, se obtiene el AFND para la expresión `digito` [*véase* Figura 3.7(a)]. A partir de este AFND, y aplicando el paso 6 de dicho algoritmo, se obtiene el AFND de la Figura 3.7(b). Por último, aplicando el paso 5, se obtiene el AFND para la expresión completa, que aparece en la Figura 3.7(c).

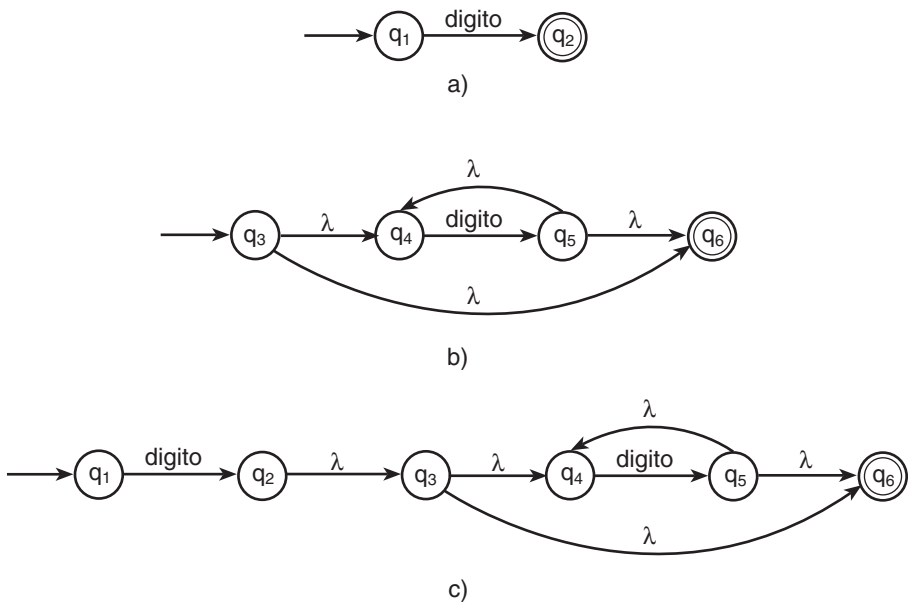


Figura 3.7. AFND para la expresión regular `digito.digito*`.

3.4 Autómata Finito Determinista (AFD) equivalente a un AFND

Un autómata finito determinista es una quíntupla `(Σ, Q, δ, q0, F)`, donde

1. Σ es un conjunto finito de símbolos de entrada o *alfabeto.*
2. `Q` es un conjunto finito de estados.

3. δ es la *función de transición* que recibe como argumentos un estado y un símbolo de entrada y devuelve un estado.
4. q0 $\in$ Q es el *estado inicial.*
5. F $\subseteq$ Q es el conjunto de estados finales.

La *función de transición extendida* recibe como argumentos un estado p y una cadena de caracteres wy devuelve el estado que alcanza el autómata cuando parte del estado p y procesa la cadena de caracteres w.

Dado un autómata finito no determinista N = (Σ, Q, f, q0, F), siempre es posible construir un autómata finito determinista D = (Σ, Q´, f´, q0´, F´) equivalente (que acepte el mismo lenguaje). Para construir dicho autómata seguiremos el siguiente procedimiento:

- Cada estado de D corresponde a un subconjunto de los estados de N. En el autómata de la Figura 3.7(c) los subconjuntos {q1}, {q3, q4} o {q2, q4, q6} serían posibles estados del autómata finito determinista equivalente.
- El estado inicial q0´ de D es el resultado de calcular el *cierre* λ del estado inicial q0 de N. El cierre λ de un estado e se representa como $\overline{e}$ y se define como el conjunto de estados alcanzables desde e mediante cero o más transiciones λ. En el autómata de la Figura 3.7(c) el cierre λ de cada uno de los estados son las siguientes:

  $\overline{q1}$ = {q1} $\quad\quad$ $\overline{q4}$ = {q4}
  $\overline{q2}$ = {q2, q3, q4, q6} $\quad\quad$ $\overline{q5}$ = {q5, q4, q6}
  $\overline{q3}$ = {q3, q4, q6} $\quad\quad$ $\overline{q6}$ = {q6}

  Por lo tanto, el estado inicial del AFD correspondiente al AFND de la Figura 3.7(c) será {q1}.

- Desde un estado P de D habrá una transición al estado Q con el símbolo a del alfabeto. Para calcular esta transición calculamos primero un conjunto intermedio P_a formado por los estados q de N tales que para algún p en P existe una transición de p a q con el símbolo a. El estado Q se obtiene calculando el cierre λ del conjunto P_a.
  Veamos esto con un ejemplo. Partiendo del AFND de la Figura 3.7(c), la transición desde el estado inicial {q1} con el símbolo digito se calcularía de la siguiente forma:

  $\{q1\}_{digito}$ = {q2}
  $\{\overline{q1}\}_{digito}$ = {q2, q3, q4, q6}

  Puesto que δ(q4,digito)=q5, la transición desde el estado {q2,q3,q4,q6} con el símbolo digito será:

  $\{q2, q3, q4, q6\}_{digito}$ = {q5}
  $\{\overline{q5}\}_{digito}$ = {q5, q4, q6}

  Puesto que δ(q4,digito)=q5, la transición desde el estado {q5,q4,q6} con el símbolo digito será:

  $\{q5, q4, q6\}_{digito}$= {q5}
  $\{\overline{q5}\}_{digito}$ = {q5, q4, q6}

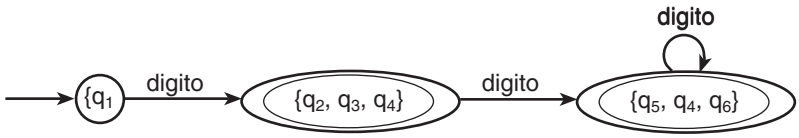


Figura 3.8. Autómata finito determinista correspondiente al AFND de la Figura 3.7.

- En el autómata finito determinista `D` un estado será final si contiene algún estado final del AFND `N`. En el AFD correspondiente al autómata de la Figura 3.7(c), serán estados finales todos aquellos que contengan el estado `q6`.

La Figura 3.8 muestra el AFD equivalente al AFND de la Figura 3.7(c).

3.5 Autómata finito mínimo equivalente a uno dado

Recordemos que el objetivo que se persigue es obtener un autómata finito que sirva para implementar un analizador morfológico, es decir, que acepte las cadenas correspondientes a las unidades sintácticas del lenguaje fuente que se va a compilar. Por este motivo, el analizador morfológico será tanto más eficiente cuanto menor sea el número de estados del autómata finito correspondiente.

Para cualquier autómata finito, existe un autómata finito mínimo equivalente. El primer paso para obtener este autómata mínimo es identificar pares de estados *equivalentes*. Decimos que los estados `p` y `q` son equivalentes si para toda cadena `w`, $\overline{\delta}$`(p,w)`, es un estado final si y sólo si $\overline{\delta}$`(q,w)` es un estado final. La relación «equivalente» es una relación de equivalencia que establece clases de equivalencia en el conjunto de estados de un autómata finito.

Dos estados son equivalentes si no son *distinguibles*. Podemos calcular los pares de estados distinguibles en un AFD mediante el *algoritmo por llenado de tabla*. Este algoritmo realiza una búsqueda recursiva de pares distinguibles aplicando las siguientes reglas:

1. Si `p` es un estado final y `q` no lo es, el par `{p,q}` es distinguible.
2. Si para dos estados `p` y `q` se cumple que existe una transición de `p` a `r` con el símbolo `a` y una transición de `q` a `s` con el símbolo `a`, y los estados `r` y `s` son distinguibles, entonces el par `{p,q}` es distinguible.

Como ejemplo, consideremos el autómata de la Figura 3.9, idéntico al de la Figura 3.8, salvo que se han renombrado los estados para simplificar. Si aplicamos la regla 1 a dicho autómata, los estados `{1, 2}` y `{1, 3}` son distinguibles.

Por lo tanto, sólo es necesario averiguar si los estados `{2, 3}` son distinguibles. Aplicando la regla 2 a estos estados, se cumple que existe una transición de `2` a `3` con el símbolo `digito` y una transición de `3` a `3` con el símbolo `digito`, pero los estados `3` y `3` no son distinguibles porque son el mismo estado. Por lo tanto, los estados `{2, 3}` no son distinguibles, es decir, son equivalentes.

Figura 3.9. Autómata finito determinista de la Figura 3.8 con estados renombrados.

Dado un autómata finito determinista `A`, el algoritmo para construir un autómata mínimo equivalente `B` puede enunciarse de la siguiente forma:

1. Cada clase de equivalencia establecida por la relación «equivalente» en el conjunto de estados de `A` es un estado de `B`.
2. El estado inicial de `B` es la clase de equivalencia que contiene el estado inicial de `A`.
3. El conjunto de estados finales de `B` es el conjunto de clases de equivalencia que contienen estados finales de `A`.
4. Sea γ la función de transición de `B`. Si `S` y `T` son bloques de estados equivalentes de `A` y `a` es un símbolo de entrada γ`(S,a) = T` si se cumple que para todos los estados `q` de `S`, δ`(q,a)` pertenece al bloque `T`.

Apliquemos este algoritmo al autómata `A` de la Figura 3.9.

1. Las clases de equivalencia establecidas por la relación «equivalente» en el conjunto de estados de `A` son `{1}` y `{2,3}`. Éstos serán los estados del autómata mínimo `B`.
2. El estado inicial de `B` es el bloque `{1}`.
3. El autómata `B` sólo tiene un estado final que es el bloque `{2,3}`, porque contiene los estados `2` y `3`, que son estados finales en `A`.
4. En el autómata `A` hay tres transiciones, todas ellas con el símbolo `digito`:
   - Del estado `1` al `2`. Pasa a ser una transición del bloque `{1}` al `{2,3}`.
   - Del estado `2` al `3`. Pasa a ser una transición del bloque `{2,3}` al `{2,3}`.
   - Del estado `3` al `3`. Pasa a ser una transición del bloque `{2,3}` al `{2,3}`.

   Las dos últimas transiciones son redundantes, por lo que sólo dejaremos una de ellas.

La Figura 3.10 muestra el autómata mínimo equivalente al autómata de la Figura 3.9.

Figura 3.10. Autómata finito determinista mínimo equivalente al de la Figura 3.9.

3.6 Implementación de autómatas finitos deterministas

El primer paso para implementar un autómata finito que sea capaz de reconocer las unidades sintácticas del lenguaje fuente que se va a compilar es identificar las expresiones regulares que representan dichas unidades sintácticas. Un problema que puede surgir es que determinadas expresiones regulares den lugar a que no exista una única forma de dividir la cadena de entrada en unidades sintácticas. Por ejemplo, utilizando la expresión regular `digito.digito*` para representar constantes numéricas, existirían varias formas de dividir la cadena de entrada `381` en unidades sintácticas:

- Dos unidades sintácticas: `3` y `81`
- Dos unidades sintácticas: `38` y `1`
- Una unidad sintáctica: `381`

Para resolver esta ambigüedad, se utiliza la regla conocida como *principio de la subcadena más larga*, que consiste en identificar siempre como siguiente unidad sintáctica la cadena de caracteres más larga posible. Si aplicamos este principio, la cadena `381` se identificaría como una única unidad sintáctica.

Para implementar este principio, puede añadirse al autómata una nueva transición con la etiqueta `otro`; *véase* la Figura 3.11. En esta figura la etiqueta `otro` aparece entre corchetes para indicar que, aunque en el caso general los autómatas avanzan una posición en la cadena de entrada cuando hacen una transición, en este caso el autómata leerá el carácter de entrada, pero sin avanzar una posición.

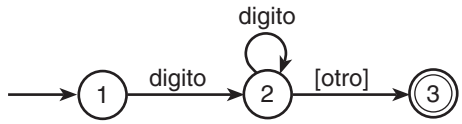


Figura 3.11. Autómata finito determinista con transición con la etiqueta `otro`.

Existen diversas formas de implementar mediante código un autómata finito. Una de ellas es utilizar el pseudocódigo que aparece en la Figura 3.12, en el que se utilizan las siguientes estructuras de datos:

- `transición`: vector de dos dimensiones indexado por estados y caracteres, que representa la función de transición del autómata.
- `final`: vector booleano de una dimensión indexado por estados, que representa los estados finales del autómata.
- `error`: vector de dos dimensiones indexado por estados y caracteres, que representa las casillas vacías en la tabla de transición.
- `avanzar`: vector booleano de dos dimensiones indexado por estados y caracteres, que representa las transiciones que avanzan en la entrada.

```
estado := estado inicial;
ch := siguiente carácter de entrada;
while not final[estado] and not error[estado,ch]
 do
            estado := transición[estado,ch];
            if avanzar[estado,ch] then
             ch := siguiente carácter de entrada;
 end
if final[estado] then aceptar;
```

Figura 3.12. Pseudocódigo que implementa un autómata finito.

En http://www.librosite.net/pulido se incluye una versión ejecutable del pseudocódigo de la Figura 3.12.

3.7 Otras tareas del analizador morfológico

Además de dividir el programa fuente en unidades sintácticas, el analizador morfológico suele llevar a cabo otras tareas auxiliares que facilitan la tarea posterior del analizador sintáctico. Una de estas tareas es la de eliminar ciertos caracteres delimitadores, como espacios en blanco, tabuladores y saltos de línea. La siguiente expresión regular representa la aparición de uno o más de estos caracteres delimitadores.

```
(blanco|tab|nuevalinea)(blanco|tab|nuevalinea)*
```

El analizador morfológico puede también encargarse de eliminar los comentarios. Consideremos un formato para comentarios como el que se utiliza en el lenguaje de programación C, es decir, cadenas de caracteres de longitud variable delimitadas por los caracteres `/*` y `*/`. Aunque es difícil encontrar una expresión regular que represente este formato, sí es posible construir un autómata finito como el que aparece en la Figura 3.13, que identifica este tipo de comentarios.

Aunque, para el analizador morfológico, una unidad sintáctica no es más que una secuencia de caracteres, cada unidad sintáctica tiene asociada una información semántica que será utiliza-

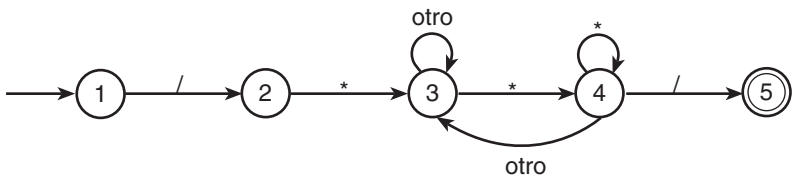


Figura 3.13. Autómata finito para comentarios tipo C.

da por el resto de los componentes del compilador. Esta información semántica se almacena en forma de *atributos* de la unidad sintáctica y se verá en detalle en el capítulo sobre el análisis semántico. En la fase de análisis morfológico, es posible calcular el valor de algunos de estos atributos como, por ejemplo, el valor de una constante numérica, o la cadena de caracteres concreta que forma el nombre de un identificador.

3.8 Errores morfológicos

El analizador morfológico puede detectar determinados tipos de error, entre los que se encuentran los siguientes:

- Símbolo no permitido, es decir, que no pertenece al alfabeto del lenguaje fuente. Por ejemplo, en la gramática de la Figura 3.1 no aparece el signo <, y sería un error morfológico que dicho símbolo apareciera en un programa fuente escrito en dicho lenguaje.
- Identificador mal construido o que excede de la longitud máxima permitida. En un lenguaje en el que el primer carácter de un identificador deba ser una letra, un identificador que comience con un dígito sería un ejemplo de este tipo de error.
- Constante numérica mal construida o que excede de la longitud máxima permitida. Por ejemplo, si el lenguaje fuente acepta números en punto fijo que constan de una parte entera y una parte decimal separadas por un punto, una constante numérica en la que se omitiera la parte decimal podría ser un error morfológico.
- Constante literal mal construida. Un ejemplo de este tipo de error sería el literal `'Pepe`, al que le falta la comilla de cierre.

Existen otros tipos de error que el analizador morfológico no será capaz de detectar. Por ejemplo, si en la entrada aparece la cadena `bgein`, el analizador morfológico lo reconocerá como un identificador, cuando probablemente se trate de la palabra reservada `begin` mal escrita.

Habitualmente, cuando el analizador morfológico detecta un error en la entrada, emite un mensaje de error para el usuario y detiene la ejecución. Un comportamiento alternativo es intentar recuperarse del error y continuar con el procesamiento del fichero de entrada. Las estrategias de recuperación de errores son variadas y se basan en la inserción, eliminación o intercambio de determinados caracteres.

Como ejemplo de recuperación de errores, se podrían incorporar al analizador morfológico algunas expresiones regulares más, que correspondan a unidades sintácticas erróneas que es probable que aparezcan en la entrada. Por ejemplo, si el lenguaje fuente acepta números en punto fijo, que corresponden a la expresión `digito*'.'digito.digito*`, podemos añadir la expresión regular `digito*'.'` para que recoja los números en punto flotante erróneos a los que les falte la parte decimal. De esta forma, cuando el analizador morfológico detecte que la entrada corresponde a esta unidad sintáctica errónea, además de mostrar un mensaje de error dirigido al usuario, puede añadir, por ejemplo, el dígito `0` a la unidad sintáctica, para transformarla en otra correcta.

3.9 Generación automática de analizadores morfológicos: la herramienta lex

Existen herramientas que generan analizadores morfológicos de forma automática. Una de las más utilizadas se llama `lex`. `Lex` recibe como entrada un fichero de texto con extensión `.l`, que contiene las expresiones regulares que corresponden a las unidades sintácticas del lenguaje que se va a compilar, al que llamaremos *fichero de especificación lex*. Como resultado del proceso del fichero de especificación, `lex` genera un fichero en código C, llamado `lex.yy.c`. Este fichero contiene una función llamada `yylex()`, que implementa el analizador morfológico que reconoce las unidades sintácticas especificadas en el fichero de entrada.

3.9.1. Expresiones regulares en lex

Al igual que en la notación general para expresiones regulares descrita en la Sección 3.2, `lex` utiliza los metacaracteres **|** y ***** para representar las operaciones de unión y cierre, respectivamente. Para representar la operación de concatenación en `lex` no se utiliza ningún meta-carácter específico: basta con escribir las expresiones regulares correspondientes de forma consecutiva. Además, en las expresiones regulares en `lex` pueden utilizarse otros metacaracteres que se describen a continuación.

El metacarácter **.** representa cualquier carácter, excepto el salto de línea `"\n"`. Por ejemplo, la expresión regular `.*0.*` representa todas las cadenas que contienen al menos un `0`.

Los corchetes **[ ]** y el guión **–** se utilizan para representar rangos de caracteres. Por ejemplo, la expresión `[a-z]` representa las letras minúsculas, y la expresión `[0-9]` representa los dígitos del `0` al `9`. Los corchetes también pueden utilizarse para representar alternativas individuales, de modo que la expresión `[xyz]` representa una «x», una «y» o una «z», y es equivalente a la expresión `x|y|z`.

El metacarácter **+** indica una o mas apariciones de la expresión que lo precede. Utilizando los metacaracteres vistos hasta ahora, podríamos representar las constantes numéricas que aparecen en la gramática de la Figura 3.1 mediante la expresión regular `[0-9]+`.

El metacarácter **~** representa cualquier carácter que no esté en un conjunto dado. Por ejemplo, la expresión `~0` representa cualquier carácter que no sea el dígito `0`.

El metacarácter **^** tiene un significado similar, combinado con los corchetes. Por ejemplo, la expresión `[^xyz]` representa cualquier carácter que no sea «x», ni «y» ni «z», y es equivalente a la expresión `~(x|y|z)`.

El metacarácter **?** sirve para indicar que una parte de una expresión es opcional. Por ejemplo, la expresión $(+|-)?[0-9]^+$ representa los números enteros como una cadena compuesta por un signo opcional, seguido por al menos un dígito entre 0 y 9.

Los metacaracteres pierden su significado, y pasan a ser caracteres normales, si los encerramos entre comillas. Por ejemplo, los números en punto fijo, como `7.51`, pueden representarse con la expresión regular $(+|-)?[0-9]^+$ `"."` $(+|-)?[0-9]^+$.

3.9.2. El fichero de especificación `lex`

La Figura 3.14 muestra la estructura del fichero de especificación `lex`, que consta de tres secciones, separadas por líneas con el separador `%%`: la sección de definiciones, la sección de reglas y la sección de rutinas auxiliares.

```
sección de definiciones

%%

sección de reglas

%%

sección de rutinas auxiliares
```

Figura 3.14. Estructura de un fichero de especificación `lex`.

a) Sección de definiciones

La sección de definiciones contiene la siguiente información:

- *Código C* encerrado entre los delimitadores `%{` y `%}`, que se copia literalmente en el fichero de salida `lex.yy.c` antes de la definición de la función `yylex()`. Habitualmente, esta sección contiene declaraciones de variables y funciones que se utilizarán posteriormente en la sección de reglas, así como directivas `#include`.
- *Definiciones* propias de `lex`, que permiten asignar nombre a una expresión regular o a una parte de ella, para utilizarlo posteriormente en lugar de la expresión. Para dar nombre a una expresión regular, se escribe el nombre en la primera columna de una línea, seguido por uno o más espacios en blanco y por la expresión regular que representa. Por ejemplo, podríamos dar nombre a la expresión regular que representa a los dígitos del 0 al 9 de la siguiente forma:

  ```
  DIGITO      [0-9]
  ```

  Para utilizar el nombre de una expresión regular en otra expresión regular, basta con encerrarlo entre llaves. Por ejemplo, utilizando la expresión regular llamada `DIGITO`, podríamos representar de la siguiente forma las constantes numéricas que aparecen en la gramática de la Figura 3.1:

  ```
  CONSTANTE   {DIGITO}+
  ```

- *Opciones* de `lex` similares a las opciones de la línea de mandatos. Estas opciones se especifican escribiendo la palabra `%option` seguida de un espacio en blanco y del nombre de la opción. Como ejemplo, en un fichero de especificación de `lex` podría aparecer la siguiente línea:

  ```
  %option noyywrap
  ```

Veamos cuál es el significado de esta línea. Existe la posibilidad de que la función `yylex()` analice morfológicamente varios ficheros, encadenando uno detrás de otro, con el siguiente mecanismo: cuando `yylex()` encuentra el fin de un fichero, llama a la función `yywrap()`. Si esta función devuelve 0, el análisis continúa con otro fichero; si devuelve 1, el análisis termina.

Para poder utilizar la función `yywrap()` en Linux, es necesario enlazar con la biblioteca de `lex` que proporciona una versión por defecto de `yywrap()`. En Windows, el usuario tiene que proporcionar el código de la función, incorporándola en la última sección del fichero de especificación.

La opción `noyywrap` provoca que no se invoque automáticamente a la función `yywrap()` cuando se encuentre un fin de fichero, y se suponga que no hay que analizar más ficheros. Esta solución es más cómoda que tener que escribir la función o enlazar con alguna biblioteca.

- Definición de *condiciones de inicio*. Estas definiciones se verán con más detalle en la Sección 3.9.4.

b) Sección de reglas

La sección de reglas contiene, para cada unidad sintáctica, la expresión regular que la describe, seguida de uno o más espacios en blanco y del código C que debe ejecutarse cuando se localice en la entrada dicha unidad sintáctica. Este código C debe aparecer encerrado entre llaves.

Como ejemplo, consideremos un analizador morfológico que reconozca en la entrada las constantes numéricas y las palabras reservadas `begin` y `end`. Cada vez que localice una de ellas, debe mostrar en la salida un mensaje de aviso de unidad sintáctica reconocida. La Figura 3.15 muestra el fichero de especificación `lex` que correspondería a dicho analizador morfológico.

c) Sección de rutinas auxiliares

Habitualmente, esta sección contiene las funciones escritas por el usuario para utilizarlas en la sección de reglas, es decir, funciones de soporte. En esta sección también se incluyen las funciones de `lex` que el usuario puede redefinir, como, por ejemplo, la función `yywrap()`. El contenido de esta sección se copia literalmente en el fichero `lex.yy.c` que genera `lex`.

Aunque, en el caso general, la función `yylex()` es llamada por el analizador sintáctico, el analizador morfológico también puede funcionar como un componente autónomo, en cuyo caso será necesario incluir en la sección de rutinas auxiliares una función `main` que realice la llamada a la función `yylex()`. Éste es el caso del fichero de especificación `lex` que aparece en la Figura 3.15. La sección de rutinas auxiliares se puede omitir, aunque sí debe aparecer el separador %%.

3.9.3. ¿Cómo funciona `yylex()`?

Una llamada a `yylex()` permite realizar el análisis morfológico hasta encontrar el fin de la entrada, siempre que ninguno de los fragmentos de código C asociado a las expresiones regulares

```
%{
#include <stdio.h> /* para utilizar printf en
                           la sección de reglas */
%}
digito         [0-9]
constante      {digito}+
%option noyywrap

%%
begin              { printf("reconocido-begin-\n"); }
end                { printf("reconocido-end-\n"); }
{constante}        { printf("reconocido-num-\n"); }

%%
int main()
{
      return yylex();
}
```

Figura 3.15. Un fichero de especificación `lex`.

correspondientes a las unidades sintácticas contenga una instrucción `return` que haga que `yylex()` termine. Cuando se encuentra el fin de la entrada, `yylex()` devuelve 0 y termina. La llamada a `yylex()` en la función `main` de la Figura 3.15 ilustra este caso.

Otra alternativa es que el código C asociado a cada expresión regular contenga una sentencia `return`. En este caso, cuando se identifica en la entrada una unidad sintáctica que satisface dicha expresión regular, se devuelve un valor al módulo que invocó a la función `yylex()`. La siguiente llamada a `yylex()` comienza a leer la entrada en el punto donde se quedó la última vez. Como en el caso anterior, cuando se encuentra el fin de la entrada, `yylex()` devuelve 0 y termina. La Figura 3.16 implementa esta alternativa en un fichero de especificación `lex`, para un analizador morfológico con la misma funcionalidad que el de la Figura 3.15.

Además de la función `main`, en el fichero de especificación de la Figura 3.16 puede apreciarse otra diferencia con respecto al que aparece en la Figura 3.15. En la sección de definiciones, aparece la instrucción `#include "tokens.h"`. Este fichero de cabeceras aparece en la Figura 3.17 y contiene un conjunto de instrucciones `#define`, que asignan un valor entero a cada unidad sintáctica que va a reconocer el analizador morfológico, y que será el valor devuelto por la función `yylex()` para cada una de ellas.

Si la función `yylex()` encuentra concordancia con más de una expresión regular, selecciona aquella que permita establecer una correspondencia de mayor número de caracteres con la entrada. Por ejemplo, supongamos que en un fichero de especificación aparecen las siguientes reglas:

```
begin     { return TOK_BEGIN; }
end       { return TOK_END; }
[a-z]+    { return TOK_ID;}
```

```
%{
#include <stdio.h>
#include "tokens.h"
%}
digito       [0-9]
constante    {digito}+

%option noyywrap

%%

begin             { return TOK_BEGIN; }
end               { return TOK_END; }
{constante}       { return TOK_NUM; }

%%

int main()
{
  int token;
  while (1)
     {
       token = yylex();
       if (token == TOK_BEGIN)  printf("reconocido-begin-
\n");
       if (token == TOK_END)    printf("reconocido-end-\n");
       if (token == TOK_NUM)    printf("reconocido-num-\n");
       if (token == 0) break;
     }
```

Figura 3.16. Un fichero de especificación `lex` con instrucciones `return`.

La entrada `beginend` concuerda con dos expresiones regulares: `begin` y `[a-z]+`, hasta que se lee la segunda «e». En ese momento se descarta la expresión regular `begin` y se selecciona la expresión regular correspondiente a los identificadores, porque es la que establece una correspondencia de mayor longitud. Por lo tanto, la entrada `beginend` será considerada como una única unidad sintáctica de tipo identificador.

```
#define TOK_BEGIN 1
#define TOK_END 2
#define TOK_NUM 129
```

Figura 3.17. El fichero `tokens.h`.

Si hay concordancia con varias expresiones regulares de la misma longitud, se elige aquella que aparece antes en la sección de reglas dentro del fichero de especificación `lex`. Por lo tanto, el orden en que se colocan las reglas es determinante. Por ejemplo, si en un fichero de especificación aparecen las siguientes reglas:

```
[a-z]+    { return TOK_ID;}
begin     { return TOK_BEGIN; }
end       { return TOK_END; }
```

la entrada `begin` será considerada como un identificador, porque concuerda con dos expresiones regulares: `begin` y `[a-z]+`, pero la expresión regular correspondiente a los identificadores aparece antes en el fichero de especificación. Al procesar con `lex` estas reglas, aparecerá un mensaje en el que se indica que las reglas segunda y tercera nunca se van a utilizar.

`Lex` declara un vector de caracteres llamado `yytext`, que contiene la cadena correspondiente a la última unidad sintáctica reconocida por la función `yylex()`. La longitud de esta cadena se almacena en la variable de tipo entero `yyleng`.

El analizador morfológico es la parte del compilador que accede al fichero de entrada y, por lo tanto, es el que conoce la posición (línea y carácter) de las unidades sintácticas en dicho fichero. Esta posición es muy importante para informar de los errores de compilación.

Para conocer la posición de las unidades sintácticas en el fichero de entrada se pueden utilizar dos variables, una que guarde el número de línea, y otra para la posición del carácter dentro de la línea. Ambas variables se pueden declarar en la sección de definiciones del fichero de especificación `lex`. Por ejemplo:

```
%{
int lineno = 1; /* número de línea */
int charno = 0; /* número de carácter */
%}
```

La actualización de las variables se realiza en el código de las reglas. Por ejemplo, cuando se analice la palabra reservada `begin`, se incrementará en 5 unidades el valor de la variable `charno`. Cuando se encuentre un identificador o un número entero, se puede utilizar el contenido de la variable `yyleng` para incrementar la variable `charno` en el número de caracteres correspondiente. De igual forma, cuando se encuentra un salto de línea, la variable `lineno` se incrementa en 1 unidad, mientras la variable `charno` se inicializa a 0.

La entrada y salida de `lex` se realiza a través de los ficheros `yyin` e `yyout`, respectivamente. Antes de llamar a la función `yylex()`, puede asignarse a cualquiera de estos dos ficheros una variable de tipo `FILE*`. Si no se realiza ninguna asignación, sus valores por defecto son la entrada estándar (`stdin`) y la salida estándar (`stdout`), respectivamente.

3.9.4. Condiciones de inicio

`Lex` ofrece la posibilidad de asociar condiciones de inicio a las reglas, lo que quiere decir que las acciones asociadas a esas reglas sólo se ejecutarán si se cumple la condición de inicio corres-

pondiente. Esta característica excede a la potencia de las expresiones regulares y de los autómatas finitos, pero resulta imprescindible para representar determinadas unidades sintácticas.

Las condiciones de inicio se especifican en la sección de definiciones del fichero de especificación `lex` utilizando líneas con el siguiente formato:

```
%s nombre_condicion
```

donde nombre representa la condición de inicio. También pueden utilizarse líneas con el formato

```
%x nombre_condicion
```

para especificar condiciones de inicio exclusivas. Ambas opciones se diferencian porque, cuando el analizador se encuentra con una condición de inicio exclusiva, sólo son aplicables las reglas que tienen asociada esa condición de inicio, mientras que si la condición no es exclusiva (si se ha definido con la opción %s), se aplican también las reglas que no tienen condición de inicio. Por ejemplo:

```
%s        uno
%x        dos

%%

abc               {printf("reconocido ");
                    BEGIN(uno);}
<uno>def          {printf("reconocido ");
                    BEGIN(dos);}
<dos>ghi          {printf("reconocido ");
                    BEGIN(INITIAL);}
```

Figura 3.18. Un fichero de especificación `lex` con condiciones de inicio.

En el ejemplo de la Figura 3.18, en la condición de inicio `uno` pueden aplicarse las reglas correspondientes a las expresiones `abc` y `def` . En el estado `dos`, sólo puede aplicarse la regla correspondiente a la expresión `ghi`.

Las condiciones de inicio aparecen en la sección de reglas del fichero de especificación precediendo a una expresión regular y encerradas entre los símbolos < y >. Por ejemplo, si asociamos la condición de inicio `comentario` a las reglas que corresponden a la identificación de comentarios, la línea siguiente, colocada en la sección de reglas, indica que, una vez detectado el comienzo de un comentario, cada salto de línea detectado en la entrada generará un incremento en el contador del número de líneas.

```
<comentario>\n      {lineno++;}
```

Se puede poner al analizador en una determinada condición de inicio escribiendo la instrucción `BEGIN(nombre_condicion)` en la parte de acción de una regla. Por ejemplo, la regla

siguiente pone al analizador en la condición de inicio `comentario` cuando se detectan los caracteres de comienzo de comentario en la entrada.

```
"/*"   {BEGIN(comentario);}
```

Para pasar a la condición de inicio normal, utilizaremos la instrucción

```
BEGIN(INITIAL)
```

En nuestro ejemplo, la regla

```
<comentario>"*"+"/" {BEGIN(INITIAL);}
```

pasa al analizador a la condición de inicio normal cuando se detecta el fin de un comentario, es decir, uno o más caracteres '*' y un carácter '/'.

El código completo que habría que incluir en el fichero de especificación `lex` de un analizador morfológico, para que identifique correctamente los comentarios de tipo C, aparece en la Figura 3.19.

```
%x comentario

%%

"/*"                          {BEGIN(comentario);}
<comentario>[^*\n]
<comentario>"*"+[^*/\n]*
<comentario>\n                {lineno++;}
<comentario>"*"+"/"           {BEGIN(INITIAL);}
```

Figura 3.19. Reglas para identificación de comentarios tipo C.

La primera regla pone al analizador en la condición de inicio `comentario` cuando se detectan en la entrada los caracteres de comienzo de comentario. Las reglas segunda y tercera no realizan ninguna acción mientras se estén leyendo caracteres *, o cualquier carácter distinto de *, / o del carácter de salto de línea. La cuarta regla incrementa el contador del número de líneas cada vez que se detecta en la entrada un salto de línea. Por último, la quinta regla pasa el analizador a la condición de inicio normal cuando se detecta el fin de un comentario, es decir, uno o más caracteres * y un carácter /.

El fichero de especificación `lex` completo para el lenguaje generado por la gramática de la Figura 3.1 aparece en http://www.librosite.net/pulido

3.10 Resumen

Este capítulo describe el funcionamiento de un analizador morfológico, cuya tarea principal es dividir el programa fuente en un conjunto de unidades sintácticas. Para ello se utiliza un sub-

conjunto de las reglas que forman la gramática del lenguaje fuente, que deberán poder representarse como expresiones regulares. A partir de estas expresiones regulares, es posible obtener un AFND que acepta el lenguaje generado por ellas y, en una segunda etapa, el AFD mínimo equivalente. Utilizando la función de transición y los estados finales de este AFD, es posible implementar el autómata que actuará como analizador morfológico.

Se describen también otras tareas auxiliares llevadas a cabo por el analizador morfológico, como la eliminación de ciertos caracteres delimitadores (espacios en blanco, tabuladores y saltos de línea), la eliminación de comentarios y el cálculo de los valores para algunos atributos semánticos de las unidades sintácticas. Se revisa también el tipo de errores que puede detectar el analizador morfológico, y cómo puede comportarse ante ellos.

Por último, se estudia con detalle la herramienta `lex`, para la generación automática de analizadores morfológicos, y se describe el fichero de especificación que requiere como entrada, así como el funcionamiento del analizador morfológico que genera como salida.

3.11 Ejercicios

3.1. Construir una gramática que represente el lenguaje de los números en punto flotante del tipo [-][cifras][.[cifras]][e[-][cifras]]. Debe haber al menos una cifra en la parte entera o en la parte decimal, así como en el exponente, si lo hay.

3.2. Construir un autómata finito determinista que reconozca el lenguaje del Ejercicio 3.1.

3.3. Construir una gramática que represente el lenguaje de las cadenas de caracteres correctas en C.

3.4. Construir un autómata finito determinista que reconozca el lenguaje del Ejercicio 3.3.

3.5. En el lenguaje APL, una cadena de caracteres viene encerrada entre dos comillas simples. Si la cadena de caracteres contiene una comilla, ésta se duplica. Construir una gramática regular que describa el lenguaje de las cadenas de caracteres válidas en APL.

3.6. Construir un autómata finito determinista que reconozca el lenguaje del Ejercicio 3.5.

3.7. Construir un autómata finito determinista que reconozca los caracteres en el lenguaje C. Ejemplos válidos: ‘a’, ‘\n’, ‘\033’ (cualquier número de cifras). Ejemplos incorrectos: ‘‘ ‘\’.

3.8. Se desea realizar un compilador para un lenguaje de programación que manejará como tipo de dato vectores de enteros. Los vectores de enteros se representarán como una lista de números enteros separados por comas. El vector más pequeño sólo tendrá un número y, en este caso, no aparecerá coma alguna. A continuación se muestran algunos ejemplos: {23}, {1,210,5,0,09}.

3.8.1. Diseñar una gramática para representar este tipo de datos.

3.8.2. Indicar qué parte de ella sería adecuado que fuera gestionada por el analizador morfológico del compilador. Justificar razonadamente la respuesta.

(c) Para cada una de las unidades sintácticas, especificar una expresión regular que la represente (puede usarse la notación de `lex`).

3.9. En un lenguaje de programación los nombres de las variables deben comenzar con la letra "V" y terminar con un dígito entre el 0 y el 9. Entre estos dos símbolos puede aparecer cualquier letra mayúscula o minúscula. Las constantes numéricas son números reales positivos que deben tener obligatoriamente las siguientes partes: parte entera (una cadena de cualquier cantidad de dígitos entre 0 y 9), separador ("**,**"), parte fraccionaria (con la misma sintaxis que la parte entera). Algunos ejemplos de expresiones correctas son las siguientes: Variable1, +Variable1 4,04, (log +V2(sen 4,54)). Especificar las expresiones regulares con notación `lex` que podría utilizar un analizador morfológico para representar las unidades sintácticas para los nombres de las variables y las constantes numéricas.

3.12 Bibliografía

[1] Alfonseca, M.; Sancho, J., y Martínez Orga, M. (1997): *Teoría de Lenguajes, Gramáticas y Autómatas,* Madrid, Promo-Soft, Publicaciones R.A.E.C.

Capítulo 4

Análisis sintáctico

Este capítulo describe algunos de los diversos métodos que suelen utilizarse para construir los analizadores sintácticos de los lenguajes independientes del contexto. Recordemos que el analizador sintáctico o *parser* es el corazón del compilador o intérprete y gobierna todo el proceso. Su objetivo es realizar el resto del análisis (continuando el trabajo iniciado por el analizador morfológico) para comprobar que la sintaxis de la instrucción en cuestión es correcta. Para ello, el analizador sintáctico considera como símbolos terminales las unidades sintácticas devueltas por el analizador morfológico.

Existen dos tipos principales de análisis:

- Descendente o de arriba abajo (*top-down*, en inglés). Se parte del axioma `S` y se va realizando la derivación $S \rightarrow^* x$. La cadena `x` (que normalmente corresponde a una instrucción o un conjunto de instrucciones) se llama meta u objetivo del análisis. La primera fase del análisis consiste en encontrar, entre las reglas cuya parte izquierda es el axioma, la que conduce a `x`. De esta manera, el árbol sintáctico se va construyendo de arriba abajo, tal como indica el nombre de este tipo de análisis.

En este capítulo se explicará con detalle un método de análisis de arriba abajo: el que se basa en el uso de gramáticas LL(1).

- Ascendente o de abajo arriba (*bottom-up*, en inglés). Se parte de la cadena objetivo `x` y se va reconstruyendo en sentido inverso la derivación $S \rightarrow^* x$. En este caso, la primera fase del análisis consiste en encontrar, en la cadena `x`, el asidero (*véase* la Sección 1.9.7), que es la parte derecha de la última regla que habría que aplicar para reducir `S` a `x`. De esta manera, el árbol sintáctico se va construyendo de abajo arriba, tal como indica el nombre de este tipo de análisis.

En este capítulo se explicarán con detalle los siguientes métodos de análisis ascendente: LR(0), SLR(1), LR(1), LALR(1) (estos dos últimos son los más generales, pues permiten analizar la sintaxis de cualquier lenguaje independiente del contexto), así como el que utiliza gramáticas de precedencia simple, el menos general de todos, pues sólo se aplica a len-

guajes basados en el uso de expresiones, pero que permite obtener mejores eficiencias en esos casos.

Como se dijo en la Sección 1.4 y en la Figura 1.1, la máquina apropiada para el análisis de los lenguajes independientes del contexto es el autómata a pila. Esto explica que, aunque en los métodos de análisis revisados en este capítulo el autómata pueda estar más o menos oculto, en todos ellos se observa la presencia de una pila. Por otra parte, el hecho de que un lenguaje sea independiente del contexto (que su gramática sea del tipo 2 de Chomsky) no siempre asegura que el autómata a pila correspondiente resulte ser determinista. Los autómatas a pila deterministas sólo son capaces de analizar un subconjunto de los lenguajes independientes del contexto. A lo largo de las páginas siguientes, se impondrá diversas restricciones a las gramáticas, en función del método utilizado. Las restricciones exigidas por un método no son las mismas que las que exige otro, por lo que los distintos métodos se complementan, lo que permite elegir el mejor o el más eficiente para cada caso concreto.

En este capítulo, es necesario introducir símbolos especiales que señalen el principio o el fin de las cadenas que se van a analizar. Cuando sólo hace falta añadir un símbolo final, se utilizará el símbolo `$`, pues es poco probable encontrarlo entre los símbolos terminales de la gramática. Cuando hace falta señalar los dos extremos de la cadena, se utilizarán los símbolos $\vdash$ y $\dashv$ para el principio y el final, respectivamente.

4.1 Conjuntos importantes en una gramática

Sea una gramática limpia $G = (\Sigma_T, \Sigma_N, S, P)$. Sea $\Sigma = \Sigma_T \cup \Sigma_N$. Sea $\in\Sigma$ un símbolo de esta gramática (terminal o no terminal). Se definen los siguientes conjuntos asociados a estas gramáticas y a este símbolo:

- Si $X\in\Sigma_N$, $primero(X) = \{V \mid X \rightarrow^+ Vx, V\in\Sigma_T, x\in\Sigma^*\}$
- Si $X\in\Sigma_T$, $primero(X) = \{X\}$

Es decir, si `X` es terminal, `primero(X)` contiene sólo a `X`; en caso contrario, `primero(X)` es el conjunto de símbolos terminales que pueden aparecer al principio de alguna forma sentencial derivada a partir de `X`.

- `siguiente(X)` $= \{V \mid S \rightarrow^+ xXVy, X\in\Sigma_N, V\in\Sigma_T, x,y\in\Sigma^*\}$, donde `S` es el axioma de la gramática.

Es decir, si `X` es un símbolo no terminal de la gramática, `siguiente(X)` es el conjunto de símbolos terminales que pueden aparecer inmediatamente a la derecha de `X` en alguna forma sentencial. Si `X` puede aparecer en el extremo derecho de alguna forma sentencial, entonces el símbolo de fin de cadena `$` $\in$ `siguiente(X)`.

Para calcular el conjunto `primero(X)` $\forall X \in \Sigma_N \cup \Sigma_T$, se aplicarán las siguientes reglas, hasta que no se puedan añadir más símbolos terminales ni λ a dicho conjunto.

(R1) Si $X \in \Sigma_T$, se hace `primero(X)`$=\{X\}$.

(R2) Si $X ::= \lambda \in P$, se añade λ a `primero(X)`.

(R3) Si $X ::= Y_1\ Y_2\ \ldots\ Y_k \in P$, se añade primero($Y_i$)−{λ} a primero(X) para i=1,2,…,j, donde j es el primer subíndice tal que Y_j no genera la cadena vacía (Y_j es terminal, o siendo no terminal no ocurre que $Y_j \rightarrow \lambda$).

(R4) Si $X ::= Y_1\ Y_2\ \ldots\ Y_k \in P$ y $Y_j \rightarrow \lambda\ \forall j \in \{1,2,\ldots,k\}$, se añade λ a primero(X).

Ejemplo 4.1 Consideremos la gramática siguiente, en la que E es el axioma:

```
(1)  E ::= TE'
(2)  E' ::= +TE'
(3)  E' ::= λ
(4)  T ::= FT'
(5)  T' ::= *FT'
(6)  T' ::= λ
(7)  F ::= (E)
(8)  F ::= id
```

En esta gramática, el conjunto primero(E) resulta ser igual al conjunto primero(T) aplicando la regla (R3) a la regla (1). Para calcular el conjunto primero(T) se aplica la regla (R3) a la regla (4), de la que se obtiene que primero(T) es igual a primero(F).

Para calcular primero(F) se aplica la regla (R3) a la regla (7), añadiendo el conjunto primero((), que es igual a {(} por la regla (R1). Al aplicar la regla (R3) a la regla (8), se añade también el conjunto primero(id), que es igual a {id} por la regla (R1). Por tanto:

primero(E)=primero(T)=primero(F)={(, id}

A continuación se calcula el conjunto primero(E'), aplicando, en primer lugar, la regla (R3) a la regla (2), que añade el conjunto primero(+), que es igual a {+} por la regla (R1). Al aplicar la regla (R2) a la regla (3), se añade también λ. Por tanto,

primero(E')={+, λ}

De una forma parecida, se calcula el conjunto primero(T'). Al aplicar la regla (R3) a la regla (5) se añade el conjunto primero(*), que es igual a {*} por la regla (R1), y aplicando la regla (R2) a la regla (6), se añade λ. Por tanto,

primero(T')={*, λ}

En alguno de los algoritmos de análisis sintáctico descritos en este capítulo será necesario extender la definición del conjunto primero para que se aplique a una forma sentencial, lo que se hará de la siguiente manera: sea $G=(\Sigma_T, \Sigma_N, S, P)$ una gramática independiente del contexto. Si α es una forma sentencial de la gramática, $\alpha \in (\Sigma_N \cup \Sigma_T)^*$; es decir, si α puede obtenerse por derivación, a partir del axioma S, en cero o más pasos, aplicando reglas de P, llamaremos primero(α) al conjunto de símbolos terminales que pueden aparecer en primer lugar en las cadenas derivadas a partir de α. Si desde α se puede derivar la cadena vacía λ, ésta también pertenecerá a primero(α).

Sea $\alpha=X_1X_2...X_n$. Para calcular primero(α), se aplicará el siguiente algoritmo hasta que no se puedan añadir más símbolos terminales o λ a dicho conjunto:

- Añadir a primero(α) todos los símbolos de primero(X_1), excepto λ.
- Si primero(X_1) contiene λ, añadir a primero(α) todos los símbolos de primero(X_2), excepto λ.
- Si primero(X_1) y primero(X_2) contienen λ, añadir a primero(α) todos los símbolos de primero(X_3), excepto λ.
- Y así sucesivamente.
- Si $\forall i \in \{1,2,..,n\}$ primero(X_i) contiene λ, entonces añadir λ a primero(α).

En la gramática del Ejemplo 4.1, para calcular el conjunto primero(T'E'id) se calcula primero(T'), por lo que se añadirá {*}. Como primero(T') contiene λ, es necesario añadir también primero(E'), por lo que se añade {+}. Como primero(E') también contiene λ, es necesario añadir también primero(id), que es igual a {id}. Por tanto:

primero(T'E'id) = {*,+,id}

Para calcular el conjunto siguiente(X) $\forall X \in \Sigma_N$, deben aplicarse las siguientes reglas, hasta que no se puedan añadir más símbolos terminales a dicho conjunto.

(R1) Para el axioma S, añadir $ a siguiente(S).

(R2) Si $A::=\alpha X\beta \in P$, añadir todos los símbolos (excepto λ) de primero(β) a siguiente(X).

(R3) Si $A::=\alpha X\beta \in P$ y $\lambda \in$ primero(β), añadir todos los símbolos de siguiente(A) a siguiente(X).

(R4) Si $A::=\alpha X \in P$, añadir siguiente(A) a siguiente(X).

Como ejemplo se considerará la gramática del Ejemplo 4.1. Se tendrá que:

siguiente(E)={$,)}

El símbolo $ se añade al aplicar la regla (R1), y el símbolo) al aplicar la regla (R2) a la regla (7).

Al aplicar la regla (R4) a las reglas (1) y (2), el conjunto siguiente(E') resulta ser igual al conjunto siguiente(E), ya que la afirmación siguiente(E')= siguiente(E') deducida de la regla (2) no añade ningún símbolo nuevo. Por tanto:

siguiente(E')={$,)}

Para calcular siguiente(T) se aplica la regla (R2) a las reglas (1) y (2) y se añade el símbolo + por pertenecer al conjunto primero(E'). Como $\lambda \in$ primero(E'), se aplica la regla (R3) a la regla (1) y se añaden también los símbolos) y $ por pertenecer a siguiente(E). Por tanto:

siguiente(T)={+,$,)}

Al aplicar la regla (R4) a la regla (4), siguiente(T') resulta ser igual a siguiente(T). Por tanto:

siguiente(T')={+,$,)}

Por último, se calcula siguiente(F). Al aplicar la regla (R2) a las reglas (4) y (5), se añade el símbolo * por pertenecer a primero(T'). Como $\lambda \in$ primero(T'), se aplica la regla (R3) a la regla (4) y se añaden los símbolos +, $ y) por pertenecer a siguiente(T). Al aplicar la regla (R3) a la regla (5), deberían añadirse los símbolos +, $ y) por pertenecer a siguiente(T'), pero no se hace porque ya pertenecen al conjunto. Por tanto:

siguiente(F)={*,+,$,)}

4.2 Análisis sintáctico descendente

Como se menciona en la introducción de este capítulo, para realizar el análisis sintáctico descendente de una palabra x, se parte del axioma S y se va realizando la derivación $S \rightarrow^* x$. Un método posible para hacerlo es el *análisis descendente con vuelta atrás*, que consiste en probar sistemáticamente todas las alternativas hasta llegar a la reducción buscada o hasta que se agoten dichas alternativas. La ineficiencia de este método se soluciona si, en cada momento del proceso de construcción de la derivación, sólo se puede aplicar una regla de la gramática. Esta idea es el origen de las gramáticas LL(1) y del método de análisis descendente selectivo que se aplica a este tipo de gramáticas.

4.2.1. Análisis descendente con vuelta atrás

En una gramática dada, sea S el axioma y sean las reglas cuya parte izquierda es el axioma:

$S ::= X_1\ X_2\ \ldots\ X_n \mid Y_1\ Y_2\ \ldots\ Y_m \mid \ldots$

Sea x la palabra que se va a analizar. El objetivo del análisis es encontrar una derivación tal que

$S \rightarrow^* x$

En el método de análisis descendente con vuelta atrás se prueba primero la regla

$S ::= X_1\ X_2\ \ldots\ X_n$

Para aplicarla, es necesario descomponer x de la forma

$x = x_1\ x_2\ \ldots\ x_n$

y tratar de encontrar las derivaciones

$X_1 \rightarrow^* x_1$
$X_2 \rightarrow^* x_2$
$\ldots$
$X_n \rightarrow^* x_n$

Cada una de las derivaciones anteriores es una submeta. Pueden ocurrir los siguientes casos:

- (Caso 1) $X_i = x_i$: submeta reconocida. Se pasa a la submeta siguiente.
- (Caso 2) $X_i \neq x_i$ y X_i es un símbolo terminal: submeta rechazada. Se intenta encontrar otra submeta válida para X_{i-1}. Si $i=1$, se elige la siguiente parte derecha para el mismo símbolo no terminal a cuya parte derecha pertenece X_i. Si ya se han probado todas las partes derechas, se elige la siguiente parte derecha del símbolo no terminal del nivel superior. Si éste es el axioma, la cadena x queda rechazada.
- (Caso 3) X_i es un símbolo no terminal. Se buscan las reglas de las que X_i es parte izquierda:

  $X_i ::= X_{i1}\ X_{i2}\ \ldots\ X_{in} \mid Y_{i1}\ Y_{i2}\ \ldots\ Y_{im} \mid \ldots$

  se elige la primera opción:

  $X_i ::= X_{i1}\ X_{i2}\ \ldots\ X_{in}$

  se descompone x_i en la forma

  $x_i = x_{i1}\ x_{i2}\ \ldots\ x_{in}$

  lo que da las nuevas submetas

  $X_{i1} \rightarrow^* x_{i1}$
  $X_{i2} \rightarrow^* x_{i2}$
  $\ldots$
  $X_{in} \rightarrow^* x_{in}$

  y continuamos recursivamente de la misma forma.

El proceso termina cuando en el Caso 2 no hay más reglas que probar para el axioma (en cuyo caso la cadena no es reconocida) o cuando se reconocen todas las submetas pendientes (en cuyo caso la cadena sí es reconocida).

Ejemplo 4.2 Consideremos la gramática siguiente:

$S ::= aSb$

$S ::= a$

Sea aabb la palabra a reconocer.

Se prueba primero la regla $S ::= aSb$ y se intenta encontrar las siguientes derivaciones:

(S1) $a \rightarrow^* a$
(S2) $S \rightarrow^* ab$
(S3) $b \rightarrow^* b$

La Figura 4.1 ilustra este primer paso. Las derivaciones primera y tercera corresponden a submetas reconocidas de acuerdo con el Caso 1. La segunda derivación corresponde al Caso 3, por lo que se elige la regla $S ::= aSb$, por ser la primera regla en cuya parte izquierda aparece S. Se obtienen dos nuevas submetas:

$a \rightarrow^* a$
$S \rightarrow^* b$

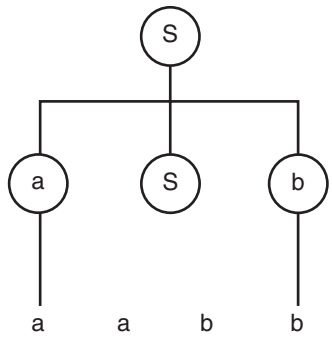


Figura 4.1. Construcción de una derivación para la palabra `aabb`: paso 1.

Como puede apreciarse en la Figura 4.2, la primera derivación corresponde a una submeta reconocida de acuerdo con el Caso 1. La segunda derivación corresponde al Caso 3, por lo que se elige la regla `S ::= aSb`, por ser la primera regla en cuya parte izquierda aparece `S`. Se obtiene una nueva submeta:

`a` $\rightarrow^*$ `b`

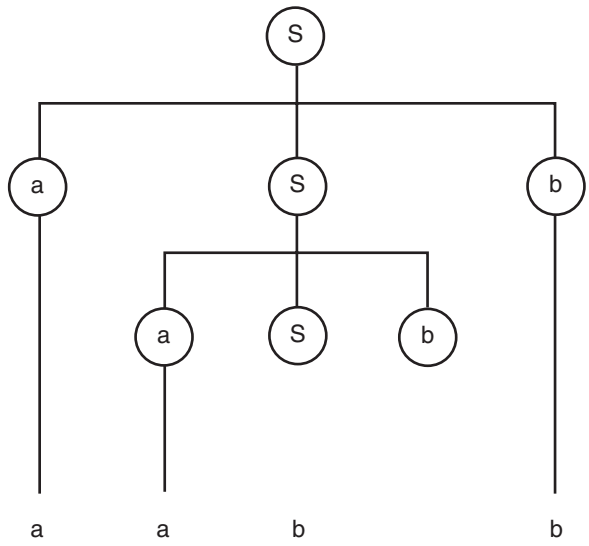


Figura 4.2. Construcción de una derivación para la palabra `aabb`: paso 2.

Esta derivación aparece en la Figura 4.3 y corresponde a una submeta rechazada, de acuerdo con el Caso 2. Como el símbolo `a` es el primer símbolo en la parte derecha de la regla `S ::= aSb`,

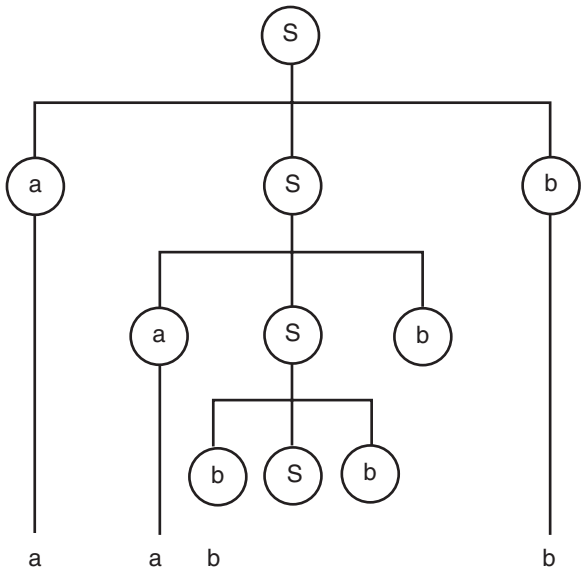


Figura 4.3. Construcción de una derivación para la palabra `aabb`: paso 3.

se elige la siguiente parte derecha para el símbolo `S`, es decir, `S ::= ab`. Como ilustra la Figura 4.4, se obtiene una nueva submeta:

`a` $\rightarrow^*$ `b`

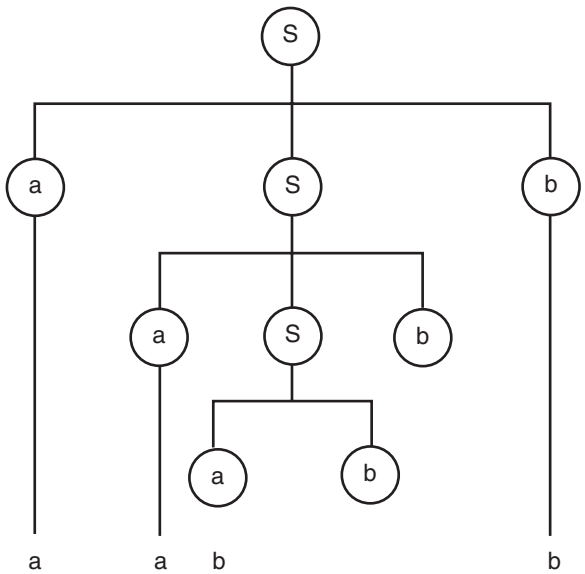


Figura 4.4. Construcción de una derivación para la palabra `aabb`: paso 4.

Esta derivación también corresponde a una submeta rechazada, de acuerdo con el Caso 2. Como ya no hay más reglas para el símbolo `S`, se vuelve a la submeta (`S2`) y se elige la siguiente parte derecha para `S`, es decir, `S ::= ab`. Se obtienen dos nuevas submetas:

```
a →* a
b →* b
```

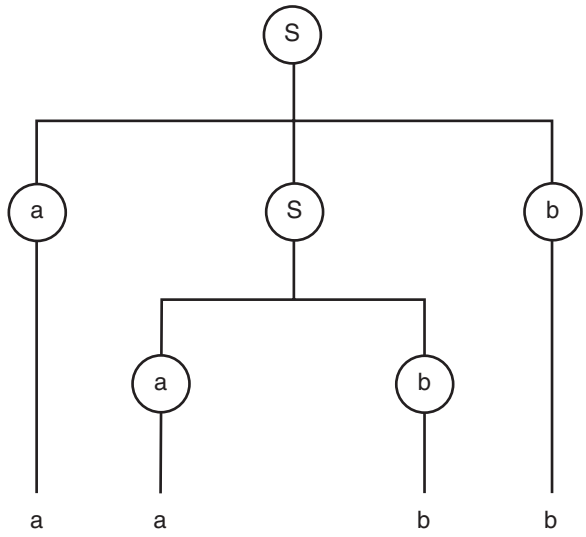


Figura 4.5. Árbol de derivación para la palabra `aabb`.

Ambas derivaciones corresponden a submetas reconocidas de acuerdo con el Caso 1. Como se han encontrado todas las derivaciones, se reconoce la cadena `aabb`. La Figura 4.5 muestra el árbol de derivación para la palabra `aabb`.

Como habrá podido observarse, este método de análisis puede ser bastante ineficiente, porque el número de submetas que hay que comprobar puede ser bastante elevado. Una forma de optimizar el procedimiento consiste en ordenar las partes derechas de las reglas que comparten la misma parte izquierda, de manera que la regla *buena* se analice antes. Una buena estrategia para ordenar las partes derechas es poner primero las de mayor longitud. Si una parte derecha es la cadena vacía, debe ser la última. Si se llega a ella, la submeta tiene éxito automáticamente. A esta variante del método se la denomina *análisis descendente con vuelta atrás rápida*.

Como ejemplo, consideremos la siguiente gramática:

```
E ::= T+E | T
T ::= F*T | F
F ::= i
```

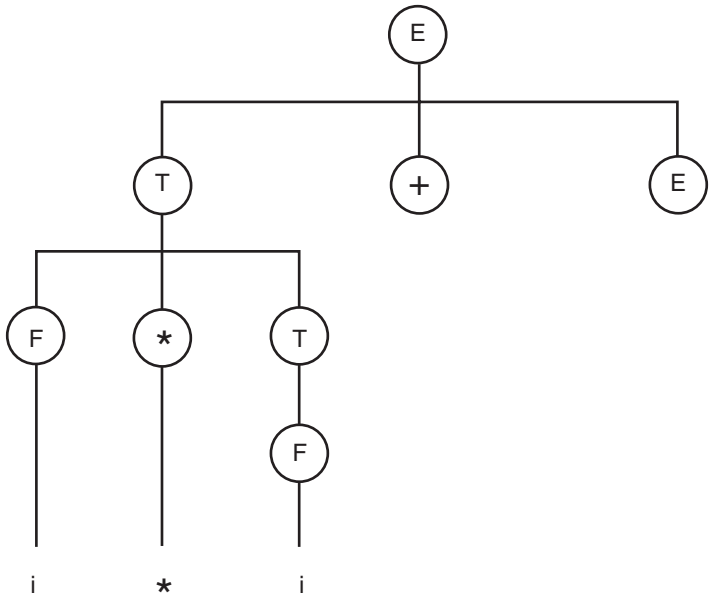


Figura 4.6. Construcción de una derivación para la palabra `i*i`: paso 1.

Se analiza la cadena `i*i`. Se prueba primero `E ::= T+E` y se obtiene el árbol de la Figura 4.6.

En cuanto se compruebe que el signo + no pertenece a la cadena objetivo, no es necesario volver a las fases precedentes del análisis, tratando de obtener otra alternativa, sino que se puede pasar directamente a probar `E::=T`. El árbol de derivación obtenido aparece en la Figura 4.7.

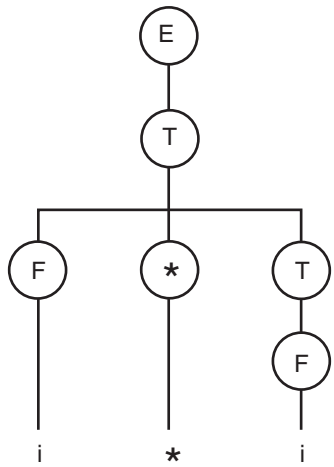


Figura 4.7. Árbol de derivación para la palabra `i*i`.

4.2.2. Análisis descendente selectivo

En el método óptimo de análisis descendente, en cada etapa de construcción del árbol de derivación sólo debería ser posible aplicar una regla. Este método de análisis se llama *análisis descendente selectivo, análisis sin vuelta atrás* o *descenso recursivo,* y las gramáticas compatibles con él reciben el nombre de *gramáticas LL(1)*. Esto puede conseguirse, por ejemplo, si en la gramática que se está utilizando para el análisis, las partes derechas de las reglas que tienen la misma parte izquierda empiezan por un símbolo terminal distinto.

Las siglas *LL(1)* se refieren a una familia de analizadores sintácticos descendentes en los que la entrada se lee desde la izquierda —***L****eft* en inglés—, las derivaciones en el árbol se hacen de izquierda —***L****eft*— a derecha, y en cada paso del análisis se necesita conocer sólo un – *1*– símbolo de la entrada.

Existen varias aproximaciones a este tipo de análisis, lo que da lugar a definiciones diferentes, no siempre equivalentes, de las gramáticas LL(1). En este capítulo se presentarán dos de ellas:

- La que se basa en el uso de la forma normal de Greibach.
- La que se basa en el uso de tablas de análisis.

4.2.3. Análisis LL(1) mediante el uso de la forma normal de Greibach

Se dice que una gramática está en *forma normal de Greibach* si todas las producciones tienen la forma $A ::= ax$, donde $A \in \Sigma_N$, $a \in \Sigma_T$, $x \in \Sigma_N^*$, es decir, si la parte derecha de todas las reglas empieza por un símbolo terminal, seguido opcionalmente por símbolos no terminales.

De acuerdo con esta definición, una *gramática LL(1)* es una gramática en forma normal de Greibach en la que no existen dos reglas con la misma parte izquierda, cuya parte derecha empiece por el mismo símbolo terminal.

Es decir, una gramática LL(1) cumple dos condiciones:

- La parte derecha de todas las reglas empieza por un símbolo terminal, seguido opcionalmente por símbolos no terminales.
- No existen dos reglas con la misma parte izquierda, cuya parte derecha empiece por el mismo símbolo terminal.

Como ejemplo, la siguiente gramática es LL(1), porque la parte derecha de todas las reglas empieza por un símbolo terminal, seguido opcionalmente por símbolos no terminales, y las dos reglas cuya parte izquierda es la misma (símbolo `R`) empiezan por un símbolo terminal distinto.

```
F ::= iRP
R ::= aAC | bZ
C ::= c
P ::= p
```

Se llama LL(1) a este tipo de gramáticas, porque en cada momento basta estudiar un carácter de la cadena objetivo para saber qué regla se debe aplicar.

Eliminación de la recursividad a izquierdas. Para convertir una gramática en otra equivalente en forma normal de Greibach, es preciso eliminar las reglas recursivas a izquierdas, si las hay. Una regla es recursiva a izquierdas si tiene la forma $A ::= Ax$, donde $x \in \Sigma^*$.

Sea la gramática independiente del contexto $G = (\Sigma_T, \Sigma_N, S, P)$, donde P contiene las reglas

$$A ::= A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

Se construye la gramática $G' = (\Sigma_T, \Sigma_N \cup \{X\}, S, P')$, donde P' se obtiene reemplazando en P las reglas anteriores por las siguientes:

$$A ::= \beta_1 X \mid \beta_2 X \mid \ldots \mid \beta_m X$$
$$X ::= \alpha_1 X \mid \alpha_2 X \mid \ldots \mid \alpha_n X \mid \lambda$$

En P' ya no aparecen reglas recursivas a izquierdas y puede demostrarse que $L(G') = L(G)$.

Ejemplo 4.3 Consideremos la gramática siguiente:

```
E ::= E + T | T
T ::= T * F | F
F ::= i
```

Para eliminar de esta gramática las reglas recursivas a izquierdas, se empieza por aquellas en cuya parte izquierda aparece el símbolo E. En este caso $\alpha_1 = +T$ y $\beta_1 = T$. Por lo tanto, deben reemplazarse las dos primeras reglas de la gramática por las siguientes:

```
E ::= TX
X ::= +TX | λ
```

Si se aplica la misma transformación a las reglas en cuya parte izquierda aparece el símbolo T ($\alpha 1 = *F$ y $\beta 1 = F$), se obtienen las reglas:

```
T ::= FY
Y ::= *FY | λ
```

Después de eliminar la recursividad a izquierdas, se obtiene la siguiente gramática equivalente a la original:

```
E ::= TX
X ::= +TX | λ
T ::= FY
Y ::= *FY | λ
F ::= i
```

Eliminación de símbolos no terminales iniciales. El algoritmo que se va a describir a continuación tiene por objeto eliminar las reglas que empiezan por un símbolo no terminal. Para ello, hay que establecer una relación de orden parcial en $\Sigma_N = \{A_1, A_2, \ldots, A_n\}$ de la siguiente

forma: A_i precede a A_j si P contiene al menos una regla de la forma $A_i ::= A_j\alpha$, donde $\alpha \in \Sigma^*$. Si existen bucles de la forma $A_i ::= A_j\alpha$, $A_j ::= A_i\beta$, se elige un orden arbitrario entre los símbolos no terminales A_i y A_j.

En la gramática obtenida en el Ejemplo 4.3, puede establecerse la relación de orden E < T < F.

Después de definir la relación de orden, se clasifican las reglas de P en tres grupos:

1. Reglas de la forma $A ::= ax$, donde $a \in \Sigma_T$, $x \in \Sigma^*$.
2. Reglas de la forma $A_i ::= A_jx$, donde $A_i < A_j$ y $x \in \Sigma^*$.
3. Reglas de la forma $A_i ::= A_jx$, donde $A_i > A_j$ y $x \in \Sigma^*$.

En la gramática obtenida en el Ejemplo 4.3, todas las reglas son de tipo 1, excepto las reglas E ::= TX y T ::= FY, que son de tipo 2.

Para obtener una gramática en forma normal de Greibach se debe eliminar las reglas de tipo 2 y 3. Para eliminar una regla de la forma $A_i ::= A_jx$, basta sustituir A_j en dicha regla por todas las partes derechas de las reglas cuya parte izquierda es A_j.

Se eliminan primero las reglas de tipo 3. Si existen varias reglas de este tipo, se trata primero aquella cuya parte izquierda aparece antes en la relación de orden establecida.

A continuación, se eliminan las reglas de tipo 2. Si existen varias reglas de este tipo, se trata primero aquella cuya parte izquierda aparece más tarde en la relación de orden establecida.

Si durante este proceso de eliminación aparecen de nuevo reglas recursivas a izquierdas, se eliminan aplicando el procedimiento descrito anteriormente. Si aparecieran símbolos inaccesibles, también deberían eliminarse (*véase* la Sección 1.12.2).

Ejemplo 4.4

En la gramática obtenida en el Ejemplo 4.3, no hay reglas de tipo 3, por lo que sólo hay que eliminar las de tipo 2: E ::= TX y T ::= FY. Como T aparece detrás de E en la relación de orden, se elimina primero la regla cuya parte izquierda es T, y se obtiene la siguiente gramática:

```
E ::= TX
X ::= +TX | λ
T ::= iY
Y ::= *FY | λ
F ::= i
```

Después de eliminar la regla en cuya parte izquierda aparece E, se obtiene la siguiente gramática:

```
E ::= iYX
X ::= +TX | λ
T ::= iY
Y ::= *FY | λ
F ::= i
```

De acuerdo con la definición dada anteriormente, en una gramática en forma normal de Greibach no pueden aparecer reglas-λ, es decir, reglas cuya parte derecha es el símbolo λ. Sin

embargo, como el objetivo es obtener una gramática a la que se pueda aplicar el método de análisis descendente selectivo, se permite una regla-λ `A ::=` λ si se cumple que `primero(A)` ∩ `siguiente(A)` = Φ.

Veamos si las reglas-λ que aparecen en la gramática obtenida en el Ejemplo 4.4 pueden permanecer en la gramática. Para la regla `X::=` λ se cumple que `primero(X) = {+,` λ`}` y `siguiente(X) = {$}`. La intersección de estos dos conjuntos es el conjunto vacío, por lo que la regla `X::=` λ puede permanecer en la gramática. Para la regla `Y::=` λ se cumple que `primero(Y) = {*,` λ`}` y `siguiente(Y) = {+,$}`. La intersección de estos dos conjuntos es el conjunto vacío, por lo que la regla `Y::=` λ puede permanecer en la gramática.

Si una regla-λ `A::=` λ no cumple la condición indicada, a veces es posible eliminarla, aplicando el siguiente procedimiento.

1. Eliminar la regla.
2. Por cada aparición de `A` en la parte derecha de una regla, añadir una nueva regla en la que se elimina dicha aparición.

Por ejemplo, si se quiere eliminar la regla `A::=` λ de una gramática en la que aparece la regla `B::= uAvAw`, deben añadirse las siguientes reglas:

```
B::= uvAw
B::= uAvw
B::= uvw
```

En la gramática obtenida en el Ejemplo 4.4, las partes derechas de todas las reglas empiezan por un símbolo terminal, seguido opcionalmente por símbolos no terminales. Puede ser que, como resultado del proceso anterior, no todos los símbolos que siguen al terminal inicial sean no terminales. En tal caso, es necesario eliminar los símbolos terminales no situados al comienzo de la parte derecha de una regla. El procedimiento que se debe aplicar en este caso es trivial. Sea, por ejemplo, la regla `A ::= abC`. Para eliminar el símbolo `b` basta reemplazar esta regla por las dos siguientes:

```
A ::= aBC
B ::= b
```

Ejemplo 4.5 Consideremos la gramática siguiente:

```
A ::= Ba | a
B ::= Ab | b
```

Inicialmente no hay ninguna regla recursiva a izquierdas. Existen dos relaciones de orden posibles: `A < B` (por la regla `A ::= Ba`) y `B < A` (por la regla `B ::= Ab`). En este caso, elegiremos arbitrariamente el orden: `B < A`.

La clasificación de las reglas de acuerdo con este orden es la siguiente:

```
A ::= Ba   tipo 3
A ::= a    tipo 1
B ::= Ab   tipo 2
B ::= b    tipo 1
```

Se elimina primero la única regla de tipo 3 que aparece en la gramática: `A ::= Ba`. El resultado es el siguiente:

```
A ::= Aba    recursiva a izquierdas
A ::= ba     tipo 1
A ::= a      tipo 1
B ::= Ab     tipo 2
B ::= b      tipo 3
```

En la gramática resultante, el símbolo no terminal `B` es inaccesible, porque no aparece en la parte derecha de ninguna regla, por lo que podemos eliminar las reglas en las que aparece como parte izquierda. El resultado es el siguiente:

```
A ::= Aba    recursiva a izquierdas
A ::= ba     tipo 1
A ::= a      tipo 1
```

Después de eliminar la regla recursiva a izquierdas, el resultado es el siguiente:

```
A ::= baX    tipo 1
A ::= aX     tipo 1
X ::= baX    tipo 1
X ::= λ      tipo 1
```

Para que esta gramática esté en forma normal de Greibach, la parte derecha de todas las reglas debe constar de un símbolo terminal seguido de símbolos no terminales. Para ello se deben transformar las reglas 1 y 3 con el siguiente resultado final:

```
A ::= bZX
A ::= aX
X ::= bZX
X ::= λ
Z ::= a
```

El siguiente paso es analizar si la regla `X ::= λ` puede permanecer en la gramática. Para ello, deben calcularse los conjuntos `primero(X) = {b,λ}` y `siguiente(X) = {$}`. Como la intersección de estos conjuntos es el conjunto vacío, la regla `X ::= λ` puede permanecer en la gramática.

Para que una gramática en forma normal de Greibach sea una gramática LL(1), debe cumplir que no existan dos reglas con la misma parte izquierda, cuya parte derecha empiece por el mismo símbolo terminal. La gramática en forma normal de Greibach obtenida para el Ejemplo 4.5 cumple esta condición, porque las dos reglas con el símbolo `A` en su parte izquierda empiezan por dos símbolos terminales distintos: `a` y `b`. La gramática obtenida en el Ejemplo 4.4 también cumple esta condición.

Si no fuese éste el caso, el procedimiento para conseguir que dicha condición se cumpla es bastante sencillo. Sea la siguiente gramática:

```
U ::= aV | aW
V ::= bX | cY
W ::= dZ | eT
```

Para que las reglas con el símbolo U en su parte izquierda cumplan la condición LL(1), se hace algo parecido a sacar factor común, reemplazando las dos reglas con parte izquierda U por

```
U ::= aK
K ::= V | W
```

Ahora las reglas de parte izquierda K no están en forma normal de Greibach. Para que lo estén, se sustituyen los símbolos V y W por las partes derechas de sus reglas.

```
U ::= aK
K ::= bX | cY | dZ | eT
```

El resultado es una gramática LL(1).

Hay que tener en cuenta que estas operaciones no siempre dan el resultado apetecido, pues no toda gramática independiente del contexto puede ponerse en forma LL(1).

Una función para cada símbolo no terminal

En una gramática LL(1), los símbolos no terminales se clasifican en dos grupos: los que tienen reglas-λ y los que no. A cada símbolo no terminal de la gramática se le hace corresponder una función que realizará la parte del análisis descendente correspondiente a dicho símbolo. La Figura 4.8 muestra la función escrita en C que corresponde a un símbolo no terminal sin regla-λ, para el que la gramática contiene las siguientes reglas:

```
U::=x X1 X2...Xn | y Y1 Y2...Ym |...| z Z1 Z2...Zp
```

Esta función recibe dos parámetros: la cadena que se va a analizar y un contador que indica una posición dentro de dicha cadena. Si el valor del segundo argumento es un número negativo, la función termina y devuelve dicho valor. La función consta de una instrucción `switch`, con un caso para cada una de las partes derechas de las reglas cuya parte izquierda es el símbolo no terminal para el que se implementa la función. En todos los casos se incrementa el valor del contador y se invocan las funciones de los símbolos no terminales que aparecen en la parte derecha correspondiente. Se añade un caso por defecto, que se ejecuta si ninguna de las partes derechas es aplicable al carácter de la cadena que se está examinando, en el que se devuelve un número negativo, para indicar que no se reconoce la palabra. Este número negativo puede ser distinto para cada función, lo que servirá para identificar cuál es la función que ha generado el error. Para el resto de los casos, se devuelve el valor final del contador.

La Figura 4.9 muestra la función escrita en C que corresponde a un símbolo no terminal con regla-λ, para el que la gramática contiene las siguientes reglas:

```
U ::= x X1 X2 ... Xn | ... | z Z1 Z2 ... Zp | λ
```

```
int U (char *cadena, int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case x:
        i++;
        i = X1 (cadena, i);
        i = X2 (cadena, i);
        . . .
        i = Xn (cadena, i);
        break;
      case y:
        i++;
        i = Y1 (cadena, i);
        i = Y2 (cadena, i);
        . . .
        i = Ym (cadena, i);
        break;
      case Z:
        i++;
        i = Z1 (cadena, i);
        i = Z2 (cadena, i);
        . . .
        i = Zp (cadena, i);
        break;
      default: return -n;
    }
    return i;
  }
```

Figura 4.8. Función para un símbolo no terminal sin regla-λ.

La única diferencia con la función que aparece en la Figura 4.8 es que en la instrucción `switch` no se incluye el caso por defecto, porque la aplicación de la regla-λ significa que la función correspondiente al símbolo no terminal debe devolver el mismo valor del contador que recibió, es decir, no debe avanzar en la cadena de entrada.

```
int U (char *cadena, int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case x:
        i++;
        i = X1 (cadena, i);
        i = X2 (cadena, i);
        . . .
        i = Xn (cadena, i);
        break;
      . . .
      case Z:
        i++;
        i = Z1 (cadena, i);
        i = Z2 (cadena, i);
        . . .
        i = Zp (cadena, i);
        break;
        }
    return i;
  }
```

Figura 4.9. Función para un símbolo no terminal con regla-λ.

Ejemplo 4.6 Consideremos la gramática siguiente:

```
E ::= T + E
E ::= T - E
E ::= T
T ::= F * T
T ::= F / T
T ::= F
F ::= i
F ::= (E)
```

En forma normal de Greibach, la gramática queda así:

```
E ::= iPTME  |(ECPTME | iDTME | (ECDTME | iME | (ECME
     | iPTSE |(ECPTSE | iDTSE | (ECDTSE | iSE | (ECSE
     | iPT   |  (ECPT | iDT   | (ECDT   | i   | (EC
T ::= iPT    |  (ECPT | iDT   | (ECDT   | i   | (EC
```

```
F ::= i | (EC
M ::= +
S ::= -
P ::= *
D ::= /
C ::= )
```

A partir de esta gramática, se obtiene la siguiente gramática LL(1):

```
E ::= iV  | (ECV
V ::= *TX | /TX   | +E | -E | λ
X ::= +E  | -E    | λ
T ::= iU  | (ECU
U ::= *T  | /T    | λ
F ::= i   | (EC
C ::= )
```

Las funciones correspondientes a los siete símbolos no terminales que aparecen en esta gramática se muestran en las Figuras 4.10 a 4.16.

```
int E (char *cadena, int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case 'i':
        i++;
        i = V (cadena, i);
        break;
      case '(':
        i++;
        i = E (cadena, i);
        i = C (cadena, i);
        i = V (cadena, i);
        break;
      default: return -1;
    }
    return i;
  }
```

Figura 4.10. Función para el símbolo no terminal E.

```
int V (char *cadena, int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case '*':
      case '/':
        i++;
        i = T (cadena, i);
        i = X (cadena, i);
        break;
      case '+':
      case '-':
        i++;
        i = E (cadena, i);
        break;
    }
    return i;
  }
```

Figura 4.11. Función para el símbolo no terminal V.

```
int X (char *cadena, int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case '+':
      case '-':
        i++;
        i = E (cadena, i);
        break;
    }
    return i;
  }
```

Figura 4.12. Función para el símbolo no terminal X.

```
int T (char *cadena, int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case 'i':
        i++;
        i = U (cadena, i);
        break;
      case '(':
        i++;
        i = E (cadena, i);
        i = C (cadena, i);
        i = U (cadena, i);
        break;
      default: return -2;
    }
    return i;
  }
```

Figura 4.13. Función para el símbolo no terminal T.

```
int U (char *cadena, int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case '*':
      case '/':
        i++;
        i = T (cadena, i);
        break;
    }
    return i;
  }
```

Figura 4.14. Función para el símbolo no terminal U.

```
int F (char *cadena, int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case 'i':
        i++;
        break;
      case '(':
        i++;
        i = E (cadena, i);
        i = C (cadena, i);
        break;
      default: return -3;
    }
    return i;
  }
```

Figura 4.15. Función para el símbolo no terminal F.

```
int C (char *cadena, int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case ')':
        i++;
        break;
      default: return -4;
    }
    return i;
  }
```

Figura 4.16. Función para el símbolo no terminal C.

Análisis de cadenas

Para analizar una cadena `x` con este método, basta invocar la función correspondiente al axioma de la gramática con los argumentos `x` (la cadena a analizar) y 0 (el valor inicial del contador). En el ejemplo, la llamada sería `E(x,0)`. Si el valor devuelto por la función coincide con la longitud de la cadena de entrada, la cadena queda reconocida. En caso contrario, la función devolverá un número negativo, que indica el error detectado.

La Figura 4.17 muestra el análisis de la palabra `i+i*i`. La llamada a la función correspondiente al axioma de la gramática devuelve el valor 5, que coincide con la longitud de la cadena de entrada, por lo que la palabra es reconocida.

```
E ("i+i*i", 0) = V ("i+i*i", 1) =
               = E ("i+i*i", 2) =
               = V ("i+i*i", 3) =
               = X ("i+i*i", T ("i+i*i", 4)) =
               = X ("i+i*i", U ("i+i*i", 5)) =
               = X ("i+i*i", 5) =
               = 5
```

Figura 4.17. Análisis de la cadena `i+i*i`.

Sin embargo, como ilustra la Figura 4.18, el análisis de la palabra `i+i*` devuelve el valor –2, lo que indica que no se reconoce la palabra, y que el error lo devolvió la llamada a la función correspondiente al símbolo no terminal `T`.

```
E ("i+i*", 0) = V ("i+i*", 1) =
              = E ("i+i*", 2) =
              = V ("i+i*", 3) =
              = X ("i+i*", T ("i+i*", 4)) =
              = -2
```

Figura 4.18. Análisis de la cadena `i+i*`.

4.2.4. Análisis LL(1) mediante el uso de tablas de análisis

Otra forma de realizar el análisis descendente selectivo utiliza una tabla de análisis, cuyas filas son los símbolos no terminales de la gramática y sus columnas son los símbolos terminales y el símbolo de fin de cadena `$`. En las celdas de la tabla aparecen reglas de la gramática. Las celdas vacías corresponden a un error en el análisis.

Las celdas de la tabla se rellenan aplicando el siguiente procedimiento para cada producción `A ::=` α de la gramática:

- Para cada símbolo terminal `a` $\in$ `primero(`α`)`, añadir la producción `A ::=` α en la celda `T[A,a]`.
- Si $\lambda \in$ `primero(`α`)`, para cada símbolo terminal `b` $\in$ `siguiente(A)`, añadir la producción `A ::=` α en la celda `T[A,b]`. Obsérvese que `b` también puede ser `$`.

Como ejemplo, considérese la gramática del Ejemplo 4.1, que es la siguiente:

```
(1)  E ::= TE'
(2)  E' ::= +TE'
(3)  E' ::= λ
(4)  T ::= FT'
(5)  T' ::= *FT'
```

```
(6)  T' ::= λ
(7)  F ::= (E)
(8)  F ::= id
```

Los conjuntos primero y siguiente para los símbolos no terminales de esta gramática son:

```
primero(E)=primero(T)=primero(F)={(,id}
primero(E')={+, λ}
primero(T')={*, λ}
siguiente(E)=siguiente(E')={),$}
siguiente(T)=siguiente(T')={+,),$}
siguiente(F)={*,+,),$}
```

La producción `E ::= TE'` debe colocarse en la fila correspondiente al símbolo `E`, en las columnas correspondientes a los símbolos terminales del conjunto `primero(TE')`. Si se aplica el procedimiento para calcular el conjunto `primero` de una forma sentencial, hay que calcular el conjunto `primero(T) = {(,id}`.

La producción `E' ::= +TE'` debe colocarse en la fila correspondiente al símbolo `E'` y en las columnas correspondientes a los símbolos terminales del conjunto `primero(+TE')`. Si se aplica el procedimiento para calcular el conjunto `primero` para una forma sentencial, hay que calcular el conjunto `primero(+) = {+}`.

La producción `E' ::= λ` debe colocarse en la fila correspondiente al símbolo `E'` y en las columnas correspondientes a los símbolos terminales del conjunto `siguiente(E') = {),$}`.

Por este procedimiento, se van rellenando las celdas de la tabla, obteniéndose la que muestra la Tabla 4.1.

Una gramática es LL(1) si en la tabla de análisis sintáctico obtenida por el procedimiento anterior aparece como máximo una regla en cada celda.

Tabla 4.1

	id	+	*	(	)	$
E	E::=TE'			E::=TE'		
E'		E'::=+TE'			E'::=λ	E'::=λ
T	T::=FT'			T::=FT'		
T'		T'::=λ			T'::=λ	T'::=λ
F	F::=id			F::=(E)		

Ejemplo 4.7 Consideremos la gramática siguiente:

```
P ::= iEtPP'
P ::= a
```

P'::= eP
P'::= λ
E ::= b

La Tabla 4.2 muestra la tabla de análisis sintáctico para esta gramática. Puede observarse que en la celda correspondiente a la intersección de la fila P' con la columna e aparecen dos reglas. El motivo es que la intersección del conjunto primero(P') = {e, λ} y siguiente(P') = {$, e} no es el conjunto vacío, por lo que la regla-λ para el símbolo P' debe colocarse en una celda que ya está ocupada por la regla P'::=eP.

Tabla 4.2

	a	b	e	i	t	$
P	P::=a			P::=iEtPP'		
P'			P'::=λ P'::=eP			P'::=λ
E		E::=b				

Análisis de cadenas

La tabla de análisis sintáctico puede utilizarse para analizar cadenas mediante el siguiente algoritmo:

1. Inicializar una pila con el símbolo $ y el axioma de la gramática y añadir el símbolo $ al final de la cadena de entrada.
2. Repetir el siguiente procedimiento: comparar el símbolo de la cima de la pila P con el siguiente símbolo de entrada e:
   - Si P y e son iguales al símbolo $, aceptar la cadena y salir.
   - Si son iguales y distintos del símbolo $, extraer un elemento de la pila y avanzar una posición en la cadena de entrada.
   - Si son distintos, y la celda de la tabla de análisis T(P,e) está vacía, emitir mensaje de error y salir.
   - Si son distintos, y en la celda de la tabla de análisis T(P,e) aparece la producción P ::= $X_1 X_2 \ldots X_n$, extraer P de la pila e insertar los símbolos $X_1 X_2 \ldots X_n$ en la pila en el orden inverso a como aparecen en la parte derecha de la producción.

Como ejemplo, la Tabla 4.3 muestra el análisis de la cadena id+id utilizando la tabla de análisis que aparece en la Tabla 4.1. En la Tabla 4.3 aparece el contenido de la pila en cada paso del análisis, así como el estado de la entrada y la acción a realizar. Veamos en detalle algunos de los pasos del análisis. En el primer paso, los símbolos E (símbolo de la cima de la pila) e id (siguiente símbolo de entrada) son distintos, por lo que se aplica la regla E::=TE' que aparece en la celda T(E,id). Como resultado, se extrae el símbolo E de la pila y se insertan los símbolos E' y T, que son los que aparecen en la parte derecha de esta regla, en sentido inverso.

Tabla 4.3. Análisis de la cadena `id+id`.

Paso	Pila	Entrada	Acción
1	$E	id+id$	Aplicar E::=TE'
2	$E'T	id+id$	Aplicar T::=FT'
3	$E'T'F	id+id$	Aplicar F::=id
4	$E'T'id	id+id$	Avanzar
5	$E'T'	+id$	Aplicar T'::=λ
6	$E'	+id$	Aplicar E'::=+TE'
7	$E'T+	+id$	Avanzar
8	$E'T	id$	Aplicar T::=FT'
9	$E'T'F	id$	Aplicar F::=id
10	$E'T'id	id$	Avanzar
11	$E'T'	$	Aplicar T'::=λ
12	$E'	$	Aplicar E'::=λ
13	$	$	Cadena aceptada

En el cuarto paso del análisis, el símbolo de la cima de la pila y el siguiente símbolo de entrada son iguales, por lo que se extrae `id` de la pila y se avanza una posición en la cadena de entrada.

Obsérvese que, como en el paso 5 del análisis, cuando la regla a aplicar es una regla-λ se extrae un símbolo de la pila y no se inserta ninguno.

4.3 Análisis sintáctico ascendente

4.3.1. Introducción a las técnicas del análisis ascendente

En contraposición a las técnicas del análisis sintáctico descendente, los analizadores ascendentes recorren el árbol de derivación de una cadena de entrada correcta desde las hojas (los símbolos terminales) a la raíz (el axioma), en una dirección gráficamente ascendente, lo que da nombre a estas técnicas. El análisis consiste en un proceso iterativo, que se aplica inicialmente a la cadena completa que se va a analizar y termina, bien cuando se completa el análisis con éxito, o bien cuando éste no puede continuar, debido a algún error sintáctico. En cada paso del análisis se intenta deducir qué regla de la gramática se tiene que utilizar en ese punto del árbol, teniendo en cuenta el estado de éste y la posición a la que se ha llegado en la cadena de entrada. En general, al final de cada paso del análisis se ha modificando la cadena de entrada, que queda preparada para continuar.

En este tipo de análisis se pueden realizar dos operaciones fundamentales: la reducción y el desplazamiento.

Reducción

Se aplica cuando se ha identificado en la cadena de entrada la parte derecha de alguna de las reglas de la gramática. Esta operación consiste en reemplazar, en la cadena de entrada, dicha parte derecha por el símbolo no terminal de la regla correspondiente. La Figura 4.19 muestra gráficamente cómo se realiza esta operación.

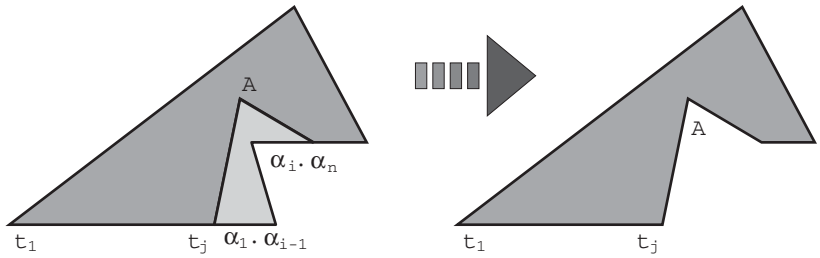


Figura 4.19. Representación gráfica de la reducción de la regla $A \rightarrow \alpha_i \cdots \alpha_n$.

Desplazamiento

Intuitivamente, los analizadores ascendentes guardan información que les permite saber, en cada momento, qué partes derechas de las reglas de la gramática son compatibles con la porción de la cadena de entrada analizada, entre todas las reglas posibles. Como, en general, las partes derechas de las reglas tienen más de un símbolo, en cada paso del análisis no siempre se puede reducir una regla. Se llama *desplazamiento* la operación mediante la cual se avanza un símbolo, simultáneamente en la cadena de entrada y en todas las reglas que siguen siendo compatibles con la porción de entrada analizada.

La Figura 4.20 muestra gráficamente un ejemplo de esta operación. La parte inferior de la figura muestra la cadena de entrada. La superior, un subconjunto de reglas de la gramática. El bloque izquierdo muestra la situación anterior al desplazamiento: la cadena de entrada ha sido analizada hasta el terminal t_j. El subconjunto de reglas contiene aquellas cuyas partes derechas han sido compatibles con la parte de la cadena de entrada analizada, hasta el símbolo t_j inclusive. El bloque derecho muestra la situación después del desplazamiento. Sólo se resaltan las reglas que siguen siendo compatibles con el siguiente símbolo de entrada (t_{j+1}).

El algoritmo básico del análisis ascendente, que se explicará con detalle en las próximas secciones, puede describirse de la siguiente manera en función de las operaciones de reducción y desplazamiento:

- Se inicia el proceso con el primer símbolo de la cadena de entrada.

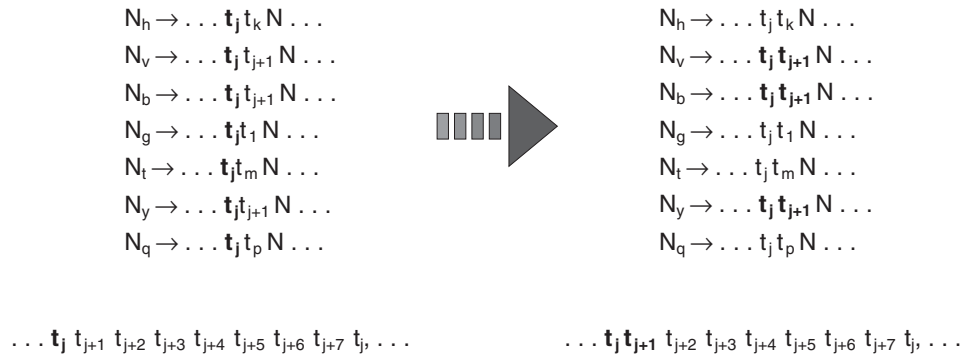


Figura 4.20. Representación gráfica de la operación de desplazamiento.

- Se realiza el siguiente paso del análisis, hasta que se determine que la cadena es sintácticamente correcta (si se ha recorrido la entrada completa, reduciéndola al axioma) o incorrecta (si en algún instante el análisis no puede continuar).

  1. Si una regla se puede reducir (toda su parte derecha se ha *desplazado*) se reduce. La nueva cadena de análisis es el resultado de reemplazar la parte de la cadena correspondiente a la parte derecha de la regla por el símbolo no terminal situado a la izquierda de la misma. El análisis continuará a partir de dicho símbolo no terminal.
  2. En otro caso, se realiza un desplazamiento sobre el símbolo correspondiente al paso de análisis actual. Esto significa que se descartan las reglas cuyo símbolo siguiente, en la parte derecha, no sea compatible con el desplazado, mientras se avanza en las partes derechas en las que sea posible y en la cadena de entrada.

4.3.2. Algoritmo general para el análisis ascendente

La mayoría de los algoritmos de análisis sintáctico de tipo ascendente se realizan en dos fases: en la primera, se construye una tabla auxiliar para el análisis sintáctico ascendente, que en el segundo paso se utilizará en el análisis de las cadenas de entrada.

Como se ha indicado previamente, los analizadores de los lenguajes independientes del contexto pueden basarse en el autómata a pila asociado a su gramática. Hay dos diferencias principales entre los analizadores ascendentes: la manera en que construyen la tabla de análisis y la información que necesitan introducir en la pila. En este último aspecto, las técnicas más potentes, LR(1) y LALR(1), necesitan más información que las más simples, LR(0) y SLR(1). Para simplificar, en este capítulo se utilizará el mismo algoritmo para todos ellos, aunque sea a costa de introducir en la pila información redundante para las técnicas LR(0) y SLR(1).

Estructura general de las tablas de análisis ascendente

Las tablas del análisis contienen una información *equivalente* a la función de transición del autómata a pila que reconocería el lenguaje asociado a la gramática que se está analizando. Dentro

de este autómata a pila se puede distinguir la parte que tiene por objeto reconocer los asideros de la gramática, que en realidad es un autómata finito. A lo largo de todo el capítulo se utilizará el nombre *autómata de análisis* para referirse a dicho autómata.

- El número de filas varía en función de la técnica utilizada para construir la tabla y coincide con el número de estados del autómata. Cada técnica puede construir un autómata distinto, con diferentes estados.
- Tendrán tantas columnas como símbolos hay en el alfabeto de la gramática. Usualmente, la tabla se divide en dos secciones, que corresponden, respectivamente, a los símbolos terminales y no terminales. Las columnas de la tabla asociadas a los símbolos terminales forman el bloque de *acción*, ya que son ellas las que determinan la acción siguiente que el analizador debe realizar. Las restantes columnas de la tabla (las columnas asociadas a los símbolos no terminales) sólo conservan información relacionada con las transiciones del autómata y forman el bloque *ir_a*. Sus casillas sólo contienen la identificación del estado al que hay que transitar.
- Para poder utilizarla en el análisis, la tabla debe especificar, en función del símbolo terminal de la cadena de entrada que se recibe y del estado en que se encuentra el autómata, cuál será el estado siguiente; qué modificaciones deben realizarse en la cadena de entrada y en la pila; si se produce un desplazamiento o una reducción, si se ha terminado con éxito el análisis o si se ha detectado algún error. Para especificar esta información, a lo largo de este capítulo se utilizará la siguiente notación:

  1. **d<e>**, donde **d** significa **d**esplazamiento y **<e>** identifica un estado del autómata de análisis. Representa la acción de desplazar el símbolo actual y pasar al estado **<e>**.
  2. **r<p>**, donde **<p>** identifica una regla de producción de la gramática. Representa la acción de **r**educción de la regla **<p>**.
  3. **Aceptar**, que representa la finalización con éxito del análisis.
  4. **Error**, que representa la finalización sin éxito del análisis.

La Figura 4.21 muestra gráficamente la estructura de estas tablas.

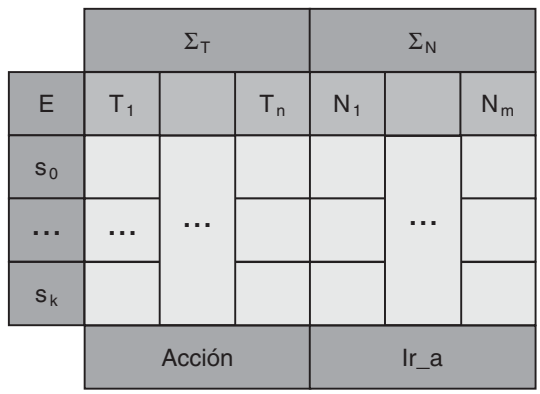


Figura 4.21. Estructura de las tablas de análisis de los analizadores sintácticos ascendentes.

Algoritmo de análisis ascendente

La especificación completa del algoritmo general de análisis ascendente describe las manipulaciones realizadas sobre la tabla, la entrada y la pila, asociadas con la acción correspondiente a cada paso del análisis. La Figura 4.22 muestra gráficamente el esquema del analizador. Tanto el significado de las columnas de *acción* como de las de *ir_a*, así como el contenido de la pila, son objeto de próximas secciones.

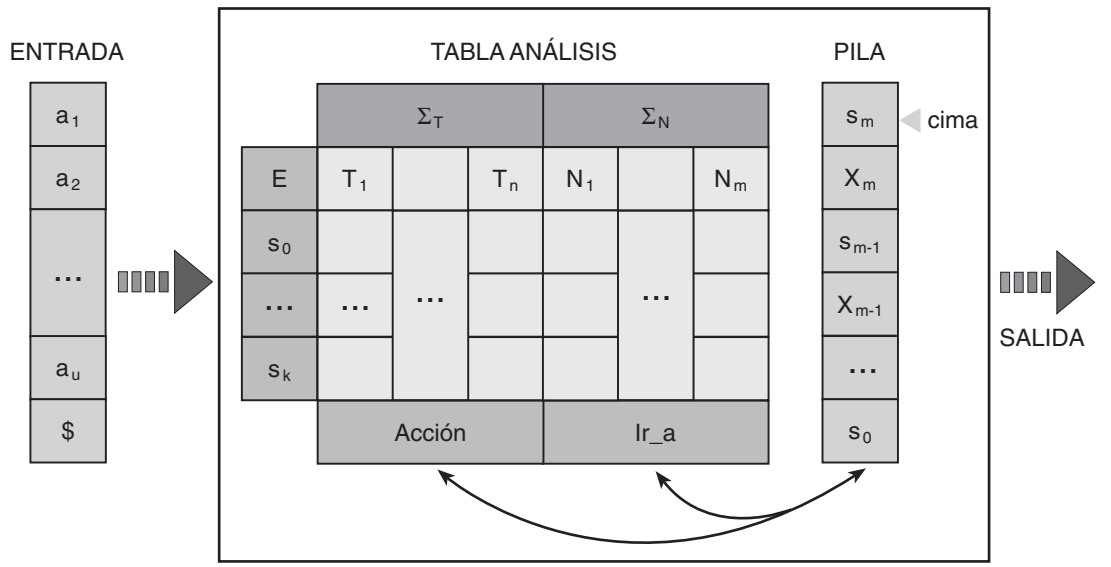


Figura 4.22. Estructura de un analizador sintáctico ascendente.

La Figura 4.23 muestra el algoritmo general de análisis en pseudocódigo.

Puede observarse que el algoritmo es un bucle en el que se consulta la tabla del análisis para descubrir la acción que hay que realizar. A continuación se estudia con detalle el tratamiento de cada tipo de operación. Se utilizará como ejemplo la cadena `i+i+i` y la siguiente gramática:

```
{     ΣT={+,i,(,),$[1]},
      ΣN={E'[1],E,T},
      E',
      {     E' ::= E$[1],
            E ::= E+T | T,
            T ::= i  | (E)} }
```

[1] Más adelante se verá que la introducción del símbolo no terminal `E'`, el símbolo terminal `$` y la regla `E'::=E$` son pasos generales del algoritmo de análisis.

```
Estado AnalizadorAscendente(tabla_análisis,
                            entrada, pila, gramática)
/* La entrada contiene la cadena w$ que se quiere analizar se
deja vacía la posición 0 para las reducciones */
{
  puntero símbolo_actual=1;
  /* la posición 0 está vacía por si fuese necesario al
  reducir */
  estado  estado_actual;
  push(pila,0);
  while( verdadero ) /* Bucle sin fin */
  { estado_actual=cima(pila);
    if (tabla_análisis[estado_actual, entrada[símbolo_actual]]
         == ds' ) {
         push(pila, entrada[símbolo_actual]);
         push(pila, s');
         símbolo_actual++;}
     else
     if ( tabla_análisis[estado_actual, entrada[símbolo_ac-
         tual]]
         == rj ) {
     /* Podemos suponer que la regla j es A::=α */
         `realizar 2*longitud(α) pop(pila)´
         entrada[--símbolo_actual] = A;
         printf("Reducción de A::=α");)
     else
     if (  tabla_análisis[estado_actual,  entrada[símbolo_ac-
         tual]]
         == aceptar )
         return CADENA_ACEPTADA;
     else /* casilla vacía */
         return CADENA_RECHAZADA:_ERROR_SINTÁCTICO);
  }}
```

Figura 4.23. Pseudocódigo del algoritmo general de análisis ascendente.

Es fácil comprobar que el lenguaje asociado está compuesto por expresiones aritméticas formadas por sumas entre paréntesis opcionales, y el símbolo `i` como único operando.

- **Inicio del análisis**

Se supone que se dispone de la tabla de análisis que se muestra en la Figura 4.24.

	Σ_T					Σ_N	
E	i	+	(	)	$	E	T
0	d1	(*)	d2			4	3
1		d3		d3	d3		
2	d1		d2			5	3
3		d2		d2	d2		
4		d6			acc		
5		d6		d8			
6	d1		d2				7
7		r1		r1	r1		
8		r4		r4	r4		
	Acción					Ir_a	

(*) Se considera que todas las casillas vacías representan la acción de error.

Figura 4.24. Una tabla de análisis correspondiente a la gramática
{Σ_T={+,i,(,),$ }, Σ_N={E',E,T}, E', {E'→E$,E→E+T|T,T→i|(E)}}.

Se añade a la entrada el símbolo $, que indica que se ha llegado al final de la cadena. La pila contendrá inicialmente el estado inicial del autómata, que en este capítulo será el estado etiquetado con el número 0. La Figura 4.25 muestra el paso inicial, realizado con la gramática del ejemplo.

Es importante resaltar que el algoritmo descrito utiliza la pila para conservar toda la información necesaria para continuar el análisis. Para ello, a excepción de esta situación inicial, en la que sólo se introduce un estado, la información mínima que se inserte o se saque de la pila será usualmente un par de datos: el estado del analizador y el símbolo de la gramática considerado en ese instante. Esto permite utilizar el mismo algoritmo de análisis en todas las técnicas, aunque para alguna de ellas bastaría con introducir el estado.

- **Desplazamiento d<e>**

Como se ha indicado, las filas de la tabla representan los estados del autómata asociado a la gramática y, para poder utilizarla en el análisis, tienen que conservar simultáneamente información sobre todas las reglas cuyas partes derechas son compatibles con la parte de la cadena de entrada que ya ha sido analizada.

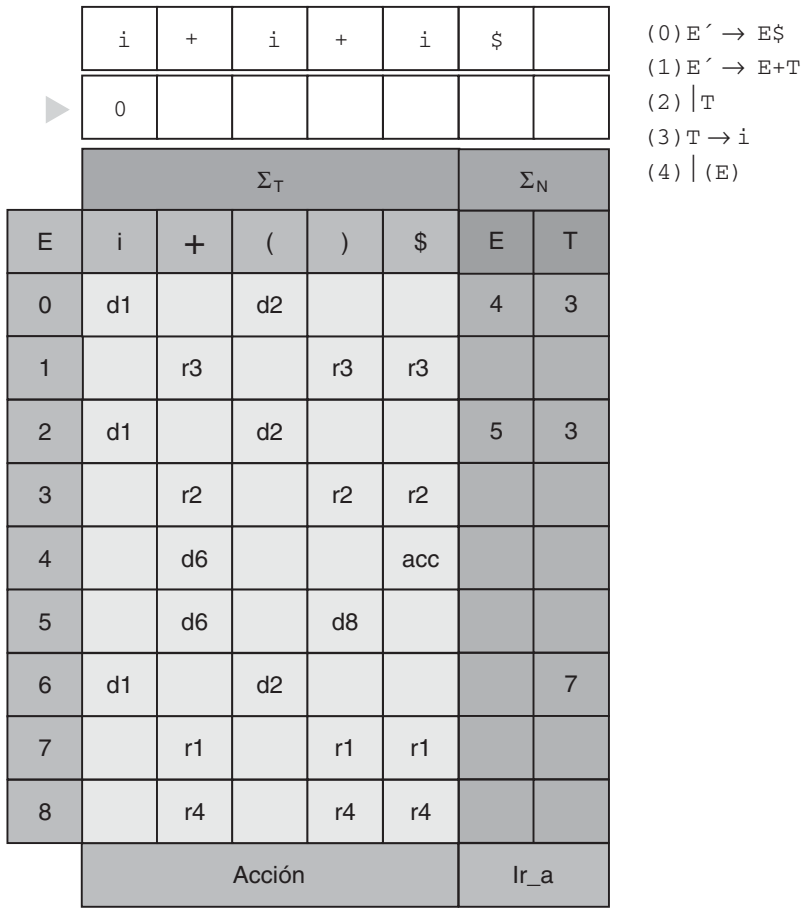


Figura 4.25. Situación inicial para el análisis de la cadena `i+i+i`.

La indicación de desplazamiento significa que el símbolo actualmente estudiado en la cadena de entrada es uno de los que espera alguna de las reglas con partes derechas parcialmente analizadas. Por lo tanto, se puede avanzar una posición en la cadena de entrada, y el autómata puede pasar al estado que corresponde al desplazamiento de ese símbolo en las partes derechas de las reglas en las que dicho desplazamiento sea posible. Esta operación implicará realizar las siguientes operaciones:

- Se introduce en la pila el símbolo de entrada.
- Para tener en cuenta el cambio de estado del autómata, el estado indicado por la operación (**<e>**) también se introduce en la pila.
- Se avanza una posición en la cadena de entrada, de manera que el símbolo actual pase a ser el siguiente al recién analizado.

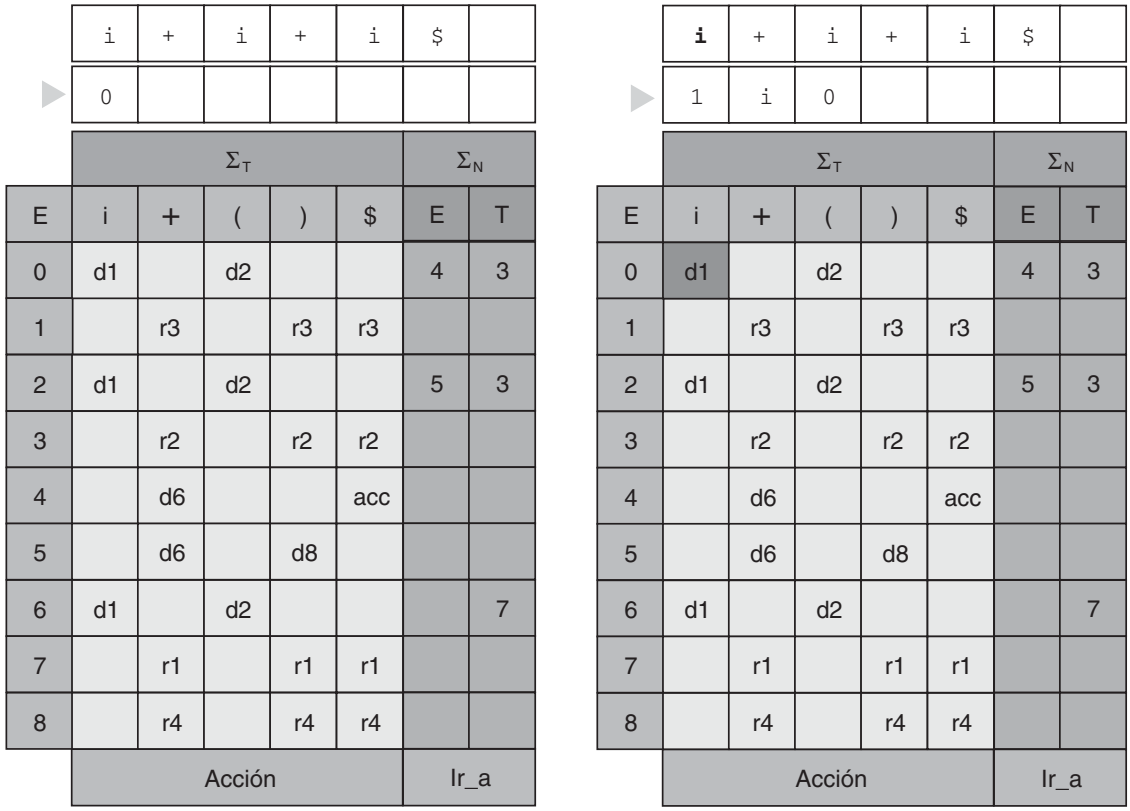


Figura 4.26. Ejemplo de operación de desplazamiento en el análisis de la cadena `i+i+i`.

En el ejemplo se puede comprobar que la primera acción de análisis en el tratamiento de la cadena `i+i+i` es un desplazamiento. El analizador está en el estado 0 y el próximo símbolo que hay que analizar es `i`. La casilla (0,`i`) de la tabla del análisis contiene la indicación **d1,** es decir, desplazamiento al estado 1. Por lo tanto, hay que introducir en la pila el símbolo `i` y el número 1. La Figura 4.26 muestra gráficamente este paso del análisis.

• Reducción r<p>

La indicación de reducción en una casilla de la tabla significa que, teniendo en cuenta el símbolo actual de la cadena de entrada, alguna de las reglas representadas en el estado actual del autómata ha desplazado su parte derecha completa, que puede ser sustituida por el símbolo no terminal de su parte izquierda. Como resultado de esta acción, el analizador debe actuar de la siguiente forma:

- Se saca de la pila la información asociada a la parte derecha de la regla **<p>**. Supongamos que la regla <p> es `N::=`α. En la pila hay dos datos por cada símbolo, por lo que tendrán que realizarse tantas operaciones `pop` como el doble de la longitud de α.

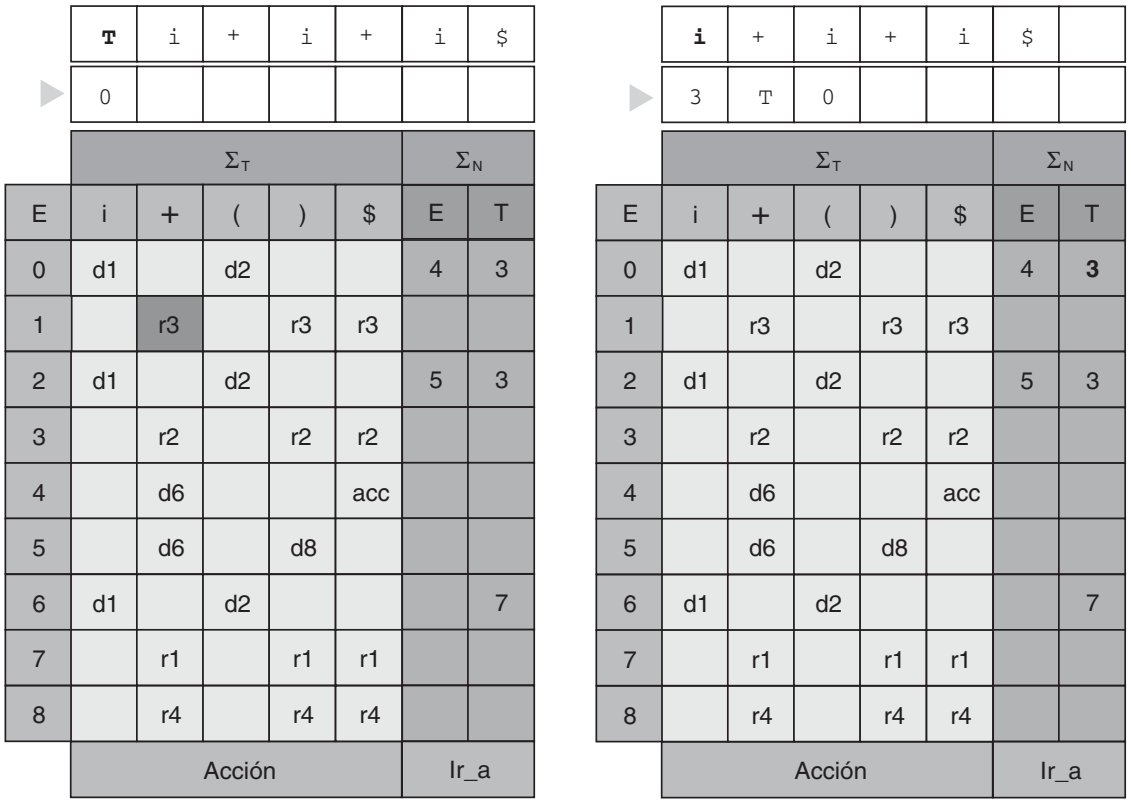


Figura 4.27. Ejemplo de operación de reducción. La casilla de la tabla (3,+) contiene la indicación **r3**.

- Se introduce el símbolo no terminal de la regla (N) a la izquierda de la posición actual de la cadena de entrada, y se apunta a dicho símbolo.

Es fácil comprobar en el ejemplo que la segunda acción, tras el desplazamiento anterior, es una reducción, ya que la casilla correspondiente al símbolo actual (+) y al estado que ocupa la cima de la pila (1), indica que debe reducirse la regla 3: `T ::= i`. Como sólo hay un símbolo en la parte derecha, hay que ejecutar dos `pop` sobre la pila e insertar a la izquierda de la porción de la cadena de entrada pendiente de analizar la parte izquierda de la regla (`T`), que pasa a ser el símbolo actual. El estado que queda en la pila es 0. La casilla (0, `T`) de la tabla de análisis indica que se tiene que ir al estado 3. La Figura 4.27 muestra gráficamente este paso del análisis.

La Figura 4.28 ilustra otra reducción cuya regla asociada tiene una parte derecha de longitud mayor que 1. Se trata de la regla `1: E ::= E+T`. En este caso habrá que ejecutar `6 (2*3)` operaciones `pop` en la pila. Tras realizarlas, la cima de la pila contiene el estado 0. Se añade en la posición correspondiente de la cadena de entrada la parte izquierda (`E`) y se continúa el análisis.

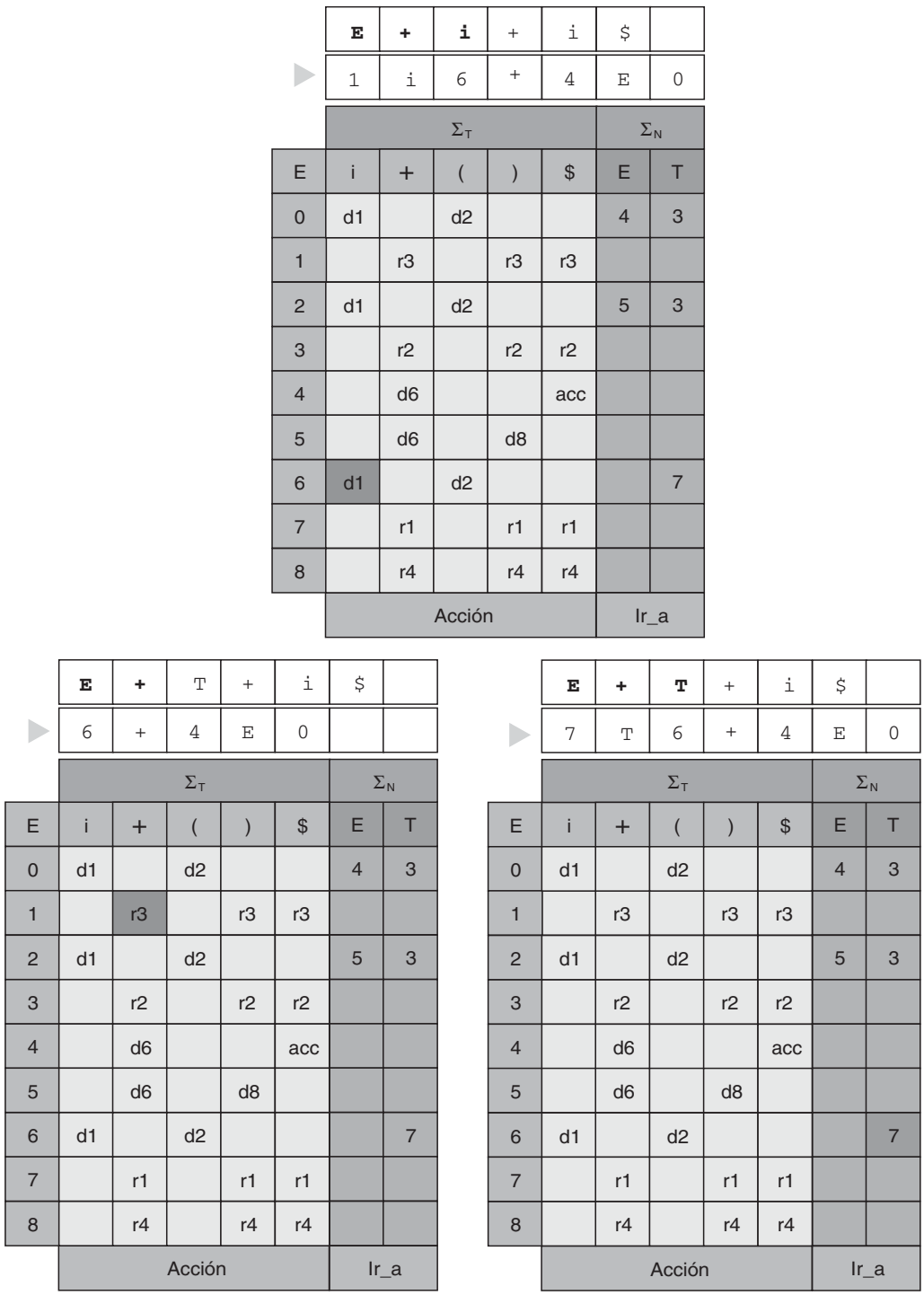


Figura 4.28. Reducción de la regla 1 en un estado intermedio del análisis de la cadena `i+i+i`.

- **Aceptación**

Cuando en la ejecución del algoritmo se llega a una casilla que contiene esta indicación, el análisis termina y se concluye que la cadena de entrada es sintácticamente correcta. La Figura 4.29 muestra el final del análisis de la cadena `i+i+i` en el caso del ejemplo de los puntos anteriores. Obsérvese el papel del símbolo `$`, que se añadió precisamente para facilitar la identificación de esta circunstancia.

	E	**+**	**E**	**+**	**E**	$	
▶	4	E	0				

	Σ_T					Σ_N	
E	i	+	(	)	$	E	T
0	d1		d2			4	3
1		r3		r3	r3		
2	d1		d2			5	3
3		r2		r2	r2		
4		d6			acc		
5		d6		d8			
6	d1		d2				7
7		r1		r1	r1		
8		r4		r4	r4		
	Acción					Ir_a	

Figura 4.29. Fin de análisis: la cadena `i+i+i` es correcta.

- **Error**

Cuando la ejecución del algoritmo llega a una casilla con esta indicación, el análisis termina y se concluye que la cadena de entrada contiene errores sintácticos. Las tablas del análisis suelen contener más casillas con esta indicación que con cualquiera de las anteriores. Es frecuente dejar estas casillas en blanco para facilitar el manejo de la tabla. La Figura 4.30 muestra un ejemplo de análisis de la cadena '`(+i)`' con la gramática

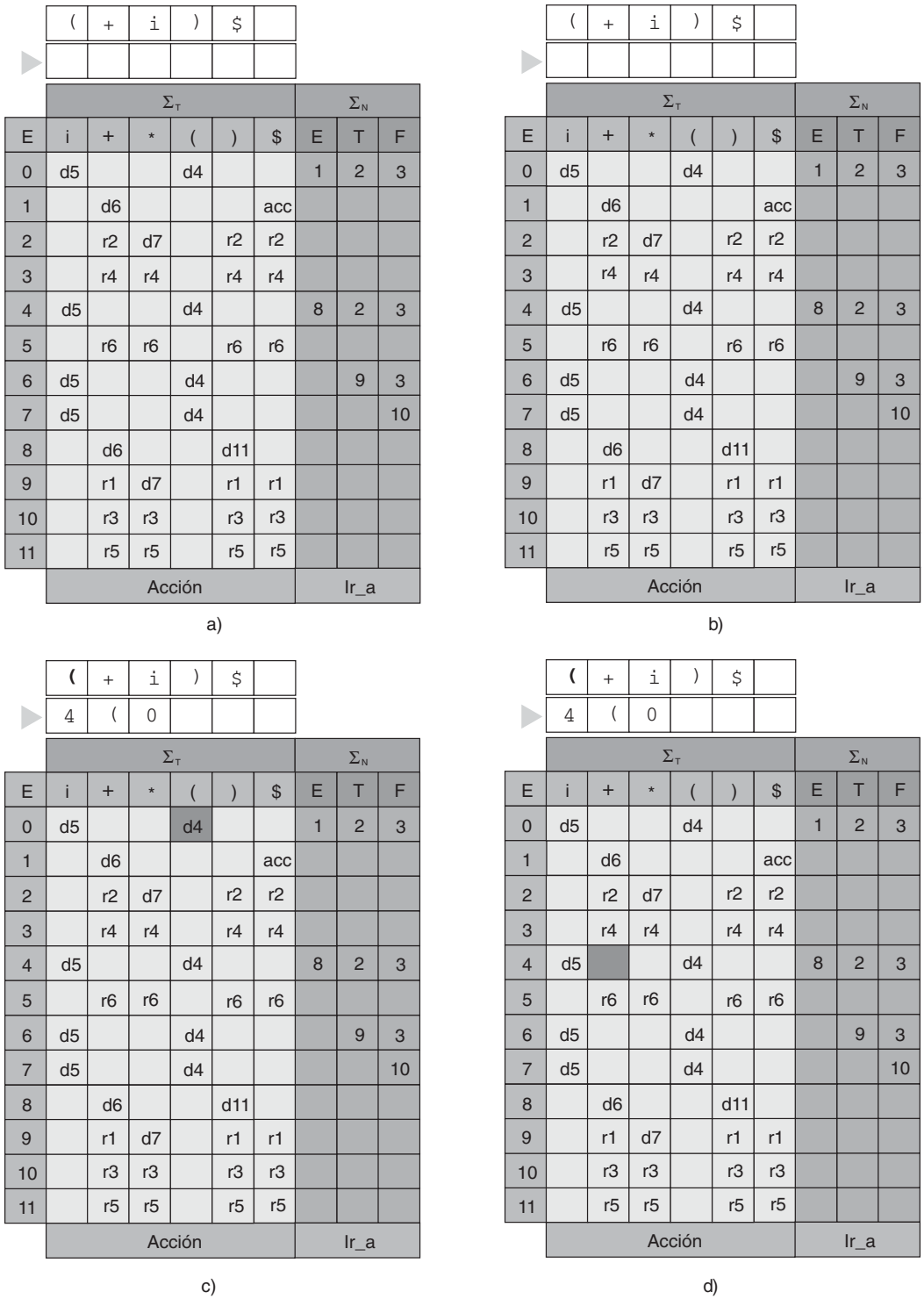


Figura 4.30. Ejemplo de análisis que termina con error sintáctico.

```
{    Σ_T={+,*,i,(,)},
     Σ_N={E,T,F},
     E,
     {      E ::= E+T | T,
            T ::= T*F | F,
            F ::= (E) | i} }
```

y que termina con error sintáctico. Es fácil comprobar que esta gramática genera expresiones aritméticas con los operadores binarios '+' y '*' y con el símbolo `i` como único operando. Las expresiones permiten el uso opcional de paréntesis.

Obsérvese:

(a) y (b) Muestran respectivamente la situación previa e inicial al análisis.

(c) La casilla (0, `(`) contiene la operación **d4,** por lo que se inserta en la pila el símbolo '`(`' y el estado 4 y se avanza una posición en la cadena de entrada; el símbolo actual es '+'.

(d) La casilla (4,+) está vacía, es decir, indica que se ha producido un error sintáctico. El error es que se ha utilizado el símbolo '+' como operador monádico cuando la gramática lo considera binario.

4.3.3. Análisis LR(0)

Las siglas LR describen una familia de analizadores sintácticos que

- Examinan la entrada de izquierda a derecha (del inglés ***L**eft to right*).
- Construyen una derivación derecha de la cadena analizada (del inglés ***R**ightmost derivation*).

Las siglas LR(k) hacen referencia a que, para realizar el análisis, se utilizan los k símbolos siguientes de la cadena de entrada a partir del *actual*. Por lo tanto, LR(0) es el analizador de esa familia que realiza cada paso de análisis teniendo en cuenta únicamente el símbolo actual.

Configuración o elemento de análisis LR(0)

Como se ha indicado anteriormente, los analizadores ascendentes siguen simultáneamente todos los caminos posibles; es decir, cada estado del autómata de análisis conserva todas las reglas cuyas partes derechas son compatibles con la porción de entrada ya analizada. Cada una de esas reglas es hipotética, pues al final sólo una de ellas será la aplicada.

En los apartados siguientes se estudiará con detalle la construcción del autómata de análisis LR(0). Para ello, es necesario explicitar en un objeto concreto cada una de las hipótesis mencionadas. Informalmente, se llamará *configuración* o *elemento de análisis* a la representación de cada una de las hipótesis.

Una *configuración* o *elemento de análisis LR(0)* es una regla, junto con una indicación respecto al punto de su parte derecha hasta el que el analizador ha identificado que dicha regla es compatible con la porción de entrada analizada.

Es posible utilizar diversas notaciones para las *configuraciones LR(0)*. En este capítulo mencionaremos las dos siguientes:

- **Notación explícita:** Consiste en escribir la regla completa y marcar con un símbolo especial la posición de la parte derecha hasta la que se ha analizado. Suele utilizarse el símbolo '•' o el símbolo '_', colocándolo entre la subcadena ya procesada de la parte derecha y la pendiente de proceso. A lo largo de este capítulo se utilizará el nombre *apuntador de análisis* para identificar este símbolo. A continuación se muestran algunos ejemplos:

  `E ::= •E+T`

  Indica que es posible que se utilice esta regla en el análisis de la cadena de entrada, aunque, por el momento, el analizador aún no ha procesado información suficiente para avanzar ningún símbolo en la parte derecha de la regla. Para que finalmente sea ésta la regla utilizada, será necesario reducir parte de la cadena de entrada al símbolo no terminal `E`, encontrar a continuación el terminal +, y reducir posteriormente otra parte de la cadena de entrada al símbolo no terminal `T`.

  `E ::= E+•T`

  Indica que es posible que en el análisis de la cadena de entrada se vaya a utilizar esta regla, y que el analizador ya ha podido reducir parte de la cadena de entrada al símbolo no terminal `E`, encontrando a continuación el símbolo terminal '+'. Antes de utilizar esta regla, será necesario reducir parte de la cadena de entrada al símbolo no terminal '`T`'.

  `E ::= E+T•`

  Indica que el analizador ya ha comprobado que toda la parte derecha de la regla es compatible con la parte de la cadena de entrada ya analizada. En este momento se podría reducir toda la parte de la cadena de entrada asociada a `E+T` y sustituirla por el símbolo no terminal `E` (la parte izquierda de la regla). Las configuraciones de este tipo se denominan **configuraciones de reducción**. Todas las configuraciones que no son de reducción son **configuraciones de desplazamiento**.

  `P ::= •`

  Esta configuración indica que es posible reducir la regla-λ `P ::=` λ. Siempre que se llegue a una configuración asociada a una regla lambda, será posible reducirla. Se trata, por tanto, de una configuración de reducción.

  Esta notación es más farragosa y menos adecuada para programar los algoritmos, pero resulta más legible, por lo que será utilizada a lo largo de este capítulo.

- **Notación de pares numéricos:** También se puede identificar una configuración con un par de números. El primero es el número de orden de la regla de que se trate en el conjunto de las reglas de producción. El segundo indica la posición alcanzada en la parte derecha de la regla, utilizando el 0 para la posición anterior al primer símbolo de la izquierda, e incrementando en 1 a medida que se avanza hacia la derecha. Esta notación es equivalente a la anterior y facilita la programación de los algoritmos, pero resulta menos legible.

A continuación se muestran las correspondencias entre ambas notaciones en los ejemplos anteriores, suponiendo que la regla `E ::= E+T` es la número 3 y la regla `P ::= λ` es la número 4.

`E ::= •E+T` ⇔ (3,0)
`E ::= E+•T` ⇔ (3,2)
`E ::= E+T•` ⇔ (3,3)
`P ::= •` ⇔ (4,0)

Ejemplo 4.8

Puesto que el analizador sintáctico LR(0) se basa en el autómata de análisis LR(0), será objetivo de los siguientes apartados describir detalladamente su construcción. Con este ejemplo se justificarán intuitivamente los pasos necesarios, que luego se formalizarán en un algoritmo. Se utilizará como ejemplo la gramática que contiene las siguientes reglas de producción (el axioma es E'; obsérvese que la primera regla ya incorpora el símbolo de fin de cadena).

```
(0) E' ::= E$
(1) E ::= E+T
(2)     | T
(3) T ::= i
(4)     |(E)
```

- **Estado inicial del autómata.** El estado inicial contiene las configuraciones asociadas con las hipótesis previas al análisis; el apuntador de análisis estará situado delante del primer símbolo de la cadena de entrada, y se trata de reducir toda la cadena al axioma. La hipótesis inicial tiene que estar relacionada con la regla del axioma. A lo largo del capítulo se verá cómo se puede asegurar que el axioma sólo tenga una regla. Esta hipótesis tiene que procesar la parte derecha completa de esa regla completa, por lo que el estado inicial tiene que contener la siguiente configuración:

  `E' ::= •E$`

  Esta configuración representa la hipótesis de que se puede reducir toda la cadena de entrada al símbolo no terminal `E`, ya que a continuación sólo hay que encontrar el símbolo terminal que indica el fin de la cadena. Un símbolo no terminal nunca podrá encontrarse en la cadena de entrada original, pues sólo aparecerá como resultado de alguna reducción. Por ello, la hipótesis representada por una configuración que espere encontrar a continuación un símbolo no terminal obligará a mantener simultáneamente todas las hipótesis que esperen encontrar a continuación cualquiera de las partes derechas de las reglas de dicho símbolo no terminal, ya que la reducción de cualquiera de ellas significaría la aparición esperada del símbolo no terminal. La Figura 4.31 muestra gráficamente esta circunstancia.

  A lo largo de la construcción del autómata aparecerán muchas configuraciones que indiquen que el analizador está situado justo delante de un símbolo no terminal. La reflexión anterior se podrá aplicar a todas esas situaciones y se implementará en la operación *cierre*, que se aplica a conjuntos de configuraciones y produce conjuntos de configuraciones. Por lo tanto, el estado inicial del autómata debe contener todas las hipótesis *equivalentes* a la

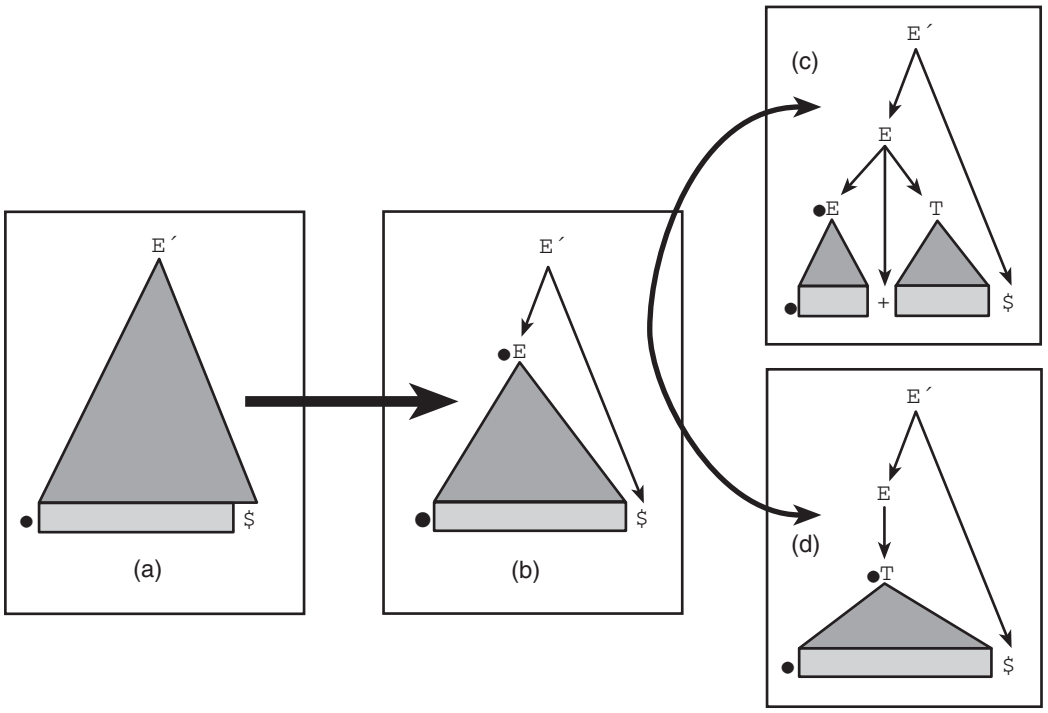


Figura 4.31. Ejemplo de cierre de configuración cuando el análisis precede un símbolo no terminal. (a) Situación previa al análisis. (b) `E'::=•E$` (c) `E::=•E+T` (d) `E::=•T`.

inicial: hay que añadir, por tanto, todas las que tengan el indicador del analizador delante de las partes derechas de las reglas del símbolo no terminal `E`.

```
E ::= •E+T
E ::= •T
```

Por las mismas razones que antes, habrá que realizar el *cierre* de estas dos configuraciones. Para la primera, no es preciso añadir al estado inicial ninguna configuración nueva, ya que el apuntador de análisis precede al mismo símbolo no terminal que acabamos de considerar, y las configuraciones correspondientes a su cierre ya han sido añadidas. Para la segunda, habría que añadir las siguientes configuraciones:

```
T ::= •i
T ::= •(E)
```

Estas dos configuraciones tienen en común que el analizador espera encontrar a continuación símbolos terminales (`i` y '`(`'). Para ello sería necesario localizarlos en la cadena de entrada, pero eso sólo ocurrirá en pasos futuros del análisis.

No queda nada pendiente en la situación inicial, por lo que, si llamamos s_0 al estado inicial, se puede afirmar que:

s_0={E′ ::= •E$,
E ::= •E+T,
E ::= •T,
T ::= •i,
T ::= •(E)}

Completada esta reflexión, se puede describir con más precisión cuál es el contenido del estado inicial de los autómatas de análisis LR(0). Se ha mencionado anteriormente que siempre se podrá suponer que hay sólo una regla cuya parte izquierda es el axioma y cuya parte derecha termina con el símbolo de final de cadena '$'. Si esa regla es A ::= α$, el estado inicial contendrá el conjunto de configuraciones resultado del cierre del conjunto de configuraciones {A ::= •α$}.

- **Justificación intuitiva de la operación de *ir de un conjunto de configuraciones* a otro mediante un símbolo.** El analizador tiene que consultar los símbolos terminales de la cadena de entrada. En una situación intermedia de análisis, tras alguna reducción, también es posible encontrar como siguiente símbolo pendiente de analizar uno no terminal. Una vez estudiada la situación inicial, se puede continuar la construcción del autómata del analizador, añadiendo las transiciones posibles desde el estado inicial.

  La primera conclusión es que hay transiciones posibles, tanto ante símbolos terminales, como ante no terminales. ¿Cómo se comportaría el analizador si, a partir del estado s_0, encuentra en la cadena de entrada un símbolo terminal distinto de i y de '(' o algún símbolo no terminal distinto de E o de T? Ya que no hay ninguna configuración que espere ese símbolo, se concluiría que la cadena de entrada no puede ser generada por la gramática estudiada, es decir, que contiene un error sintáctico. Por lo tanto, todas las transiciones desde el estado inicial con símbolos distintos de i, '(', E o T conducirán a un estado de *error*. En teoría de autómatas, es habitual omitir los estados erróneos al definir las transiciones de un autómata, de forma que las transiciones que no se han especificado para algún símbolo son consideradas como erróneas.

  La segunda conclusión es que, en el autómata del analizador, habrá tantas transiciones a partir de un estado que conduzcan a estados no erróneos, como símbolos sigan al apuntador de análisis en alguna de las configuraciones del estado de partida. Esto significa que desde el estado inicial sólo se podrá ir a otros estados mediante los símbolos terminales i o '(' o mediante los no terminales E o T.

  En la situación inicial, cuando en la cadena de entrada se encuentra el símbolo E, sólo las dos primeras configuraciones (E′ ::= •E$ y E ::= •E+T) representan hipótesis que siguen siendo compatibles con la entrada. En esta situación, se puede desplazar un símbolo hacia la derecha, tanto en la cadena de entrada como en estas configuraciones. Si se utiliza el símbolo s_1 para identificar el nuevo estado, tienen que pertenecer a él las configuraciones que resultan de este desplazamiento:

  E′ ::= E•$
  E ::= E•+T

Para completar el estado resultante de la operación *ir a*, hay que aplicar la operación *cierre* a las nuevas configuraciones, del mismo modo que se vio antes. En este caso, dado que el apuntador de análisis precede sólo a símbolos terminales, no se tiene que añadir ninguna otra configuración. Por lo tanto, se puede afirmar que:

```
s1={E' ::= E•$,
    E ::= E•+T}
```

La Figura 4.32 muestra los dos estados calculados hasta este momento, y la transición que puede realizarse entre ellos.

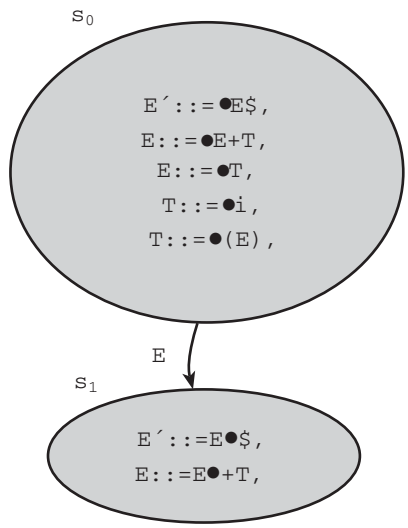


Figura 4.32. Estados s_0 y s_1 y transición entre ellos del autómata de análisis LR(0) del ejemplo.

- **Estados de aceptación y de reducción.** Es interesante continuar la construcción del autómata con una de las transiciones posibles del estado s_1: la del símbolo '$'. Tras aplicar el mismo razonamiento de los puntos anteriores, es fácil comprobar que el estado siguiente contiene el cierre de la configuración `E' ::= E$•`. Esta configuración es de reducción: al estar el apuntador de análisis al final de la cadena, no precede a ningún símbolo, terminal o no terminal, por lo que el cierre no añade nuevas configuraciones al conjunto.

  Cuando el autómata de análisis LR(0) se encuentra en un estado que contiene una configuración de reducción, ha encontrado una parte de la entrada que puede reducirse (un asidero), esto es, reemplazarse por el correspondiente símbolo no terminal. Es decir, ha concluido esta fase del análisis. Por lo tanto, se puede considerar que el autómata debe reconocer esa porción de la entrada y el *estado debe ser final*. Se utilizará la representación habitual (trazo doble) para los estados finales del autómata.

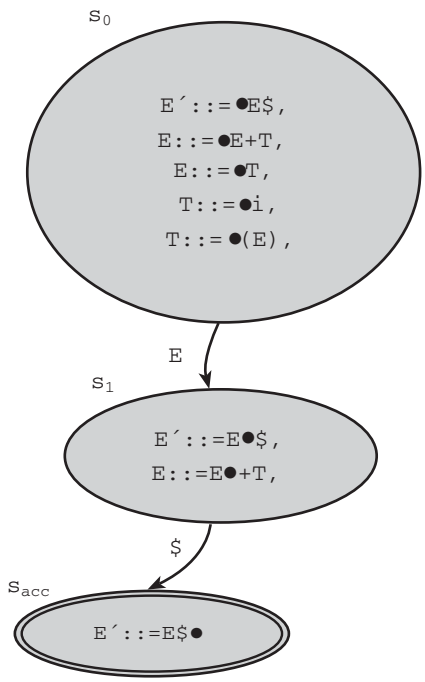


Figura 4.33. Estados s_0, s_1 y de aceptación (s_{acc}) del autómata de análisis LR(0) del ejemplo.

Obsérvese también que la regla de esta configuración es especial: se trata de la regla única asociada al axioma, cuya parte derecha termina con el símbolo especial de fin de cadena. Este estado de reducción también es especial: es el *estado de aceptación* de la cadena completa. Cuando el analizador llega a este estado, significa que la reducción asociada a su configuración ha terminado el análisis y sustituye toda la cadena de entrada por el axioma. Se utilizará para este estado el nombre s_{acc}. La Figura 4.33 representa gráficamente esta parte del diagrama de estados.

Los razonamientos anteriores pueden aplicarse tantas veces como haga falta, teniendo en cuenta que cada estado debe aparecer una sola vez en el diagrama, y que el orden en que aparezcan las configuraciones en el estado no es relevante. La Figura 4.34 presenta el diagrama completo, una vez obtenido. Obsérvese que hay cuatro estados finales más, que no son de aceptación: s_3, s_4, s_7 y s_8. También hay varios estados a los que se llega por diferentes transiciones: s_2, s_4, s_5 y s_7. Esto significa que dichos estados aparecen más de una vez en el proceso descrito anteriormente.

Autómata asociado a un analizador LR(0): definiciones formales

La descripción formal del algoritmo de diseño del autómata de análisis LR(0) precisa de la definición previa del concepto auxiliar de *gramática aumentada* y de las operaciones de *cierre de un*

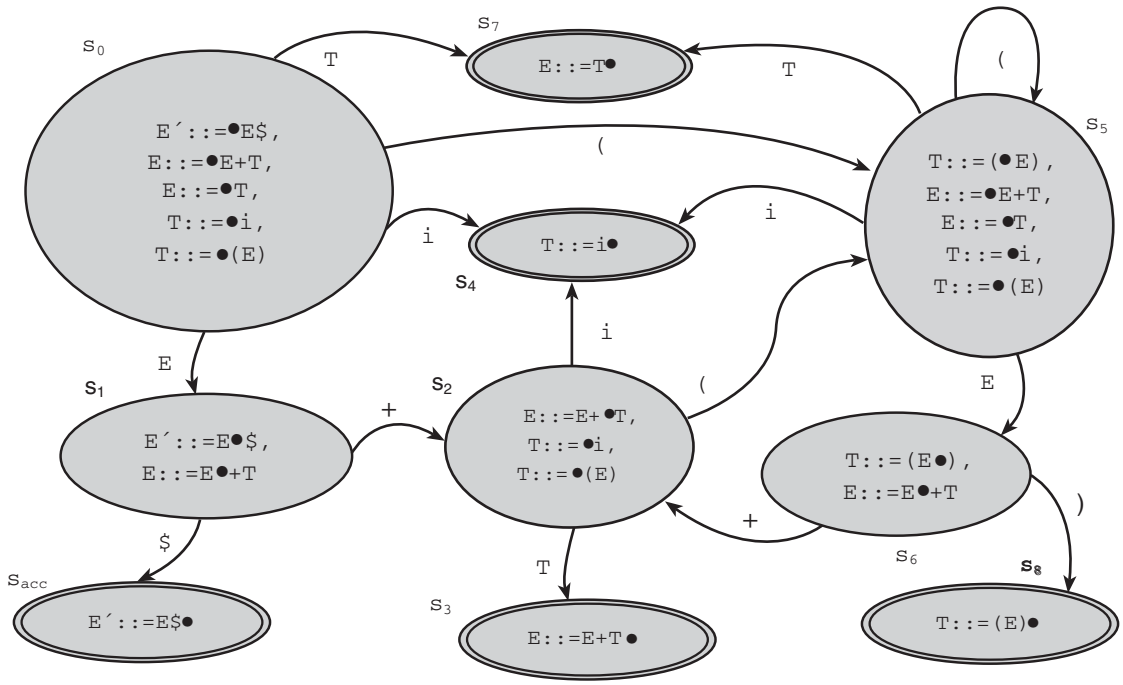


Figura 4.34. Diagrama de estados completo del analizador LR(0) del ejemplo.

conjunto de configuraciones y paso de un conjunto de configuraciones a otro mediante un símbolo (*ir a*).

- **Gramática aumentada.** Dada una gramática independiente del contexto $G=<\Sigma_T, \Sigma_N, A, P>$, se define la gramática extendida para LR(0), en la que se cumple que $A' \notin \Sigma_N$ y que $\$ \notin \Sigma_T$:

  $$G'=<\Sigma_N \cup \{A'\}, \Sigma_T \cup \{\$\}, A', P \cup \{A' ::= A\$\}>$$

  Es fácil comprobar que el lenguaje generado por `G'` es el mismo que el generado por `G`. Recuérdese que el objetivo de esta gramática es asegurar que sólo hay una regla para el axioma.

- **Operación de cierre.** Sea `I` un conjunto de elementos de análisis o configuraciones referido a la gramática `G'` del apartado anterior. Se define `cierre(I)` como el conjunto que contiene los siguientes elementos:

- $\forall\, c \in I \Rightarrow c \in \text{cierre}(I)$.

- $A::=\alpha \bullet B\beta \in \text{cierre}(I) \wedge B::=\gamma \in P \Rightarrow B::=\bullet\gamma \in \text{cierre}(I)$.

La Figura 4.35 muestra un posible pseudocódigo para esta operación.

```
ConjuntoConfiguraciones Cierre(ConjuntoConfiguraciones I,
                               GramáticaIndependienteContexto
                               Gic)

{
  ConjuntoConfiguraciones Cierre := I;
  Configuración c;
  ReglaProducción r;

  while( `se añaden configuraciones a Cierre en la itera-
ción´ )
  {
    `repetir para cada elemento c en Cierre y r en
Reglas(Gic)´
    /* Se supondrá que c es de la forma A::=αBβ y r B::=γ */
    if (B::=•γ ∉ Cierre)
        Cierre := Cierre ∪ { B::=•γ };
  }
  return Cierre;
}
```

Figura 4.35. Pseudocódigo para la operación de cierre de un conjunto de configuraciones.

- **Operación *ir a*.** Sea `I` un conjunto de elementos de análisis o configuraciones y `X` un símbolo (terminal o no) de la gramática `G'` del apartado anterior. Se define la operación `ir_a(I,X)` así:

  $\cup_{A::=\alpha\bullet X\beta \in I}$ `cierre( {A ::= αX•β} )`

  No se muestra ningún pseudocódigo, ya que la operación `ir_a` se reduce a una serie de aplicaciones de la operación `cierre`.

- **Grafo de estados y transiciones del autómata**[2]**.** En lo siguiente, `estados` y `transiciones` serán los nombres de los conjuntos de estados (nodos) y transiciones (arcos) del autómata. A partir de la gramática aumentada `G'`, se puede definir formalmente el grafo de transiciones del autómata de análisis LR(0), de la siguiente manera:

  1. `cierre( {A' ::= •A$} )` $\in$ `estados(G')`.
  2. $\forall$ `I` $\in$ `estados(G')`
     (1) $\forall X \in \Sigma_N \cup \Sigma_T$, `J=ir_a(I,X)` $\in$ `estados(G')` $\wedge$ `(I,J)` $\in$ `transiciones(G')`.

[2] Algunos autores llaman a los estados de este grafo *conjunto de configuraciones canónicas LR(0).*

(2) I es final $\Leftrightarrow \exists N \in \Sigma \wedge \gamma \in (\Sigma_N \cup \Sigma_T)^*$ tales que N ::= γ•∈ I.

(3) I es de aceptación ⇔ A' ::= A$•∈ I.

La Figura 4.36 muestra un posible pseudocódigo para el cálculo de este grafo.

```
Grafo GrafoLR(0) (GramáticaIndepenienteContexto Gic)
{
  ConjuntoConfiguraciones estados[];
  ParEnteros transiciones[];
  ParEnteros aux_par;
/* ParEnteros, tipo de datos con dos enteros: o y d
(de origen y destino) */

  entero i,j,k,it;
  i:=0;        /* índice de estados */
  it:=0;       /* índice de transiciones */

  estados[i]:=cierre({axioma(Gic)'::=•axioma(Gic).'$')};
  /*. es la concatenación de cadenas de caracteres */


  `repetir para cada j≤i´
  {
       `repetir para cada elemento X en Σ_N∪Σ_T´
       {
          if ( (ir_a(s[j],X)≠∅ ∧
                (∀ k ∈ [0,i] s[k]≠ir_a(s[j],X) )
                      )
               {
                aux_par = nuevo ParEnteros;
                aux_par.o = i; aux_par.d = j;
                transiciones[ia++]=aux_par;
                estado[i++]=ir_a(estado[j],X);
               }
       }
       j++;
  }
  return s;
}
```

Figura 4.36. Pseudocódigo para el cálculo del diagrama de transiciones del autómata de análisis LR(0).

Construcción de la tabla de análisis a partir del autómata LR(0)

Puede abordarse ahora la construcción de la tabla de análisis a partir de este autómata. Para ello, hay que identificar las condiciones en las que se anotará cada tipo de operación en las casillas de la tabla.

- **Desplazamientos.** Se obtienen siguiendo las transiciones del diagrama. Si el autómata transita del estado s_i al estado s_j mediante el símbolo x, en la casilla (i,x) de la tabla se añadirá **dj** si $x \in \Sigma_T$ y **j** si $x \in \Sigma_N$.
- **Reducciones.** Se obtienen consultando los estados finales del diagrama, excepto el estado de aceptación. Por definición, cada estado final contendrá una configuración de reducción. Si el estado final es s_i y su configuración de reducción es $N ::= \gamma\bullet$ (donde $N ::= \gamma$ es la regla k), se añadirá la acción **rk** en todas las casillas de la fila i y las columnas correspondientes a símbolos terminales (la parte de la tabla llamada *acción*).
- **Aceptación.** Se obtienen consultando los estados que transitan al estado de aceptación (con el símbolo '$'). Para todas las casillas (i, $), donde i representa un estado s_i que tiene una transición con el símbolo '$' al estado de aceptación, se añade la acción de aceptar.
- **Error.** Todas las demás casillas corresponden a errores sintácticos. Como se ha dicho anteriormente, estas casillas suelen dejarse vacías.

La Figura 4.37 muestra la tabla de análisis del diagrama de transiciones del Ejemplo 4.8.

	Σ_T					Σ_N	
E	i	+	(	)	$	E	T
0	d4		d5			1	7
1		d2			acc		
2	d4		d5				3
3	r1	r1	r1	r1	r1		
4	r3	r3	r3	r3	r3		
5	d4		d5			6	7
6		d2		d8			
7	r2	r2	r2	r2	r2		
8	r4	r4	r4	r4	r4		
	Acción					Ir_a	

Figura 4.37. Tabla de análisis LR(0) correspondiente al diagrama de la Figura 4.34.

4.3.4. De LR(0) a SLR(1)

Ejemplo 4.9

El análisis LR(0) presenta limitaciones muy importantes. Para comprobarlo, se plantea la construcción de la tabla de análisis LR(0) de la gramática G_B que contiene el siguiente conjunto de reglas de producción, que aparecen numeradas, y en las que el axioma es el símbolo <Bloque>.

```
(1)<Bloque> ::=  begin <Decs> ; <Ejecs> end
(2)<Decs>  ::=    dec
(3)        |    <Decs>;dec
(4)<Ejecs>  ::=      ejec
(5)        |    ejec ; <Ejecs>
```

Obsérvese que esta gramática representa la estructura de los fragmentos de programas compuestos por bloques delimitados por los símbolos `begin` y `end`, que contienen una sección declarativa, compuesta por una lista de símbolos `dec` (declaraciones), separados por '`;`', seguida por una serie de instrucciones ejecutables, que consta de una lista de símbolos `ejec`, separados también por '`;`'.

La gramática extendida añade la siguiente regla de producción:

```
(0)<Bloque'> ::= <Bloque>$
```

Siguiendo los algoritmos y explicaciones de las secciones anteriores, se puede comprobar que el diagrama de estados del autómata de análisis LR(0) es el que muestra la Figura 4.38 y la tabla de análisis LR(0) es la de la Figura 4.39. Para simplificar, se utilizan las siguiente abreviaturas:

Símbolo original	Abreviatura
<Bloque'>	B'
<Bloque>	B
<Decs>	D
<Ejecs>	E
begin	b
dec	d
end	f
ejec	e

Este diagrama tiene una situación peculiar no estudiada hasta este momento: el estado S_7={<Ejecs> ::= ejec•, <Ejecs> ::= ejec•;<Ejecs>} es un estado final que contiene más de una configuración. A continuación se describirá con detalle qué repercusiones tiene esta situación.

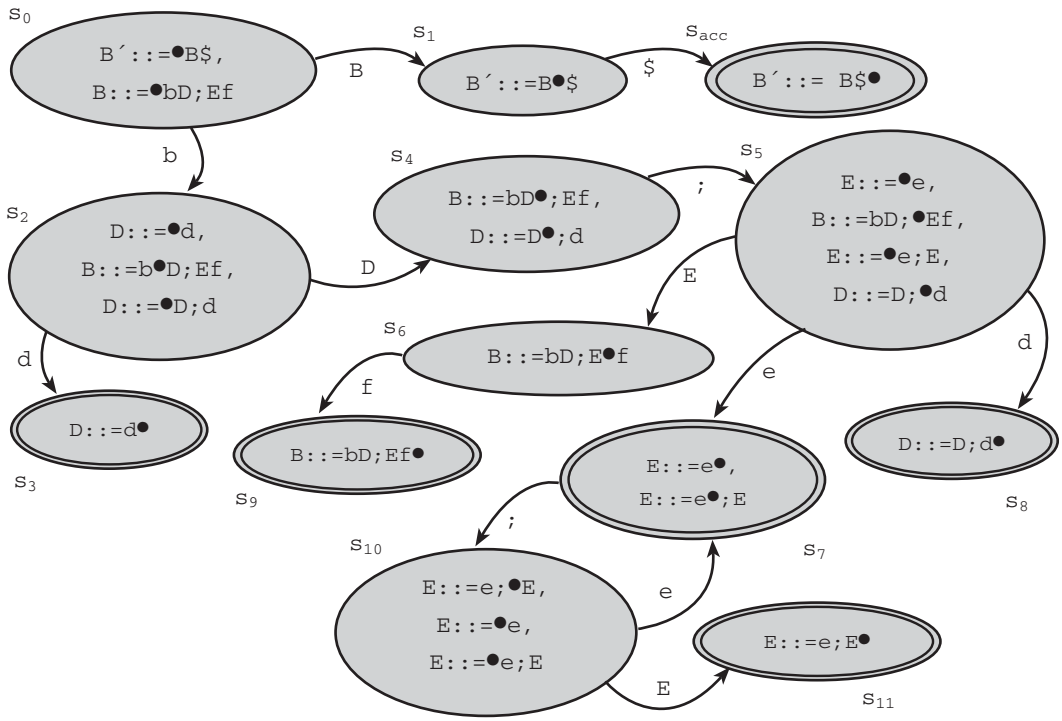


Figura 4.38. Diagrama del autómata de análisis LR(0) del ejemplo sobre los límites de LR(0).

La casilla (7,`;`) presenta otra anomalía: según los algoritmos analizados, la presencia de la configuración de reducción `<Ejecs>::=ejec•` obliga a añadir en las casillas de las columnas de la sección *acción* de la fila 7 la indicación **r4** (4 es el identificador de la regla `<Ejecs>::=ejec`). La presencia adicional de una configuración de desplazamiento con el apuntador de análisis antes del símbolo '`;`' posibilita que con ese símbolo se transite al estado correspondiente (s_{10}) y, por lo tanto, obliga a añadir a la misma casilla la indicación **d10**. Eso significa que, en este estado, en presencia del símbolo '`;`', no se sabe si se debe reducir o desplazar.

Definiciones

Se llama *conflicto* a la circunstancia en que una casilla de una tabla de análisis contiene más de una acción.

Se llama *conflicto reducción / desplazamiento* a la circunstancia en que una casilla contiene una configuración de reducción y otra de desplazamiento.

Se llama *conflicto reducción / reducción* a la circunstancia en que una casilla contiene más de una configuración de reducción.

	Σ_T						Σ_N		
E	d	e	b	;	f	$	B	D	E
0			d2				1		
1						acc			
2	d3							4	
3	r2	r2	r2	r2	r2	r2			
4				d5					
5	d8	d7							6
6					d9				
7	r4	r4	r4	d10/ r4	r4	r4			
8	r3	r3	r3	r3	r3	r3			
9	r1	r1	r1	r1	r1	r1			
10		d7							11
11	r5	r5	r5	r5	r5	r5			
	Acción						Ir_a		

Figura 4.39. Tabla de análisis LR(0) correspondiente al diagrama de la Figura 4.38.

Una gramática independiente del contexto G es una *gramática LR(0)* si y sólo si su tabla de análisis LR(0) es determinista, es decir, no presenta conflictos.

4.3.5. Análisis SLR(1)

SLR(1) es una técnica de análisis que simplifica la técnica LR(1), que se verá posteriormente. Toma su nombre de las siglas en inglés de la expresión *LR(1) sencillo*.

El estudio del ejemplo de conflicto de la sección anterior sugiere una solución. Es fácil comprobar que la gramática G_B puede generar la palabra

```
begin dec ; ejec ; ejec end
```

La Figura 4.40 muestra su árbol de derivación.

Al llegar a la segunda aparición del símbolo '`;`' (situada entre los dos símbolos `ejec`), el analizador LR(0) se encuentra en el estado s_7. El símbolo siguiente que hay que procesar de la cadena de entrada es '`;`'. A continuación se analizará el efecto de cada una de las dos opciones sobre el árbol de derivación.

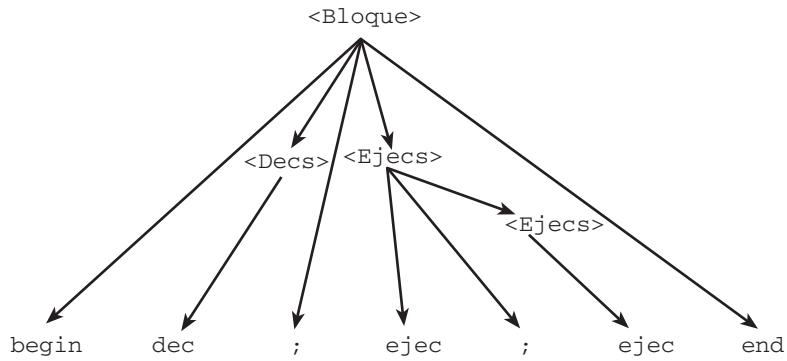


Figura 4.40. Árbol de derivación de la palabra `begin dec; ejec; ejec end` por parte de la gramática G_B.

La Figura 4.41 muestra lo que ocurriría si se eligiese la opción de la reducción.

Si se reduce la regla `<Ejecs>::=ejec`, el resto del subárbol que tiene a `<Ejecs>` como raíz, que está resaltado en la Figura 4.41, no puede ser analizado tras la reducción, porque `<Ejecs>` habría sido ya totalmente analizado. La Figura 4.42 refleja gráficamente los pasos del analizador sobre el diagrama de transiciones y contiene una flecha que indica la secuencia en la que se visita cada estado. Se resaltan los estados y las transiciones de ese camino.

Desde el estado inicial, tras desplazar el símbolo `begin`, se llega al estado s_2. Al desplazar el siguiente terminal de la cadena de entrada (`dec`), se transita al estado s_3, en el que se reduce la regla `<Decs>::=dec`. Cuando se reduce una regla, el algoritmo de análisis elimina de la pila los símbolos almacenados en relación con su parte derecha, y vuelve al estado en que se encontraba antes de comenzar a procesar dicha parte derecha. Ese estado se encuentra ahora con el símbolo no terminal que forma la parte izquierda de la regla que se está reduciendo. En este caso,

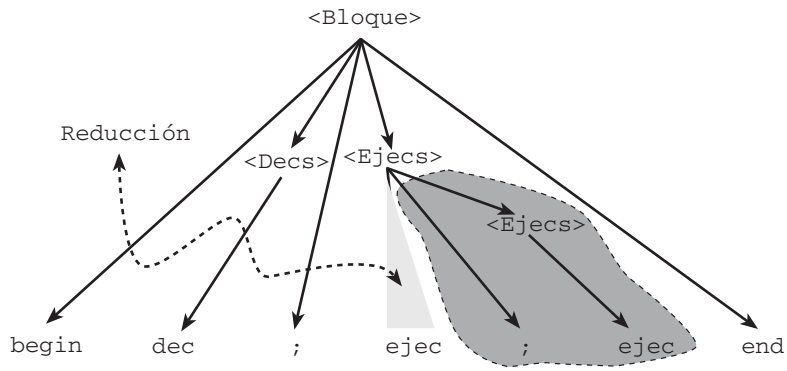


Figura 4.41. Efecto de la reducción de la regla `<Ejecs>`→`ejec`.

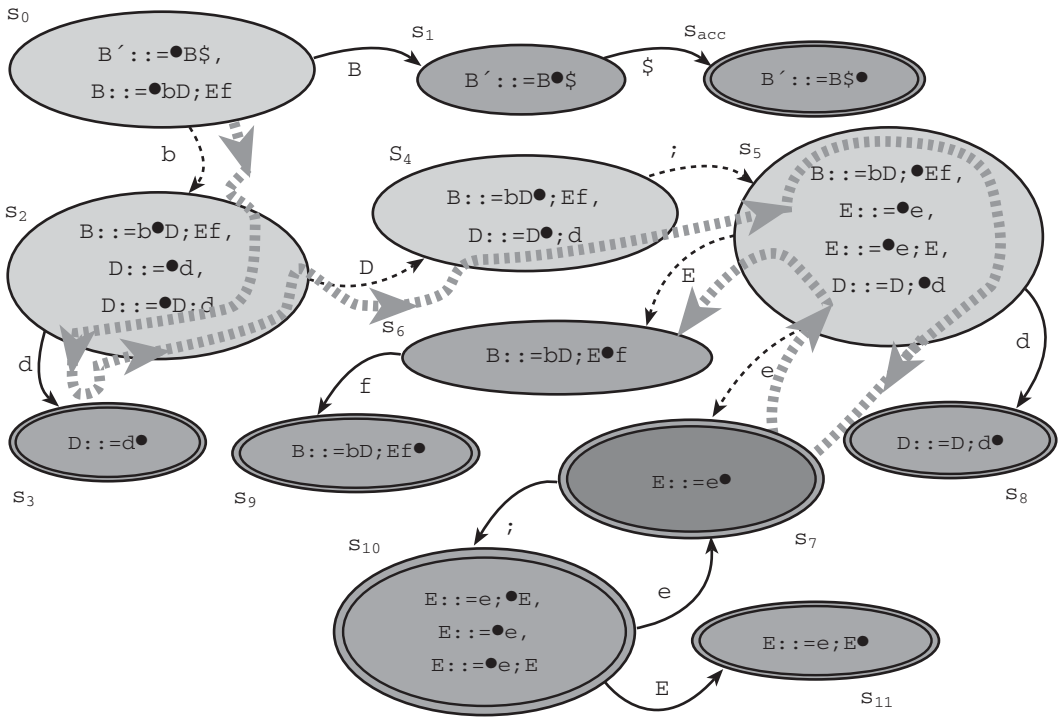


Figura 4.42. Recorrido del analizador sintáctico sobre la cadena `begin dec; ejec; ejec end` si el estado s_7 fuera sólo de reducción.

antes de analizar la parte derecha 'dec', el analizador estaba en el estado s_2 (eso es lo que representa el fragmento de la flecha que vuelve desde el estado s_3 al s_2). A continuación, con el símbolo <Decs> se transita desde el estado s_2 al estado s_6. Los dos siguientes símbolos terminales (';' y 'ejec') también dan lugar a desplazamientos a los estados s_5 y s_7, respectivamente. En este último se reduce la regla <Ejecs>::=ejec y se vuelve al estado anterior al proceso de la parte derecha (ejec), que es, de nuevo, s_5. El símbolo no terminal de la parte izquierda de la regla (<Ejecs>) hace que se transite a s_6. Desde este estado sólo se espera desplazar el terminal end para llegar al estado en el que se puede reducir un bloque completo. Sin embargo, el símbolo que hay que analizar en este instante es el terminal ejec. La transición asociada a este símbolo no está definida, por lo que el análisis terminaría indicando un error sintáctico. Intuitivamente, el error se ha originado porque se interpretó que el símbolo ';' indicaba el final de la lista de sentencias ejecutables (ejec), cuando realmente era su separador.

El analizador sólo tiene un comportamiento correcto posible: considerar el símbolo ';' como lo que es, un separador, y desplazarlo. La Figura 4.43 muestra, en el árbol de derivación, que este desplazamiento posibilita el éxito del análisis.

Al desplazar el símbolo ';', se posibilita la reducción posterior, primero de la segunda aparición de ejec al símbolo no terminal <Ejecs>, y luego de ejec;<Ejecs> al símbolo no ter-

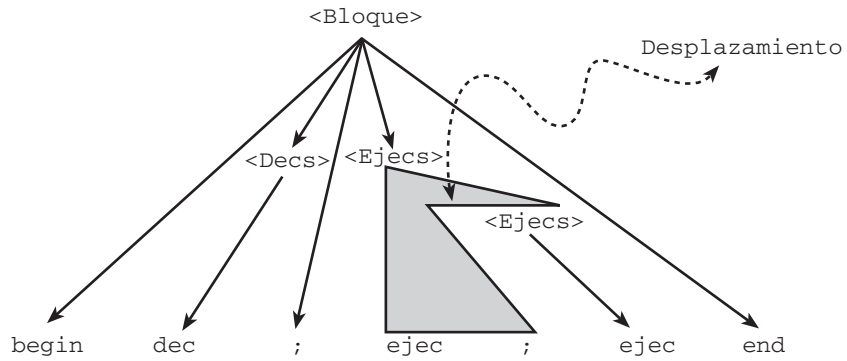


Figura 4.43. Efecto de desplazar el símbolo ';' en el análisis de la palabra `begin dec; ejec; ejec end`.

minal <Ejecs>. De esta forma, el análisis puede terminar con éxito. Las Figuras 4.44 a 4.46 muestran el recorrido por el diagrama de estados correspondiente al análisis completo.

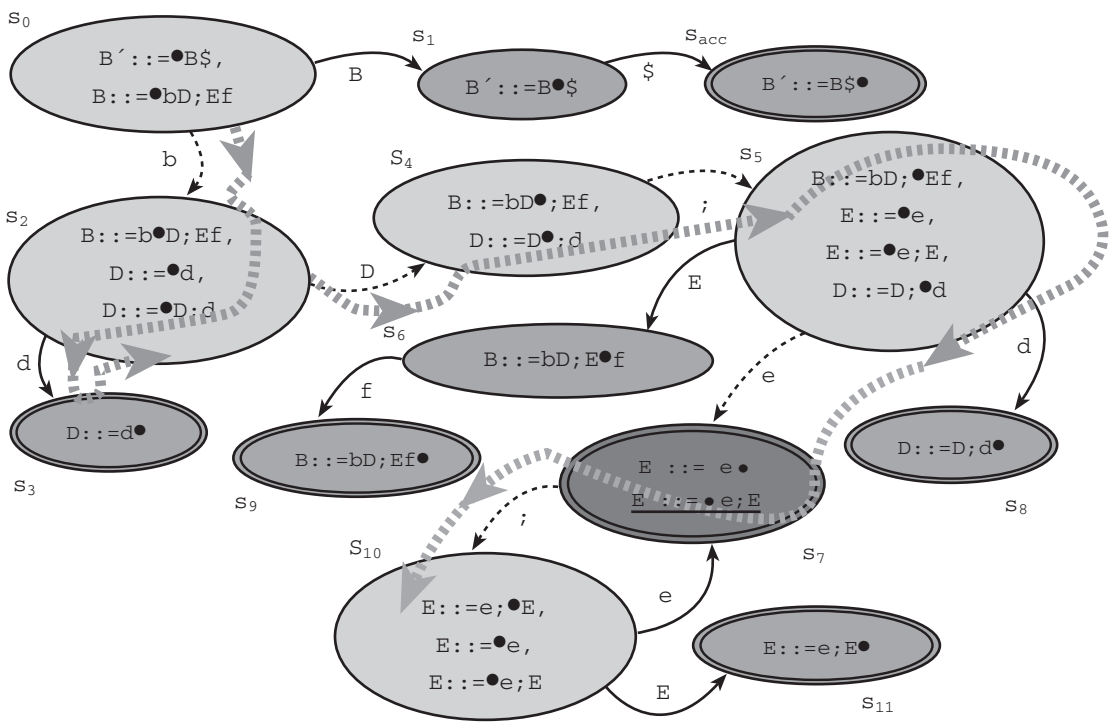


Figura 4.44. Recorrido hasta el estado s_7 que el análisis sintáctico debería realizar sobre el diagrama del analizador sintáctico para analizar correctamente la cadena `begin dec; ejec; ejec end`.

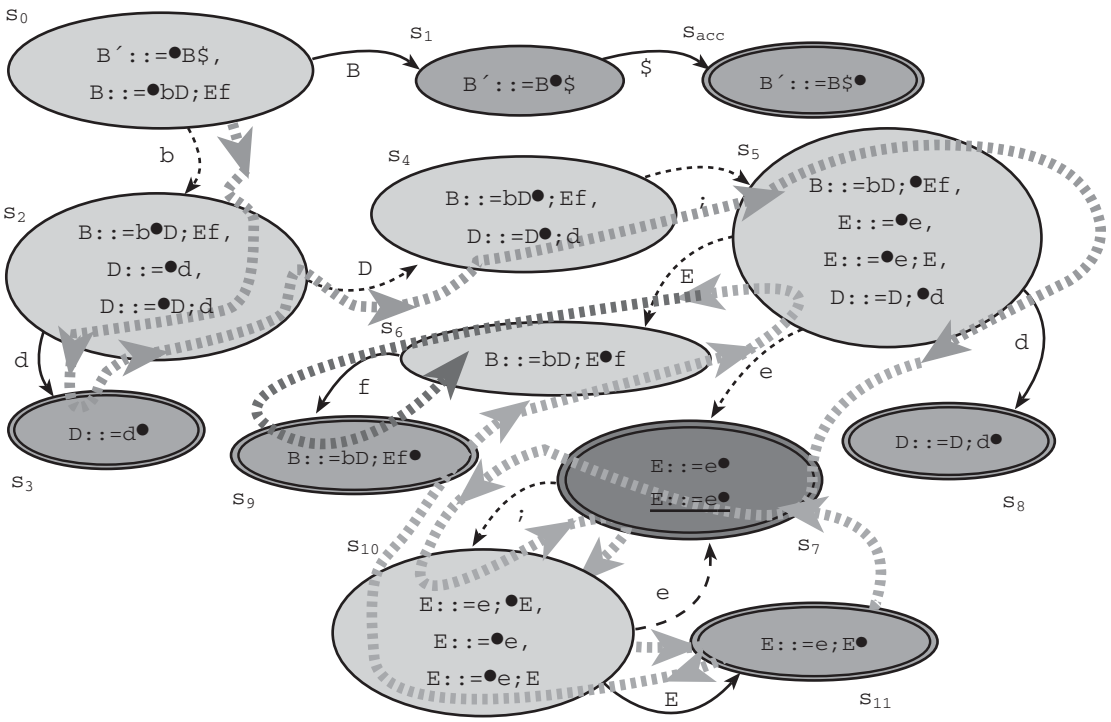


Figura 4.45. Recorrido hasta la última reducción que el análisis sintáctico debería realizar sobre el diagrama del analizador sintáctico para analizar correctamente la cadena `begin dec; ejec; ejec end`.

Obsérvese que, en este caso, se tienen que considerar las dos configuraciones del estado s_7. Primero se aplica la configuración de desplazamiento, que aparece subrayada en la Figura 4.44. De esta manera se llega al estado s_{10}. Con el desplazamiento correspondiente a la siguiente aparición del terminal `ejec` se vuelve al estado s_7, pero ahora, en presencia de `end`, que es el siguiente símbolo terminal analizado, sólo se puede reducir y sustituir `ejec` por <Ejecs>.

La Figura 4.45 muestra los pasos de análisis siguientes. Tras la reducción, que supone eliminar de la pila todo lo que corresponde a la parte derecha de la regla, el analizador se encuentra de nuevo en el estado s_{10} y tiene que procesar el símbolo no terminal recién incorporado a la cadena de entrada (<Ejecs>). Así se llega al estado s_{11} en el que se reducen las dos apariciones de `ejec`. El analizador vuelve al estado en el que se encontraba antes de la primera aparición de `ejec`, es decir, en el estado s_5. Con el símbolo no terminal de la parte izquierda (<Ejecs>) se transita de s_5 a s_6. El siguiente símbolo para analizar es `end`. Su desplazamiento lleva al estado en el que se reduce el bloque de sentencias completo.

La Figura 4.46 muestra los pasos finales del análisis. Tras reducir el bloque completo, se vuelve al estado inicial. Con el símbolo no terminal de la parte izquierda de la regla reducida (<Bloque>) se llega a s_1, desde donde se desplaza el símbolo final de la cadena y se llega al estado de aceptación, lo que completa con éxito el análisis.

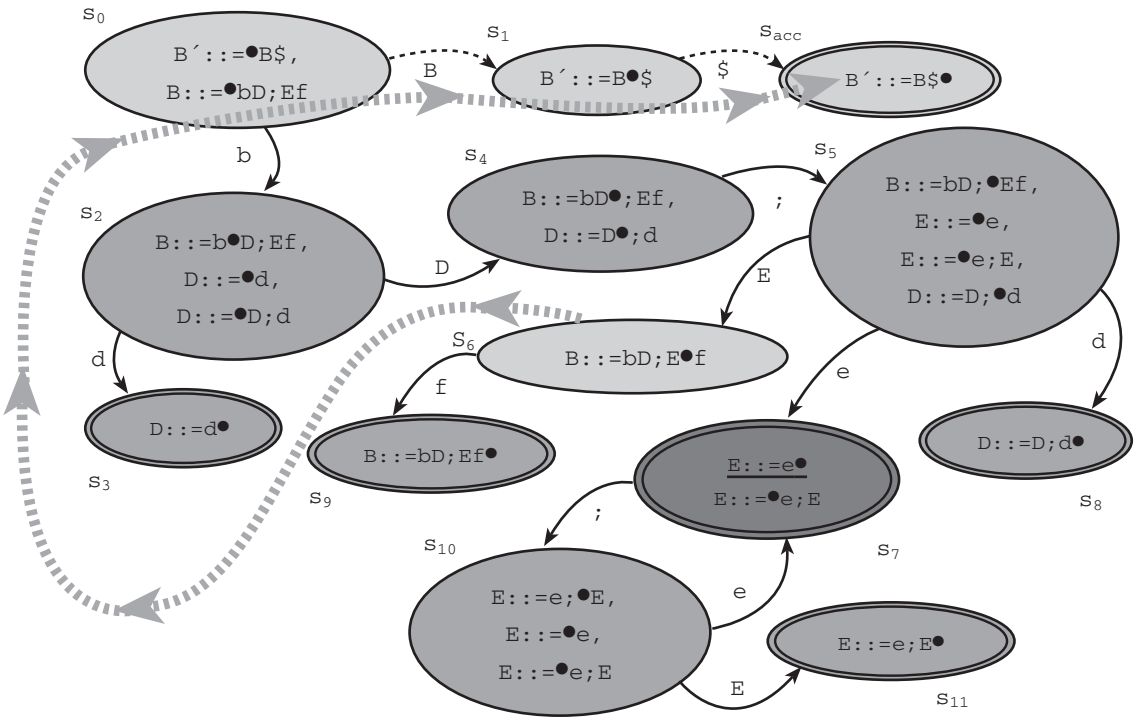


Figura 4.46. Últimos pasos del recorrido que el análisis sintáctico debería realizar sobre el diagrama del analizador sintáctico para analizar correctamente la cadena `begin dec; ejec; ejec end`.

Lo más relevante de este análisis es la razón por la que el analizador no debe reducir en la casilla del conflicto: en el estado s_7, el símbolo '`;`' debe interpretarse siempre como separador de sentencias ejecutables. La reducción sólo debe aplicarse cuando se ha llegado al final de la lista de símbolos `ejec`, y esto ocurre sólo cuando aparece el símbolo terminal `end`. Con el análisis LR(0) es imposible asociar el estado s_7 y el terminal `end`. En la próxima sección se verá con detalle que esta solución se basa en el hecho de que *el símbolo no terminal* `<Ejecs>` (la parte izquierda de la regla que se reduce en s_7) *siempre debe venir seguido por el símbolo terminal* `end`.

Este conflicto no se habría producido si *en lugar de reducir la regla en todas las columnas de la parte de acción de la fila 7, sólo se hubiera hecho en las columnas de los símbolos terminales que pueden seguir a* `<Ejecs>`, *que es la parte izquierda de la regla que se va a reducir*.

Construcción de tablas de análisis

En la Sección 4.1 se presentaron dos conjuntos importantes de las gramáticas independientes del contexto. Uno de ellos, el conjunto *siguiente*, contiene el conjunto de símbolos que pueden *seguir a otro en alguna derivación*. Este conjunto puede utilizarse para generalizar las reflexiones de la sección anterior, y es el origen de la técnica llamada SLR(1).

	Σ_T						Σ_N		
E	d	e	b	;	f	$	B	D	E
0			d2				1		
1						acc			
2	d3							4	
3	r2	r2	r2	r2	r2	r2			
4				d5					
5	d8	d7							6
6					d9				
7	r4	r4	r4	d10/ r4	r4	r4			
8	r3	r3	r3	r3	r3	r3			
9	r1	r1	r1	r1	r1	r1			
10		d7							11
11	r5	r5	r5	r5	r5	r5			
	Acción						Ir_a		

(0)B´→ B$
(1)B→ bD;Ef
(2)D → d
(3)D→ D;d
(4)E→ e
(5)E→ e;E

siguiente (B)={$}
siguiente (D)={;}
siguiente (E)={f}

	Σ_T						Σ_N		
E	d	e	b	;	f	$	B	D	E
0			d2				1		
1						acc			
2	d3							4	
3				r2					
4				d5					
5	d8	d7							6
6					d9				
7				d10	r4				
8				r3					
9						r1			
10		d7							11
11					r5				
	Acción						Ir_a		

Figura 4.47. Tablas de análisis LR(0) y SLR(1) para la gramática del ejemplo. La parte superior muestra la tabla LR(0) y resalta las casillas que cambiarán de contenido. La parte inferior muestra la tabla SLR(1). A su izquierda están los conjuntos auxiliares que justifican su contenido.

La tabla de análisis se construye de la misma manera que con la técnica LR(0) (*véase* la Sección 4.3.3), excepto por las reducciones:

- **Reducciones.** Igual que en el caso LR(0), se consultan los estados finales del diagrama, excepto el estado de aceptación. Si el estado final es s_i y su configuración de reducción es N::=γ• (donde N::=γ es la regla k), la acción **rk** se añadirá sólo en las casillas correspondientes a las columnas de los símbolos terminales del conjunto *siguiente(N)*, ya que sólo ellos pueden seguir a las derivaciones de N.

Veamos el contenido de los conjuntos siguiente de los símbolos no terminales del Ejemplo 4.9:

```
siguiente(<Bloque>)={$}
siguiente(<Decs>)={;}
siguiente(<Ejecs>)={end}
```

Esto significa que en las filas correspondientes a los estados en los que se reduce una regla cuya parte izquierda sea <Bloque>, sólo hay que colocar la acción de reducción en la casilla correspondiente al símbolo '$'. En los estados en que se reduzca una regla cuya parte izquierda sea <Decs>, hay que hacerlo sólo en la columna correspondiente a ';', y en los estados en que se reduzca una regla cuya parte izquierda sea <Ejecs>, hay que hacerlo sólo en la columna encabezada por end.

La Figura 4.47 compara las tablas de análisis LR(0) y SLR(1) para el Ejemplo 4.9. Puede comprobarse que ha desaparecido el conflicto reducción / desplazamiento de la tabla LR(0).

Definición de gramática SLR(1)

Una gramática independiente del contexto G es una *gramática SLR(1)* si y sólo si su tabla de análisis SLR(1) es determinista, es decir, no presenta conflictos.

4.3.6. Más allá de SLR(1)

¿Hay alguna gramática independiente del contexto que no sea SLR(1)? Es decir, ¿existen gramáticas independientes del contexto cuyas tablas de análisis SLR(1) siempre presentan conflictos?

Ejemplo 4.10

Considérese la gramática G_{axb} que contiene las siguientes reglas de producción y cuyo axioma es el símbolo S:

```
(1)S ::= A
(2)S ::= xb
(3)A ::= aAb
(4)A ::= B
(5)B ::= x
```

Es fácil comprobar que esta gramática genera el lenguaje $\{xb, a^n xb^n \mid n \geq 0\}$. A continuación se va a construir su tabla de análisis SLR(1). En primer lugar se aumenta la gramática con la producción (0)S'::=S$ y se construye el diagrama de estados del autómata de análisis LR(0) que muestra la Figura 4.48.

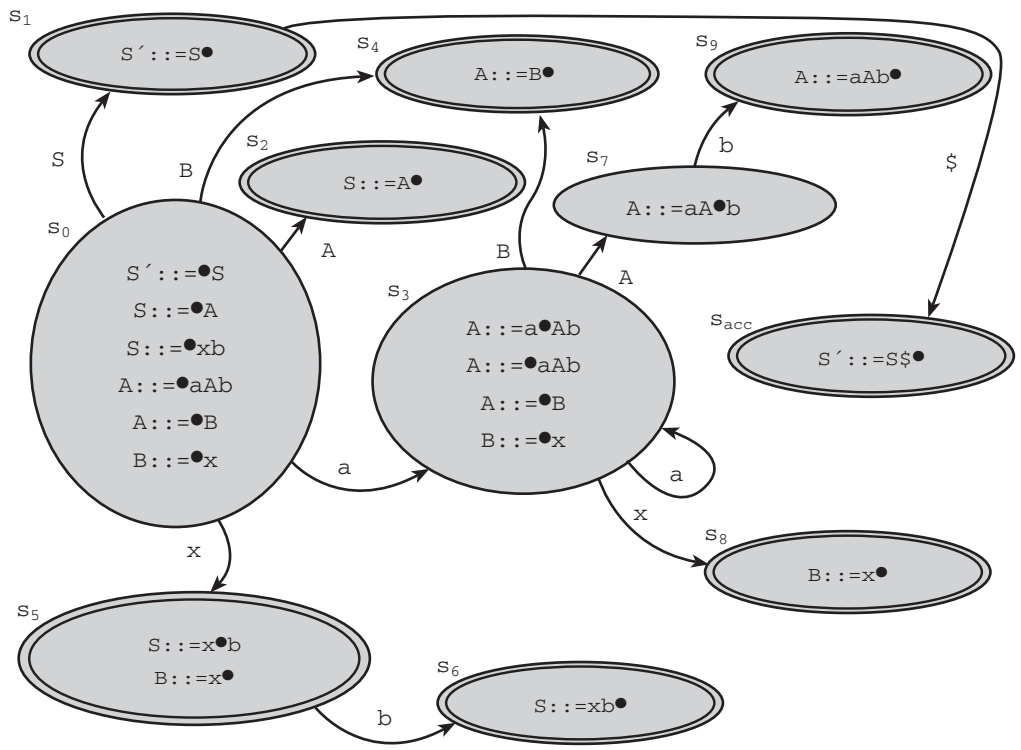


Figura 4.48. Diagrama de estados de la gramática G_{axb}.

Es fácil calcular el valor el conjunto *siguiente*:

```
siguiente(A)={$,b}
siguiente(B)={$,b}
siguiente(S)={$}
```

La tabla de análisis SLR(1) de G_{axb} se muestra en la Figura 4.49.

La presencia del estado s_5, que contiene la reducción de la regla `(5)B::=x`, y el hecho de que `b` ∈ `siguiente(B)`, y que con `b` se pueda transitar desde s_5 a s_6, originan un conflicto de tipo reducción / desplazamiento.

4.3.7. Análisis LR(1)

Considérese G_{axb}, ejemplo de una gramática que no es SLR(1). El símbolo `b` pertenece a `siguiente(B)` a causa de las reglas `A::=B` y `A::=aAb`. Por la primera se llega a la conclusión de que los símbolos que sigan al árbol de derivación de `B` tienen que contener a los que sigan al de `A`. Por la segunda queda claro que `b` es uno de esos símbolos. Por lo tanto, para que haya una `b` detrás del árbol de derivación de `B`, tiene que ocurrir que antes del árbol hubiera una `a` (ya que

siguiente(A)={$,b}
siguiente(B)={$,b}
siguiente(S)={$}

	Σ_T				Σ_N		
E	a	b	x	$	S	A	B
0	d3		d5		1	2	4
1				acc			
2				r1			
3	d3		d8			7	4
4		r4		r4			
5		r5/d6		r5			
6				r2			
7							
8		r5		r5			
9		r3		r3			
	Acción				Ir_a		

Figura 4.49. Tabla de análisis SLR(1) de la gramática G_{axb}.

A→aAb→aBb→axb). El árbol de derivación debe reducir x a B, que a su vez se reducirá a A. Tras hacer todo esto es cuando se puede encontrar la b. Todo lo anterior asegura que la reducción de B::=x sería posible (en las circunstancias descritas) antes de una b.

En el diagrama de estados (*véase* Figura 4.48) hay dos estados (s_5 y s_8) en los que se puede reducir la regla B→x, que corresponden a dos fragmentos distintos de análisis desde el estado inicial, que se pueden comparar en la Figura 4.50. Para llegar a s_8 desde s_0, es necesario desplazar previamente el terminal a y luego el x. Para llegar a s_5 basta con el símbolo x.

Por lo tanto, la reducción de B::=x antes de una b es la que se realiza en el estado s_8. ¿Cuándo sería correcto reducirla en el estado s_5? La Figura 4.51 muestra el árbol de la derivación S′→S$→A$→B$→x$.

En este caso, la reducción correspondería al estado s_5, ya que desde el estado inicial se ha desplazado una x no precedida de una a. Por lo tanto, es cierto que los símbolos terminales que pueden seguir a B son $ y b, pero en algunos estados del diagrama (s_8) la reducción sólo puede ser seguida por $ y en otros (s_5) sólo por b. La tabla de análisis SLR(1), que está dirigida por el conjunto siguiente, carece de la precisión suficiente para gestionar estas gramáticas.

Obsérvese que una posible solución consistiría en que las configuraciones de reducción aparecieran en los estados junto con la información relacionada con los símbolos en presencia de los cuales la reducción es posible. En este caso, s_5 estaría ligado a B→x:$ y s_8 a B→x:b.

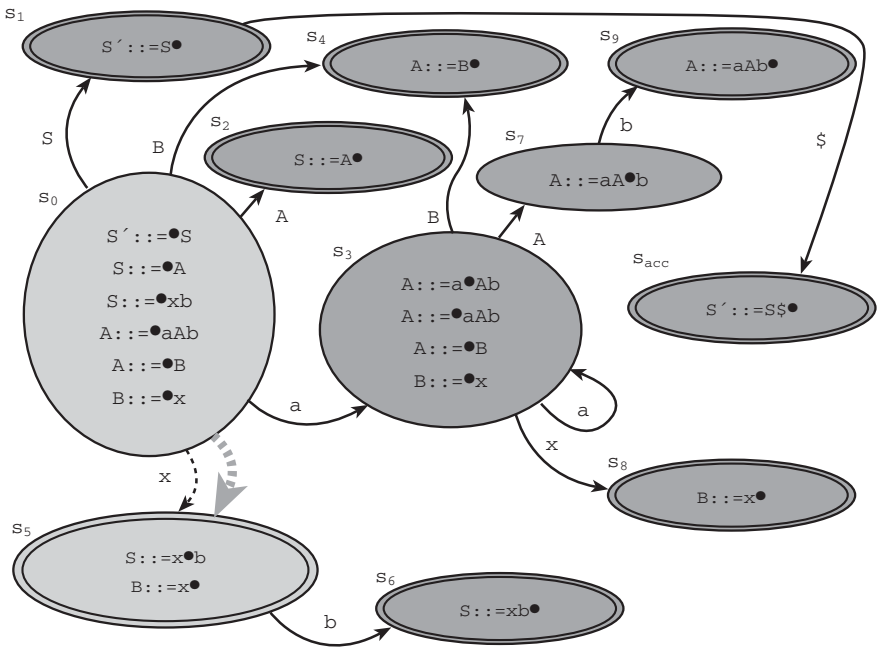


Figura 4.50. Comparación entre los dos posibles recorridos previos a las reducciones de la regla `B::=x`: (a) en el estado s_5, (b) en el estado s_8.

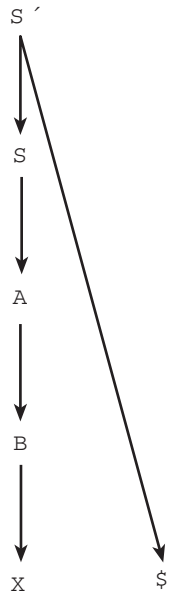


Figura 4.51. Árbol de derivación de la cadena x$ por la gramática G_{axb}'.

Descripción intuitiva del análisis LR(k), k>0

La posible mejora sugerida al final de la sección anterior es generalizable. A lo largo de las próximas secciones, se formalizará mediante el *conjunto de símbolos de adelanto*, para describir la técnica de análisis LR(k) con k>0.

A partir de este punto, se llamará *conjunto de símbolos de adelanto* a los símbolos que, en un estado concreto del diagrama, se espera encontrar en cada una de sus configuraciones. Hasta ahora, una configuración representaba una hipótesis en curso, en el proceso del análisis, definida por la posición del apuntador de análisis en la parte derecha de una regla compatible con la porción de cadena de entrada analizada. En este nuevo tipo de análisis se añadirá a cada configuración los símbolos de adelanto correspondientes. Tras añadir esta información, la configuración

$$N ::= \alpha \bullet \gamma \ \{\sigma_1, \ldots, \sigma_m\}, \ N \in \Sigma_N \wedge \alpha, \gamma \in (\Sigma_N \cup \Sigma_T)^* \wedge \{\sigma_1, \ldots, \sigma_m\} \subseteq \Sigma_T$$

significa que, en este instante, una de las hipótesis posibles está relacionada con la regla $N \rightarrow \alpha\gamma$; en particular, el prefijo α de la parte derecha es compatible con la porción de la entrada analizada hasta este momento; además, esto sólo es posible si, tras terminar con esta regla, el siguiente símbolo terminal pertenece al conjunto $\{\sigma_1, \ldots, \sigma_m\}$.

En esto se basa el aumento de precisión del análisis LR(1) sobre SLR(1). Lo que en SLR(1) era una única hipótesis, se multiplica ahora con tantas posibilidades como conjuntos diferentes de símbolos de adelanto.

Introducción al cálculo de símbolos de adelanto. Vamos a construir el diagrama de estados del autómata de análisis LR(1) de la gramática G_{axb} del Ejemplo 4.10. Se mantendrá el mismo

algoritmo básico, al que se incorpora el cálculo de los conjuntos de símbolos de adelanto de cada configuración de cada estado.

Empezaremos con un mecanismo para calcular el conjunto de símbolos de adelanto:

- De la configuración inicial del estado inicial (`A'::=•A$`).
- De las configuraciones del *cierre* de una configuración.
- De las configuraciones resultado de *ir a* otro conjunto de configuraciones mediante un símbolo.

En el ejemplo, la configuración inicial del estado inicial es `S'::=•S$`. Para preservar la intuición es frecuente, en el análisis LR(1), considerar que la primera configuración de la nueva regla de la gramática ampliada es `S'::=•S`, prescindiendo del símbolo final (`$`). Expresada de esta forma, la hipótesis inicial indica que el apuntador de análisis se encuentra antes del primer símbolo de la cadena de entrada y que se espera poder reducirla toda ella al símbolo no terminal `S`. Resulta claro que el único símbolo que se puede esperar, tras procesar completamente la regla, es el símbolo que indica el final de la misma. Por tanto, *el conjunto de símbolos de adelanto de la configuración inicial del estado inicial en el análisis LR(1) es* `{$}`.

Para completar el estado inicial, hay que añadir las configuraciones de `cierre({S'::=•S {$}})`, lo que implica calcular los conjuntos de símbolos de adelanto para `S::=•A`, `A::=•B`, `B::=•x`, `A::=•aAb` y `S::=•xb`.

Hemos visto que, tras procesar por completo `S'::=•S`, hay que encontrar el símbolo '`$`'. Esto implica que, si `S` se redujo mediante la regla `S::=A`, tras `A` se puede encontrar lo mismo que se encontraría tras `S`, es decir, `$`. Este razonamiento vale para todas las configuraciones y justifica que el nuevo conjunto de símbolos de adelanto sea `{$}`, lo que esquematiza gráficamente la Figura 4.52, que representa los cierres sucesivos mediante árboles de derivación concatenados.

La Figura 4.53 muestra gráficamente cómo se completa el cálculo del estado inicial, que resulta ser:

s_0={s' ::= •S {$},
S ::= •A {$},
A ::= •B {$},
B ::= •x {$},
A ::= •aAb {$},
S ::= •xb {$}}

El caso analizado en este estado no es el más general que puede aparecer al realizar la operación `cierre`. De hecho, puede inducir a engaño que en este caso no se modifique el conjunto de símbolos de adelanto porque, como se verá en los próximos párrafos, es esta operación la que puede modificarlos. Como sugiere la Figura 4.52, la razón por la que en este caso no se modifican los símbolos de adelanto es que el cierre se ha aplicado en todos los casos a configuraciones con la estructura:

$N ::= \alpha \bullet A \{\sigma_1, \ldots, \sigma_m\},\ N, A \in \Sigma_N \wedge \alpha \in (\Sigma_N \cup \Sigma_T)^* \wedge \{\sigma_1, \ldots, \sigma_m\} \subseteq \Sigma_T$

Tras el símbolo no terminal que origina el cierre, no hay ningún otro símbolo terminal. Por lo tanto, lo que se encontrará tras él (en este caso `A`) es lo mismo que se encuentra cuando se termina de procesar cualquiera de sus reglas (`A ::=` γ).

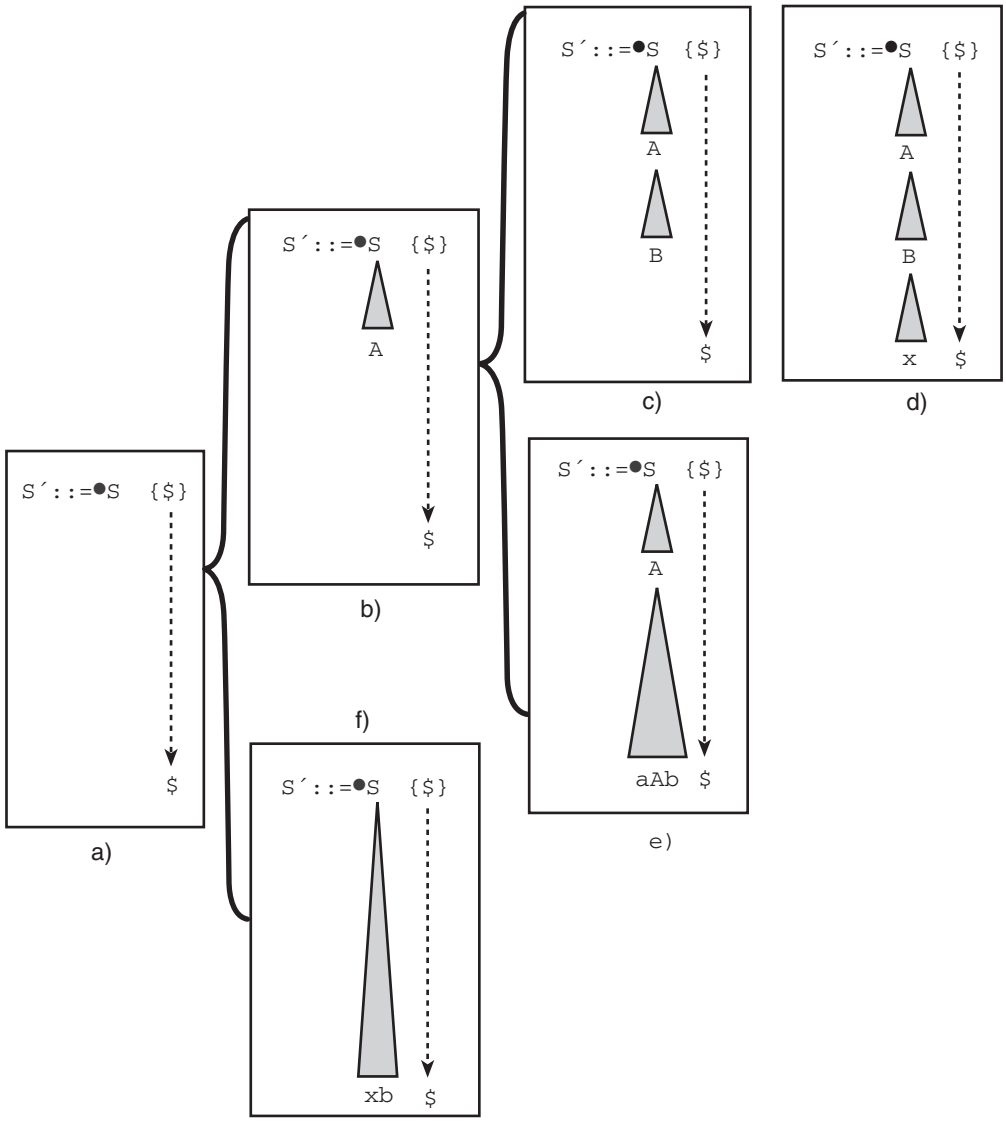


Figura 4.52. Ejemplo de situación de cierre que no modifica el conjunto de símbolos de adelanto: (a)`S'::=•S {$}` (b) `S::=•A {$}` (c) `A::=•B {$}` (d) `B::=•x {$}` (e) `A::=•aAb {$}` (f) `S::=•xb {$}`

Para ilustrar esta situación, considérese a continuación la operación que calcula s_5=`ir_a(`s_0`,a).A::=•aAb{$}` es la única configuración de s_0 relacionada con esta operación. Hay que calcular, por tanto, el conjunto de símbolos de adelanto de `A::=a•Ab`. Parece lógico concluir que lo que se espere encontrar tras terminar con la parte derecha de la regla no cambiará porque se vayan procesando símbolos en ella. El conjunto buscado coincide con `{$}` y se puede extraer la siguiente conclusión: *El conjunto de símbolos de adelanto de una configuración no varía cuando se desplaza el apuntador de análisis hacia la derecha a causa de transiciones entre estados.*

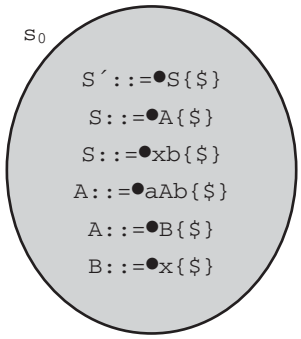


Figura 4.53. Estado inicial del autómata de análisis LR(1) de la gramática G_{axb}.

Para concluir los cálculos para la operación `ir_a`, se realiza el cierre de la configuración resultado de desplazar `a`:

```
ir_a(s0,a)=cierre( { A ::= a•Ab {$}} )
```

Hay que calcular los conjuntos de símbolos de adelanto para `A::=•xb` y `A::=•B`. En este caso, lo fundamental es la `b` que sigue a `A` (`A ::= a•A`***b***). Aplicando el mismo razonamiento de los párrafos anteriores, la hipótesis de esa configuración es que la próxima porción de la cadena de entrada tiene que permitir reducir alguna regla de `A`, y luego desplazar la `b` ha de que seguirla obligatoriamente. Por eso, tras procesar completas las reglas de `A`, se tendrá que encontrar una `b` (la resaltada en las líneas anteriores) y `{b}` es el conjunto de símbolos de adelanto buscado. El resto de las configuraciones no presentan novedades respecto a lo expuesto anteriormente. Obtendremos, por tanto, el siguiente estado:

```
s5={A ::= a•Ab {$},
    A ::= aA•b {b},
    A ::= •B {b},
    B ::= •x {b}}
```

La Figura 4.54 muestra el diagrama de estados correspondiente a esta situación.

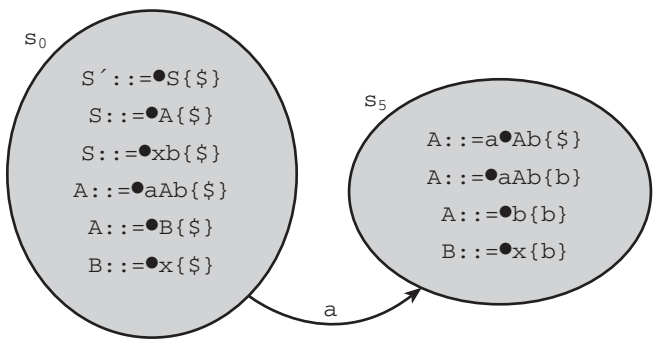


Figura 4.54. Diagrama con los dos primeros estados del autómata del analizador LR(1) de la gramática G_{axb}.

Como se ha visto, el conjunto de símbolos de adelanto puede variar cuando se cierra una configuración con la siguiente estructura:

$P::=\alpha\bullet N\gamma\ \{\sigma_1,\ldots,\sigma_m\},,P,N\in\Sigma_N\wedge\alpha\in(\Sigma_N\cup\Sigma_T)^*\wedge\gamma\in(\Sigma_N\cup\Sigma_T)^+\wedge\{\sigma_1,\ldots,\sigma_m\}\subseteq\Sigma_T$

El caso analizado contiene una cadena γ que se reduce a un único símbolo no terminal. En general, γ puede ser cualquier cadena.

Autómata asociado a un analizador LR(1): definiciones formales

A continuación se describirán las diferencias entre los análisis LR(0) y LR(1).

- **Gramática aumentada.** Como se ha descrito informalmente en las secciones anteriores, la interpretación del símbolo de adelanto para la regla añadida a la gramática de partida origina algunas diferencias en el análisis LR(1).
  Dada cualquier gramática independiente del contexto $G=<\Sigma_T, \Sigma_N, A, P>$, la gramática extendida para LR(1) se define así, donde $A'\notin\Sigma_N$ y $\$\notin\Sigma_T$:

  $G'=<\Sigma_T, \Sigma_N\cup\{A'\}, A', P\cup\{A' ::= A\}>$

  Es fácil comprobar que el lenguaje generado por G' es el mismo que el generado por G. Obsérvese que el símbolo '$' no aparece de forma explícita en G'. Sin embargo, la restricción impuesta sobre él es necesaria, porque en la construcción del autómata de análisis LR(1) el símbolo '$' se utilizará como el único símbolo de adelanto para la configuración del estado inicial.

- **Construcción de los conjuntos de símbolos de adelanto.** El único cambio en los conceptos descritos formalmente en la sección *Autómata asociado a un analizador LR(0): definiciones formales* es la incorporación del cálculo del conjunto de símbolos de adelanto al algoritmo general:

- La configuración inicial del estado inicial ($A'::=\bullet A$) tiene como conjunto de símbolos de adelanto $\{\$\}$. Dicho de otro modo:

  $A' ::= \bullet A\ \{\$\} \in s_0$

- Dado un estado cualquiera (s_i) del autómata, cuando se calcula $cierre(s_i)$
  $\forall P::=\alpha\bullet N\beta\ \{\sigma_1,\ldots,\sigma_m\}\in s_i,, P,N\in\Sigma_N\wedge\alpha,\beta\in(\Sigma_N\cup\Sigma_T)^*\wedge\{\sigma_1,\ldots,\sigma_m\}\subseteq\Sigma_T$
  $\Rightarrow N::=\bullet\gamma\ primero\_LR(1)(\beta.\{\sigma_1,\ldots,\sigma_m\})\in cierre(s_i).$

  Donde:

  1. $\beta.\{\sigma_1,\ldots,\sigma_m\}$ no define ninguna operación, sino que es una notación que se refiere a la concatenación de una cadena β y un conjunto de símbolos $\{\sigma_1,\ldots,\sigma_m\}$.

  2. $primero\_LR(1)$ es un conjunto que se puede definir en función del conjunto $primero$ de la siguiente manera:

     $primero\_LR(1)(\beta.\{\sigma_1,\ldots,\sigma_m\})=\cup_{i=1,\ldots,m}\{primero\ (\beta\sigma_i)\}$

     donde $\beta\sigma_i$ representa la concatenación habitual de símbolos.

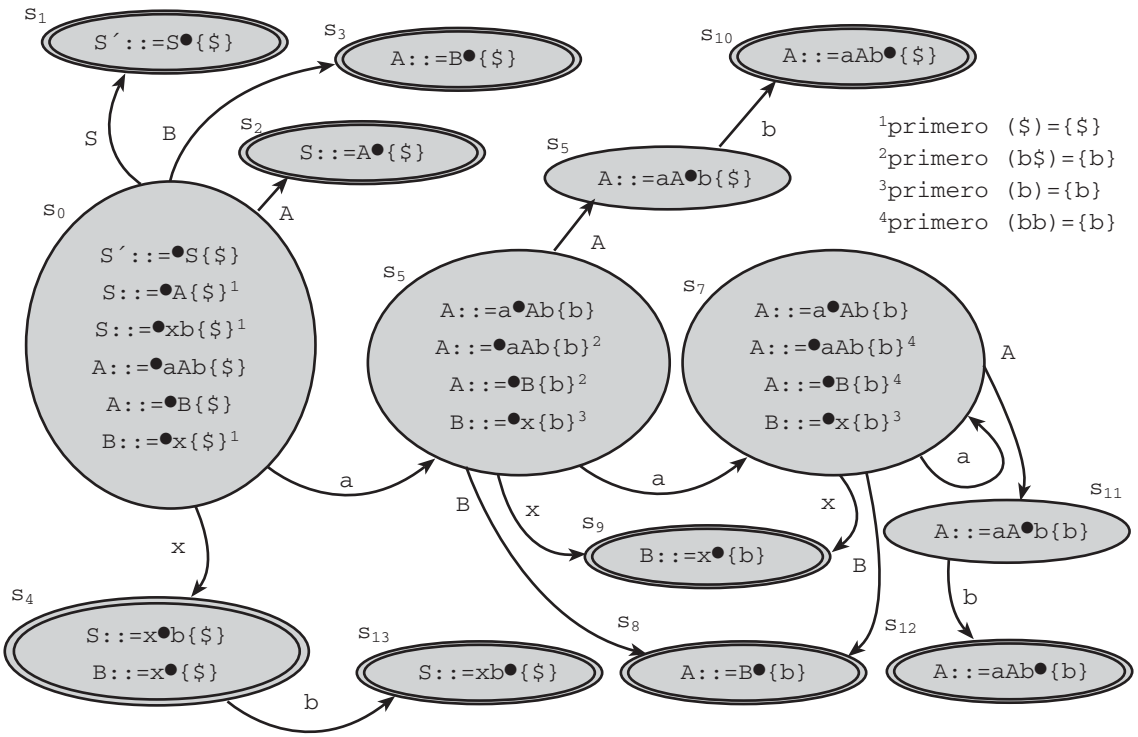


Figura 4.55. Diagrama de estados del autómata del analizador LR(1) de la gramática G_{axb}.

También es posible calcular este conjunto de la siguiente manera:

$$\texttt{primero\_LR(1)}(\beta.\{\sigma_1,\ldots,\sigma_m\}) = \begin{cases} \texttt{primero}(\beta) \text{ si } \lambda \notin \texttt{primero}(\beta) \\ \texttt{primero}(\beta)-\lambda \cup \{\sigma_1,\ldots,\sigma_m\} \text{ en otro caso} \end{cases}$$

- Dado un estado cualquiera (s_i) del autómata, y un símbolo cualquiera ($X \in \Sigma_N \cup \Sigma_T$), cuando se calcula $\texttt{ir\_a}(s_i,X)$ $\forall P::=\alpha\bullet X\beta\ \Omega \in s_i \Rightarrow \texttt{ir\_a}(s_i,X) \supseteq \texttt{cierre}(\{P::=\alpha X\bullet\beta\Omega\})$.

La Figura 4.55 muestra el diagrama de estados completo del analizador LR(1) de la gramática G_{axb}.

Observaciones sobre la naturaleza del conjunto de símbolos de adelanto

El conjunto de símbolos de adelanto no siempre contiene símbolos aislados. La Sección 4.3.8 se dedica al análisis LALR(1), en el que, de forma natural, se construyen conjuntos de símbolos de adelanto con más de un símbolo. Otra circunstancia en la que puede aparecer este tipo de conjuntos es en el cálculo del diagrama de estados, cuando en alguno de ellos surge la necesidad de incluir varias veces la misma configuración con diferentes símbolos de adelanto. En

este caso, el conjunto de símbolos de adelanto tiene que incluir todos los símbolos identificados.

Como ejemplo, considérese la siguiente gramática independiente del contexto para un fragmento de un lenguaje de programación de alto nivel que permite el tipo de dato *apuntador*, con una notación similar a la del lenguaje C. La gramática describe algunos aspectos de las asignaciones a los identificadores que incluyen posibles accesos a la información apuntada por un puntero, mediante el uso del operador '*'.

```
G*={  Σ_T={=,*,id},
      S,
      Σ_N={S,L,R},
      {     S→L=R | R,
            L→*R | id,
            R→L } }
```

La Figura 4.56 muestra el diagrama de estados del autómata de análisis LR(1).

Puede observarse en el estado s_0 la presencia de dos configuraciones cuyos conjuntos de símbolos de adelanto contienen dos símbolos. En el caso de L::=•*R{=,$} la presencia del símbolo '=' se justifica porque es el que sigue al no terminal L en la configuración que se está cerrando (S::•L=R{$}). Es necesario añadir también el símbolo $, ya que también hay que cerrar la configuración R::•L{$}, y en esta ocasión el símbolo no terminal L aparece al final de

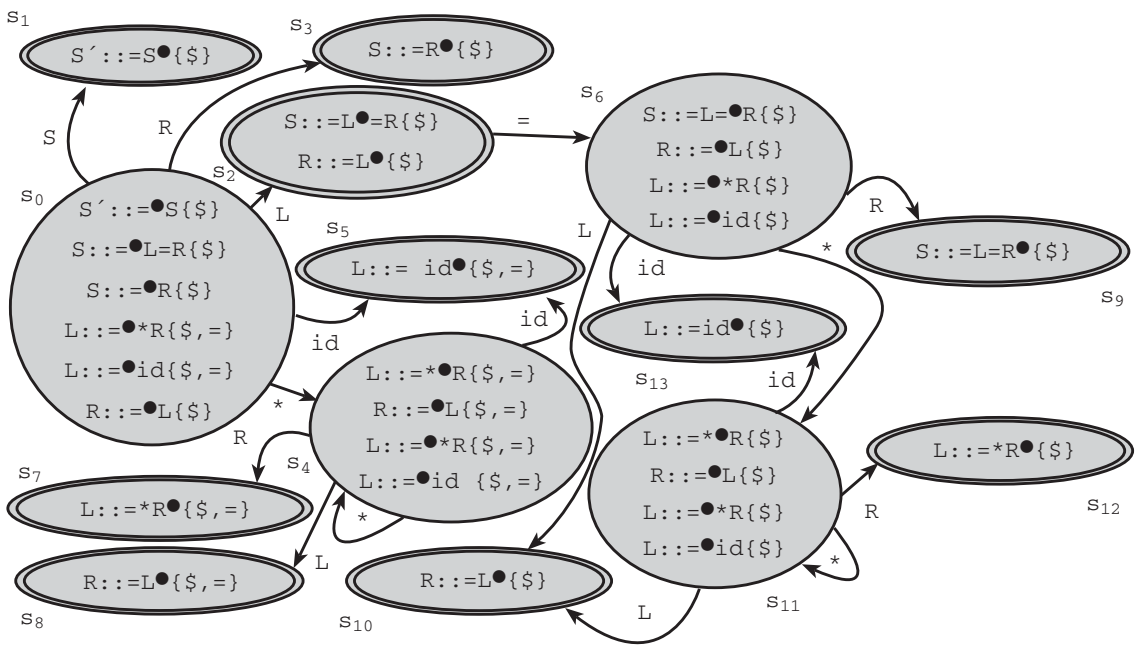


Figura 4.56. Ejemplo de diagrama de estados de autómata de análisis LR(1) con conjuntos de símbolos de adelanto no unitarios.

la parte derecha de la regla y no se modifican los símbolos de adelanto ({$}). El análisis para la configuración L::=•id{=,$} es similar.

Construcción de tablas de análisis LR(1)

La tabla de análisis tiene la misma estructura que la tabla LR(0), con la excepción de que en LR(1) la columna del símbolo de final de cadena ('$') aparece por convenio, ya que no pertenece estrictamente al conjunto de terminales.

- **Desplazamientos.** Igual que en LR(0), se obtienen siguiendo las transiciones del diagrama. Si el autómata transita del estado s_i al estado s_j mediante el símbolo x, en la casilla (i,x) de la tabla se añadirá **dj** si $x \in \Sigma_T$ y **j** si $x \in \Sigma_N$.
- **Reducciones.** No se aplica el mismo proceso que en las tablas LR(0) ni SLR(1). Igual que en LR(0), se consultan los estados finales del diagrama, pero en este caso se utilizan los conjuntos de símbolos de adelanto de las configuraciones de reducción. Si el estado final es s_i y su configuración de reducción es $N ::= \gamma \bullet \{\sigma_1, \ldots \sigma_m,\}$ (donde $N ::= \gamma$ es la regla número k), la acción **rk** se añadirá sólo en las casillas correspondientes a las columnas de los símbolos de adelanto $\{\sigma_1, \ldots \sigma_m,\}$.
- **Aceptación.** No se aplica el mismo proceso que en las tablas LR(0), ya que el símbolo final de la cadena '$' no forma parte explícitamente de la gramática, sino que aparece sólo en los símbolos de adelanto. La acción de aceptación se escribe en la casilla (i, $), siempre que la i represente al estado s_i que contiene la configuración A::=A'•{$}.
- **Error.** Igual que en las tablas LR(0), todas las demás casillas corresponden a errores sintácticos.

Como ejemplo, la Figura 4.57 muestra la tabla de análisis LR(1) de la gramática G_{axb}.

Puede comprobarse que se ha resuelto el conflicto que hacía que G_{axb} no fuese SLR(1).

Definición de gramática LR(1)

Una gramática independiente del contexto G es una *gramática LR(1)* si y sólo si su tabla de análisis LR(1) es determinista, es decir, no presenta conflictos.

Evaluación de la técnica

Comparando el tamaño de los diagramas de estado y de las tablas de análisis SLR(1) y LR(1) para la gramática G_{axb}, que aparecen respectivamente en las Figuras 4.48, 4.49, 4.55 y 4.56, se comprueba que el aumento de precisión para solucionar el conflicto implica un aumento considerable en el tamaño de ambos elementos. Es fácil ver, sobre todo en los diagramas de estado, que los nuevos estados s_7, s_8, s_{11} y s_{12} se originan como copias, respectivamente, de los antiguos estados s_3, s_4, s_7 y s_9, con símbolos de adelanto distintos.

Se puede demostrar que LR(1) es el algoritmo de análisis más potente entre los que realizan el recorrido de la cadena de entrada de izquierda a derecha con ayuda de un símbolo de adelanto.

También tiene interés la extensión de este algoritmo de análisis a un conjunto mayor de símbolos de adelanto. Como se ha dicho previamente, estas extensiones se denominan LR(k), donde

	Σ_T				Σ_N			
E	a	b	x	$	S´	S	A	B
0	d5		d4			1	2	3
1				acc				
2				r1				
3				r4				
4		d13		r5				
5	d7		d9				6	8
6		d10						
7	d7		d9				11	8
8		r4						
9		r5						
10				r3				
11		d12						
12		r3						
13				r2				
	Acción				Ir_a			

Figura 4.57. Tabla de análisis LR(1) para la gramática G_{axb}.

k representa la longitud de los elementos de los conjuntos de adelanto. En la práctica, las gramáticas LR(1) son capaces de expresar las construcciones presentes en la mayoría de los lenguajes de programación de alto nivel. El incremento observado en el tamaño de los diagramas de estado y las tablas de análisis se acentúa cuando se utiliza un valor de k mayor que 1. Por ello, en la práctica, los compiladores e intérpretes no suelen utilizar valores de k mayores que 1.

En la sección siguiente no se intentará incrementar la potencia expresiva de los analizadores ascendentes, sino sólo mitigar la ineficiencia derivada del aumento del tamaño de las tablas de análisis al pasar de los analizadores LR(0) y SLR(1) a LR(1).

4.3.8. LALR(1)

Las siglas LALR hacen referencia a una familia de analizadores sintácticos ascendentes que utilizan símbolos de adelanto (de la expresión inglesa ***L**ook-**A**head-**L**eft-to-**R**ight*). El número que acompaña a LALR tiene el mismo significado que en LR(k).

Motivación

Después de recorrer las diferentes técnicas del análisis ascendente, desde LR(0) hasta LR(1), pasando por SLR(1), se llega a la conclusión de que la potencia del análisis LR(1), y la falta de precisión del análisis SLR(1), se deben a que, aunque los conjuntos de símbolos de adelanto y los conjuntos `siguiente` pueden estar relacionados (los símbolos de adelanto de una configuración parecen, intuitivamente, estar incluidos en el conjunto `siguiente` del no terminal de la parte izquierda de su regla), tienen significados distintos. Que un símbolo terminal pueda seguir a la parte izquierda de una regla no significa que tenga que aparecer, cada vez que se reduzca, a continuación de ella. De hecho, los conjuntos `siguiente` dependen sólo de la regla, mientras que los de adelanto dependen de la configuración y de su historia.

El estudio del diagrama de la Figura 4.55 muestra la existencia de estados que sólo se diferencian en los símbolos de adelanto de sus configuraciones. Ante esta situación cabe formularse la siguiente pregunta: ¿sería posible minimizar el número de estados distintos, realizando la unión de todos los símbolos de adelanto y excluyendo de los conjuntos siguientes los símbolos que realmente no pueden aparecer inmediatamente después de reducir la regla de la configuración correspondiente? Las próximas secciones se dedicarán a comprobar que la respuesta es afirmativa y a articular una nueva técnica de análisis ascendente que hace uso de ella.

Ejemplo 4.11

A continuación se revisará el ejemplo de la gramática G_{axb} desde el punto de vista descrito en la sección anterior. La Figura 4.58 resalta cuatro parejas de estados (s_3 y s_8, s_5 y s_7, s_6 y s_{11}, s_{10} y s_{12}) en el diagrama de estados del autómata de análisis LR(1). Son cuatro parejas de estados distintos, que sólo difieren en los símbolos de adelanto de sus configuraciones.

Se intentará reducir el tamaño del diagrama agrupando esos estados. El lector familiarizado con la teoría de autómatas reconocerá esta situación como la de minimización de un autómata. En cualquier caso, la reducción es un proceso iterativo, en el que dos estados se transformarán en uno solo siempre que sean equivalentes. La equivalencia de estados se basa en las siguientes condiciones:

- Que sólo difieran en los símbolos de adelanto de las configuraciones. El estado que se obtiene al unir los de partida, contendrá en cada configuración la unión de los símbolos de adelanto.
- El nuevo estado debe mantener las transiciones del diagrama, es decir, tienen que llegar a él todas las transiciones que llegaran a los de partida y salir de él todas las que salieran de ellos.

El proceso termina cuando no se puedan agrupar más estados.

- **Pareja s_3 - s_8**: La Figura 4.59 muestra el proceso de unión de estos dos estados, que da lugar al estado nuevo s_{3_8}.

Obsérvese que la unión es posible porque

- La configuración de los dos estados sólo difiere en los símbolos de adelanto.
- Las dos transiciones que llegan desde s_0 a s_3 y desde s_5 y s_7 a s_8 pueden sin problemas llegar a s_{3_8}.
- No hay transiciones que salgan de s_3 ni de s_8.

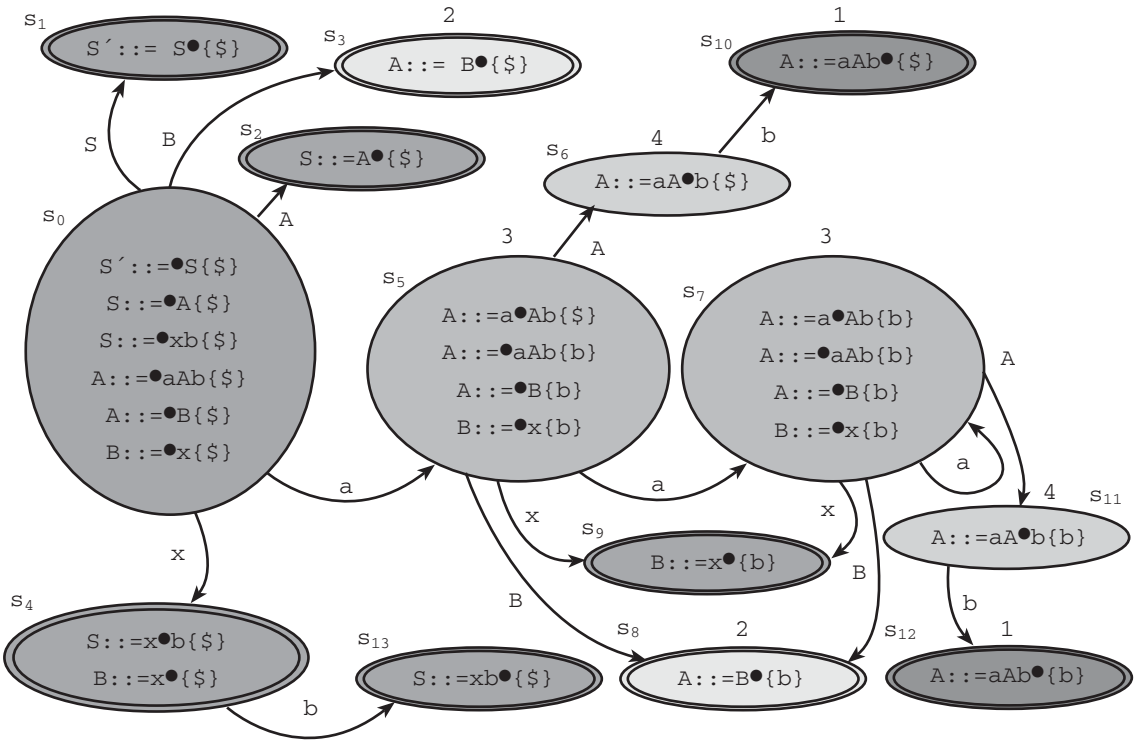


Figura 4.58. Diagrama de estados del autómata de análisis LR(1) de la gramática G_{axb} en el que se indican los conjuntos de símbolos distintos que sólo difieren en los símbolos de adelanto de sus configuraciones.

El estado resultante es: s_{3_8}={A::=B• {$,b}}

- **Pareja s_{10} - s_{12}**: La Figura 4.60 muestra el proceso para esta pareja.

Por razones análogas, la unión de los dos estados es posible y su resultado es s_{10_12}={A::=aAb• {$,b}}.

- **Pareja s_6 - s_{11}**: La Figura 4.61 muestra el proceso para esta pareja.

Este caso presenta una situación nueva: tanto s_6 como s_{11} tienen transiciones de salida mediante el símbolo b. La unión es posible, porque las dos llegan al estado nuevo s_{10_12}, por lo que el estado resultado (s_{6_11}) tendrá una transición con el símbolo b al estado s_{10_12}.

Es conveniente reflexionar acerca del orden en que se realizan las uniones. Si se hubiera intentado unir esta pareja antes que s_{10} - s_{12}, la unión no habría sido posible. No debe preocupar esta situación, ya que, al ser el proceso iterativo, tarde o temprano se habría unificado la pareja 10 - 12, y después de ella también la 6 - 11.

En cualquier caso el resultado es s_{6_11}={A::=aA•b {$,b}}

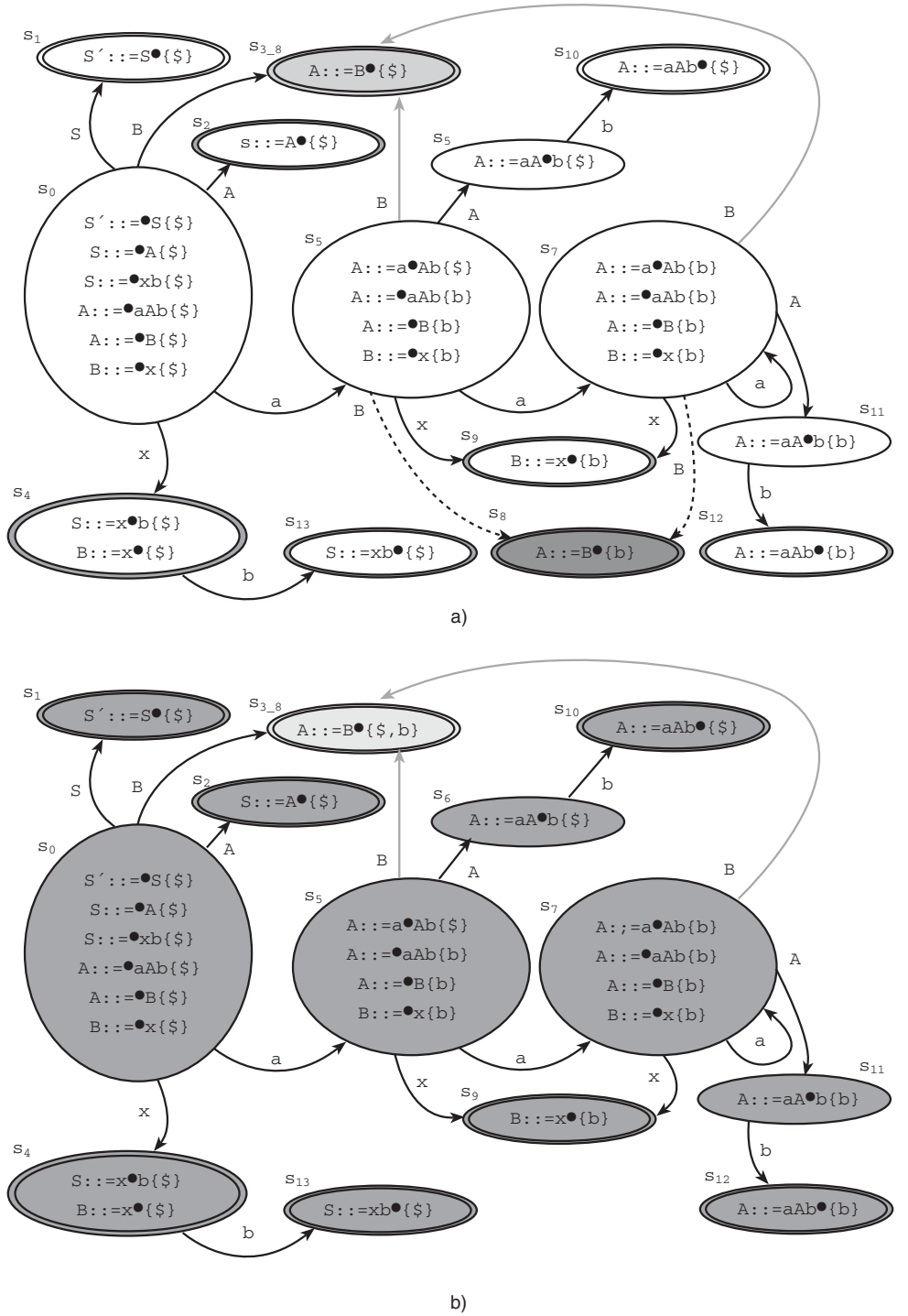
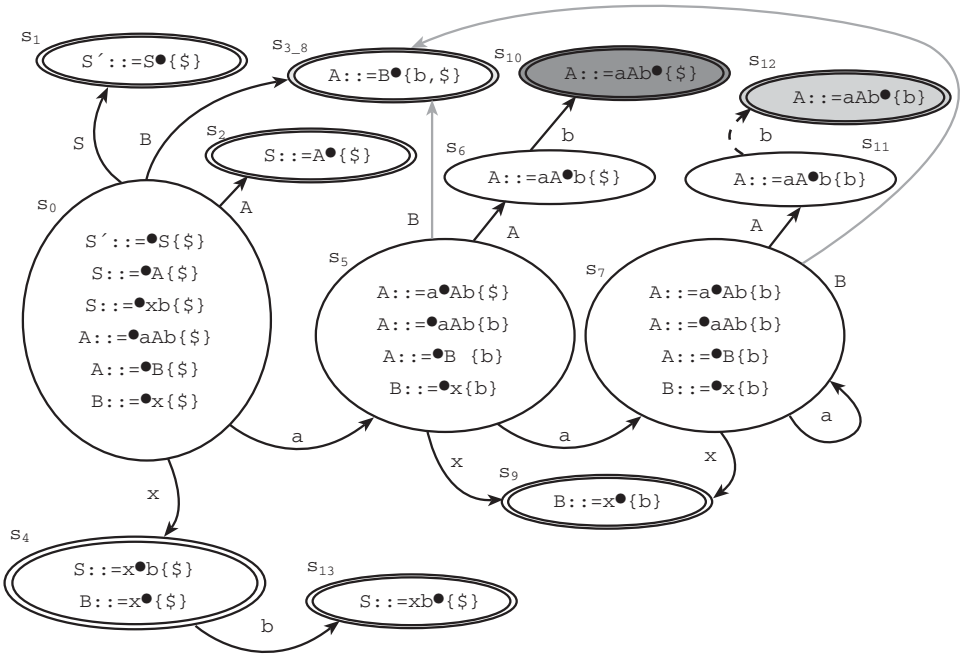
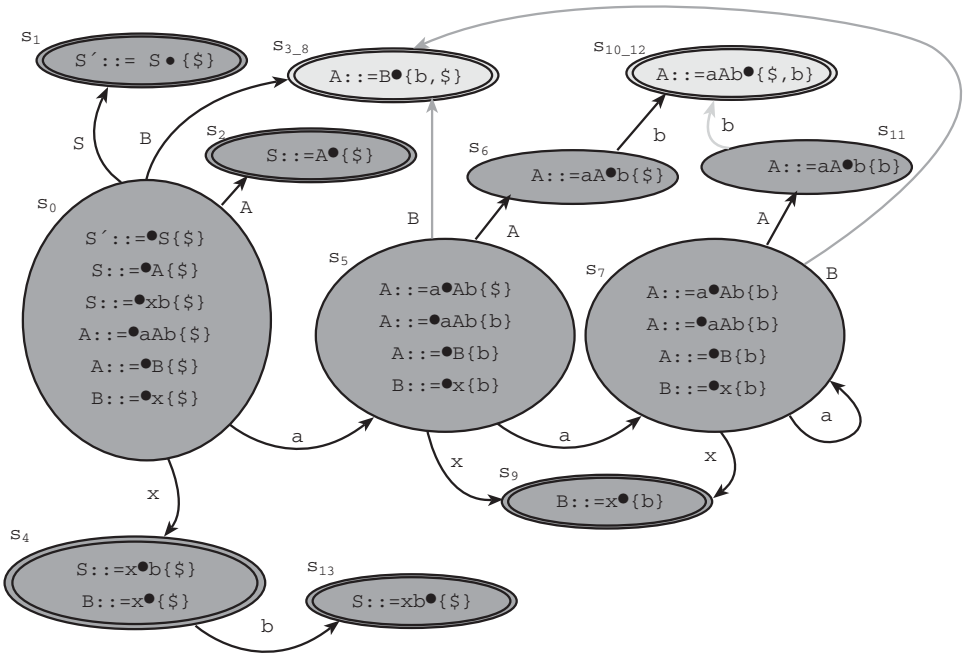


Figura 4.59. Unión de los estados s_3 y s_8 en el nuevo estado s_{3_8}. (a) Antes de la unión: se resaltan las transiciones afectadas. (b) Después de la unión: se resalta el estado resultado.

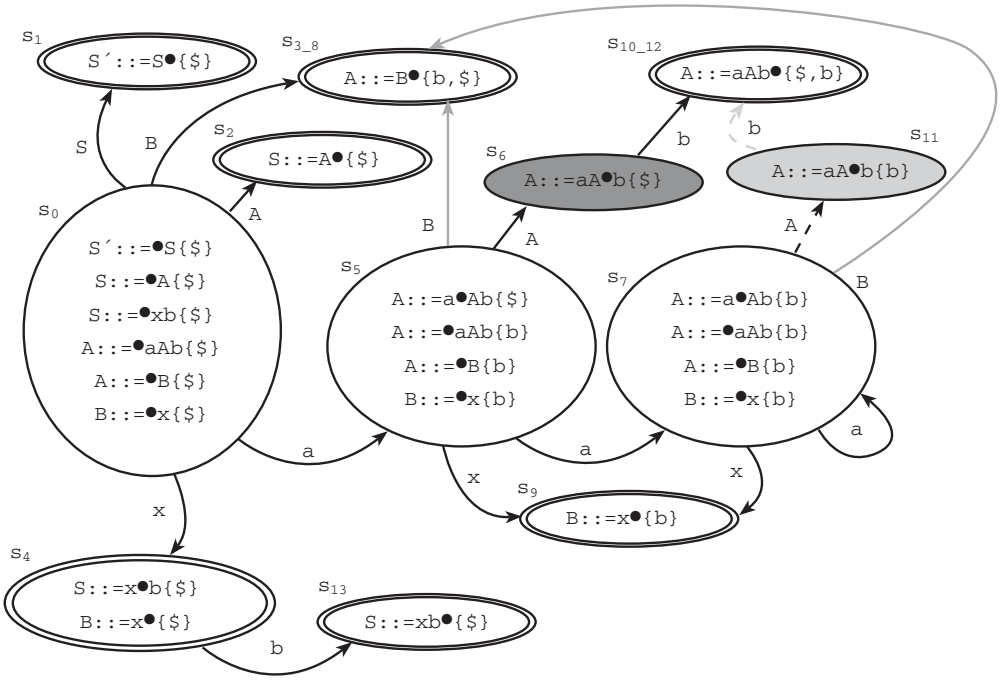


a)

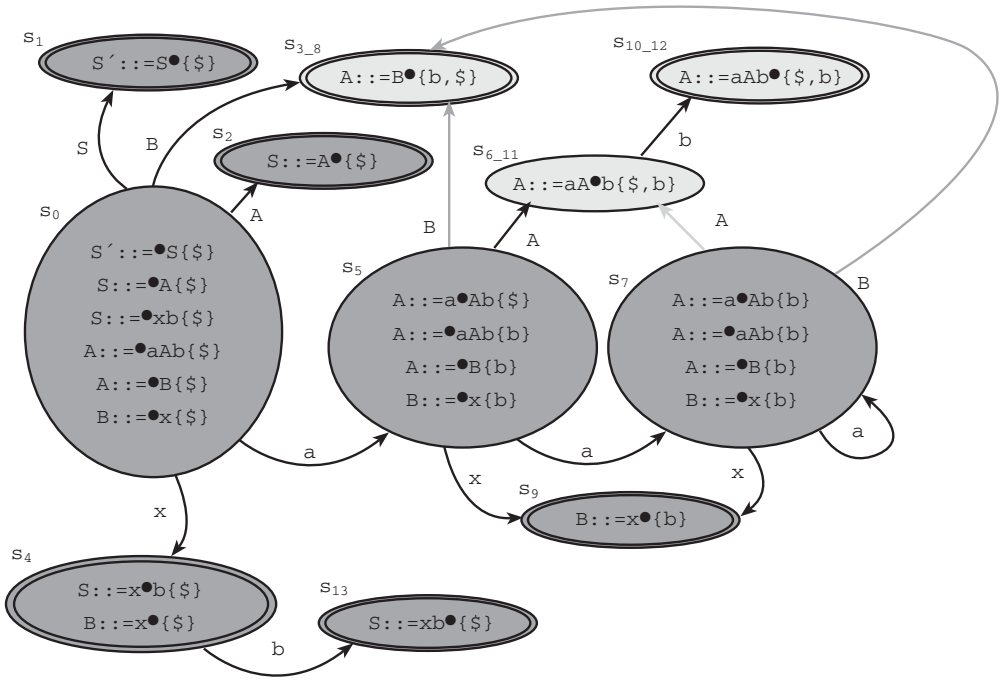


b)

Figura 4.60. Unión de los estados s_{10} y s_{12} en el nuevo estado s_{10_12}.



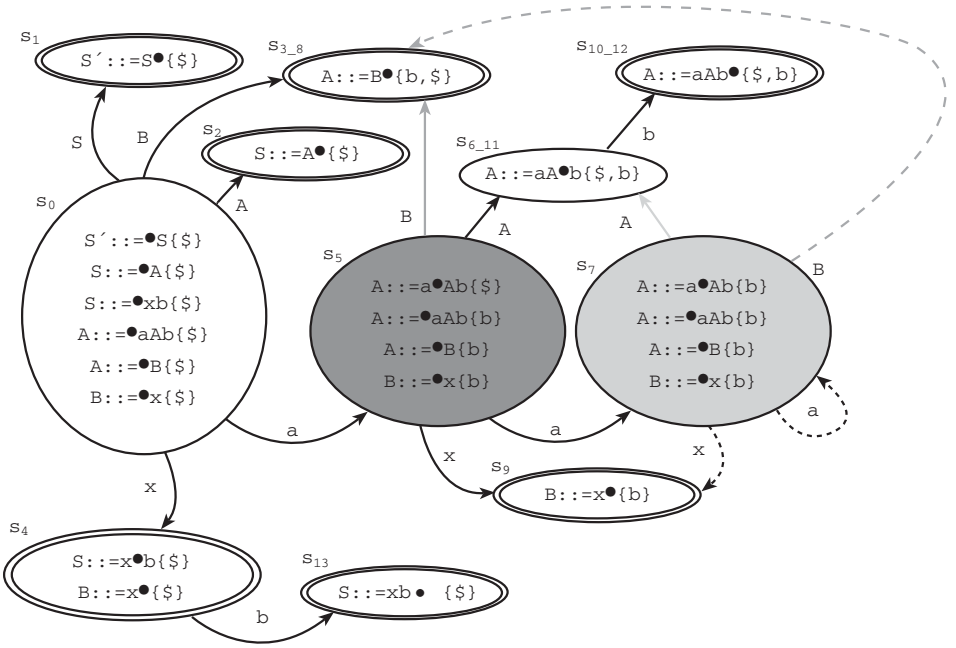
a)



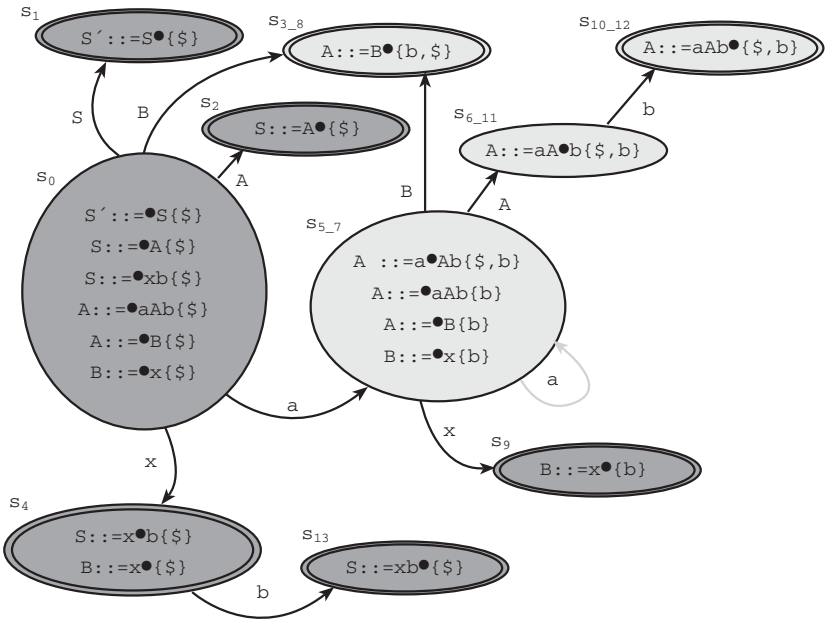
b)

Figura 4.61. Unión de los estados s_6 y s_{11} en el nuevo estado s_{6_11}.

- **Pareja s_5 - s_7**: La Figura 4.62 muestra el proceso para esta pareja.



a)



b)

Figura 4.62. Unión de los estados s_5 y s_7 en el nuevo estado s_{5_7}.

Por razones análogas (en este caso las transiciones potencialmente peligrosas llegan a s_{3_8} y a s_{6_11}, que ya están unificados) la unión es posible y el resultado es

s_{5_7}={A::=a•Ab {$,b},
A::= •aAb {b},
A::= •B {b},
B::= •x {b}}

Construcción del autómata de análisis LALR(1) a partir del autómata LR(1)

Se puede formalizar, en forma de pseudocódigo, el proceso descrito anteriormente, para calcular el diagrama de estados del analizador LALR(1) a partir del autómata del analizador LR(1). En el siguiente pseudocódigo, para representar la transición desde el estado s_p al estado s_d mediante el símbolo a, se utilizará la siguiente notación:

$(s_o,a) ::= s_d$

Mientras haya cambios en el diagrama de estados:

Para cada pareja de estados s_i y s_j que cumplan que sus configuraciones sólo difieren en los símbolos de adelanto se realizará, si se puede, la siguiente unificación:

Se crea un nuevo estado s_{i_j} cuyo contenido se calcula mediante el siguiente proceso:

Para cada pareja de configuraciones

c^i=x $\{s_1,\ldots,s_n\}\in s_i$
c^j=x $\{d_1,\ldots,d_m\}\in s_j$

se añade a s_{i_j} la configuración

x $\{s_1,\ldots,s_n\}\cup\{d_1,\ldots,d_m\}$

cuyas transiciones se calculan de la siguiente manera:

- Cada transición $(s_o,a)\rightarrow s_i$ [ídem. $(s_o,a)\rightarrow s_j$)] del autómata de análisis LR(1) origina en el autómata de análisis LALR(1) una transición $(s_p,a)\rightarrow s_{i\_j}$.
- Cada transición $(s_i,a)\rightarrow s_d$ [ídem. $(s_j,a)\rightarrow s_d$)] del autómata de análisis LR(1) origina en el autómata de análisis LALR(1) una transición $(s_{i\_j},a)\rightarrow s_d$. Obsérvese que esta operación es la que podría causar que la unificación fuera imposible, ya que el autómata tiene que seguir siendo determinista, y esto no sería posible si existiera algún símbolo para el que las transiciones desde s_i y desde s_j no terminaran en el mismo estado.

A modo de ejemplo, la parte b) de la Figura 4.62 muestra el diagrama de estados del autómata de análisis LALR(1) para la gramática G_{axb}.

Construcción de tablas de análisis LALR(1) a partir del autómata LALR(1)

El algoritmo de creación de la tabla de análisis LALR(1) es el mismo que en LR(1).

La Figura 4.63 muestra la tabla de análisis LALR(1) de la gramática G_{axb}.

	Σ_T				Σ_N			
E	a	b	x	$	S´	S	A	B
0	d5_7		d4			1	2	3_8
1				acc				
2				r1				
3_8		r4		r4				
4		d13		r5				
5_7	d5_7		d9				6_11	3_8
6_11		d10_12						
9		r5						
10_12		r3		r3				
13				r2				
	Acción				Ir_a			

Figura 4.63. Tabla de análisis LALR(1) para la gramática G_{axb}.

Otros algoritmos para construir LALR(1) sin pasar por LR(1)

El algoritmo descrito en este capítulo para llegar a LALR(1) mediante LR(1) no es el más eficiente para la generación automática de analizadores LALR(1). Existen versiones de este algoritmo que construyen directamente el diagrama de estados del analizador LALR(1) sin necesidad de construir el analizador LR(1).

Evaluación de la técnica

Es fácil comprobar que, debido al mecanismo de construcción del analizador LALR(1), no se pueden añadir conflictos reducción / desplazamiento a los que ya tuviera el analizador LR(1). Por otra parte, aunque no en todos los casos se consigue reducir el tamaño de las tablas y de los diagramas, a veces se consigue la potencia de un analizador LR(1) con el tamaño de un analizador LR(0). Esto hace que LALR(1) sea la técnica de análisis ascendente más extendida.

De hecho, existen herramientas informáticas de libre distribución (como `yacc` o `bison`) que construyen automáticamente analizadores de este tipo. Usualmente estas herramientas no sólo generan el analizador sintáctico, sino que añaden más componentes, proporcionando esqueletos de compiladores e intérpretes. En capítulos sucesivos, tras describir otras componentes de los compiladores necesarias para entender estas herramientas, se proporcionará una breve descripción de las mismas. También se puede consultar en `http://www.librosite.net/pulido` enlaces de interés, documentación detallada y ejemplos de uso.

4.4 Gramáticas de precedencia simple

El método del análisis sintáctico mediante gramáticas de precedencia simple tiene utilidad, y puede ser más eficiente que otros, en los lenguajes de expresiones, que desempeñan un papel importante en el acceso a bases de datos, en las fórmulas de las hojas de cálculo, y en otras aplicaciones.

En esta sección se utilizarán dos conjuntos especiales, llamados `first(U)` y `last(U)`, que se definen de la siguiente manera:

Sea una gramática $G = (\Sigma_T, \Sigma_N, S, P)$. Sea $\Sigma = \Sigma_T \cup \Sigma_N$. Sea $U \in \Sigma$ un símbolo de esta gramática. Se definen los siguientes conjuntos asociados a estas gramáticas y a este símbolo:

- $first(U) = \{V \mid U \rightarrow^+ Vx, V \in \Sigma, x \in \Sigma^*\}$
- $last(U) = \{V \mid U \rightarrow^+ xV, V \in \Sigma, x \in \Sigma^*\}$

Obsérvese que, en estas funciones, $U \in \Sigma_N$, es decir, U tiene que ser no terminal.

Estos conjuntos se calculan con facilidad aplicando conceptos de la teoría algebraica de relaciones, aunque esto nos fuerza a establecer las siguientes restricciones en la gramática:

- Los símbolos no terminales distintos del axioma no contienen reglas no generativas (reglas de la forma $U ::= \lambda$). Si existieran reglas así, se puede obtener una gramática equivalente que no las contenga, aplicando el procedimiento explicado en la Sección 1.12.5.

- Si el axioma genera la palabra vacía, las reglas que definen sus derivaciones directas no deben ser recursivas. Si lo fuesen, se puede construir una gramática equivalente que cumpla esta condición, introduciendo un nuevo axioma que genere directamente el axioma antiguo, y aplicando el procedimiento de la Sección 1.12.5 para trasladar la cadena vacía al nuevo axioma.

Por ejemplo, sea la gramática `({a,b,c,d}, {S,B}, S, P)`, donde P contiene las siguientes reglas de producción:

$S ::= a\ S\ b \mid B$
$B ::= c\ B\ d \mid \lambda$

Esta gramática tiene una regla no generativa en el símbolo B, que no es el axioma. Para eliminarla, hay que añadir las reglas que se obtienen sustituyendo B por λ en todas las partes derechas, de donde resulta:

```
S ::= a S b | B | λ
B ::= c B d | cd
```

Esta gramática cumple la primera condición, pero no la segunda, pues el axioma es recursivo y genera la palabra vacía. Aplicando el método propuesto, se puede transformar en la siguiente gramática equivalente:

```
S' ::= S
S ::= a S b | B | λ
B ::= c B d | cd
```

Ahora se elimina la regla no generativa, con lo que se obtiene la siguiente gramática:

```
S' ::= S | λ
S ::= a S b | a b | B
B ::= c B d | cd
```

Esta gramática cumple las dos restricciones anteriores y es totalmente equivalente a la gramática de partida (genera el mismo lenguaje).

4.4.1. Notas sobre la teoría de relaciones

Sea un conjunto A. Una relación sobre los elementos de A se define como $R \subset A \times A$, es decir, un conjunto de pares de elementos de A.

Sea $(a,b) \in R$ un par de elementos de A que están en relación R (se representa aRb). La relación R se puede definir también por enumeración de sus elementos: $R = \{(a,b) \mid aRb\}$.

- Sea una relación R entre elementos de A. Se llama *relación transpuesta* de R a la relación R' definida así: $aR'b \Leftrightarrow bRa$.
- Una relación R se llama *reflexiva* si todos los elementos $a \in A$ cumplen que aRa.
- Una relación R se llama *transitiva* si todos los elementos $a,b,c \in A$ cumplen que $aRb \wedge bRc \Rightarrow aRc$.
- Sean dos relaciones R, P entre elementos de A. Se dice que dos elementos $a,b \in A$ están en la relación *producto* de R y P, y se representa aRPb, si existe un elemento $c \in A$ tal que $aRc \wedge cPb$. El producto de relaciones cumple la propiedad asociativa.
- Se llama *potencia* de una relación R, y se representa R^n, al producto de R por sí misma n veces. Se define $R^1=R$ y $R^0=I$, donde I es la relación *identidad*, definida así: $aIb \Leftrightarrow a=b$.
- Se llama clausura transitiva de una relación R a la siguiente relación:

$$R^+ = \bigcup_{i=1}^{\infty} R^i$$

Obviamente, $aRb \Rightarrow aR^+b$. Además, cualquiera que sea R, R^+ es transitiva.

- Se llama *clausura reflexiva y transitiva* de una relación R a la siguiente relación:

$$R^* = \bigcup_{i=0}^{\infty} R^i$$

Obviamente, $aRb \Rightarrow aR^*b$. Además, cualquiera que sea R, R^* es transitiva y reflexiva.

Teorema 4.4.1. Si A es un conjunto finito de n elementos y R es una relación sobre los elementos de A, entonces $aR^+b \Rightarrow aR^kb$ para algún k positivo menor o igual que n. Esto significa que, si A es finito,

$$R^+ = \bigcup_{i=1}^{n} R^i$$

Demostración:

$aR^+b \Rightarrow aR^pb$ para algún $p>0$. Pero $aR^pb \Rightarrow \exists\ s_1, s_2, \ldots, s_p$ tal que $a=s_1$, s_1Rs_2, s_2Rs_3, … , $s_{p-1}Rs_p$, s_pRb (por definición de R^p). Supongamos que p es mínimo y $p>n$. Entonces, como A sólo contiene n elementos, mientras que la sucesión de s_i contiene $p>n$, debe haber elementos repetidos en dicha sucesión. Sea $s_i=s_j$, $j>i$. Entonces, $a=s_1$, s_1Rs_2, s_2Rs_3, … , $s_{i-1}Rs_i$, $s_i=s_j$, s_jRs_{j+1}, … , $s_{p-1}Rs_p$, s_pRb, y por tanto existe una sucesión más corta, con $p-(j-i)$ términos intermedios, para pasar de a a b. Esto contradice la hipótesis de que $p>n$ sea mínimo. Luego p tiene que ser menor o igual que n.

4.4.2. Relaciones y matrices booleanas

Se llama *matriz booleana* aquella cuyos elementos son valores lógicos, representados por los números 1 (verdadero) y 0 (falso). Las operaciones lógicas clásicas ($\vee$, $\wedge$) pueden aplicarse a los elementos o a las matrices en la forma usual. El *producto booleano* de matrices se define igual que el producto matricial ordinario, sustituyendo la suma por la operación $\vee$ y la multiplicación por la operación $\wedge$. El producto booleano de dos matrices B y C se representa con el símbolo $B\vee.\wedge C$.

Sea A un conjunto finito de n elementos, y sea R una relación sobre los elementos de A. Se puede representar R mediante una matriz booleana B de n filas y n columnas, donde $b_{ij}=1 \Leftrightarrow a_iRa_j$, y 0 en caso contrario. La aplicación M(R)=B, que pasa de las relaciones a las matrices booleanas, es un isomorfismo, pues tiene las siguientes propiedades:

- $M(R') = (M(R))'$

  La matriz booleana de la relación transpuesta de R es la matriz transpuesta de la matriz correspondiente a R.

- $M(R \cup P) = M(R) \vee M(P)$

  La matriz booleana de la unión de dos relaciones es la unión lógica de las dos matrices booleanas correspondientes.

- $M(RP) = M(R) \vee.\wedge M(P)$

  La matriz booleana de la relación producto de otras dos es el producto booleano de las dos matrices correspondientes.

- $M(R^n) = (M(R))^n$

  La matriz booleana de la potencia enésima de una relación es la potencia enésima de la matriz booleana de la relación. Se llama potencia enésima de B el producto booleano de B por sí misma n veces. Además, B^0 es la matriz unidad.

- $M(R^+) = (M(R))^+$

  La matriz booleana de la clausura transitiva de una relación es la clausura transitiva de la matriz de la relación, definida esta última operación como:

$$B^+ = \bigcup_{i=1}^{\infty} B^i$$

  Como consecuencia del Teorema 4.4.1, se cumple que

$$B^+ = \bigcup_{i=1}^{n} B^i$$

4.4.3. Relaciones y conjuntos importantes de la gramática

Recuérdese la definición del conjunto `first(U)` al principio de la Sección 4.4:

$$first(U) = \{V \mid U \rightarrow + Vx,\ V \in \Sigma,\ x \in \Sigma^*\}$$

Esta expresión define el conjunto `first(U)` en función de la relación $\rightarrow+$, que a su vez actúa sobre elementos de Σ^*, que es un conjunto infinito, por lo que no se puede aplicar el Teorema 4.4.1. Para obtener un algoritmo que permita calcular fácilmente `first(U)`, se puede definir la siguiente relación, que actúa sobre conjuntos finitos:

$$U\ \mathcal{F}\ V \Leftrightarrow U ::= Vx \in P,\ V \in \Sigma,\ x \in \Sigma^*$$

Se calcula el cierre transitivo de $\mathcal{F}$:

$$U\ \mathcal{F}^+\ V \Leftrightarrow U ::= V_1 x_1 \in P,\ V_1 ::= V_2 x_2 \in P,\ \ldots,\ V_n ::= V x_{n+1} \in P$$

$$V_1, V_2, \ldots, V_n \in \Sigma_N,\ V \in \Sigma,\ x_1, x_2, \ldots, x_{n+1} \in \Sigma^*$$

De la expresión anterior se deduce que

$$U\ \mathcal{F}^+\ V \Leftrightarrow u \rightarrow + Vx$$

por tanto,

$\texttt{first(U)} = \{V \mid U\ \mathcal{F}^+\ V,\ V \in \Sigma\}$

Pero la relación $\mathcal{F}$ está definida sobre un conjunto finito Σ, y se puede aplicar el Teorema 4.4.1.

De la misma forma en que se ha definido la relación $\mathcal{F}$ y el conjunto `first(U)`, se puede definir la siguiente relación y el siguiente conjunto:

- $U\ \mathcal{L}\ V \Leftrightarrow U ::= xV \in P,\ V \in \Sigma,\ x \in \Sigma^*$
- $\texttt{last(U)} = \{V \mid U\ \mathcal{L}^+\ V\}$

Ejemplo 4.12 Sea la gramática `({0,1,2,3,4,5,6,7,8,9}, {N,C}, N, {N::=NC|C, C::=0|1|2|3|4|5|6|7|8|9})`. Para calcular `first(C)`, se empieza definiendo la relación $\mathcal{F}$:

Regla	Relación
N::=NC	N $\mathcal{F}$ N
N::=C	N $\mathcal{F}$ C
C::=0	C $\mathcal{F}$ 0
C::=1	C $\mathcal{F}$ 1
C::=2	C $\mathcal{F}$ 2
C::=3	C $\mathcal{F}$ 3
C::=4	C $\mathcal{F}$ 4
C::=5	C $\mathcal{F}$ 5
C::=6	C $\mathcal{F}$ 6
C::=7	C $\mathcal{F}$ 7
C::=8	C $\mathcal{F}$ 8
C::=9	C $\mathcal{F}$ 9

Por tanto,

$\mathcal{F} = \{(N,N), (N,C), (C,0), (C,1), (C,2), \ldots, (C,9)\}$

La matriz booleana equivalente a $\mathcal{F}$ es la siguiente matriz B:

```
   N C 0 1 2 3 4 5 6 7 8 9
N: 1 1 0 0 0 0 0 0 0 0 0 0
C: 0 0 1 1 1 1 1 1 1 1 1 1
0: 0 0 0 0 0 0 0 0 0 0 0 0
1: 0 0 0 0 0 0 0 0 0 0 0 0
2: 0 0 0 0 0 0 0 0 0 0 0 0
3: 0 0 0 0 0 0 0 0 0 0 0 0
4: 0 0 0 0 0 0 0 0 0 0 0 0
5: 0 0 0 0 0 0 0 0 0 0 0 0
6: 0 0 0 0 0 0 0 0 0 0 0 0
7: 0 0 0 0 0 0 0 0 0 0 0 0
8: 0 0 0 0 0 0 0 0 0 0 0 0
9: 0 0 0 0 0 0 0 0 0 0 0 0
```

donde las filas y las columnas corresponden a los símbolos {N,C,0,1,2,3,4,5,6,7,8,9}, en ese orden. De aquí se puede calcular que $B^2 = B^3 = \ldots =$ la siguiente matriz:

```
   N C 0 1 2 3 4 5 6 7 8 9
N: 1 1 1 1 1 1 1 1 1 1 1 1
C: 0 0 0 0 0 0 0 0 0 0 0 0
0: 0 0 0 0 0 0 0 0 0 0 0 0
1: 0 0 0 0 0 0 0 0 0 0 0 0
2: 0 0 0 0 0 0 0 0 0 0 0 0
3: 0 0 0 0 0 0 0 0 0 0 0 0
4: 0 0 0 0 0 0 0 0 0 0 0 0
5: 0 0 0 0 0 0 0 0 0 0 0 0
6: 0 0 0 0 0 0 0 0 0 0 0 0
7: 0 0 0 0 0 0 0 0 0 0 0 0
8: 0 0 0 0 0 0 0 0 0 0 0 0
9: 0 0 0 0 0 0 0 0 0 0 0 0
```

(En cuanto dos potencias de la matriz coinciden, las restantes son todas iguales). B^+ es la unión booleana de todas las matrices (las dos) anteriores:

```
   N C 0 1 2 3 4 5 6 7 8 9
N: 1 1 1 1 1 1 1 1 1 1 1 1
C: 0 0 1 1 1 1 1 1 1 1 1 1
0: 0 0 0 0 0 0 0 0 0 0 0 0
1: 0 0 0 0 0 0 0 0 0 0 0 0
2: 0 0 0 0 0 0 0 0 0 0 0 0
3: 0 0 0 0 0 0 0 0 0 0 0 0
4: 0 0 0 0 0 0 0 0 0 0 0 0
5: 0 0 0 0 0 0 0 0 0 0 0 0
6: 0 0 0 0 0 0 0 0 0 0 0 0
7: 0 0 0 0 0 0 0 0 0 0 0 0
8: 0 0 0 0 0 0 0 0 0 0 0 0
9: 0 0 0 0 0 0 0 0 0 0 0 0
```

Por tanto,

```
F+ = {(N,N), (N,C), (N,0), (N,1), (N,2), ..., (N,9),
      (C,0), (C,1), (C,2), ..., (C,9)}
```

Y así se tiene que:

```
first(C) = {0, 1, 2, ..., 9}
```

Ahora se calcula el conjunto `last(N)`. Para ello, se define la relación $\mathcal{L}$:

Regla	Relación $\mathcal{L}$
N::=NC	N $\mathcal{L}$ C
N::=C	N $\mathcal{L}$ C
C::=0	C $\mathcal{L}$ 0
C::=1	C $\mathcal{L}$ 1
C::=2	C $\mathcal{L}$ 2
C::=3	C $\mathcal{L}$ 3
C::=4	C $\mathcal{L}$ 4
C::=5	C $\mathcal{L}$ 5
C::=6	C $\mathcal{L}$ 6
C::=7	C $\mathcal{L}$ 7
C::=8	C $\mathcal{L}$ 8
C::=9	C $\mathcal{L}$ 9

La matriz de la relación $\mathcal{L}^+$ se calcula igual que la de $\mathcal{F}^+$ y sale:

```
   N C 0 1 2 3 4 5 6 7 8 9
N: 0 1 1 1 1 1 1 1 1 1 1
C: 0 0 1 1 1 1 1 1 1 1 1
0: 0 0 0 0 0 0 0 0 0 0 0
1: 0 0 0 0 0 0 0 0 0 0 0
2: 0 0 0 0 0 0 0 0 0 0 0
3: 0 0 0 0 0 0 0 0 0 0 0
4: 0 0 0 0 0 0 0 0 0 0 0
5: 0 0 0 0 0 0 0 0 0 0 0
6: 0 0 0 0 0 0 0 0 0 0 0
7: 0 0 0 0 0 0 0 0 0 0 0
8: 0 0 0 0 0 0 0 0 0 0 0
9: 0 0 0 0 0 0 0 0 0 0 0
```

Luego `last(N)` es igual a `{C,0,1,2,3,4,5,6,7,8,9}`.

4.4.4. Relaciones de precedencia

Dada una gramática limpia G de axioma S, se definen las siguientes relaciones de precedencia entre los símbolos del vocabulario $\Sigma = \Sigma_T \cup \Sigma_N$. Para todo U, V$\in \Sigma$:

U =. V $\Leftrightarrow \exists$ W::=xUVy $\in$ P
U <. V $\Leftrightarrow \exists$ W::=xUTy $\in$ P, T $\mathcal{F}^+$ V en P.
U >. V $\Leftrightarrow \exists$ W::=xTRy $\in$ P, T $\mathcal{L}^+$ U, R $\mathcal{F}^*$ V en P.
U <.= V $\Leftrightarrow$ R <. S ó R =. S
U >.= V $\Leftrightarrow$ R >. S ó R =. S

Teorema 4.4.2. *U =. V $\Leftrightarrow$ UV aparece en el asidero de alguna forma sentencial.*

Prueba:

- $\Rightarrow$

  U =. V $\Rightarrow \exists$ W::=xUVy $\in$ P
  G es limpia $\Rightarrow$ S $\rightarrow^*$ uWv $\Rightarrow$ existe un árbol que genera uWv.
  Se poda ese árbol hasta que U esté en un asidero (todo símbolo ha de ser alguna vez parte de un asidero).
  Se aplica la regla W::=xUVy. Como U era parte del asidero, el asidero de este árbol nuevo debe ser xUVy, q.e.d.

- $\Leftarrow$

  UV es parte de un asidero. Por definición de asidero existe una regla W::=xUVy $\Rightarrow$ U =. V, q.e.d.

Teorema 4.4.3. *U >. V $\Leftrightarrow$ existe una forma sentencial* xUVy *donde U es el símbolo final de un asidero.*

Prueba:

- $\Rightarrow$

  U >. V $\Rightarrow \exists$ W::= xTRy $\in$ P, T $\mathcal{L}^+$ U, R $\mathcal{F}^*$ V en P
  G es limpia $\Rightarrow$ S $\rightarrow^*$ uWv $\rightarrow+$ uxTRyv
  T $\mathcal{L}^+$ U $\Rightarrow$ T $\rightarrow+$ tU $\Rightarrow$ S $\rightarrow+$ uxtURyv

  Se dibuja este árbol y se reduce hasta que U sea parte del asidero. Por construcción, tendrá que ser su último símbolo.

  R $\mathcal{F}^*$ V $\Rightarrow$ R $\rightarrow^*$ Vw $\Rightarrow$ S $\rightarrow+$ uxtUVwyv

  Se añade al árbol anterior esta última derivación. El asidero no ha cambiado, U sigue siendo el último símbolo y va seguido por V, q.e.d.

- $\Leftarrow$

  U es la cola de un asidero y va seguido por V. Se reduce hasta que V esté en el asidero. Esto da un árbol para la forma sentencial xTVy, donde T $\rightarrow^*$ tU.

Si T y V están los dos en el asidero, existe W ::= uTVv ⇒ U >. V, q.e.d.

Si V es cabeza del asidero, reducimos hasta llegar a xTWw donde T está en el asidero. Ahora, W →+ V… y W tiene que estar en el asidero, pues, si no, T sería la cola y habría sido asidero antes que V. Luego U >. V, q.e.d.

Teorema 4.4.4. *U <. V ⇔ existe una forma sentencial* xUVy *donde V es el símbolo inicial de un asidero.*

Se demuestra de forma análoga.

4.4.5. Gramática de precedencia simple

Definición: *G es una Gramática de Precedencia Simple o Gramática de Precedencia (1,1) si:*

1. G cumple las condiciones especificadas al principio de la Sección 4.4.
2. Sólo existe, como mucho, una relación de precedencia entre dos símbolos cualesquiera del vocabulario.
3. No existen en G dos producciones con la misma parte derecha.

Se llama también Gramática de Precedencia (1,1) porque sólo se usa un símbolo a la izquierda y la derecha de un posible asidero para decidir si lo es.

Si no se cumplen las condiciones anteriores, a veces se puede manipular la gramática para intentar que se cumplan. La recursividad a izquierdas o a derechas suele dar problemas (salen dos relaciones de precedencia con los símbolos que van antes o después del recursivo). Para evitarlo, se puede estratificar la gramática, añadiendo símbolos nuevos, pero, si hay muchas reglas, esto es muy pesado. De todos modos, no siempre es posible obtener una gramática de precedencia simple equivalente a una dada, pues no todos los lenguajes independientes del contexto pueden representarse mediante gramáticas de precedencia simple.

Supondremos que toda forma sentencial queda encuadrada entre dos símbolos especiales de principio y fin de cadena, ⊢ y ⊣, que no están en Σ y tales que, para todo U∈Σ, ⊢+ <. U y U >. ⊣.

Teorema 4.4.5. *Una gramática de precedencia simple no es ambigua. Además, el asidero de una forma sentencial $U_1 \ldots U_n$ es la subcadena $U_i \ldots U_j$, situada más a la izquierda tal que*:

$U_{i-1} <. U_i =. U_{i+1} =. U_{i+2} =. \ldots =. U_{j-1} =. U_j >. U_{j+1}$

Prueba:

- Si $U_i \ldots U_j$ es asidero, se cumple la relación por los teoremas anteriores.
- Reducción al absurdo. Se cumple la relación y no es asidero. Entonces ninguna poda la hará asidero, pues si en algún momento posterior resultara ser la rama completa más a la izquierda, ya lo es ahora.

  Si no es asidero, cada uno de los símbolos de $U_{i-1} U_i \ldots U_j U_{j+1}$ debe aparecer más pronto o más tarde como parte de un asidero. Sea U_k el primero que aparece.

1. Si $U_k = U_{i-1}$, de los teoremas se sigue que $U_{i-1} =. U_i$ o $U_{i-1} >. U_i$, lo que contradice que $U_{i-1} <. U_i$ (no puede haber dos relaciones entre dos símbolos).
2. Si $U_k = U_{j+1}$, de los teoremas se sigue que $U_j =. U_{j+1}$ o $U_j <. U_{j+1}$, lo que contradice que $U_j >. U_{j+1}$.
3. Si $i-1<k<j+1$, ocurre lo siguiente:
   a. Que U_{i-1} no puede estar en el asidero (pues entonces $U_{i-1} =. U_i$, contradicción).
   b. Que U_{j+1} no puede estar en el asidero (pues entonces $U_j =. U_{j+1}$, contradicción).
   c. Que ningún símbolo U_p, $i<p<=k$ puede ser cabeza del asidero (pues entonces $U_{p-1} <. U_p$, contradicción).
   d. Que ningún símbolo U_p, $k<=p<j$ puede ser cola del asidero (pues entonces $U_p <. U_{p+1}$, contradicción).
   e. Luego la cabeza del asidero debe ser U_i y la cola U_j, lo que contradice que $U_i \ldots U_j$ no era el asidero, q.e.d.

Como el asidero es único (por construcción) y sólo puede ser parte derecha de una regla (por ser la Gramática de Precedencia Simple), sólo se puede aplicar una regla para reducir. Luego la gramática no es ambigua.

4.4.6. Construcción de las relaciones

- La relación $=.$ se construye por simple observación de las partes derechas de las reglas (véase la definición de $=.$).
- La relación $<.$ se construye fácilmente utilizando la representación matricial de las relaciones y teniendo en cuenta que:

  $$<. \Leftrightarrow =. \vee.\wedge \mathcal{F}^+$$

  donde $\vee.\wedge$ es el producto booleano de matrices. (Por definición del producto de relaciones y la definición de $<.$ y $=.$).
- La relación $>.$ se construye de manera parecida:

  $$>. \Leftrightarrow (\mathcal{L}^+)' \vee.\wedge =. \vee.\wedge \mathcal{F}^*$$

  donde $'$ es la transposición de matrices. La demostración queda como ejercicio.

4.4.7. Algoritmo de análisis

1. Se almacenan las producciones de la gramática en una tabla.
2. Se construye la matriz de precedencia MP de dimensiones N×N (N = cardinal o número de elementos de Σ), tal que

MP(i,j) = 0 si no existe relación entre U_i y U_j
= 1 si U_i <. U_j
= 2 si U_i =. U_j
= 3 si U_i >. U_j

3. Se inicializa una pila con el símbolo ⊢ y se añade el símbolo ⊣ al final de la cadena de entrada.
4. Se compara el símbolo situado en la cima de la pila con el siguiente símbolo de entrada.
5. Si no existe ninguna relación, error sintáctico: cadena rechazada (fin del algoritmo).
6. Si existe la relación <. o la relación =., se introduce el símbolo de entrada en la pila y se elimina de la entrada. Volver al paso 4.
7. Si la relación es >., el asidero termina en la cima de la pila.
8. Se recupera el asidero de la pila, sacando símbolos hasta que el símbolo en la cima de la pila esté en relación <. con el último sacado.
9. Se compara el asidero con las partes derechas de las reglas.
10. Si no coincide con ninguna, error sintáctico: cadena rechazada (fin del algoritmo).
11. Si coincide con una, se coloca la parte izquierda de la regla en el extremo izquierdo de la cadena que queda por analizar.
12. Si en la pila sólo queda ⊢ y la cadena de entrada ha quedado reducida al axioma seguido del símbolo ⊣, la cadena ha sido reconocida (fin del algoritmo). En caso contrario, volver al paso 4.

Ejemplo 4.13 Sea la gramática G = ({a,b,c}, {S}, S, {S ::= aSb | c}). Las matrices de las relaciones son:

```
=. :  0 0 1 0     F: 0 1 0 1     L: 0 0 1 1
      1 0 0 0        0 0 0 0        0 0 0 0
      0 0 0 0        0 0 0 0        0 0 0 0
      0 0 0 0        0 0 0 0        0 0 0 0
F+ = F
L+ = L
F* : 1 1 0 1
     0 1 0 0
     0 0 1 0
     0 0 0 1
```

Aplicando las expresiones de la Sección 4.4.3, se obtiene:

```
<. :  0 0 0 0   >. : 0 0 0 0
      0 1 0 1        0 0 0 0
      0 0 0 0        0 0 1 0
      0 0 0 0        0 0 1 0
```

Con lo que la matriz de precedencia queda:

```
   S  a  b  c  ┤
S     =.    >.
a =. <.    <.  >.
b       >.    >.
c       >.    >.
├ <. <. <. <.
```

Se analizará ahora la cadena `aacbb`:

Pila	Relación	Entrada	Asidero	Regla a aplicar
├	<.	aacbb┤		
├<.a	<.	acbb┤		
├<.a<.a	<.	cbb┤		
├<.a<.a.<.c	>.	bb┤	c	S::=c
├<.a<.a	=.	Sbb┤		
├<.a<.a=.S	=.	bb┤		
├<.a<.a=.S=.b	>.	b┤	aSb	S::=aSb
├<.a	=.	Sb┤		
├<.a=.S	=.	b┤		
├<.a=.S=.b	>.	┤	aSb	S::=aSb
├	<.	S┤		

Luego la cadena es aceptada.

Se analizará ahora la cadena `aabb`:

Pila	Relación	Entrada	Asidero	Regla a aplicar
├	<.	aabb┤		
├<.a	<.	abb┤		
├<.a<.a	No hay	bb┤		

Luego la cadena es rechazada.

Se analizará ahora la cadena acbb:

Pila	Relación	Entrada	Asidero	Regla a aplicar
├	<.	acbb┤		
├<.a	<.	cbb┤		
├<.a<.c	>.	bb┤	c	S::=c
├<.a	=.	Sbb┤		
├<.a=.S	=. bb┤			
├<.a=.S=.b	>.	b┤	aSb	S::=aSb
├	<.	Sb┤		
├<.S	=.	b┤		
├<.S=.b	>.	┤	Sb	No hay

Luego la cadena es rechazada.

4.4.8. Funciones de precedencia

La matriz de precedencias ocupa N×N posiciones de memoria. A veces es posible construir dos funciones de precedencia que ocupen sólo 2×N. Esta operación se llama linealización de la matriz. Dichas funciones, de existir, no son únicas (hay infinitas).

Para el ejemplo anterior valen las dos funciones siguientes:

```
   ├  S  a  b  c ┤
f  0  2  2  3  3
g     2  3  2  3  0
```

Si es posible construir f y g, se verifica que

- f(U)=g(V) ⇒ U =. V
- f(U)<g(V) ⇒ U <. V
- f(U)>g(V) ⇒ U >. V

Existen matrices que no se pueden linealizar. Ejemplo:

```
  A B
A = >
B = =
```

En este caso, debería cumplirse que:

$f(A) = g(A)$
$f(A) > g(B)$
$f(B) = g(A)$
$f(B) = g(B)$

Es decir:

$$f(A) > g(B) = f(B) = g(A) = f(A) \Rightarrow f(A) > f(A)$$

Con lo que se llega a una contradicción. Cuando no hay inconsistencias, el siguiente algoritmo construye las funciones de precedencia:

- Se dibuja un grafo dirigido con $2 \times N$ nodos, llamados $f_1, f_2, \ldots, f_N, g_1, g_2, \ldots, g_N$, con un arco de f_i a g_j si $U_i >.= U_j$ y un arco de g_j a f_i si $U_i <.= U_j$.
- A cada nodo se le asigna un número igual al número total de nodos accesibles desde él (incluido él mismo). El número asignado a f_i se toma como valor de $f(U_i)$ y el asignado a g_i es $g(U_i)$.

Demostración:

- Si $U_i =. U_j$, hay una rama de f_i a g_j y viceversa, luego cualquier nodo accesible desde f_i es accesible desde g_j y viceversa. Luego $f(U_i) = g(U_j)$.
- Si $U_i >. U_j$, hay una rama de f_i a g_j. Luego cualquier nodo accesible desde g_j es accesible desde f_i. Luego $f(U_i) >= g(U_j)$.

  Si $f(U_i) = g(U_j)$, existe un camino cerrado $f_i \rightarrow g_j \rightarrow f_k \rightarrow g_l \rightarrow \ldots \rightarrow g_m \rightarrow f_i$, lo que implica que $U_i >. U_j$, $U_k <.= U_j$, $U_k >.= U_l$, $\ldots$ $U_i <.= U_m$ y reordenando: $U_i >. U_j >.= U_k >.= U_l >.= \ldots >.= U_m >.= U_i$, es decir: $f(U_i) > f(U_i)$, con lo que se llega a una contradicción. Luego el caso de igualdad de valores de las funciones queda excluido y se deduce que $f(U_i) > g(U_j)$.

- Si $U_i <. U_j$, la demostración es equivalente.

La Figura 4.64 describe la aplicación de este algoritmo al ejemplo anterior, y explica cómo se obtuvieron las funciones mencionadas al principio de esta sección.

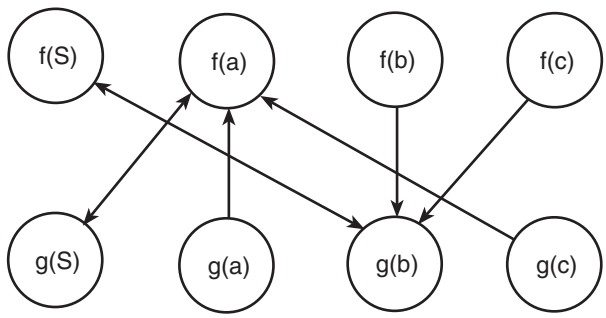


Figura 4.64. Grafo utilizado para la construcción de las funciones de precedencia.

El algoritmo descrito puede mecanizarse. El grafo descrito equivale a la relación *existe un arco del nodo x al nodo y*. Dicha relación posee su matriz Booleana correspondiente, de dimensiones 2N×2N, que llamaremos B, y que puede representarse así:

```
(0)      (>.=)
(<.=)'   (0)
```

A partir de B, construimos B*. Entonces se verifica que $f(U_i)$ es igual al número de unos en la fila i, mientras $g(U_i)$ es igual al número de unos en la fila N+i.

En el ejemplo anterior, B resulta ser la matriz:

```
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
```

A partir de B, obtenemos B*, que resulta ser la matriz:

```
1 0 0 0 0 0 1 0
0 1 0 0 1 0 0 0
1 0 1 0 0 0 1 0
1 0 0 1 0 0 1 0
0 1 0 0 1 0 0 0
0 1 0 0 1 1 0 0
1 0 0 0 0 0 1 0
0 1 0 0 1 0 0 1
```

Con lo que las dos funciones f y g resultan ser las indicadas al principio de esta sección. Para completarlas, basta añadir los símbolos de principio y fin de cadena, a los que se asigna el valor 0. A continuación se muestran de nuevo los resultados obtenidos. Obsérvese que, si se suma cualquier número entero a todos los valores de f y g, se obtienen dos nuevas funciones que también cumplen todas las condiciones. Por eso, si existe un par de funciones f y g, tendremos infinitas, todas equivalentes.

```
   ├ S  a  b  c ┤
f  0  2  2  3  3
g     2  3  2  3  0
```

El uso de las funciones supone cierta pérdida de información respecto al uso de la matriz, pues desaparecen los lugares vacíos en la tabla de precedencias, que conducían directamente a la detección de algunos casos de error. Sin embargo, estos casos serán detectados más tarde, de otra manera. Para comprobarlo, analicemos por medio de las funciones la cadena aabb y veremos que en este caso basta con un paso más para detectar la condición de error.

Pila	f(x)	g(x)	Relación	Entrada	Asidero	Regla a aplicar
├	0	3	<.	aabb┤		
├<.a	2	3	<.	abb┤		
├<.a<.a	2	2	=.	bb┤		
├<.a<.a=.b	3	2	>.	b┤	ab	No hay

4.5 Resumen

En este capítulo se describen algunos de los métodos que suelen utilizarse para construir los analizadores sintácticos de los lenguajes independientes del contexto. En una primera sección del capítulo se describen los conjuntos primero y siguiente asociados a una gramática, pues son necesarios para describir los métodos que aparecen en el resto del capítulo. Estos métodos de análisis pueden clasificarse en dos categorías: análisis descendente y análisis ascendente. Dentro del análisis descendente se describe, en primer lugar, el análisis descendente con vuelta atrás cuya ineficiencia se soluciona con las gramáticas LL(1) y el método de análisis descendente selectivo.

Las operaciones fundamentales de los algoritmos de análisis ascendente son el desplazamiento de los símbolos de entrada necesarios para reconocer los asideros y la reducción de los mismos. Estas técnicas se basan en la implementación del autómata a pila para la gramática independiente del contexto del lenguaje considerado y éste, a su vez, en el autómata finito que reconoce los asideros del análisis. Hay diferentes maneras de construir este autómata finito. Los analizadores estudiados en orden creciente de potencia, son LR(0), SLR(1), LR(1) y LALR(1). Su principal diferencia consiste en que, para reducir una regla, comprueban condiciones más estrictas respecto a los próximos símbolos que el análisis encontrará en la entrada: SLR(1) tiene en cuenta el símbolo siguiente y LR(1) y LALR(1) consideran de forma explícita un símbolo de adelanto. Los algoritmos LR y LALR se podrían generalizar para valores de k>1 pero la complejidad que implica gestionar más de un símbolo de adelanto desaconseja su uso.

Otro método de análisis ascendente, que no se utiliza mucho en compiladores completos, pero sí en analizadores de expresiones, hojas de cálculo, bases de datos, etc., utiliza gramáticas de precedencia simple. El método se basa en tres relaciones, llamadas de precedencia, que permiten localizar los asideros del análisis con un algoritmo muy sencillo y eficiente. Estas relaciones pueden construirse fácilmente, por simple observación de las reglas de producción, o de forma automática, mediante operaciones realizadas sobre matrices booleanas. Por último, es posible mejorar el algoritmo sustituyendo las matrices por dos funciones de precedencia.

4.6 Ejercicios

1. Considérese el lenguaje de los átomos en lenguaje Prolog. Todos ellos tienen un identificador, y pueden tener cero, uno o varios argumentos. Cuando el número de argumentos es ma-

yor o igual que 1, éstos aparecen entre paréntesis, y separados por comas; en caso contrario, no se escriben los paréntesis. Finalmente, cada uno de los argumentos de un átomo puede ser otro átomo, un número o una variable (que comienza con letra mayúscula). Supondremos, además, que al final de cada palabra del lenguaje hay un punto. Por ejemplo, las siguientes expresiones serían válidas:

```
atomo.
pepe(a(b,c)).
paco(0,1,2,X).
```

Supondremos que el analizador morfológico ya ha identificado los identificadores, los números y las variables, por lo que no es necesario incluir las reglas para reconocer éstos en la gramática del lenguaje. Proporcionar una gramática LL(1) para este lenguaje. Construir la matriz de análisis LL(1) para la gramática proporcionada.

2. Con la matriz de análisis LL(1) del ejercicio anterior, analizar las siguientes cadenas:

```
atomo.
pepe(a(b,c)).
paco(variable(0)).
```

3. Sea el lenguaje $\{a^n b^{m+n} a^m \mid m,n>=0\}$. Construir una gramática LL(1) que lo reconozca. Construir el analizador sintáctico correspondiente. Analizar las siguientes palabras: aabbba, aaabb.

4. Dado el lenguaje L = {w | w tiene un número par de ceros y unos}, construir una gramática LL(1) que lo describa. Construir el analizador sintáctico correspondiente.

5. Construir una gramática independiente del contexto que describa el lenguaje $\{a^n b^p c^{m+p} d^{n+m} \mid m,n,p>0\}$. Construir una gramática LL(1) que describa el mismo lenguaje.

6. Sea el lenguaje formado por todas las palabras con las letras a y b, con doble número de a que de b, pero con todas las b seguidas, situadas en un extremo de la cadena. Construir una gramática LL(1) que lo reconozca.

7. Sea el lenguaje $\{ab(ab)^n ac(ac)^n \mid n>=0\}$. Construir una gramática LL(1) que lo reconozca y el analizador sintáctico correspondiente. Analizar las siguientes palabras: ababacac, abacac.

8. Dada la gramática

```
A ::= B a | a
B ::= A b | b
C ::= A c | c
```

Construir una gramática LL(1) equivalente y un analizador sintáctico descendente que analice el lenguaje correspondiente. Analizar las siguientes palabras: abab, baba, ababa, babab.

9. Sea el lenguaje formado por todas las palabras no vacías con las letras a y b, con el mismo número de a que de b, pero con todas las b seguidas, sin ninguna restricción para las a. Construir una gramática LL(1) que lo reconozca. Construir un analizador sintáctico descendente que la analice. Analizar las siguientes palabras: `bbaa, baba`.

10. Sea el siguiente lenguaje sobre el alfabeto `{a,b}`: $(aa^*+\lambda)b+bb^*$. Construir una gramática LL(1) que reconozca el mismo lenguaje. Escribir el analizador sintáctico descendente correspondiente. Utilizando el analizador anterior, analizar las siguientes cadenas: `aaab, aabb`.

11. Sea el siguiente lenguaje sobre el alfabeto `{a,b}`: $\{a^m\ b\ c^m\ a^n\ b\ c^n \mid m,n>0\}\ U\ \{b^p \mid p>0\}$. Construir una gramática LL(1) que reconozca el mismo lenguaje. Escribir el analizador sintáctico *top-down* correspondiente. Utilizando el analizador anterior, analizar las siguientes cadenas: `abbc, bbb`.

12. Construir una gramática que reconozca el lenguaje $\{a^n\ b^m \mid 0<=n<m\}$. Convertir la gramática anterior a la forma normal de Greibach. Convertir la gramática anterior a la forma LL(1). Escribir un analizador sintáctico descendente que analice el lenguaje anterior.

13. Sea el siguiente lenguaje sobre el alfabeto `{a,b}`: $\{\ a^n\ b^n \mid n\geq 0\ \}\ U\ \{\ a\ \}$. Construir una gramática LL(1) que reconozca el mismo lenguaje. Escribir el analizador sintáctico *top-down* correspondiente. Utilizando el analizador anterior, analizar las siguientes cadenas: `a, b, ab, aab, aabb`.

14. Construir una gramática LL(1) que reconozca el lenguaje $\{a^n\ b^m\ c^{n+m} \mid n,m>=0\}$. Escribir un analizador sintáctico descendente que analice el lenguaje anterior.

15. Construir una gramática LL(1) para el lenguaje $\{a^n\ b^m\ c^n \mid n,m>=0\}$ y programar el analizador sintáctico correspondiente.

16. Utilizar la tabla de análisis de la Figura 4.30 para realizar el análisis sintáctico de la cadena `id*id+id`. ¿Es sintácticamente correcta la cadena?

17. Utilizar la tabla de análisis de la Figura 4.37 para realizar el análisis sintáctico de la cadena `i+(i+i)`. ¿Es sintácticamente correcta la cadena?

18. Utilizar la tabla de análisis de la Figura 4.37 para realizar el análisis sintáctico de la cadena `(i+i`. ¿Es sintácticamente correcta la cadena?

19. Utilizar la tabla de análisis SLR(1) de la Figura 4.47 para realizar el análisis sintáctico de la siguiente cadena y determinar si es sintácticamente correcta o no.

```
begin
     dec;
     ejec;
     ejec
end
```

20. Utilizar la tabla de análisis SLR(1) de la Figura 4.47 para realizar el análisis sintáctico de la siguiente cadena y determinar si es sintácticamente correcta o no.

```
begin
     dec;
     ejec;
end
```

21. Dada la gramática independiente del contexto que se puede deducir de las siguientes reglas de producción en las que el axioma es el símbolo `E`:

```
(1)E::=E+E
(2)E::=E*E
(3)E::=i
```

Construir el diagrama de estados del analizador SLR(1) y la tabla de análisis para determinar si la gramática es o no SLR(1). Obsérvese que esta gramática es ambigua ya que no establece prioridad entre las operaciones aritméticas. Analizar el efecto de la ambigüedad en los posibles conflictos de la gramática y el significado que aporta a la ambigüedad solucionarlos mediante la selección de una de las operaciones en conflicto. Comprobar los resultados en el análisis de la cadena `i*i+i*i+i`.

22. Utilizar la tabla de análisis LR(1) de la Figura 4.56 para realizar el análisis sintáctico de la cadena `aaxbb` y determinar si es sintácticamente correcta o no.

23. Utilizar la tabla de análisis LR(1) de la Figura 4.56 para realizar el análisis sintáctico de la cadena `ax` y determinar si es sintácticamente correcta o no.

24. Repetir el ejercicio anterior con la tabla LALR(1) de la Figura 4.62.

25. Construir la tabla de análisis sintáctico ascendente para la siguiente gramática que corresponde a al subconjunto (dedicado a la declaración de variables) de la gramática de un lenguaje de programación imaginario.

```
<declaration>  ::= <mode> <idlist>
<mode> ::= bool
<mode> ::= int
<idlist> ::= <id>
<idlist> ::= <id> , <idlist>
```

Obsérvese que en la regla 1 aparece un espacio en blanco entre los símbolos no terminales `<mode>` e `<idlist>`. Considerar el símbolo `<declaration>` como el axioma.

Contestar razonadamente a las siguientes preguntas

- La gramática del apartado anterior ¿es una gramática LR(0)? ¿Por qué?
- La gramática del apartado anterior ¿es una gramática SLR(1)? ¿Por qué?
- Utilizando la tabla de análisis desarrollada en el apartado (1), analizar la siguiente declaración:

```
int x,y
```

26. La tabla de análisis sintáctico SLR(1) para la siguiente gramática está incompleta. Considerar que `S` es el axioma.

```
S ::= id (L)
S ::= id
L ::= λ
L ::= S Q
Q ::= λ
Q ::=, S Q
```

	Acción					Ir a		
	id	(	)	,	$	S	L	Q
0	**d2**					**1**		
1					**acc**			
2								
3	**d2**					**5**	**4**	
4			**d6**					
5				**d8**				**7**
6								
7								
8	**d2**					**9**		
9								**10**
10								

26.1. Completar las casillas sombreadas detallando los cálculos realizados al efecto.

26.2. Rellenar las casillas que correspondan a operaciones de reducción detallando los cálculos realizados al efecto.

27. Realizar el análisis de la cadena `var int inst` utilizando la tabla de análisis SLR(1) que se proporciona y que corresponde a la gramática

```
S ::= S inst
S ::= S var D
S ::= λ
D ::= D ident E
D ::= D ident sep
D ::= int
D ::= float
E ::= S fproc
```

	Acción							Ir a			
	inst	var	ident	sep	int	float	fproc	$	S	D	E
0	r3	r3					r3	r3	1		
1	d2	d3						acc			
2							r1	r1			
3					d5	d6					
4	r2	r2	d7				r2	r3			
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	d2					d11				
10	r5	r5	r5				r5	r5			
11	r8	r8	r8				r8	r8			

28. Dada la siguiente gramática (considerar que E es el axioma)

```
E ::= (L)
E ::= a
L ::= L,E
L ::= E
```

¿Cuál sería el resultado de aplicar la operación de clausura o cierre al estado formado por el elemento LR(1) E::=(•L) {$} (o en una notación equivalente, el elemento LR(1) (1,1,$))? Contestar razonadamente.

29. Sea el lenguaje sobre el alfabeto {a,b} formado por todas las palabras que empiezan por a y acaban por b. Construir una gramática SLR(1) que reconozca el mismo lenguaje y su tabla de análisis.

30. Construir una gramática SLR(1), con la cadena vacía en alguna parte derecha, que reconozca el lenguaje $\{a^n b^m \mid 0 \leq n < m\}$. Construir también la tabla de análisis y utilizarla en el análisis de las cadenas abb, aab. La gramática anterior ¿es LR(0)? ¿Por qué?

31. Construir una gramática SLR(1) que describa el lenguaje $\{a^n b^{n+p+q} a^p c^q \mid n,p,q >= 0\}$. Construir la tabla del análisis correspondiente. Analizar las cadenas abbac, abbbac y abbbbac.

32. Dado el lenguaje L = { w | w tiene un número par de ceros y unos }, construir una gramática SLR(1) que lo describa. Construir la tabla del análisis correspondiente.

33. Construir una gramática SLR(1) y la tabla de análisis para el lenguaje $\{a^n c^m b^n \mid m>0, n>=0\}$.

34. Sea el siguiente lenguaje sobre el alfabeto {a,b}: $\{a^n b^n \mid n>=0\} \cup \{a\}$.

34.1. Construir una gramática SLR(1) que reconozca el mismo lenguaje.
34.2. Construir la tabla de análisis.
34.3. Utilizando el analizador anterior, analizar las siguientes cadenas: a, b, ab, aab, aabb.

35. Sea el siguiente lenguaje sobre el alfabeto $\{a,b\} : \{a^m bc^m a^n bc^n \mid m,n>0\} \cup \{b^p \mid p>0\}$.

35.1. Construir una gramática SLR(1), con la cadena vacía en alguna parte derecha, que reconozca el mismo lenguaje.
35.2. Construir la tabla de análisis.
35.3. Utilizando la tabla anterior, analizar las siguientes cadenas: abcabc, abbc.

36. Sea el siguiente lenguaje sobre el alfabeto {a,b} representado mediante su expresión regular (aa*+λ)b+bb*, donde λ es la palabra vacía.

36.1. Construir una gramática SLR(1), con la palabra vacía en alguna parte derecha, que reconozca el mismo lenguaje.
36.2. Construir la tabla de análisis.
36.3. Utilizando la tabla anterior, analizar las siguientes cadenas: aaab, aabb.

37. Sea el lenguaje del Ejercicio 4.29.

37.1. Construir una gramática de precedencia simple que lo reconozca.
37.2. Construir la matriz de relaciones de precedencia.

38. Para el lenguaje del Ejercicio 4.30.

38.1. Construir una gramática sin la cadena vacía que lo reconozca.
38.2. Construir la matriz de relaciones de precedencia. Explicar por qué no es de precedencia simple.

39. En el lenguaje Smalltalk, los operadores binarios (+,–,*,/) no tienen precedencia intrínseca, sino posicional: el operador situado más a la izquierda se ejecuta primero, salvo por la presencia de paréntesis, que modifican la precedencia de la manera habitual. Los operandos básicos pueden ser identificadores o constantes numéricas:

39.1. Construir una gramática que represente el lenguaje de las expresiones binarias en Smalltalk.
39.2. ¿Es de precedencia simple esta gramática?

40. Construir una gramática de precedencia simple y una matriz de precedencia para el lenguaje del Ejercicio 4.33.

41. Sea el lenguaje del Ejercicio 4.34.

41.1. Construir una gramática de precedencia simple que lo reconozca.
41.2. Construir la matriz de relaciones de precedencia.

41.3. Utilizando la matriz anterior y el algoritmo estándar, analizar las siguientes cadenas: `a, b, ab, aab, aabb`.

42. Sea el lenguaje del Ejercicio 4.35.

42.1. Construir una gramática de precedencia simple, sin la cadena vacía en ninguna parte derecha, que lo reconozca.
42.2. Construir la matriz de relaciones de precedencia.
42.3. Utilizando la matriz anterior y el algoritmo estándar, analizar las siguientes cadenas: `abcabc, abbcb, abbc`.

43. Sea el lenguaje del Ejercicio 4.36.

43.1. Construir una gramática de precedencia simple, sin la cadena vacía en ninguna parte derecha, que lo reconozca.
43.2. Construir la matriz de relaciones de precedencia.
43.3. Utilizando la matriz anterior y el algoritmo estándar, analizar las siguientes cadenas: `aaab, aabb`.

44. Encontrar una gramática cuyo lenguaje sea el conjunto de los números enteros pares.

45. En la gramática anterior, calcular las relaciones $\mathcal{F}$, $\mathcal{L}$, $\mathcal{F}^+$, $\mathcal{L}^+$.

46. En la gramática anterior, construir los conjuntos `first(S)`, `last(S)`, donde `S` es el axioma.

47. Demostrar que R^+ es transitiva, cualquiera que sea `R`.

Capítulo 5

Análisis semántico

5.1 Introducción al análisis semántico

5.1.1. Introducción a la semántica de los lenguajes de programación de alto nivel

El análisis semántico es la fase del compilador en la que se comprueba la corrección semántica del programa.

En la Sección 1.13.4 se reflexionó acerca de la distinción entre la sintaxis y la semántica de los lenguajes de programación de alto nivel. También se explicó que las gramáticas del tipo 0 de Chomsky, el único tipo de gramática que tiene la expresividad necesaria para representar todos los aspectos de estos lenguajes, presenta demasiadas dificultades para su diseño y gestión. De esta forma se justifica que, en el tratamiento de los lenguajes de programación, se distingan las construcciones sintácticas (usualmente independientes del contexto) de las semánticas (usualmente dependientes).

En el Capítulo 4 se han tratado con detalle los algoritmos necesarios para el análisis sintáctico, que normalmente aborda los aspectos independientes del contexto. Se ha podido comprobar que todos ellos son relativamente simples. El objetivo de este capítulo es incorporar la semántica al análisis del programa que se está compilando.

Sería deseable disponer de una herramienta parecida a las gramáticas independientes del contexto, a la que se pudiera incorporar de forma sencilla la comprobación de las condiciones semánticas. Si se dispusiera de ella, el analizador semántico se reduciría a una extensión de los algoritmos de análisis sintáctico, para incorporar la gestión de los aspectos semánticos.

En este capítulo se verá que las gramáticas de atributos proporcionan una herramienta muy adecuada para el análisis semántico, se explicará cómo pueden solucionar los problemas asociados con la semántica de los programas compilados y se describirán algunas aplicaciones existentes, que reciben como entrada gramáticas de atributos y generan de forma automática analizadores semánticos.

5.1.2. Objetivos del analizador semántico

El analizador semántico es la parte del compilador que realiza el análisis semántico. Suele estar compuesto por un conjunto de subrutinas independientes, que pueden ser invocadas por los analizadores morfológico y sintáctico.

Se puede considerar que el analizador semántico recibe, como entrada, el árbol del análisis del programa, una vez realizado el análisis morfológico y sintáctico. Esta distinción es más bien conceptual, ya que, en los compiladores reales, a menudo estas fases se entremezclan. Suele describirse el análisis semántico como un proceso mediante el cual se añade al árbol de derivación una serie de anotaciones, que permiten determinar la corrección semántica del programa y preparar la generación de código. Por lo tanto, la salida que genera el análisis semántico, en el caso de que no haya detectado errores, es un árbol de derivación con anotaciones semánticas. Dichas anotaciones se pueden usar para comprobar que el programa es semánticamente correcto, de acuerdo con las especificaciones del lenguaje de programación. Hay que comprobar, por ejemplo, que:

- Cuando se utiliza un identificador, éste ha sido declarado previamente.
- Se ha asignado valor a las variables antes de su uso.
- Los índices para acceder a los *arrays* están dentro del rango válido.
- En las expresiones aritméticas, los operandos respetan las reglas sobre los tipos de datos permitidos por los operadores.
- Cuando se invoca un procedimiento, éste ha sido declarado adecuadamente. Además, el número, tipo y posición de cada uno de sus argumentos debe ser compatible con la declaración.
- Las funciones contienen al menos una instrucción en la que se devuelve su valor al programa que las invocó.

Ejemplo 5.1 Se está compilando el siguiente programa, escrito en un lenguaje de programación ficticio, cuya sintaxis resultará fácil de comprender para cualquier programador:

```
begin
      int A;
      A := 100;
      A := A + A;
      output A
end
```

Resulta claro que el programa declara una variable de tipo entero y nombre `A`, le asigna inicialmente el valor 100 y posteriormente el de la suma de `A` consigo misma; finalmente, se escri-

be en algún medio externo el último valor de A. La Figura 5.1 resume la acción del análisis semántico en relación con este programa.

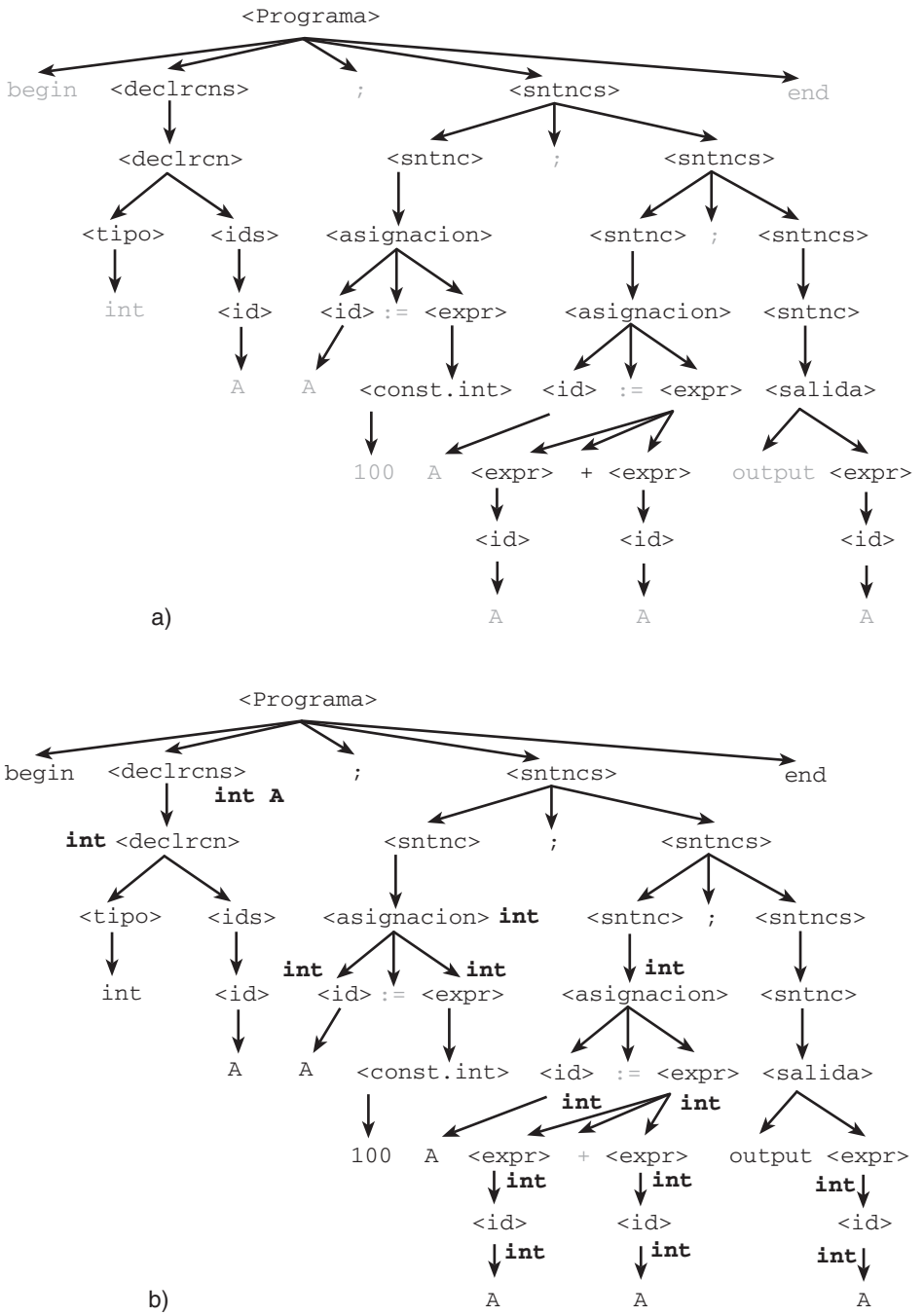


Figura 5.1. Ejemplo de un posible resultado del análisis semántico. a) Entrada del analizador semántico: el árbol de derivación. b) Salida: el *árbol anotado*.

La parte a) de la Figura 5.1 muestra el árbol de derivación del programa, de acuerdo con una gramática que no hace falta especificar. Este árbol sería la entrada que recibe el analizador semántico. La parte b) muestra el árbol, tras añadirle la siguiente información:

- En el símbolo no terminal `<dclrcn>`, asociado con la declaración de la variable `A`, se ha añadido el tipo de ésta: `int`.
- Al símbolo no terminal `<dclrcns>`, se le ha añadido la lista de identificadores declarados con sus tipos: `int A`. Esta lista podría utilizarse para añadir en la tabla de símbolos la información correspondiente.
- En la primera aparición del símbolo no terminal `<id>` como primer hijo del símbolo `<asignacion>`, se ha anotado que el tipo del identificador `A`, que se conoce desde su declaración, es `int`.
- En el símbolo no terminal `<expr>` que aparece como hermano del recién analizado `<id>`, se ha anotado que el tipo de la expresión es también entero, ya que la constante `100`, que es el fragmento de la entrada derivado de `<expr>`, es un número entero.
- Las anotaciones de los dos últimos puntos pueden servir para comprobar que la asignación es correcta, ya que los tipos del identificador y de la expresión son compatibles.
- En el nodo del símbolo `<asignacion>`, padre del subárbol estudiado, puede anotarse que se ha realizado una asignación correcta de valores de tipo entero.
- El subárbol cuya raíz es la última aparición de `<asignacion>` presenta un caso análogo al anteriormente descrito: las apariciones del identificador `A` en la parte derecha de la asignación obligan a consultar el tipo con que fue declarado. Se trata, por lo tanto, de asignar una expresión de tipo entero a un identificador de tipo entero. Las anotaciones de este subárbol permiten realizar todas las comprobaciones necesarias.
- El subárbol correspondiente a la última aparición de `<expr>` en la instrucción que imprime el valor de la variable `A` contiene anotaciones con el tipo de la variable y el de la expresión.

5.1.3. Análisis semántico y generación de código

En función del procedimiento utilizado para generar el programa objeto, se distinguen los siguientes tipos de compiladores (*véase* la Figura 5.2):

- **Compiladores de un solo paso:** integran la generación de código con el análisis semántico. Estos compiladores generan directamente el código a partir del árbol de la derivación. En este caso, las llamadas a las rutinas que escriben el código ensamblador suelen entremezclarse con el análisis semántico.
- **Compiladores de dos o más pasos:** en el primer paso de la compilación, el analizador semántico genera un código abstracto denominado *código intermedio*. En un segundo paso se realiza la generación del código definitivo a partir del código intermedio. A veces se separa también la optimización de código, la cual se realiza en un tercer paso independiente.

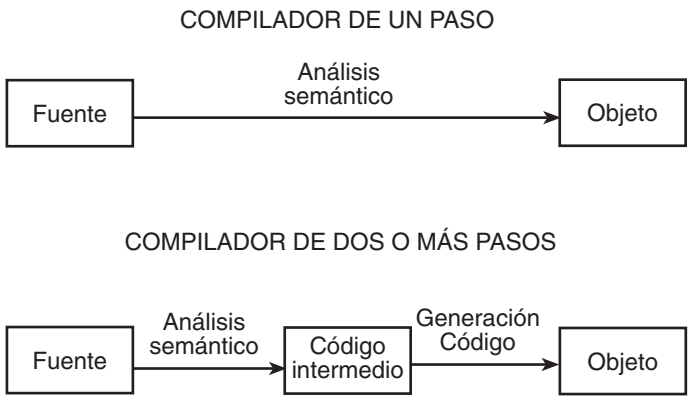


Figura 5.2. Esquema reducido de la compilación en uno y dos pasos.

Las representaciones intermedias facilitan la optimización de código. En este libro se van a describir dos tipos de representaciones intermedias: la *notación sufija*, que se utiliza especialmente para las expresiones aritméticas, y la que utiliza *tuplas* o *vectores* (usualmente *cuádruplas*) para representar las instrucciones que deben ser ejecutadas.

La notación sufija intenta sacar provecho de que la mayoría de los procesadores disponen de una pila para almacenar datos auxiliares, y de que la evaluación de expresiones con esta notación puede realizarse con facilidad mediante el uso de una pila auxiliar.

Para la representación intermedia que utiliza tuplas, se abstraen primero las operaciones disponibles en un lenguaje ensamblador hipotético, lo suficientemente genérico para poder representar cualquier ensamblador real. El objetivo de esa abstracción es decidir el número de componentes de las tuplas y la estructura de la información que contienen. Por ejemplo, es frecuente considerar que la primera posición sea ocupada por la operación que se va a realizar, las dos siguientes por sus operandos y la cuarta y última por el resultado. La cercanía de esta representación a los lenguajes simbólicos (procesados por ensambladores) facilita la generación del código. La abstracción introducida por las tuplas independiza esta representación de los detalles correspondientes a una máquina concreta, lo que ofrece ventajas respecto a su portabilidad.

Cuando se utilizan representaciones intermedias, la generación de código se reduce a un nuevo problema de traducción (de la representación intermedia al lenguaje objeto final), con la ventaja de que las representaciones intermedias son mucho más fáciles de traducir que los lenguajes de programación de alto nivel.

Las representaciones intermedias podrían considerarse parte del análisis semántico, ya que proporcionan formalismos para la representación de su resultado. Sin embargo, en este libro se ha decidido describirlas con detalle en el capítulo dedicado a la generación de código. Por un lado, las técnicas y algoritmos necesarios para generar tuplas son análogos a los necesarios para generar código simbólico o en lenguaje de la máquina, lo que aconseja que ambos tipos de generadores de código sean descritos en el mismo capítulo. Para simplifi-

car la exposición, también se incluirá en el capítulo de generación de código la otra representación intermedia: la notación sufija.

En general, los compiladores de un solo paso suelen ser más rápidos, pero más complejos, por lo que existen muchos compiladores comerciales construidos en dos y tres pasos.

5.1.4. Análisis semántico en compiladores de un solo paso

En los compiladores de un paso, no se utilizan representaciones intermedias, ya que la generación del código objeto se entremezcla con el análisis semántico. En estos compiladores resulta más complicado utilizar técnicas de optimización de código y de gestión de memoria. La Figura 5.3 muestra un esquema de esta situación.

En la parte izquierda de la figura se ven las dos primeras fases de la compilación, que han sido explicadas en los capítulos anteriores. En la parte inferior se muestra la cadena de unidades sintácticas correspondientes al programa del Ejemplo 5.1:

```
(<palabra clave>,begin)
    (<tipo>,int)(<id>,A)(<simb>,;)
    (<id>,A)(<simbm>,:=)(<cons int>,100)(<simb>,;)
    (<id>,A)(<simbm>,:=)(<id>,A)(<simb>,+)(<id>,A)(<simb>,;)
    (<palabra clave>,output)(<id>,A)
(palabra clave>,end)
```

Se ha utilizado una representación de pares para cada unidad sintáctica, en los que el primer elemento representa el tipo y el segundo el texto concreto que corresponde en el programa a esa unidad. En este ejemplo se utilizan los siguientes nombres de unidades sintácticas:

- `<palabra clave>`, para las palabras claves del lenguaje.
- `<tipo>`, para las palabras que representan tipos en las declaraciones de variables y procedimientos.
- `<id>`, para los identificadores de las variables y los procedimientos.
- `<simb>`, para caracteres especiales.
- `<simbm>`, para palabras formadas por más de un carácter especial.
- `<cons int>`, para números enteros.

En el árbol de la parte izquierda de la Figura 5.3 aparece el resultado del análisis sintáctico: el árbol de derivación de la parte a). A veces es necesario modificar la tabla de símbolos durante los análisis morfológico y sintáctico. La parte superior muestra esa posibilidad.

La parte derecha de la Figura 5.3 contiene el resultado del análisis semántico. En la mitad superior está el árbol de derivación con anotaciones semánticas; en la inferior, un posible código simbólico equivalente al programa de partida. Como el compilador es de un solo paso, el código debe generarse mientras se realiza el análisis semántico.

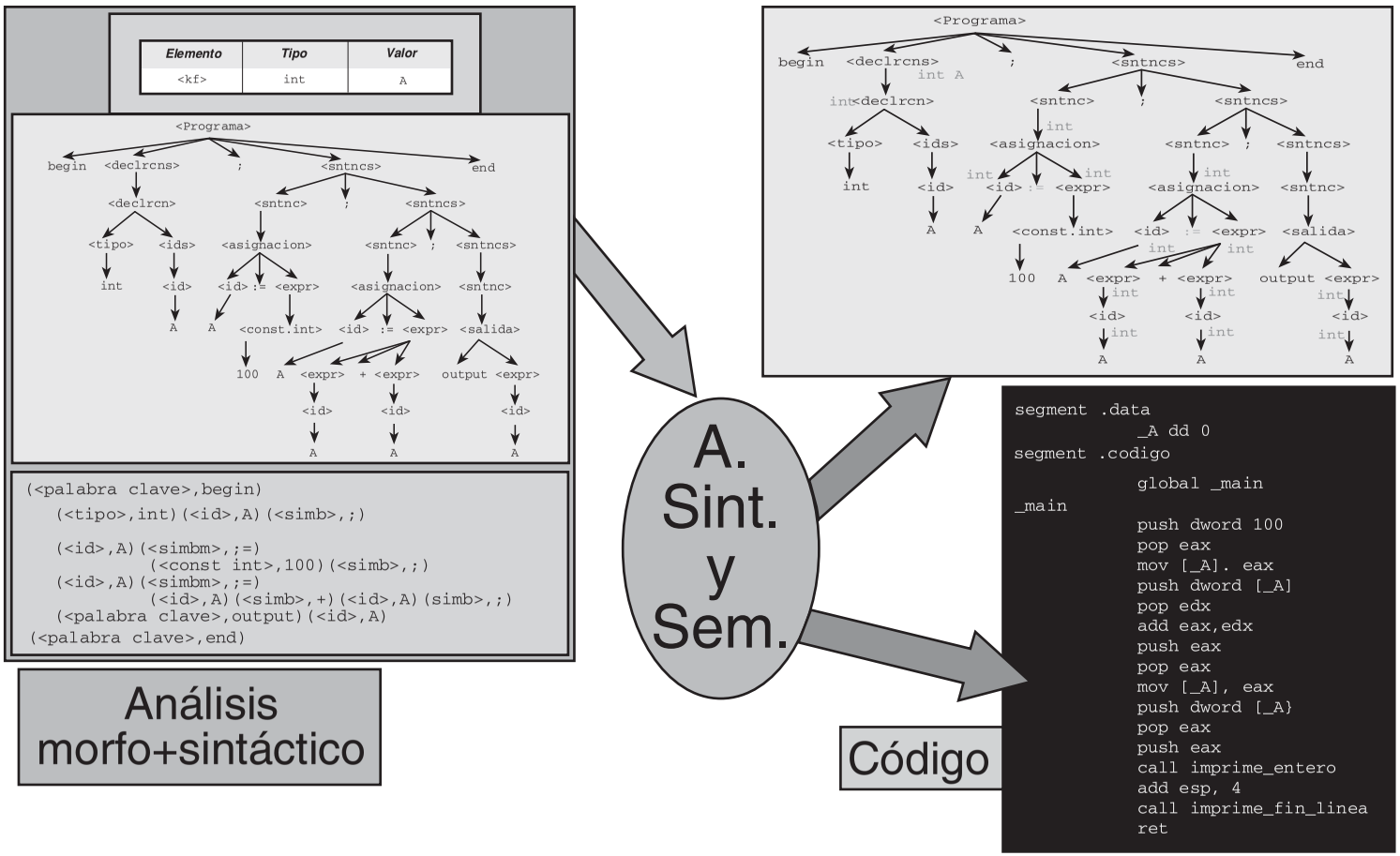


Figura 5.3. Esquema gráfico detallado del proceso de compilación en un solo paso.

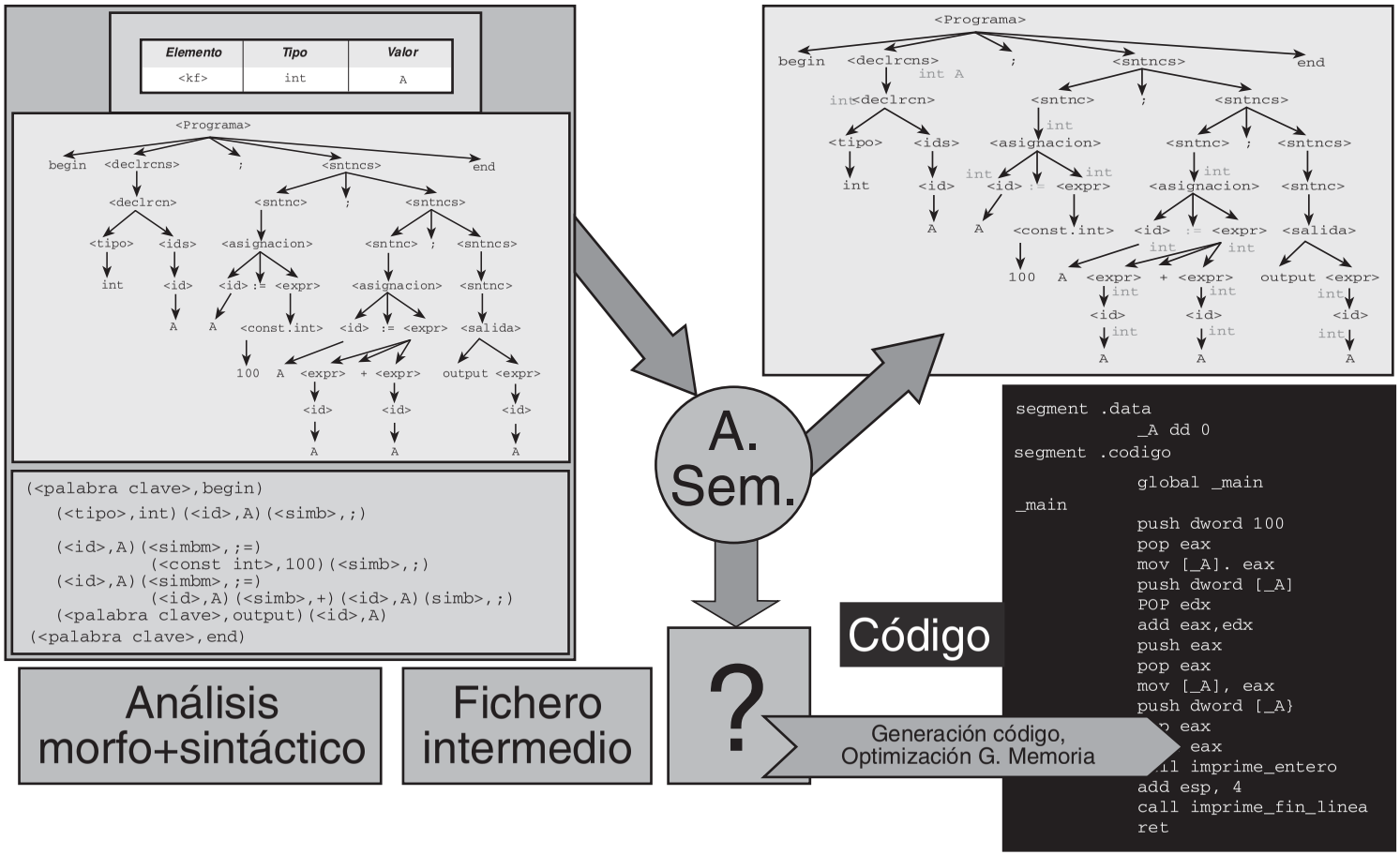


Figura 5.4. Esquema gráfico detallado del proceso de compilación en más de un paso.

5.1.5. Análisis semántico en compiladores de más de un paso

Cuando se diseña un compilador de más de un paso, se utilizan representaciones intermedias para facilitar las fases de optimización de código y gestión de memoria que se realizarán en los pasos siguientes. La Figura 5.4 muestra gráficamente esta situación.

Obsérvese que la salida del análisis semántico es una representación intermedia, que en la figura aparece contenida en un fichero. Las representaciones intermedias no tienen necesariamente que ocupar espacio en disco. En la Figura 5.4 se ve cómo un compilador, construido de acuerdo con este esquema, puede terminar la fase de generación de código optimizado realizando pasos adicionales sobre la representación intermedia.

5.2 Gramáticas de atributos

5.2.1. Descripción informal de las gramáticas de atributos y ejemplos de introducción

Informalmente, se llamará *atributos de un símbolo de la gramática* a toda información añadida en el árbol de derivación por el analizador semántico, asociada a los símbolos de los nodos anotados. El Ejemplo 5.1 describe de forma simplificada el valor de esos atributos y sugiere cómo calcularlos. La otra componente importante de las gramáticas de atributos es el algoritmo de cálculo de los valores. Todos estos aspectos se presentarán con más detalle mediante ejemplos.

Ejemplo 5.2 Considérese la gramática independiente del contexto para las expresiones aritméticas enteras asociada a las siguientes reglas de producción:

```
E::=E+E
E::=-E
E::=E*E
E::=(E)
E::=i
E::=c
```

Se supone que los símbolos terminales `c` e `i` hacen referencia a las constantes numéricas y los identificadores, respectivamente.

Los lenguajes de programación suelen proporcionar reglas para determinar correctamente el valor y el tipo de las expresiones aritméticas. En nuestro ejemplo, el último aspecto es trivial, ya que todas van a ser de tipo entero, pero en futuros ejemplos se podrá comprobar su importancia. Es evidente que los símbolos de la gramática que representan a los operandos tienen que poseer información respecto al tipo y el valor de éstos, mientras los que se refieren a los operadores deben realizar la operación correspondiente, es decir, aplicar las reglas del lenguaje para calcular el valor y el tipo de la expresión en la que aparecen.

Si se considera la expresión `(3+4)*5`, que pertenece al lenguaje de la gramática anterior, se puede concluir que su valor será 35 y su tipo entero. La corrección del valor se basa en que 3+4=7 y 7*5=35. El tipo es entero, porque todos los operandos elementales lo son y los operadores no modifican el tipo. Es decir, como 3 y 4 son enteros, 7 (su suma) también lo es, y el producto de la suma por otro entero (5) es también un valor entero (35).

Es evidente también que algunas partes de la expresión tienen que ser procesadas antes que otras. Por ejemplo, no se puede evaluar la expresión completa antes que la subexpresión (3+4). Lo mismo ocurre con el tipo. La Figura 5.5 muestra el árbol del análisis de esta expresión, anotado para gestionar su tipo y su valor.

Un rectángulo con línea discontinua resalta las tareas propias del analizador morfológico: al reconocer en la entrada las constantes `3`, `4` y `5`, tiene que indicar al analizador sintáctico que ha identificado la unidad `c`, que sus valores son `3`, `4` y `5`, y su tipo entero (representado por `i` de ***i***`nt`). Las flechas ascendentes sugieren el orden en el que se pueden realizar las anotaciones: primero las hojas, y las dos subexpresiones `(3+4)` y `5` antes que la expresión completa.

La Figura 5.5 indica que las acciones que hay que realizar para evaluar la expresión y deducir su tipo se pueden resumir de la siguiente forma: el valor de las constantes es el del número asociado a ellas, y su tipo es entero; el valor de la suma es la suma de los valores de sus operandos, y su tipo es entero, como ellas; lo mismo ocurre con el producto. Los paréntesis no modifican el valor ni el tipo de la expresión que contienen. Es fácil comprobar que estas acciones no

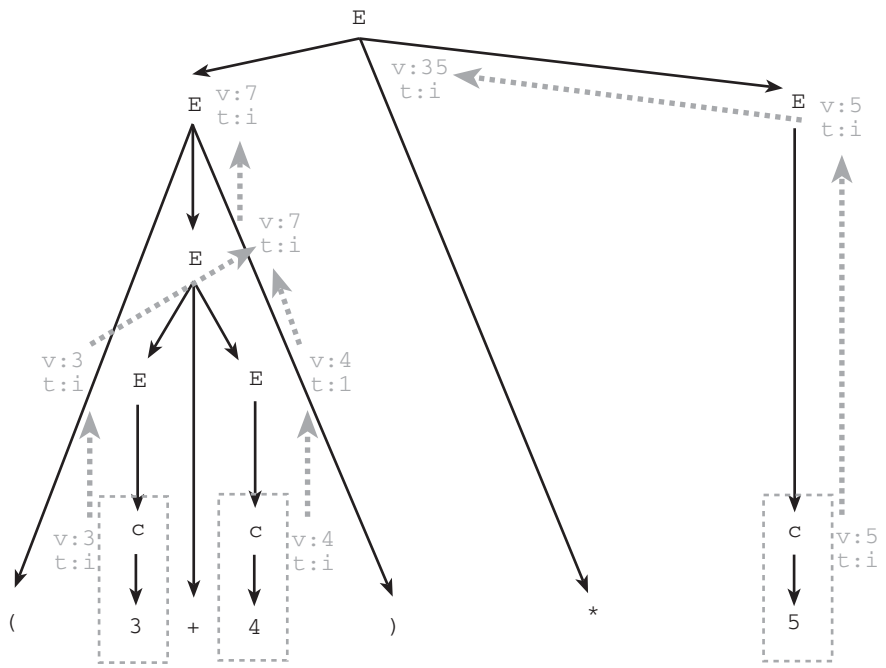


Figura 5.5. Árbol de análisis de la expresión `(3+4)*5`, anotado para calcular su tipo y su valor.

dependen de la expresión concreta, sino de la regla, es decir, que todas las sumas, productos, constantes y expresiones entre paréntesis se tratan de la misma manera.

Otra observación importante tiene que ver con el hecho de que, en este ejemplo, todas las flechas que aparecen en la figura, que indican el orden de realización de las operaciones, son *ascendentes*, es decir: *la información que se asigna a la parte izquierda de la regla se ha calculado utilizando únicamente información procedente de la parte derecha de la misma regla*.

Se puede plantear la posibilidad de que no siempre ocurra esto: ¿existen casos en los que la información que se desea asociar a algún símbolo de la parte derecha necesite de la información asociada a otro símbolo de la parte derecha o de la parte izquierda de la regla? El próximo ejemplo tiene como objeto responder esta cuestión.

Ejemplo 5.3 Considérese la gramática asociada a las siguientes reglas de producción, donde se considera que el símbolo `D` es el axioma:

```
D::=TL
T::=int
T::=real
L::=L,i
L::=i
```

Esta gramática podría pertenecer a la parte declarativa de algún lenguaje de programación, ya que `T` representa diferentes tipos de datos (en particular entero o `int`, o `real`). Es fácil comprobar que esta gramática genera declaraciones de identificadores de tipo entero o real, de forma que en cada instrucción puede declararse un único identificador o una lista de ellos separados por comas.

Vamos a estudiar la instrucción `int var1, x, y`, en la que se declaran tres identificadores de tipo entero. El objetivo de este ejemplo es asociar a cada uno de ellos el tipo con que han sido declarados. Dicho tipo se conoce desde que se detecta el símbolo terminal `int`. El tipo es el mismo para todos, y tiene que anotarse para cada uno de los identificadores de la lista.

La Figura 5.6 muestra el árbol anotado con los tipos de los identificadores. Por convenio, inicialmente se asigna a los identificadores un valor igual a 0.

Obsérvese que las flechas que sugieren un posible orden en las anotaciones indican que es imprescindible conocer en primer lugar el tipo del símbolo no terminal `T`. Dicho tipo puede usarse para anotar, si es necesario, el tipo de la raíz del árbol (`D`) y el del símbolo `L`, hermano de `T`. A partir de aquí, se sabe que el identificador `y` es de tipo entero, así como `var1` y `x`, mediante las dos apariciones más profundas del símbolo `L`.

Las acciones que completarían las anotaciones del árbol pueden resumirse así: el tipo del símbolo no terminal `D` debe ser igual al de su primer hijo `T`, y también pasará a ser el de su segundo hijo `L`. El tipo de este símbolo (`L`) pasará a ser el de su primer hijo (`L`) y el del identificador `y`. El tipo del identificador `x` será el de su padre (`L`) y también lo será del otro hijo, que termina en el identificador `var1`. Por lo tanto, en la regla `D::=TL`, el tipo de la raíz se calcula utilizando el de su hijo `T`, y el de `L` se calcula también de la misma manera. En las dos apariciones de la regla `L::=L,i` y en la de la regla `L::=i`, el tipo del padre se utiliza para calcular el de sus hijos (`L` e `i`).

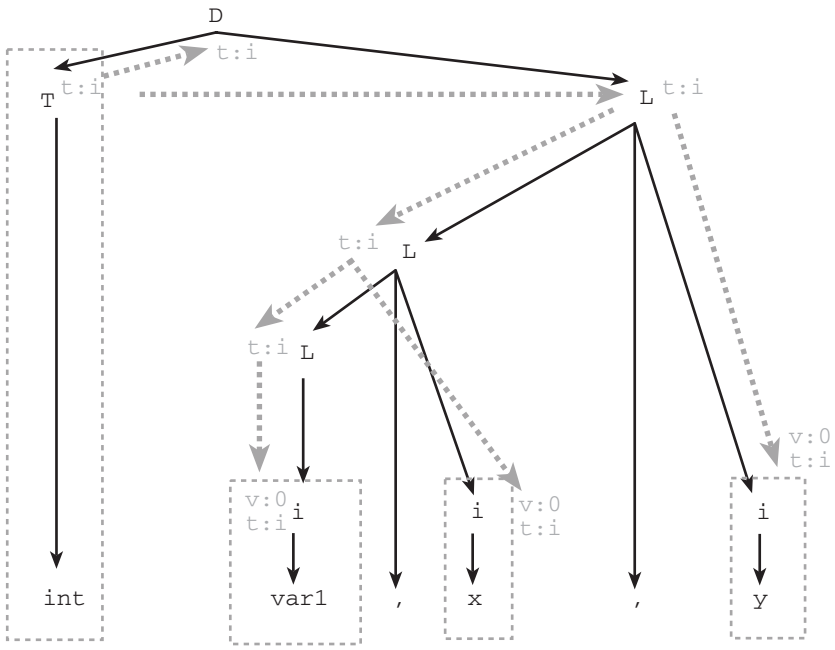


Figura 5.6. Árbol de análisis de la expresión `int var1, x, y`, anotado para calcular su tipo.

Es fácil imaginar cómo se podría incorporar aquí la gestión de la tabla de símbolos de un compilador. Una vez que se conoce el nombre y el tipo de cada identificador, se podría comprobar que no colisiona con ningún otro elemento de la tabla de símbolos, antes de realizar su inserción en ella.

Las conclusiones de estos ejemplos son las siguientes:

- Para la realización del análisis semántico se necesita asociar cierta información a cada símbolo de la gramática, así como describir las acciones necesarias para calcular el valor de dicha información en cada punto del árbol de análisis. En las próximas secciones se verá que la información semántica se formaliza mediante los *atributos semánticos*, que se calculan mediante la ejecución de *acciones semánticas*.
- Puede ser conveniente disponer de información *global*, que no dependa de ningún símbolo y sea accesible desde cualquier regla de la gramática.
- La información que se asocia a cada símbolo depende sólo de éste, es decir, será la misma para todas las apariciones del mismo símbolo. Evidentemente, el valor concreto que reciba dicha información dependerá del lugar en el que aparezca cada símbolo.
- Las acciones que hay que realizar para calcular los valores de los atributos dependen de las reglas, no de los símbolos: en cada aplicación de la misma regla en el árbol de análisis se aplicarán las mismas acciones para calcular los valores asociados a sus símbolos.

5.2.2. Descripción formal de las gramáticas de atributos

El objetivo de esta sección es definir formalmente los conceptos que se han presentado informalmente mediante ejemplos en la sección anterior.

Se llama *gramática de atributos*[1] a una extensión de las gramáticas independientes del contexto a las que se añade un *sistema de atributos*. Un *sistema de atributos* está formado por:

- Un conjunto de *atributos semánticos* que se asocia a cada símbolo de la gramática.
- Los datos *globales* de la gramática, accesibles desde cualquiera de sus reglas, pero no asociados a ningún símbolo concreto.
- Un conjunto de *acciones semánticas*, distribuidas por las reglas de producción.

De forma semejante a las variables en los lenguajes de programación de alto nivel, un *atributo semántico* se define como un par, compuesto por un tipo de datos (la especificación de un dominio o conjunto) y un nombre o identificador. En algunos ejemplos de este capítulo, debido a su simplicidad, el dominio de los atributos es irrelevante y se omitirá. En cada momento, cada atributo semántico puede tener un valor único que tiene que pertenecer a su dominio. Dicho valor puede modificarse en las acciones semánticas, sin ningún tipo de restricción.

Una *acción semántica* es un algoritmo asociado a una regla de la gramática de atributos, cuyas instrucciones sólo pueden referirse a los atributos semánticos de los símbolos de la regla y a la información global de la gramática de atributos. El objetivo de la acción semántica es calcular el valor de alguno de los atributos de los símbolos de su regla, sin restricciones adicionales.

En este capítulo se utilizará la siguiente notación para las gramáticas de atributos:

$$A = (\sum_T, \sum_N, S, P, K)$$

- Se mantiene la estructura de las gramáticas independientes del contexto.
- A cada símbolo de la gramática lo acompaña la lista de sus atributos semánticos entre paréntesis. El nombre de cada atributo sigue al de su dominio. Se utiliza la misma notación que en el resto de los ejemplos de pseudocódigo. En los símbolos sin atributos semánticos se omitirán los paréntesis.
- Las reglas de producción se modifican para distinguir distintas apariciones del mismo símbolo. A cada regla de producción lo acompaña su acción semántica, en la que también se utilizará la misma sintaxis de los ejemplos de pseudocódigo. Las instrucciones de la acción se escriben entre llaves. Para referirse a un atributo semántico de un símbolo dentro de las acciones semánticas, se escribirá el nombre del atributo tras el del símbolo, separados por un punto '`.`'. Si alguna regla no necesita realizar ninguna acción semántica, se escribirá '`{}`'.
- La información global se añade como nueva componente adicional, al final de la gramática de atributos (`K`).

---

[1] A lo largo de la historia de los lenguajes formales ha habido diversas ideas que han desembocado en las *gramáticas de atributos.* Algunas sólo proponen nombres distintos para el mismo concepto; otras sí describen aspectos diferentes y han tenido relevancia en distintos momentos. En este texto se utilizará la denominación *gramáticas de atributos* para unificar y clarificar la exposición.

Ejemplo 5.4 A continuación se muestra la gramática de atributos del Ejemplo 5.2:

A_{5_4}={
Σ_T={+, *, (,), c(valor,tipo), i(valor,tipo)},
Σ_N={E(valor,tipo)},
E,
P={
E::=E_i+E_d
{ E.valor = E_i.valor+E_d.valor;
E.tipo = E_i.tipo;},
E::=-E_d,
{ E.valor = -E_d.valor;
E.tipo = E_d.tipo; },
E::=E_i*E_d
{ E.valor = E_i.valor*E_d.valor;
E.tipo = E_i.tipo;},
E::=(E_d)
{ E.valor = E_d.valor;
E.tipo = E_d.tipo},
E::=i
{ E.valor = i.valor,
E.tipo = i.tipo},
E::=c
{ E.valor = c.valor,
E.tipo = c.tipo}
},
K=Φ}

Ejemplo 5.5 A continuación se muestra la gramática de atributos del Ejemplo 5.3:

A_{5_5}={
Σ_T={int(tipo), real(tipo), i(tipo)},
Σ_N={D(tipo),T(tipo),L(tipo)},
D,
P={
D::=TL
{ L.tipo = T.tipo;
D.tipo = L.tipo;},
T::=int {T.tipo = entero;},
T::=real {T.tipo = real;},
L::=L_d,i
{ L_d.tipo = L.tipo;
i.tipo = L.tipo;},
L::=i {i.tipo = L.tipo;}
},
K=Φ}

5.2.3. Propagación de atributos y tipos de atributos según su cálculo

Se llama *propagación* al cálculo del valor de los atributos en función del valor de otros. Se define así una relación de dependencia entre los atributos que aparecen en el árbol de análisis de una instrucción, ya que los atributos asociados a un nodo dependen de los atributos necesarios para calcular su valor.

Las flechas que sugieren el orden de cálculo de los atributos en las Figuras 5.5, 5.6 y 5.7 muestran realmente el proceso de la propagación de atributos o, lo que es lo mismo, su relación de dependencia. En la Sección 5.3.5 se estudiará con más detalle dicha relación de dependencia y su importancia para el análisis semántico.

Los atributos semánticos se pueden clasificar según la posición, en la regla de producción, de los símbolos cuyos atributos se utilicen en el cálculo. Se distinguen dos grupos:

- **Atributos sintetizados:** Son los atributos asociados a los símbolos no terminales de la parte izquierda de las reglas de producción cuyo valor se calcula utilizando los atributos de los símbolos que están en la parte derecha correspondiente.

  Obsérvese la siguiente regla de la gramática del Ejemplo 5.4:

  ```
  E::=E_i+E_d
  { E.valor = E_i.valor+E_d.valor;
    E.tipo = E_i.tipo;},
  ```

  Los atributos `valor` y `tipo` del símbolo no terminal de la parte izquierda (`E`) se calculan a partir de los de la parte derecha (E_i y E_d), por lo que se puede afirmar que son sintetizados.

- **Atributos heredados:** El resto de las situaciones originan atributos heredados, es decir, atributos asociados a símbolos de la parte derecha de las reglas cuyo valor se calcula utilizando los atributos de la parte izquierda o los de otros símbolos de la parte derecha. En este grupo es necesaria la siguiente distinción:

  — Atributos heredados *por la derecha*, cuando el cálculo del valor de un atributo utiliza atributos de los símbolos que están situados a su derecha.
  — Atributos heredados *por la izquierda*, cuando sólo se utilizan los que están a su izquierda, ya sea en la parte derecha de la regla o en la parte izquierda.

  Obsérvese la siguiente regla de la gramática del Ejemplo 5.5:

  ```
  D::=TL
  { L.tipo = T.tipo;
    D.tipo = L.tipo;}
  ```

  El atributo `tipo` del símbolo no terminal `L` lo hereda de su hermano izquierdo (`T`). Véanse también las siguientes reglas:

  ```
  L::=L_d,i
  { L_d.tipo = L.tipo;
  ```

```
    i.tipo = L.tipo;}
L::=i {i.tipo = L.tipo;}
```

El atributo `tipo` del símbolo terminal `i` se hereda del padre (en ambos casos `L`).

Ejemplo 5.6 Muestra una gramática de atributos que reúne los dos últimos ejemplos, para describir un lenguaje de programación un poco más realista. El lenguaje contiene la parte declarativa del Ejemplo 5.5 y las expresiones aritméticas del Ejemplo 5.4. Se añade un nuevo axioma, para indicar que esta gramática genera programas completos, y un nuevo símbolo no terminal (`A`), que realiza la asignación de una expresión a un identificador mediante el símbolo '='. La gramática incorpora una tabla de símbolos, donde se conserva el tipo con el que se declaran los atributos, que ayuda a comprobar la corrección semántica de las instrucciones en las que aparecen dichos atributos.

```
A5_6={
      ΣT={+, *, (, ), c(valor,tipo), i(valor,tipo),
          int(tipo), real(tipo), i(tipo), = },
      ΣN={Programa, A(tipo),
          E(valor,tipo), D(tipo),T(tipo),L(tipo)},
      Programa,
      P={
         Programa::=DA {},
         A::=i = E
         { SI 'i∈TablaSimbolos'∧ E.tipo=i.tipo
           ENTONCES A.tipo=i.tipo;}
         E::=Ei+Ed
         { E.valor = Ei.valor+Ed.valor;
           E.tipo = Ei.tipo;},
         E::=-Ed,
         { E.valor = -Ed.valor;
           E.tipo = Ed.tipo; },
         E::=Ei*Ed
         { E.valor = Ei.valor*Ed.valor;
           E.tipo = Ei.tipo;},
         E::=(Ed)
         { E.valor = Ed.valor;
           E.tipo = Ed.tipo},
         E::=i
         { E.valor = i.valor,
           E.tipo = i.tipo},
         E::=c
         { E.valor = c.valor,
           E.tipo = c.tipo}
         D::=TL
         { L.tipo = T.tipo;
```

```
        D.tipo = L.tipo;},
      T::=int {T.tipo = entero;},
      T::=real {T.tipo = real;},
      L::=Ld,i
      { Ld.tipo = L.tipo;
        i.tipo = L.tipo;
        insertar(TablaSimbolos, i, L.tipo);},
      L::=i
      { i.tipo = L.tipo;
        insertar(TablaSimbolos, i, L.tipo);}
      },
    K={TablaSimbolos}}
```

En esta gramática, `TablaSimbolos` es una tabla de símbolos en la que se conserva (al menos) información sobre el tipo de cada identificador. Asociada a esta tabla se dispone de la operación `insertar(TablaSimbolos, identificador, tipo)`, mediante la que se deja constancia en la tabla de que se ha declarado el identificador cuyo nombre es el segundo argumento y cuyo tipo es el tercero.

Obsérvese que la regla

```
A::=i = E
{ SI 'i∈TablaSimbolos'∧E.tipo=i.tipo
  ENTONCES A.tipo=i.tipo;}
```

realiza las comprobaciones semánticas sobre la correspondencia de tipos entre la expresión y el identificador al que se asigna su valor. Obsérvese también que las dos reglas en las que aparece el símbolo terminal `i` en la parte declarativa insertan dicho identificador en la tabla de símbolos.

```
L::=Ld,i
{ Ld.tipo = L.tipo;
  i.tipo = L.tipo;
  insertar(TablaSimbolos, i, L.tipo);}
L::=i
{ i.tipo = L.tipo;
  insertar(TablaSimbolos, i, L.tipo);}
}
```

La Figura 5.7 muestra el análisis del programa

```
int var1, x, y
x = (3+4)*5
```

También puede observarse el contenido de la tabla de símbolos al final del análisis, que inicialmente se encuentra vacía.

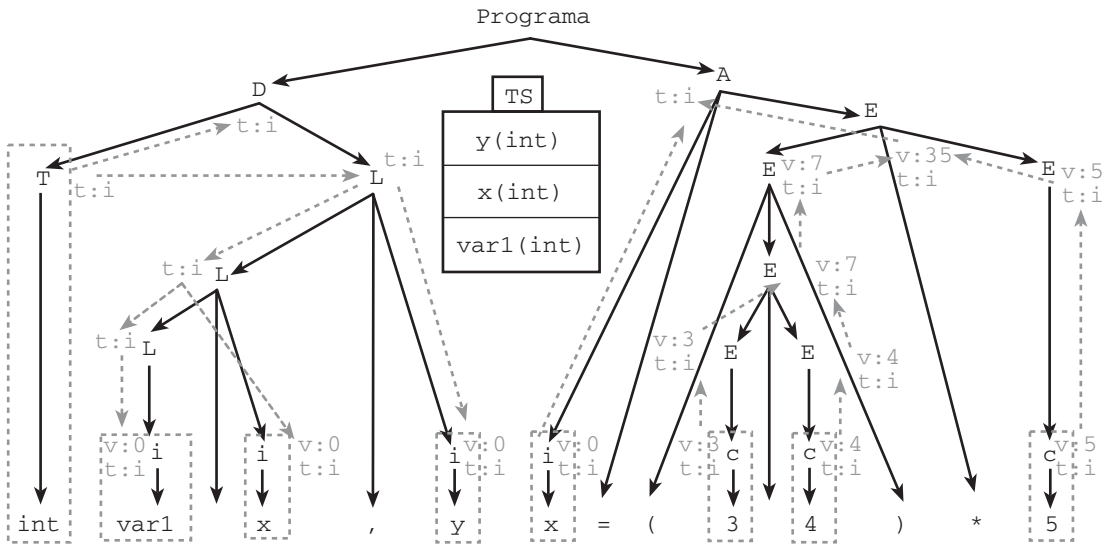


Figura 5.7. Árbol de análisis del Ejemplo 5.6.

Las flechas que sugieren el orden de evaluación de los atributos muestran que es necesario procesar primero el subárbol izquierdo que contiene la parte declarativa. Dentro de este subárbol se aplica lo explicado en la Figura 5.6. Al terminar el proceso de dicho subárbol, la tabla de símbolos contiene toda la información que necesita.

Al analizar la asignación, el identificador al que se le asigna nuevo valor (`x`) es buscado en la tabla de símbolos, para comprobar que ha sido declarado antes de ser usado y para consultar el tipo con el que se lo declaró (`int`). Esta información está contenida en la anotación del nodo del símbolo `i`, padre del nodo `x`.

A continuación puede procesarse el subárbol de la expresión (el que tiene como raíz la aparición menos profunda del símbolo no terminal `E`). En este subárbol se aplica lo explicado en la Figura 5.5. Finalmente, la asignación puede concluirse correctamente en el nodo etiquetado con el símbolo no terminal `A`. Tanto el identificador como la expresión son de tipo entero.

5.2.4. Algunas extensiones

Es frecuente que las gramáticas de atributos contengan acciones semánticas situadas entre los símbolos de la parte derecha de las reglas. Hay varias razones para permitirlo: por un lado, se ofrece mayor flexibilidad al diseñador de la gramática; por otro, hay algunos analizadores semánticos que pueden procesar las acciones en esa posición.

Ejemplo 5.7 La siguiente gramática de atributos presenta una versión simplificada de las instrucciones de asignación de expresiones a identificadores:

```
A5_7={
      ΣT={=, +, cte(valor, tipo), id(valor, tipo)},
      ΣN={ Asignación(valor, tipo),
           Expresión(valor, tipo),
           Término{valor, tipo}},
      Asignación,
      P={
          Asignación ::= id
          { 'Comprobación en la tabla de símbolos de que el
            identificador ha sido declarado y recuperación de
            su tipo'; }
                      = Expresión
          { 'Comprobación de la compatibilidad de los tipos
            Expresión.tipo e id.tipo';
            id.valor = Expresión.valor;},

          Expresión ::= Expresiónd + Término
          { 'Comprobación de compatibilidad de los tipos
            Expresiónd.tipo y Término.tipo';
            Expresión.tipo = [2]
                 tipoSuma(Expresiónd.tipo, Término.tipo);
            Expresión.valor=Expresiónd.valor+Término.valor;},

          Expresión ::= Término
          { Expresión.tipo = Término.tipo;
            Expresión.valor = Términao.valor;},

          Término ::= id
          { 'Comprobación en la tabla de símbolos de que el
            identificador ha sido declarado y recuperación de
            su tipo';
            Término.tipo = id.tipo;
            Término.valor = id.valor;},

          Término ::= cte
          { Término.tipo = cte.tipo;
            Término.valor = cte.valor; }
      },
      K=Φ}
```

En este ejemplo, la información relacionada con el tipo y los valores se almacena con el mismo criterio de los ejemplos anteriores. La primera acción semántica no está situada al final de la primera regla.

[2] La función `tipoSuma` toma como argumentos dos tipos y determina el tipo que le correspondería, con esos tipos, a la operación suma de dos operandos.

En este capítulo se supondrá, casi siempre, que las gramáticas de atributos tienen las acciones semánticas situadas al final de la regla, y cuando esto no se aplique se indicará explícitamente que la gramática no cumple dicha condición. Siempre es posible transformar una gramática de atributos con acciones semánticas en cualquier posición en otra equivalente con todas las acciones semánticas al final de las reglas. El algoritmo correspondiente realiza los siguientes pasos, mientras existan reglas de producción con la siguiente estructura:

$$N ::= X_1\ X_2\ \ldots\ X_{i-1}\ \{\text{Instrucciones}\}\ X_i\ \ldots\ X_n$$

1. Se añade un nuevo símbolo no terminal a la gramática (por ejemplo, Y) sin atributos semánticos asociados.
2. Se añade la siguiente regla de producción:

$$Y ::= \lambda\ \{\text{Instrucciones}\}$$

3. Se sustituye la regla inicial por la siguiente:

$$N ::= X_1\ X_2\ \ldots\ X_{i-1}\ Y\ X_i\ \ldots\ X_n$$

Es fácil comprobar que el lenguaje generado por ambas gramáticas independientes del contexto es el mismo, y que la información semántica no ha cambiado.

En este ejemplo se obtendría la siguiente gramática:

```
A5_7,={ΣT, ΣN∪{M}, Asignación,
       P∪
       {M::=λ
         {   'Comprobación en la tabla de símbolos de que el
             identificador ha sido declarado y recuperación de
             su tipo'; }},
       K=Φ}
```

El problema de este enfoque es que los nuevos símbolos no terminales no recogen ninguna semántica del problema, ya que son meros marcadores. Además, es posible que no se desee añadir reglas-λ. En situaciones reales, en las que puede haber muchas reglas de producción, a veces conviene modificar la gramática un poco más, para que cada símbolo y cada regla retengan algo de significado.

En tal caso, podría conseguirse lo mismo con el siguiente algoritmo:

1. Se añade un nuevo símbolo no terminal a la gramática, para representar todos los símbolos de la parte derecha, desde su comienzo hasta el símbolo seguido por la acción semántica (por ejemplo, X_{1_i-1}). Este nuevo símbolo no tiene atributos semánticos asociados.
2. Se añade la siguiente regla de producción:

$$X_{1\_i-1} ::= X_1\ X_2\ \ldots\ X_{i-2}\ X_{i-1}\ \{\text{Instrucciones}\}$$

3. Se sustituye la regla inicial por la siguiente:

$$N ::= X_{1\_i-1}\ X_i\ \ldots\ X_n$$

Por ejemplo, se podría pensar que en la gramática anterior se necesitan realmente dos reglas de producción para la asignación. La primera conservaría la estructura de la actual, pero sustituyendo el símbolo terminal `id` por el nuevo no terminal `id_asignado`. El nuevo símbolo tendrá una regla en la que, junto con la acción semántica, también se *arrastra* el resto de la derivación:

```
Asignación ::= id_asignado Expresión
{ 'Comprobación de la compatibilidad de los tipos
  Expresión.tipo e id.tipo';
  id_asignado.valor = Expresión.valor;},


id_asignado ::= id
{ 'Comprobación en la tabla de símbolos de que el
  identificador ha sido declarado y recuperación de
  su tipo';
  id_asignado.tipo = id.tipo;},
```

Resulta fácil comprobar la equivalencia del resultado de este enfoque, basado en el segundo algoritmo y las dos gramáticas anteriores. Obsérvese que no se ha añadido ninguna regla-λ, aunque sí una regla de redenominación.

5.2.5. Nociones de programación con gramáticas de atributos

En el desarrollo del analizador semántico de un compilador, es necesario solucionar con gramáticas de atributos las dificultades asociadas a todas las construcciones del lenguaje de programación considerado. En esta sección se va a reflexionar, mediante ejemplos, sobre algunas características del diseño de las gramáticas de atributos. Es fácil comprobar que la posibilidad de especificar cualquier algoritmo en las acciones semánticas de las reglas implica que las gramáticas de atributos tengan la misma potencia que las Máquinas de Turing: son capaces de expresar el algoritmo asociado a cualquier tarea computable. Por lo tanto, podría considerarse que las gramáticas de atributos constituyen un lenguaje de programación en sí mismo. El objetivo de los próximos puntos es destacar las peculiaridades de este *nuevo lenguaje de programación*.

Todos los ejemplos que se van a exponer en esta sección están relacionados con el *lenguaje de los paréntesis*, compuesto por expresiones escritas exclusivamente con paréntesis equilibrados, en los que cada paréntesis abierto se cierra dentro de la expresión siguiendo las normas habituales de las expresiones matemáticas. El lenguaje considerado en estos párrafos tiene la

peculiaridad de que la expresión completa está encerrada siempre entre paréntesis. Es decir, las siguientes palabras pertenecen al lenguaje descrito:

```
( () () )
( ( ()(())() ) )
```

Pero las siguientes no pertenecen a él:

```
( ()
() ()
```

Todos los ejemplos utilizan como punto de partida la siguiente gramática independiente del contexto:

G_{5_8}={
Σ_T={ (,) },
Σ_N={ <lista>, <lista_interna>},
<lista>,
P={<lista> ::= (<lista_interna>)
<lista_interna>::=<lista_interna>(<lista_interna>)
<lista_interna>::=λ}
}

El axioma representa la expresión completa: una *lista interna* entre paréntesis. La *lista interna* más sencilla es la palabra vacía. La regla recursiva para la lista interna la describe como una lista interna junto a otra encerrada entre paréntesis. Es fácil comprobar la correspondencia entre esta gramática y el lenguaje especificado.

Consideraremos los dos problemas siguientes:

- **Cálculo de la profundidad** (o nivel de anidamiento) de la expresión. El concepto queda definido por los siguientes ejemplos:

  `()`, tiene profundidad 1
  `( () ( ( ) ) )`, tiene profundidad 3, debido a la sublista derecha

- **Cálculo del número de listas** de la expresión por ejemplo,

  `()`, contiene una lista
  `( () ( ( ) ) )` contiene 3 listas
  `( () () () () )` contiene 5 listas
  `( ( ((())) () ))` contiene 6 listas

Aunque los ejemplos son casos particulares pequeños, las conclusiones extraídas de ellos se pueden aplicar al diseño de gramáticas de atributos en general.

Ejemplo 5.8

La siguiente gramática de atributos calcula la profundidad de la lista estudiada. El diseño de esta gramática se basa en el cálculo clásico de la profundidad de una lista con un algoritmo recursivo que aproveche la estructura de las reglas de la gramática. De esta forma, basta con introducir un único atributo de tipo entero (`profundidad`).

En la regla del axioma <lista>::=(<lista_interna>), habrá que añadir una unidad a la profundidad de la <lista_interna>.

Una de las reglas para el símbolo no terminal <lista_interna> sirve para finalizar la recursividad: <lista_interna>::=λ, que corresponde claramente a una lista de profundidad 0. En cambio, en la regla recursiva:

<lista_interna>::=<lista_interna>[1](<lista_interna>[2])

hay que determinar cuál de las sublistas tiene la mayor profundidad. Si es la primera, la profundidad de la lista interna completa coincidirá con la suya; si es la segunda, será necesario incrementarla en una unidad, pues está encerrada entre paréntesis. Véase la gramática completa:

```
A5_8={
      ΣT={ (, ) },
      ΣN={<lista>(entero profundidad;),
          <lista_interna>(entero profundidad)},
      <lista>,
      P={
         <lista>::= ( <lista_interna> )
         { <lista>.profundidad=
                <lista_interna>.profundidad + 1;
           IMPRIMIR("PROFUNDIDAD:",<lista>.profundidad);},

         <lista_interna>::=<lista_interna>1
                (<lista_interna>2)
         { SI  ( <lista_interna>1.profundidad >
                 <lista_interna>2.profundidad )
                 <lista_interna>.profundidad =
                      <lista_interna>1.profundidad ;
           EN OTRO CASO
                 <lista_interna>.profundidad =
                      <lista_interna>2.profundidad + 1 ;},

         <lista_interna>::=λ
         { <lista_interna>.profundidad = 0;}
      },
      K=Φ}
```

La Figura 5.8 muestra el cálculo de la profundidad de la lista (()(())) mediante la gramática de atributos de este ejemplo. Se han utilizado flechas discontinuas para indicar la síntesis de los atributos. Los nombres de los atributos han sido sustituidos por sus iniciales.

Se puede observar que es posible solucionar este problema utilizando únicamente atributos sintetizados, porque lo que se desea calcular sólo depende de una parte concreta de toda la expresión: la sublista más profunda.

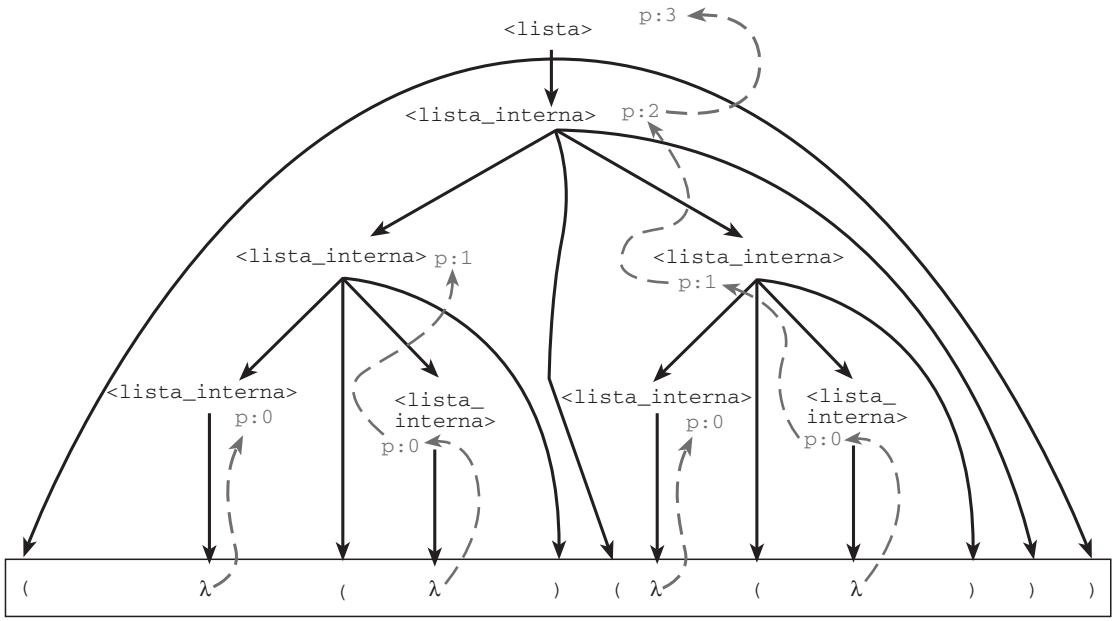


Figura 5.8. Cálculo de la profundidad de la lista `(()(()))` para el Ejemplo 5.8.

Ejemplo 5.9 La siguiente gramática de atributos calcula el número de listas de la expresión. La estructura de las reglas de la gramática independiente del contexto sugiere que la única regla asociada al axioma, `<lista>::=(<lista_interna>)`, debe sintetizar un atributo semántico cuyo valor sea el número total de listas de la expresión. Si este atributo se llama `num_listas_total`, está claro que su valor correcto se obtendrá sumando una unidad al número de listas, calculado por el símbolo no terminal `<lista_interna>`.

Para ilustrar el cálculo de listas de una lista interna se puede considerar el siguiente ejemplo:

((***((()))*** ()))

Se supondrá que la cadena se recorre de izquierda a derecha. Es fácil comprobar que el número de listas correspondiente a la lista resaltada es igual a 3. A su derecha hay una lista más, y el conjunto está incluido en otras dos listas. Eso hace un total de 6 (3+1+2). Este proceso acumulativo sugiere que cada lista interna debe calcular el número de sus listas, sumarlo al número de listas encontradas en la cadena que la precede y ofrecer el valor total al resto del análisis. Puesto que las listas tienen que estar equilibradas, si se identifica una nueva lista por su paréntesis de apertura, al llegar a la sublista resaltada anterior se llevarán ya contabilizadas 2 listas. El cálculo que debe realizar la lista resaltada consistirá en añadir las suyas (3) y ofrecer el valor total (5) al resto del proceso. Tras la parte resaltada sólo queda una lista más, que completará el valor correcto. Puede utilizarse un atributo para recibir el número de listas de la parte procesada (se lo denominará `num_listas_antes`) y otro para el que se ofrezca al resto del análisis (`num_listas_despues`).

Hay que distribuir el cálculo del número de listas del símbolo <lista_interna> entre las reglas donde aparece. Puesto que sus reglas son recursivas, comenzaremos por aquella que permite dar por terminada la recursión: <lista_interna>::=λ, donde es evidente que no hay que hacer nada, pues esta regla no añade nuevas listas, por lo que el valor de num_listas_despues coincidirá con el de num_listas_antes.

También está claro el cálculo necesario para la regla recursiva <lista_interna>::= <lista_interna>[1](<lista_interna>[2]).

- El valor de <lista_interna>.num_listas_antes es el número de listas encontradas hasta el momento, y es el mismo que habrá hasta <lista_interna>[1].
- El número de listas antes de procesar <lista_interna>[2] tiene que añadir 1 (por la lista que aparece de forma explícita) después de procesar <lista_interna>[1].
- El número de listas después de procesada esta regla, coincide con el obtenido después de procesar <lista_interna>[2].

Falta asociar un valor inicial al atributo num_listas_antes del símbolo no terminal <lista_interna> en la regla del axioma. Es evidente que la lista que aparece de forma explícita en dicha regla es la primera de la expresión completa, por lo que el valor del atributo tiene que ser 1. Véase la gramática de atributos completa:

```
A5_9={
      ΣT={ (, ) },
      ΣN={ <lista>(entero num_listas_total;),
           <lista_interna>( entero num_listas_antes;
                            entero num_listas_despues)},
      <lista>,
      P=
      {<lista> ::= ( <lista_interna> )
         {<lista_interna>.num_listas_antes = 1;
          <lista>.num_listas_total =
                <lista_interna>.num_listas_despues;
          IMPRIMIR("ELEMENTOS",<lista>.num_listas_total);},
      <lista_interna>::=<lista_interna>1(<lista_interna>2)
        { <lista_interna>1.num_listas_antes =
                <lista_interna>.num_listas_antes;
          <lista_interna>2.num_listas_antes =
                <lista_interna>1.num_listas_despues+1;
          <lista_interna>.num_listas_despues =
                <lista_interna>2.num_listas_despues;},

      <lista_interna>::=λ
        { <lista_interna>.num_listas_despues =
                <lista_interna>.num_listas_antes;}},
    K=Φ}
```

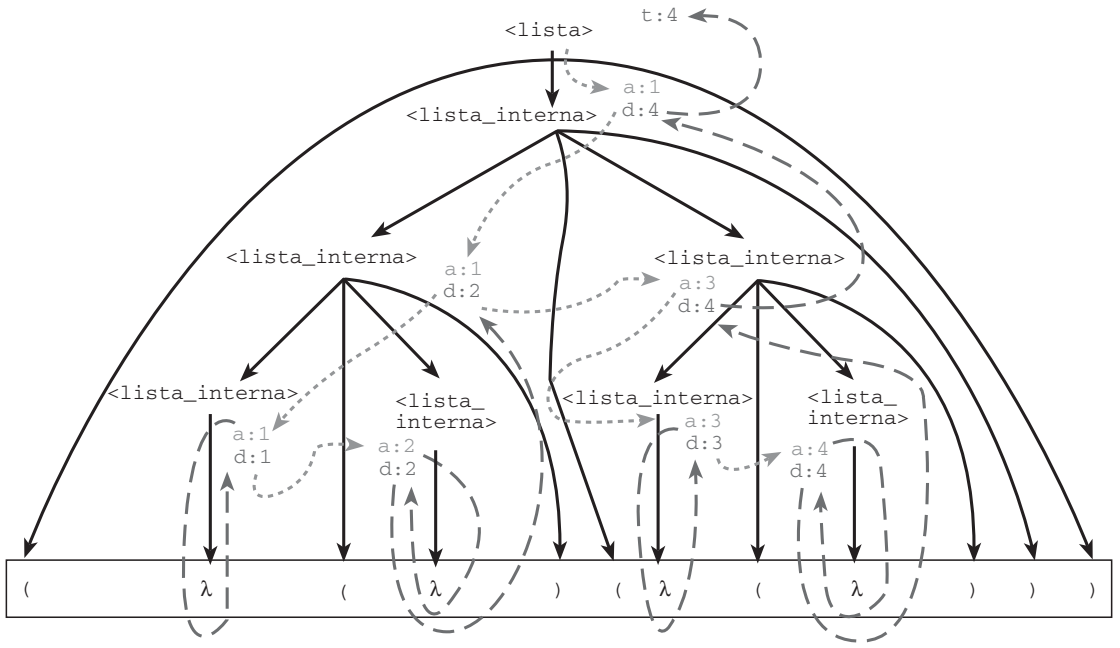


Figura 5.9. Cálculo del número de listas de la expresión `(()(()))` para el Ejemplo 5.9.

La Figura 5.9 muestra el cálculo del número de listas de la expresión `(()(()))` mediante la gramática de atributos de este ejemplo. Se han utilizado dos tipos de flechas discontinuas para distinguir la herencia de la síntesis de atributos.

Se pueden extraer las siguientes conclusiones:

- El axioma tiene asociado un atributo sintetizado de tipo entero (`num_listas_total`), que representa el número total de listas de la expresión.
- Las listas internas tienen dos atributos de tipo entero, uno (`num_listas_antes`) heredado a veces de su padre, a veces de su hermano más a la izquierda, que indica el número de listas que había en la expresión antes del proceso de esta lista interna, y el otro (`num_listas_despues`) sintetizado, que recoge las modificaciones en el número de listas debidas a la regla. El atributo heredado es necesario, porque el valor que se quiere calcular puede depender de partes diferentes de la expresión.

Ejemplo 5.10 A continuación se muestra otra gramática de atributos que también calcula el número de listas de la expresión. Puede compararse con la del ejemplo anterior. En este caso se ha aprovechado la posibilidad de utilizar, como información global, una variable de tipo entero (`num_elementos`) que se declara e inicializa con el valor 0 en la última componente de la gramática (`K`). Para obtener el valor correcto, bastará con incrementar el valor de dicho atributo en una unidad cada vez que aparece una lista de forma explícita en la parte derecha de alguna regla.

```
A5_10={
      ΣT={ (, ) },
      ΣN={<lista>(), <lista_interna>()},
      <lista>,
      P={
         <lista>::=( <lista_interna> )
         { num_elementos = num_elementos + 1;
           IMPRIMIR("HAY ",num_elementos, " LISTAS" );},

         <lista_interna>::= <lista_interna>
                ( <lista_interna> )
         { num_elementos++;},
         <lista_interna>::=λ{ }},
  K={entero num_elementos=0;}
}
```

La comparación de esta gramática con la del Ejemplo 5.9 ofrece una conclusión interesante: los atributos heredados pueden sustituirse fácilmente por información global. La importancia de esta conclusión se analizará con más detalle en las próximas secciones, en las que se verá que esta sustitución puede ser necesaria en determinadas circunstancias.

5.3 Incorporación del analizador semántico al sintáctico

5.3.1. ¿Dónde se guardan los valores de los atributos semánticos?

La primera pregunta que hay que responder, al incorporar el análisis semántico al sintáctico, es dónde se pueden guardar los valores de los atributos que se han añadido a la gramática independiente del contexto. El analizador sintáctico manipula los símbolos de la gramática moviéndolos entre la entrada y la pila del análisis. De forma general, los analizadores semánticos sustituyen los símbolos manipulados por el analizador semántico por una estructura de datos, que contendrá, además de la unidad sintáctica que se va a analizar, los atributos semánticos especificados en la gramática. Desde este momento se supondrá que los algoritmos descritos en el Capítulo 4, en lugar de *símbolos,* manipulan *estructuras que contienen la información semántica de cada símbolo*.

Esta misma modificación es necesaria en el analizador morfológico. La estructura de cada unidad sintáctica generada por dicho analizador debe incorporar la información semántica que lleva asociada. Por ejemplo, cuando se reconozca el nombre de una variable, el analizador morfológico debe proporcionar dicho nombre, porque es necesario para las acciones semánticas de búsqueda e inserción en la tabla de símbolos; esta información puede sustituirse por un puntero al elemento de la tabla asociado a la variable. De igual manera, cuando se analice una constante, se debe proporcionar también su valor.

5.3.2. Orden de recorrido del árbol de análisis

En las secciones y capítulos anteriores se ha dicho que el analizador sintáctico y la anotación semántica pueden recorrer en un orden determinado el árbol de derivación de los programas compilados. En esta sección se reflexionará con más profundidad sobre las repercusiones que tiene el orden de recorrido en la construcción de compiladores.

Los **analizadores descendentes** siguen el que se conoce como *recorrido en profundidad por la izquierda con vuelta atrás*, lo que significa que, entre todos los nodos posibles, siempre se continúa por uno de los más profundos (*profundidad*). En caso de empate, se toma el nodo que aparezca lo más a la izquierda posible (*por la izquierda*). Siempre que se llegue a una hoja, hay que volver hasta la última decisión tomada y elegir uno de los nodos no visitados (*con vuelta atrás*).

La Figura 5.10 muestra gráficamente el recorrido que se haría siguiendo este orden sobre el árbol de la Figura 5.6.

Obsérvese que los nodos se visitan en el siguiente orden (se resaltan en cursiva los nodos visitados más de una vez): D, T, int, *T*, *D*, L, L, L, i, var1, *i*, *L*, *L*, ‘,’, *L*, i, x, *i*, *L*, *L*, ‘,’, *L*, i, y, *i*, *L*, *D*.

Los **analizadores ascendentes** siguen el orden determinado por la reducción de los asideros que se van encontrando, realizando desplazamientos cuando la reducción no es posible. La Figura 5.11 muestra gráficamente este recorrido para el árbol de la Figura 5.6.

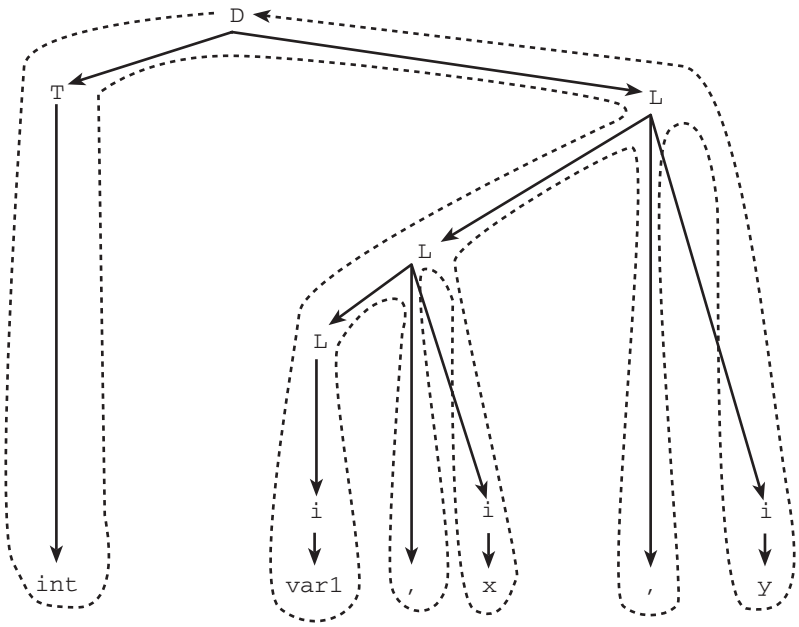


Figura 5.10. Recorrido de un analizador descendente sobre el árbol de análisis de la Figura 5.6.

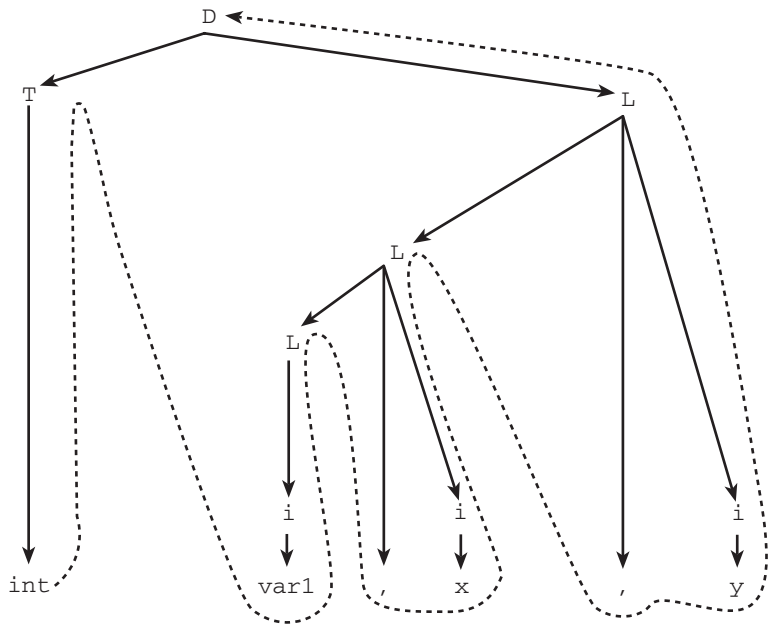


Figura 5.11. Recorrido de un analizador ascendente sobre el árbol de análisis de la Figura 5.6.

Obsérvese que los nodos se visitan en el siguiente orden: `int`, `T`, `var1`, `i`, `L`, ‘`,`’, `x`, `i`, `L`, ‘`,`’, `y`, `i`, `L`, `D`.

En las secciones anteriores de este capítulo se ha señalado que las acciones semánticas definen una relación de dependencia en función de la propagación de los atributos. Esa relación induce el orden en que se puede añadir la información semántica mientras se recorre el árbol. Las Figuras 5.5, 5.6 y 5.7 mostraban gráficamente, mediante flechas, dicha relación. Por lo tanto, al procesar el programa de entrada, se realizarán al menos dos recorridos que requieren un orden determinado. Cabría plantearse las siguientes preguntas: ¿el orden en que se realiza el análisis sintáctico y el exigido por el análisis semántico son siempre compatibles? En caso negativo, ¿es necesario conseguir que lo sean? En caso afirmativo, ¿es siempre posible hacerlos compatibles?

Para contestar a la primera pregunta basta analizar la Figura 5.12. En ella se superponen los órdenes de los analizadores de las Figuras 5.10 y 5.11 al que sugieren las dependencias de los atributos.

En la parte a), que corresponde al analizador descendente, ambos órdenes son compatibles, ya que el recorrido de la flecha discontinua encuentra los nodos en el mismo orden que sugieren las flechas continuas. Sin embargo, en la parte b), que corresponde al analizador ascendente, los órdenes no son compatibles. Aunque el nodo `T` se visita en el orden adecuado (antes que cualquiera de los otros símbolos no terminales), al llegar al nodo `var1`, y posteriormente al `L`, no es posible añadir la información de su tipo porque, a pesar de que es el mismo que el de `T`, lo recibe de su hermano `L` y ese nodo todavía no ha sido visitado.

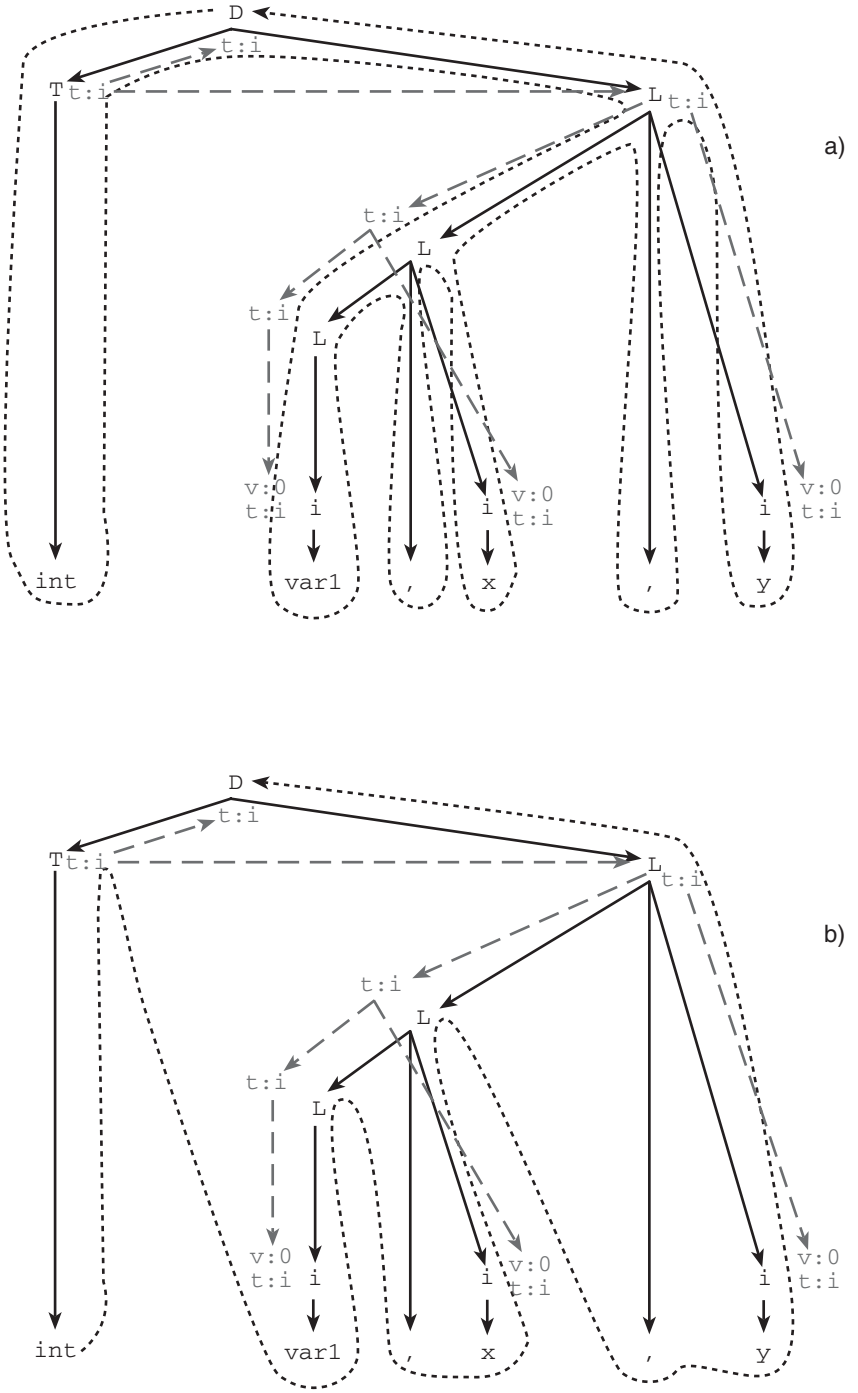


Figura 5.12. Comparación de los órdenes de recorrido del árbol por las dependencias entre los atributos y por los analizadores sintácticos.

Para contestar a la segunda pregunta hay que tener en cuenta el número de pasos del compilador. Si el compilador va a realizar dos o más pasos, no será necesario: en el primer paso, el analizador sintáctico construirá el árbol de análisis; en el segundo, el analizador semántico lo anotará. En las próximas secciones se verá que este esquema da lugar a la técnica más general del análisis semántico. En cambio, si el compilador es de sólo un paso, resulta necesario conseguir que los dos órdenes sean compatibles, ya que, en otro caso, sería imposible realizar el análisis semántico.

Para contestar a la tercera pregunta, es preciso definir algún concepto auxiliar adicional.

5.3.3. Tipos interesantes de gramáticas de atributos

Esta sección introduce algunos subconjuntos interesantes de las gramáticas de atributos:

- **Gramáticas de atributos con atributos sintetizados.** Son aquellas en las que todos los atributos son *sintetizados*. Las gramáticas de atributos A_{5_4}, A_{5_8} y A_{5_10}, de los ejemplos con el mismo número, son de este tipo. Es interesante señalar el caso de la gramática A_{5_10}, en la que se utiliza también información global.
- **Gramáticas de atributos que dependen únicamente de su izquierda.** Son aquellas en las que todos sus atributos son o bien *sintetizados* o bien *heredados de sus padres* o de *símbolos* que aparecen en la parte derecha de la regla *a la izquierda del símbolo estudiado*. Las gramáticas de atributos A_{5_5}, A_{5_6} y A_{5_9}, de los ejemplos con el mismo número, son de este tipo. Es interesante señalar el caso de la gramática A_{5_9}, ya que sus atributos dependen sólo de su izquierda, y es equivalente a la gramática A_{5_10}, que sólo tiene atributos sintetizados e información global.

Cabe ahora plantearse la siguiente cuestión: ¿existen gramáticas de atributos que no pertenezcan a alguno de los dos tipos anteriores? La respuesta a esta pregunta es afirmativa. El hecho de que todos los ejemplos de este capítulo puedan incluirse en alguna de las dos categorías anteriores es puramente casual.

Ejemplo 5.11

La siguiente gramática proporciona un ejemplo en el que se hereda de símbolos a la derecha del estudiado. Es una variante del Ejemplo 5.3, en el que la especificación del tipo se escribe a la derecha de la lista de identificadores. Sus reglas de producción son las siguientes (se ha resaltado la regla modificada):

```
D::=LT
T::=int
T::=real
L::=L,i
L::=i
```

Si se mantiene la semántica del Ejemplo 5.5, es fácil comprobar que el atributo `tipo` de `L` depende del atributo `tipo` de `T` que está a su derecha. Lo extraño de este tipo de construcciones no debe sugerir que no sean posibles, sino constatar la naturalidad con la que se ha incorporado

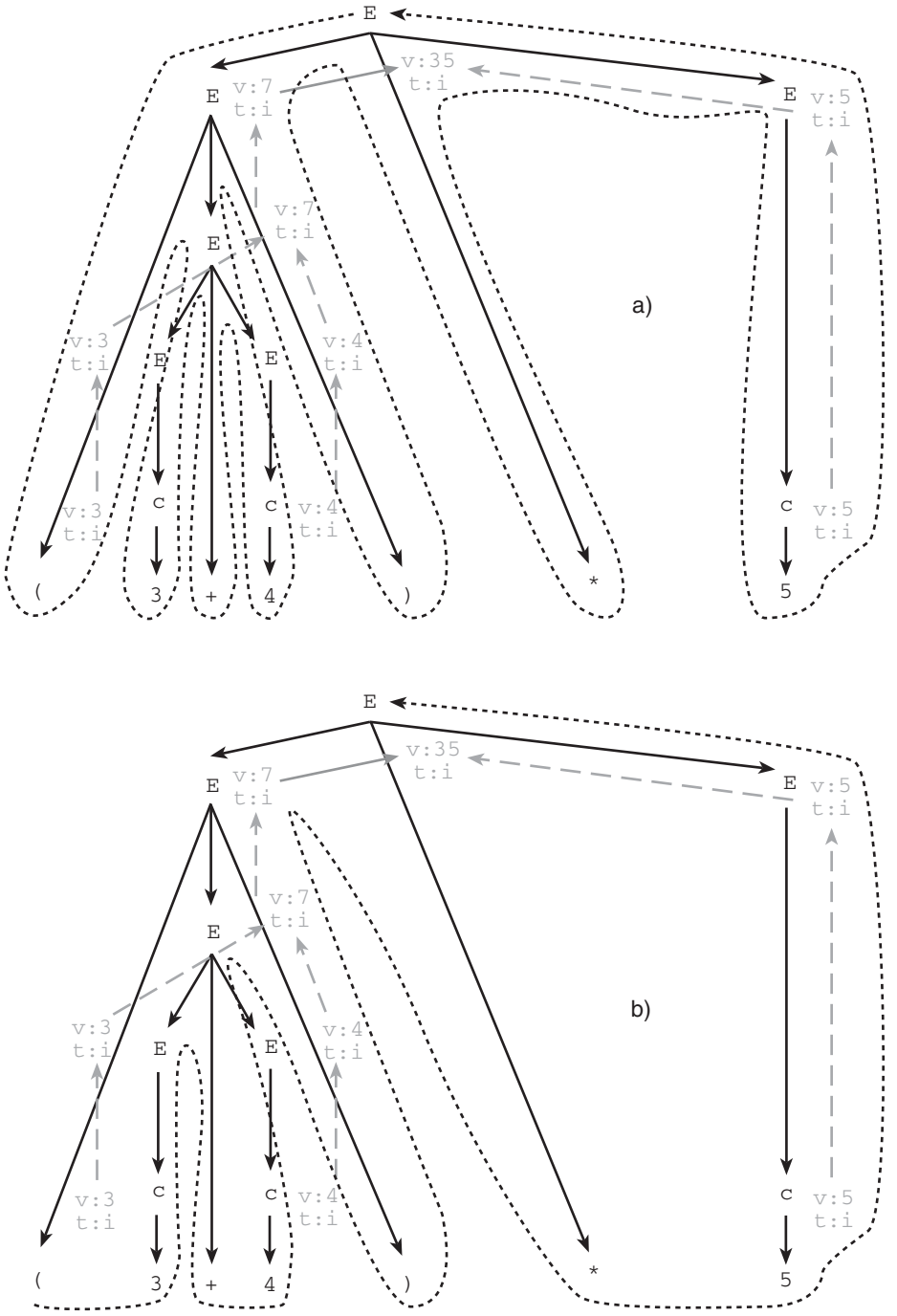


Figura 5.13. Comparación de los órdenes de recorrido del árbol por los analizadores sintácticos y por la propagación de atributos sintetizados.

a nuestra intuición el diseño de lenguajes cuyas gramáticas resultan más adecuadas para la construcción de compiladores e intérpretes.

Las gramáticas de atributos que dependen de su izquierda y las gramáticas de atributos sintetizados pueden representar las construcciones de todos los lenguajes de programación de alto nivel con los que se suele trabajar. Los Ejemplos 5.9 y 5.10 muestran cómo los atributos heredados de la gramática A_{5_9} pueden sustituirse por el uso de información global. Algunos autores distinguen entre *gramáticas de atributos* y *definiciones dirigidas por la sintaxis*, de forma que las primeras son un subconjunto de las segundas que excluyen efectos laterales, es decir, manipulación de información global. De mantener esa categoría, lo que en este capítulo se llama *gramática de atributos* sería lo que otros autores llaman *definición dirigida por la sintaxis*. Como se ha indicado en la nota 1, se ha decidido unificar estos conceptos en el de *gramática de atributos*, para ofrecer una visión más compacta y actual.

El interés de estos tipos de gramáticas de atributos no se limita a la discusión de su potencia expresiva. Son de importancia crucial para la comunicación entre los analizadores sintácticos y semánticos.

- **Gramáticas de atributos que dependen de su izquierda y analizadores descendentes:** Es fácil comprobar que el orden inducido por el recorrido de los dos analizadores es compatible en las gramáticas de este tipo. En este sentido, el ejemplo de la Figura 5.12.a) puede generalizarse. La razón es clara: el analizador descendente visita primero los nodos padre y luego los hermanos, de izquierda a derecha, y ése es precisamente el orden necesario para la propagación de los atributos de las gramáticas con dependencia de su izquierda.

  La síntesis de atributos también es compatible con los analizadores descendentes gracias al mecanismo de *vuelta atrás*. La Figura 5.13.a) muestra gráficamente un ejemplo. El recorrido de los nodos es el siguiente: `E`, `E`, `(`, *E*, `E`, `E`, `c`, `3`, *c*, *E*, *E*, `+`, *E*, `E`, `c`, `4`, *c*, *E*, *E*, *E*, `)`, *E*, *E*, `*`, *E*, `E`, `c`, `5`, *c*, *E*, *E*. La síntesis de los valores y tipos de las expresiones se puede completar cuando, al volver atrás, se visita de nuevo las partes izquierdas de las reglas.

- **Gramáticas con atributos sintetizados y analizadores ascendentes:** Es fácil comprobar que el orden inducido por el recorrido de los analizadores ascendentes y de las gramáticas con atributos sintetizados es compatible. La Figura 5.13.b) muestra gráficamente un ejemplo. La razón es que los analizadores ascendentes desplazan la entrada hasta encontrar un asidero, momento en el que se reduce la regla para continuar el análisis, buscando la siguiente reducción posible. La síntesis de los atributos sólo puede realizarse cuando se tiene seguridad de haber procesado la parte derecha completa de la regla, por lo que la reducción es el momento instante ideal para la propagación.

5.3.4. Técnica general del análisis semántico en compiladores de dos o más pasos

En general, puede presentarse el caso de disponer de una gramática de atributos que no se desea modificar, junto con un analizador sintáctico cuyo orden de análisis sea incompatible con el de

aquélla. En tal caso, es de interés disponer de una técnica general para realizar el análisis semántico. Dicha técnica exige un compilador de dos o más pasos y puede resumirse en el siguiente esquema:

1. Construir el árbol del análisis sintáctico.
2. Determinar las *dependencias entre los atributos* mediante el estudio de las acciones semánticas de la gramática.
3. Determinar un orden entre los atributos del árbol, compatible con las dependencias obtenidas en el paso anterior.
4. Establecer un recorrido del árbol compatible con el orden del paso 3.
5. La ejecución de las acciones semánticas que aparecen al recorrer el árbol según indica el paso 4 completa el análisis semántico del programa compilado.

Sin embargo, todos los ejemplos de este capítulo y la mayoría de los problemas reales pueden solucionarse sin necesidad de utilizar esta técnica general. En los próximos párrafos se verá que la técnica general transforma algunos aspectos del análisis semántico en la solución de un problema clásico de álgebra: la construcción de un grafo que representa una relación y la determinación de un recorrido sobre el grafo, compatible con la relación y que visite todos los nodos.

- **Determinación de las dependencias entre los atributos:** En los casos más sencillos, puede hacerse por simple inspección visual, como se hizo en los ejemplos de las Figuras 5.5, 5.6 y 5.7. También se puede utilizar, como técnica general, un grafo de dependencias. Para ello hay que tener en cuenta las siguientes consideraciones:

  — Las instrucciones de las acciones semánticas pueden representarse de la siguiente manera, que tiene en cuenta únicamente la propagación de los atributos:

  $$b = f(c_1, \ldots, c_n)$$

  donde tanto b como c_i, , i∈{1, ., n} son atributos, y f representa el cálculo mediante el cual, a partir de los valores de $c_1, \ldots, c_n$, se obtiene el de b. En este caso, se dirá que *el atributo b depende de los atributos* $c_1, \ldots, c_n$.

  — Los efectos laterales que pueden modificar la información global de la gramática pueden representarse de la misma manera:

  $$g(K, c_1, \ldots, c_n)$$

  donde K es la información global. Antes de aplicar el algoritmo de creación del grafo, se crea un nuevo atributo ficticio (a) para que la expresión anterior se transforme en la siguiente:

  $$a = g(K, c_1, \ldots, c_n)$$

  que se trata como cualquier otra instrucción.

La Figura 5.14 muestra el pseudocódigo de un algoritmo para la construcción del grafo de dependencias.

```
grafo ConstruirGrafoDependencias
      (arbol as, gramatica_atributos ga)
{
   nodo n; atributo_semántico a; accion_semantica acc;
   grafo gd = vacío;
   `Recorrer cada nodo (n) del árbol sintáctico as´
     `Recorrer cada atributo (a) del símbolo de n´
          AñadirNodo (nuevo_nodo(a), gd);
   `Recorrer cada nodo (n) del árbol sintáctico as´
     `Recorrer acciones (acc=b:=f(c1,...,ck)) de n´
          `Para i de 1 a k´
                AñadirArco (nuevo_arco(ci,b),gd);
}
```

Figura 5.14. Pseudocódigo para la construcción del grafo de dependencias entre los atributos de una gramática.

- **Determinación de un orden compatible con las dependencias:** Lo más frecuente es que esto pueda hacerse directamente sobre el árbol de análisis. Las Figuras 5.15, 5.16 y 5.17 muestran un orden posible para los árboles de las Figuras 5.5, 5.6 y 5.7.

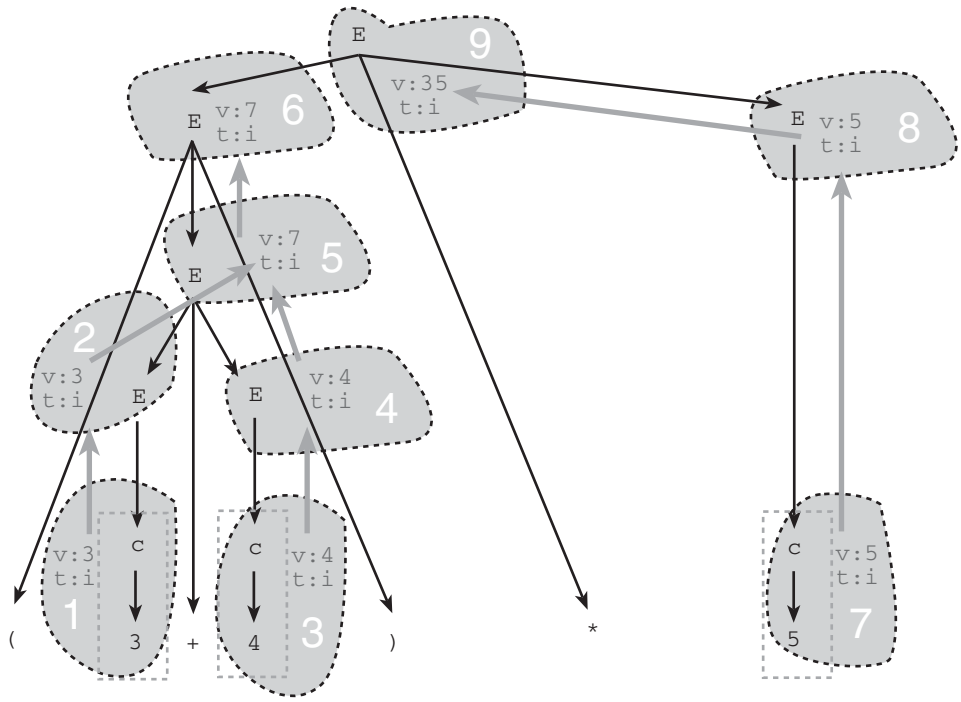


Figura 5.15. Un orden compatible con las dependencias entre atributos del árbol de la Figura 5.5.

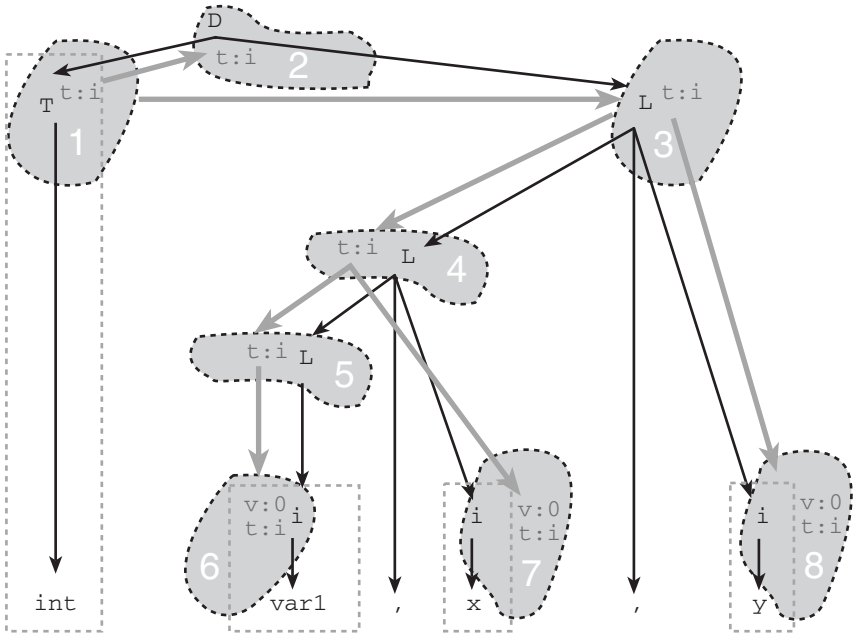


Figura 5.16. Un orden compatible con las dependencias entre atributos del árbol de la Figura 5.6.

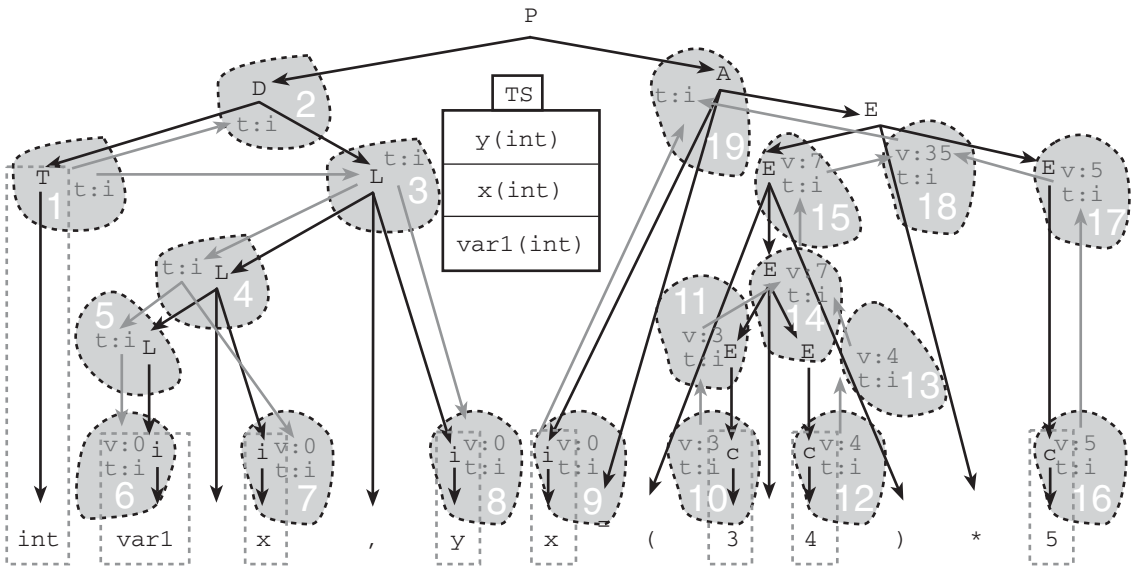


Figura 5.17. Un orden compatible con las dependencias entre atributos del árbol de la Figura 5.7.

- **Dependencias circulares:** Se dice que una gramática tiene *dependencias circulares* cuando existen al menos dos atributos `b` y `c`, e instrucciones en las acciones semánticas con la siguiente estructura (el orden que ocupan los atributos como argumentos de las funciones es irrelevante):

$$b = f_b(c, d_1, \ldots, d_n)$$
$$c = f_c(b, a_1, \ldots, a_m)$$

Esto significa que es posible que exista un árbol en el que aparezca un nodo etiquetado con el símbolo `b` y otro con el símbolo `c`, tal que, de acuerdo con la primera instrucción, `b` tenga que ser analizado antes que `c`, y de acuerdo con la segunda tenga que seguirse el orden inverso.

Las dependencias circulares presentan una dificultad insalvable para el análisis semántico. La única solución es considerar que las gramáticas con dependencias circulares están mal diseñadas, y refinarlas hasta que se elimine el problema. Si las gramáticas de atributos se consideran como un nuevo *lenguaje de programación*, este error de diseño sería similar al de programar, con un lenguaje de programación imperativo, un bucle o una función recursiva sin condición de salida, lo que daría lugar a una ejecución permanente.

5.3.5. Evaluación de los atributos por los analizadores semánticos en los compiladores de sólo un paso

Se ha dicho anteriormente que en el diseño de compiladores de un solo paso es necesario compatibilizar el orden de recorrido inducido por el analizador sintáctico utilizado con el que precisa la relación de dependencia entre los atributos. También se ha dicho que las gramáticas de atributos que dependen de su izquierda aseguran la compatibilidad con los analizadores descendentes, mientras que las que sólo tienen atributos sintetizados aseguran la compatibilidad con los analizadores ascendentes.

Para completar el análisis semántico en este caso, sólo queda describir cómo se puede realizar la evaluación de los atributos. De forma general, se pueden seguir las siguientes indicaciones:

- En los analizadores que utilizan tablas de análisis (por ejemplo ascendentes) se pueden evaluar los atributos cuando se completa la parte derecha de las reglas (en el momento de su reducción). En ese instante se sacan de la pila los símbolos asociados con la parte derecha de la regla (junto con su información semántica) y lo único que hay que añadir al algoritmo es el cálculo de la información semántica del símbolo no terminal de la parte izquierda, antes de ubicarlo en la posición adecuada para continuar el análisis. Este cálculo es posible, ya que los atributos sintetizados sólo necesitan la información semántica de los símbolos de la parte derecha, y esa información está disponible cuando se realiza la reducción.

- Los analizadores que permiten más libertad en la posición de las acciones semánticas (*véase* la Sección 5.2.4) pueden ejecutarlas a medida que las encuentran. Un ejemplo de esta situación son los analizadores descendentes recursivos. La técnica para su construcción, descrita en el Capítulo 4, codificaba una función recursiva para cada símbolo no terminal de la gramática. Lo único que hay que añadir al algoritmo es la codificación de las

rutinas semánticas, e invocarlas en la posición que ocupen en la parte derecha de la regla. La corrección del diseño de la gramática asegurará que se dispone de los valores de todos los atributos necesarios para la ejecución correcta de la acción semántica.

En el desarrollo de cada compilador concreto, siempre es posible extender este modelo, aunque no sea mediante el uso de una técnica general. Todos los analizadores semánticos utilizan una pila semántica para guardar los valores de los atributos de los símbolos. Las técnicas generales proponen un tratamiento estándar de la pila mediante las funciones `push` y `pop`. En el desarrollo de un compilador concreto, sería posible consultar la pila semántica de una manera más flexible, ya que se trata de una estructura propia del analizador. En ese caso, se podría calcular un conjunto más amplio de atributos: todos los que dependan de los valores de los atributos que se encuentran en la pila en un momento dado.

5.4 Gramáticas de atributos para el análisis semántico de los lenguajes de programación

El objetivo de esta sección es sugerir, de la manera más genérica posible, cómo se pueden solucionar los problemas relacionados con las construcciones más frecuentes que se utilizan en los lenguajes de programación de alto nivel. Se supondrá que se está especificando una gramática de atributos sintetizados con información global, pues este tipo de gramáticas es compatible con los compiladores de un solo paso que utilicen analizadores sintácticos, tanto ascendentes como descendentes. El lector será capaz de solucionar otros problemas concretos adaptando el contenido de esta sección mediante el uso de las nociones de la Sección 5.2.5.

A continuación se recuerdan algunos de los aspectos más generales de los lenguajes de programación de alto nivel:

- El tipo de dato *apuntador* o, más popularmente, *puntero*. Algunos lenguajes de programación permiten declarar un tipo de dato que apunta a objetos de otros tipos. Su peculiaridad principal es que sólo ocupa la memoria precisa para contener la dirección a la que apunta, y no depende del tamaño del objeto apuntado. Su existencia permite modificar los datos apuntados mediante uno cualquiera de los punteros que apunten a ellos. El siguiente fragmento de código C muestra un ejemplo:

Ejemplo 5.12

```
int * p_int;
int *** p_p_p_int;
int a;

a = 5;
p_int = &a;
/* Se imprime 5, el valor de a y de *p_int */
printf("%d\n", *p_int);
(*p_int)++;
```

```
/* Se imprime 6, el valor de a y de *p_int */
printf("%d\n", a);
```

En la primera instrucción se declara un apuntador a un dato de tipo entero.

La segunda demuestra la posibilidad de anidar niveles múltiples de *punteros*.

En la quinta instrucción, `p_int` pasa a apuntar al identificador `a` (`&a` es *la dirección del identificador* `a`), de forma que el valor apuntado por el puntero (`*p_int`) es el mismo que el de la variable `a`, como indica la sexta instrucción, y se puede modificar el valor de `a` mediante `p_int`, como demuestran las dos últimas instrucciones.

El tipo de dato apuntador abre la discusión sobre dos formas de gestionar la memoria: la *memoria estática* y la *memoria dinámica*. En la Sección 10.1 se encontrará una descripción detallada de este tema.

- Los *procedimientos*, *funciones* o *subrutinas* presentan dificultades, tanto en su declaración como en su invocación. Entre el analizador semántico y el generador de código, se tienen que gestionar la memoria asignada a las variables automáticas (las variables locales de los procedimientos) y el convenio de llamadas utilizado: el mecanismo mediante el que el programa que invoca comunica al programa invocado el valor de sus argumentos y la manera en la que el procedimiento o función devuelve el control al programa que la invoca, así como el valor de retorno, si existe. En la Sección 10.1 se encontrará una descripción completa de estos aspectos.

5.4.1. Algunas observaciones sobre la información semántica necesaria para el análisis de los lenguajes de programación de alto nivel

Para la gestión de los lenguajes de alto nivel es necesario tener en cuenta cierta información, que podría organizarse de la siguiente manera:

- **Atributos semánticos asociados a los operandos:** Se llama *operando* a los elementos de un programa asociados a un valor, como variables, funciones, etiquetas, etc. Estos datos deben llevar la siguiente información semántica asociada:
  - — Su *nombre*: el identificador por el que se los reconoce.
  - — Su *tipo*: véase, por ejemplo, la Tabla 6.2, en el capítulo siguiente, dedicado a la generación de código.
  - — Su *dirección*: para una variable, puede ser una posición en la memoria, un puntero a la tabla de símbolos, o el nombre de un registro. En variables de tipo *array* indexadas (como en la expresión `v[3]` del lenguaje C) es preciso especificar dos valores: el nombre de la variable y el desplazamiento necesario para localizar el elemento concreto. En el caso de los punteros, puede que baste con especificar su nombre (como en la instrucción `*p` del lenguaje C) o que se necesite también un desplazamiento u *offset*, como en `*(p+4)`.

— Su *número de referencias*: si el lenguaje permite utilizar el tipo de dato *apuntador*, puede ser necesario anotar su nivel de anidamiento.

- **Información global:** Los mecanismos mencionados a continuación afectan a atributos que suelen ser heredados, o almacenados en información global.

  — *Tabla de símbolos*. Tiene que ser accesible desde todas las reglas de la gramática. Desde algunas, será actualizada para insertar nuevos identificadores o modificar la información que se conoce sobre ellos. Desde otras, será consultada para comprobar la corrección del programa.

  — *Lista de registros usados*. Se utiliza en la generación de código y se refiere a los atributos relacionados con la dirección donde está almacenado un objeto, para el caso de los que ocupan registros. Para asegurar la corrección del programa objeto resultado de la compilación, el analizador semántico tiene que proporcionar mecanismos para que los registros no sean modificados por error, cuando su información todavía es útil.

  — *Información para la gestión de etiquetas*. Como se explicará en la Sección 6.1, en el capítulo sobre generación de código, la estructura básica de control de flujo en los lenguajes simbólicos (ensambladores) y de la máquina es el salto, tanto condicional como incondicional, a una dirección o *etiqueta*, situada en el espacio de instrucciones. Utilizando únicamente saltos a etiquetas, un compilador que genere código simbólico o de máquina tiene que generar código equivalente a las estructuras de control del flujo de programa que se utilizan en los lenguajes de alto nivel (instrucciones condicionales, bucles, etc.). El hecho de que las etiquetas tengan que tener un nombre único dentro del código, junto con la posibilidad de mezclar estructuras o estructuras anidadas en el programa fuente (como instrucciones del tipo *if-then-else* dentro de otras estructuras *if-then-else*), obliga a articular mecanismos para controlar que las etiquetas sean distintas y que los saltos que conducen a ellas sean coherentes.

  — *Información para la gestión del tipo de los identificadores*. En los lenguajes de programación se dan dos circunstancias frecuentes que suelen requerir atributos heredados o información global para su representación con gramáticas de atributos. La primera es la gestión del tipo de los identificadores en las instrucciones en las que se puede declarar una lista de variables del mismo tipo en una sola instrucción. La segunda es la declaración de los argumentos de las funciones.

Las próximas secciones resumen de manera intuitiva la gestión semántica asociada con las construcciones más frecuentes de los lenguajes de programación. Se puede encontrar un ejemplo completo de la aplicación de estas ideas a la construcción de un compilador para un lenguaje sencillo en `http://www.librosite.net/pulido`

5.4.2. Declaración de identificadores

La mayoría de los lenguajes de programación proporcionan una sintaxis para la declaración de identificadores, semejante a la siguiente regla de producción:

```
<declaracion> ::= <tipo> <identificador>
```

Algunos permiten especificar una lista de identificadores en lugar de uno solo. El analizador semántico tendrá que encargase de las siguientes tareas:

- Tras procesar el símbolo no terminal `<tipo>`, el analizador morfológico debe proporcionar, como valor de su atributo, el tipo de dato que se está declarando. Esta información tendrá que propagarse, modificando tal vez de forma adecuada la información global correspondiente, para que esté accesible más tarde, cuando se haya procesado el nombre del identificador.
- Tras procesar el símbolo no terminal `<identificador>`, debe consultarse la tabla de símbolos para comprobar que no se ha declarado previamente la misma variable, recuperar el tipo de la declaración, e insertar el identificador nuevo en la tabla de símbolos.
- Los distintos lenguajes de programación facilitan diversos tipos de datos (*arrays*, apuntadores), junto con las condiciones que tienen que satisfacer para su declaración correcta. La acción semántica de esta regla debe gestionar toda la información necesaria para cumplir esas condiciones.

5.4.3. Expresiones aritméticas

Las expresiones aritméticas son, posiblemente, la parte del lenguaje que resulta más compleja para el análisis semántico y la generación de código. El analizador semántico tiene que asegurar que se satisfacen las restricciones de tipo de los operandos que aparecen en las expresiones. Esto supone aplicar las normas de transformación de tipos compatibles (enteros de distinto tamaño, reales de distinta precisión, transformaciones entre los tipos entero y real, etc.). Es tarea del generador obtener un programa objeto que evalúe correctamente las expresiones.

La Sección 6.1.2 explica un procedimiento general para la gestión de estos dos aspectos. Este tipo de generación requiere que el analizador semántico mantenga y actualice la información relativa al tipo, número de referencias y dirección de almacenamiento de los operandos, así como la lista de registros disponibles.

Una de las dificultades para la generación de código para calcular expresiones es la determinación del lugar donde se almacenarán los resultados intermedios. El sistema ofrece un conjunto limitado de registros. El analizador semántico tendrá que comprobar si quedan registros disponibles, y en caso contrario localizar espacio en la pila o en la memoria estática. Una vez asignada memoria a un resultado intermedio, tendrá que actualizar la información semántica correspondiente para continuar el análisis adecuadamente.

5.4.4. Asignación de valor a los identificadores

La mayoría de los lenguajes de programación proporcionan una sintaxis para la asignación de valor a los identificadores, semejante a la siguiente regla de producción:

`<asignacion> ::= <identificador> ← <expresion>`

El analizador semántico tendrá que encargase de las siguientes tareas:

- El analizador morfológico propaga la información semántica del nombre del identificador.
- Tras procesar el símbolo `<identificador>`, hay que consultar la tabla de símbolos para comprobar que ya ha sido declarado y recuperar su información semántica asociada, que incluye su tipo.
- Tras procesar el símbolo no terminal `<expresion>`, habrá que comprobar la compatibilidad entre los tipos de la expresión y el identificador, para que la asignación sea correcta.

5.4.5. Instrucciones condicionales

La mayoría de los lenguajes de programación utilizan la estructura de instrucciones condicionales indicada por la siguiente regla de producción:

```
<condicional>::=if <expresion> then <instruccion>
```

Como se ha mencionado anteriormente, al generar código en lenguaje simbólico, hay que construir un fragmento de programa objeto equivalente con saltos a etiquetas, lo que obliga a que el analizador semántico gestione las etiquetas de manera adecuada. A continuación se muestra el esquema de un fragmento de código simbólico equivalente a esta instrucción condicional:

```
        ...
; Código de la expresión (el valor está en el registro EAX)

CMP EAX, 0
JE FIN_THEN
        ...
; Código de la instrucción de la rama 'ENTONCES'
        ...
FIN_THEN:
```

Bastará con que la instrucción de la rama `then` contenga otra instrucción condicional para que las dos etiquetas **`FIN_THEN`** colisionen, originando un error en el programa. Este problema puede solucionarse fácilmente si se añade a la etiqueta algún carácter que la haga única. Esta solución obliga al analizador semántico a conservar la información necesaria para que cada estructura tenga accesible la etiqueta adecuada. También tendría que asegurar que la expresión que se comprueba en la condición es del tipo permitido por las especificaciones del lenguaje de programación (entero, booleano, etc.).

La mayoría de los lenguajes de programación facilitan instrucciones más potentes, como el *if-then-else*, que se tratará de manera análoga:

```
<condicional>::=if <expresion> then <instruccion>
                    else <instruccion>
```

5.4.6. Instrucciones iterativas (bucles)

Casi todos los lenguajes de programación proporcionan instrucciones iterativas con una sintaxis parecida a la de la siguiente regla de producción:

```
<bucle> ::= while <expresion> do <instruccion>
```

Estas instrucciones presentan las mismas peculiaridades que las condicionales, tanto en lo relativo a las etiquetas, como en lo referente a las comprobaciones de tipo de la condición.

5.4.7. Procedimientos

La mayoría de los lenguajes de programación permiten declarar funciones y subrutinas con una sintaxis parecida a la de la siguiente regla de producción:

```
<subrutina>::=<tipo><identificador>(<lista_argumentos>)
              <declaracion> <instruccion>
```

El analizador semántico se encargará de realizar las siguientes comprobaciones:

- El tipo y el identificador se tratan de manera análoga a la de la declaración de variables, excepto que se tiene que indicar en la tabla de símbolos que el identificador representa una función.
- La declaración de la lista de argumentos implica actualizar la información que se conserva al respecto, que tiene que ser accesible en el momento de la invocación del procedimiento. Lo más frecuente es incluirla en la tabla de símbolos.

La mayoría de los lenguajes de programación realizan llamadas o invocaciones a los procedimientos con una sintaxis similar a la de la siguiente regla de producción:

```
<expresion>::=<identificador>(<lista_expresiones>)
```

El analizador semántico debe realizar las siguientes tareas:

- Tras procesar el símbolo `<identificador>`, cuyo nombre debe propagarse, hay que comprobar en la tabla de símbolos que se ha declarado un procedimiento con ese nombre, y recuperar la información que describe la lista de sus argumentos.
- Tras procesar el paréntesis de cierre, se podrá completar la verificación de la correspondencia del número, tipo y orden de los argumentos de la invocación y los de la declaración.

5.5 Algunas herramientas para la generación de analizadores semánticos

En la actualidad se encuentran disponibles en Internet, de forma gratuita, diversas herramientas que generan analizadores sintácticos a partir de gramáticas en BNF que cumplan ciertas condiciones, y que a veces pueden generar también el analizador semántico, si se les proporciona una

gramática de atributos. Una de las más populares y conocidas es *yacc* (***y**et **a**nother **c**ompiler **c**ompiler*), objetivo de esta sección. Yacc está incluida en las distribuciones estándares de Unix y Linux. Otra herramienta, compatible con yacc, es *Bison*, que está disponible tanto para Windows como para Linux. Aunque lo que se diga en esta sección es aplicable tanto a yacc como a Bison, por motivos históricos sólo se hará referencia a yacc.

La aplicación yacc toma como entrada un fichero, que contiene una gramática con atributos sintetizados e información global, y genera una aplicación escrita en el lenguaje de programación C, que implementa un analizador sintáctico ascendente LALR(1), junto con el correspondiente analizador semántico, aunque excluye el analizador morfológico, que debe obtenerse de manera independiente.

Esta sección no tiene por objeto proporcionar un manual exhaustivo de la herramienta, sino sugerir indicaciones prácticas para comprobar el funcionamiento de algunas de las gramáticas de atributos utilizadas en el capítulo y para que, posteriormente, el lector pueda beneficiarse de su ayuda en el diseño de las mismas. Puede encontrarse documentación más detallada sobre yacc, tanto en Internet (http://www.librosite.net/pulido), como en la bibliografía especializada [1].

El resto de la sección explicará la estructura básica del fichero fuente de yacc, las directivas que se utilizan para describir la gramática de atributos de entrada, la notación con la que deben escribirse las reglas y las acciones semánticas, así como algunas consideraciones prácticas fundamentales para trabajar con yacc. Estos conceptos se ilustrarán mediante la solución con yacc de los ejemplos A_{5_8} y A_{5_10} de gramáticas de atributos.

5.5.1. Estructura del fichero fuente de yacc

El fichero de entrada para yacc tiene tres secciones separadas por una línea, que sólo contiene los caracteres %%, sin espacios a la izquierda. Las tres secciones son:

- La sección de definiciones.
- La sección de reglas.
- La sección de código de usuario.

La Figura 5.18 muestra un esquema del fichero de entrada de yacc.

Sección de definiciones:

```
%{
     /* delimitadores de código C */
%}

%%
```

Sección de reglas:

```
%%
```

Sección de funciones de usuario

Figura 5.18. Estructura del fichero de entrada para yacc.

5.5.2. Sección de definiciones

En esta sección se incluyen las definiciones propias de `yacc` y las declaraciones escritas en lenguaje C que necesite el analizador semántico. Estas últimas deben separarse de las primeras, delimitándolas mediante dos líneas que sólo contienen, respectivamente, los caracteres `%{` y `%}`, sin espacios a la izquierda (*véase* la Figura 5.18). Entre estas dos líneas se escribirán las instrucciones declarativas necesarias, escritas en C.

En la parte de declaraciones propias de `yacc` se especifican aspectos generales, mediante el uso de *directivas*.

- **Directiva `%union`:** permite declarar los atributos que se van a asociar con los símbolos de la gramática. Su efecto es equivalente a declarar una estructura de datos de tipo `union` en el lenguaje C.
- **Directiva `%type`:** asocia los atributos especificados mediante la directiva `%union` con los símbolos no terminales de la gramática. Por lo tanto, tiene que haber una directiva `%type` por cada símbolo que posea atributos semánticos.
- **Directiva `%token`:** define los símbolos terminales de la gramática que no están representados literalmente. Puede utilizarse también para indicar los atributos semánticos asociados a los símbolos terminales, con la misma notación que la directiva anterior.
- **Directiva `%start`:** especifica el axioma de la gramática. Es opcional; si no se utiliza, el axioma será la parte izquierda de la regla que aparece en primer lugar en el fichero de entrada de `yacc`.

Ejemplo 5.13 A lo largo de esta sección se va a solucionar con `yacc` la gramática de atributos A_{5_8}. A ello se dedicará el Ejemplo 5.12, utilizándose siempre el mismo número de ejemplo en los pasos sucesivos de la escritura del fichero fuente `yacc` para dicha gramática.

En este ejemplo, en las acciones semánticas y en otras funciones de usuario, se escribirán mensajes por la salida estándar mediante la instrucción

```
fprintf(stdout, "...");
```

Para poder utilizar la función `fprintf`, es preciso incluir el archivo de definiciones `stdio.h`. Esto tiene que hacerse en la sección de declaraciones, en la parte reservada a las instrucciones en lenguaje C. Por tanto, el fichero fuente para este ejemplo debe contener la siguiente sección de declaración C:

```
%{
#include <stdio.h>
%}
```

Por otra parte, todos los símbolos de la gramática A_{5_8} utilizan el mismo atributo de tipo entero, llamado `profundidad`, por lo que la directiva `%union` necesaria es la que se muestra a continuación:

```
%union
{
      int profundidad;
}
```

Para especificar que todos los símbolos no terminales de la gramática A_{5_8} tienen el atributo `profundidad`, tienen que especificarse las siguientes instrucciones:

```
%type <profundidad> lista
%type <profundidad> lista_interna
```

Esta gramática no asigna atributos semánticos a los símbolos terminales, por lo que no se necesita la directiva `%token`. Por otra parte, el axioma es el símbolo no terminal `lista`. Se puede añadir la siguiente instrucción:

```
%start lista
```

Uniendo las componentes anteriores, la sección de definiciones del archivo de entrada para la gramática A_{5_8} será:

```
%{
#include <stdio.h>
%}
%union
{
      int profundidad;
}
%type <profundidad> lista
%type <profundidad> lista_interna
%start lista
```

Ejemplo 5.14 Como segundo ejemplo, se preparará el fichero fuente `yacc` para la gramática A_{5_10}. En este caso también se utilizará el archivo de definiciones `stdio.h`. También se debe inicializar la información global de la gramática. En este ejemplo, se utilizará una variable global de tipo entero para representar el número de elementos, que inicialmente tomará el valor 0. La sección de declaraciones en lenguaje C será:

```
%{
#include <stdio.h>
int num_elementos = 0;
%}
```

En la gramática A_{5_10} los símbolos no tienen atributos, porque sólo se utiliza información global. Por tanto, no se necesita la directiva `%union`. Por la misma razón, tampoco se precisa la di-

rectiva `%type`. Tampoco se asignan atributos semánticos a los símbolos terminales, por lo que no se utilizará la directiva `%token`. Finalmente, el axioma es el símbolo no terminal `lista`. Se puede utilizar la siguiente instrucción:

```
%start lista
```

Uniendo las componentes anteriores, la sección de definiciones del archivo de entrada para la gramática A_{5_10} será:

```
%{
#include <stdio.h>
int num_elementos = 0;
%}
%start lista
```

5.5.3. Sección de reglas

- **Sintaxis para las reglas de producción independientes del contexto:** Se utiliza una notación parecida a la notación BNF. Por ejemplo, la regla $X ::= Y_1 Y_2 \ldots Y_n$ se escribe de la siguiente forma:

  $X : Y_1 Y_2 \ldots Y_n$

  donde los espacios alrededor del símbolo '`:`' son opcionales. Cuando en la regla aparecen símbolos terminales, pueden escribirse literalmente si se cumplen las siguientes condiciones:

- Su longitud es igual a 1, como en los símbolos de apertura y cierre de paréntesis, `(` y `)`.

- El analizador morfológico los devuelve literalmente, mediante instrucciones del tipo

  ```
  return '(';
  return ')';
  ```

- Se utiliza la notación del lenguaje C para representar caracteres. Por ejemplo:

  ```
  lista: '(' lista_interna ')'
  ```

Es posible utilizar otros mecanismos de comunicación entre los analizadores morfológico y sintáctico. Por ejemplo, se podría haber definido el símbolo `INICIO_LISTA` en lenguaje C, para representar el símbolo terminal '`(`'. Para ello, tendrían que haberse realizado las siguientes acciones:

- El analizador semántico tiene que saber que existe un símbolo terminal llamado `INICIO_LISTA`. Eso se hace en la sección de declaraciones, con la directiva `%token`, de la siguiente manera:

  ```
  %token INICIO_LISTA
  ```

Gracias a esto, en el programa C generado por `yacc` se definirá un símbolo con este nombre. Cuando encuentre el carácter '`(`', el analizador morfológico tiene que devolver al analizador semántico este símbolo, por ejemplo, así:

```
if((c=fgetc(stdin))=='(')return INICIO_LISTA;
```

- En las reglas de producción se utilizará el nombre del símbolo en los lugares donde aparecía el símbolo terminal '`(`'.

```
lista: INICIO_LISTA lista_interna ')'
```

Para especificar una regla-λ, es suficiente omitir la parte derecha de la regla. Así, la regla `lista_interna::=`λ se escribiría en `yacc` así:

```
lista_interna:
```

Las reglas deben separarse mediante el símbolo '`;`'. Por ejemplo:

```
lista_interna:    lista_interna '(' lista_interna ')' ;
lista_interna:    ;
```

Cuando varias reglas comparten la misma parte izquierda, puede utilizarse el símbolo '`|`' para separar sus partes derechas, como en la notación BNF. El ejemplo anterior podría escribirse también de la siguiente manera:

```
lista_interna:   lista_interna '(' lista_interna ')' ;
        |    ;
```

- **Sintaxis para las acciones semánticas:** Las acciones semánticas asociadas a una regla se escriben a continuación de ésta encerradas entre llaves. En el caso de que una regla no tenga ninguna acción semántica asociada, debe escribirse

```
{}
```

Dentro de la acción semántica se escriben instrucciones en el lenguaje C. En ellas, pueden aparecer los símbolos de la tabla 5.1, que tienen significado especial para `yacc`, y que permiten acceder a la información semántica de la gramática. Se supone, en la tabla, que se está describiendo la regla `X : `Y_1 Y_2`...`Y_n).

Tabla 5.1. Símbolos que se pueden utilizar en las acciones semánticas asociadas a las reglas `yacc`.

Símbolo	Significado
`$$`	Valor semántico de `X`
`$1`	Valor semántico de Y_1
...	...
`$n`	Valor semántico de Y_n (`n` tiene que ser un número)

Ejemplo 5.15 A continuación se muestra la sección de reglas `yacc` para la gramática A_{5_8}.

```
lista: '(' lista_interna ')'
       {
          $$ = $2 + 1;
          fprintf(stdout, "PROF. TOTAL= %d\n", $$);
       } ;
lista_interna: lista_interna '(' lista_interna ')'
        {
          if ( $1 > $3 )
              $$ = $1 ;
          else
              $$ = $3+1 ;
        } ;
lista_interna:
        {
          $$ = 0;
        } ;
```

Ejemplo 5.16 A continuación se muestra la sección de reglas `yacc` para la gramática A_{5_10}.

```
lista: '(' lista_interna ')'
       {
          num_elementos ++;
          fprintf(stdout, "NUM. LISTAS= %d\n",
               num_elementos );
       } ;
lista_interna: lista_interna '(' lista_interna ')'
       {
          num_elementos++;
       } ;
lista_interna:
       {
       } ;
```

5.5.4. Sección de funciones de usuario

En esta sección, el programador debe incluir las funciones escritas en lenguaje C que considere oportunas y que puedan utilizarse en las acciones semánticas. Entre ellas, las más significativas son las siguientes:

- La función `int main()`: Es el programa principal y debe invocar a una función llamada `yyparse`, que realiza el análisis semántico completo.
- La función `int yylex()`: `Yacc` supone que esta función realiza el análisis morfológico. Sólo hay que tener en cuenta que tiene que devolver el valor 0 cuando se localiza el final

del fichero de entrada, y que las unidades sintácticas se representan mediante números enteros. En la dirección de Internet `http://www.librosite.net/pulido` se dan indicaciones sobre el uso de unidades sintácticas de estructura más compleja.

Ejemplos 5.15 y 5.16

En ambos casos, puede utilizarse el siguiente código:

```
int main()
{
     return( yyparse());
}

int yylex()
{
     int c;
     c=fgetc(stdin);
     while (( c != EOF ) &&
            ( c != '(' ) &&
            ( c != ')' )
            )
          c = fgetc(stdin);
     if  ( c == EOF ) return 0;
     else return c;
}
```

5.5.5. Conexión entre `yacc` y `lex`

Ya se ha explicado en la Sección 3.9 que `lex` es una herramienta, similar a `yacc`, que genera automáticamente analizadores morfológicos. La función que genera `lex` para que se pueda invocar el analizador se llama `int yylex()`. La coincidencia de nombres no es casual, pues es muy frecuente utilizar `lex` para generar el analizador morfológico empleado por `yacc`.

En la dirección de Internet `http://www.librosite.net/pulido` está disponible una explicación del uso de estas herramientas para la implementación de un compilador completo.

5.6 Resumen

Tras el estudio de este capítulo, el lector será capaz de abordar el desarrollo de un analizador semántico y de incorporarlo al compilador o intérprete del lenguaje de programación estudiado. Para ello, se describen previamente los objetivos generales del analizador semántico y su relación con el resto de las componentes de un compilador o de un intérprete, tanto en los casos de análisis en un paso como en los de dos o más pasos.

Posteriormente se dedica una sección a la descripción detallada de la herramienta más utilizada en el análisis semántico: las gramáticas de atributos. La sección comienza con una presentación informal, mediante ejemplos, de las extensiones que hay que añadir a las gramáticas independientes para que puedan hacerse cargo del análisis semántico. Así surgen de forma natural los conceptos de *atributo semántico* y las diferentes formas de calcular sus valores, a saber, *síntesis* y *herencia*. Como consecuencia de esto se descubre la existencia de una relación de dependencia entre los atributos, que sugiere un orden en el recorrido del árbol de análisis para poder completar el proceso de anotación semántica. Tras esta introducción informal, se da una definición formal de los conceptos presentados previamente, lo que completa la explicación de las gramáticas de atributos.

A continuación se analizan distintas técnicas para el diseño de las gramáticas de atributos para la solución de problemas concretos. Mediante ejemplos de fácil comprensión, se resaltan las condiciones que tiene que cumplir un problema para que se pueda resolver con cada uno de los tipos de atributos estudiados: sintetizados y heredados. Se indica cómo se pueden sustituir los atributos heredados por información global a la gramática. Aunque los ejemplos son casos particulares sencillos, las conclusiones se pueden generalizar.

Tras explicar qué son y cómo funcionan las gramáticas de atributos, se dedican dos secciones a su uso en el análisis semántico de los lenguajes de programación. La primera describe cómo se conecta una gramática de atributos con los analizadores sintácticos para completar el analizador semántico, tanto en el caso de los analizadores ascendentes como en el de los descendentes. La segunda sección analiza las dificultades asociadas a las construcciones más frecuentes de los lenguajes de programación de alto nivel, y cómo se solucionan mediante una gramática de atributos.

La última sección del capítulo describe `yacc`, una herramienta de libre distribución que genera automáticamente analizadores sintácticos y semánticos a partir de la descripción de su gramática de atributos. El objetivo de esa sección es dotar al lector de una herramienta que ayuda a comprobar la corrección de las gramáticas de atributos.

5.7 Bibliografía

[1] Levine, J. R.; Mason, T., y Brown, D.: *Lex & yacc. Unix Programming Tools*, O'Reilly & Associates, Inc., 1995.

5.8 Ejercicios

1. Construir una gramática de atributos que represente el lenguaje de los números en punto flotante del tipo `[-][cifras][.[cifras]][e[-][cifras]]`. Debe haber al menos una cifra en la parte entera o en la parte decimal, así como en el exponente, si lo hay. La gramática de atributos tiene que ser capaz de calcular el valor del número.

2. Construir una gramática de atributos que represente el lenguaje de las cadenas de caracteres correctas en el lenguaje de programación C. La gramática de atributos tienen que ser capaz de almacenar en una variable auxiliar global la cadena procesada.

3. Diseñar una gramática de atributos para expresiones aritméticas en las que los operadores son la división (`/`), la suma (`+`) y el producto (`*`). Los operandos pueden ser letras del alfabeto. La gramática de atributos tiene que gestionar el tipo de las expresiones. Para ello aplicará las siguientes reglas:

- Las letras del alfabeto se supone que representan variables declaradas como enteras.
- Las sumas y productos tienen el mismo tipo que sus operandos; en el caso de mezclar enteros y reales, la expresión completa será de tipo real.
- La división genera una expresión de tipo real independientemente del tipo de sus operandos.
- Construir el árbol de propagación de atributos en el anál0isis semántico de la siguiente expresión:

```
a/(b+c*d)
```

Capítulo 6

Generación de código

El módulo de generación de código de un compilador tiene por objeto generar el código equivalente al programa fuente escrito en un lenguaje diferente. En función del tipo de lenguaje objetivo, se distinguen distintos tipos de compiladores:

- ***Cross-compilers* (compiladores cruzados):** traducen de un lenguaje de alto nivel a otro lenguaje de alto nivel.
- **Compiladores que generan código en lenguaje simbólico**: generan un código intermedio que después deberá ser procesado por un ensamblador.
- **Compiladores que generan código en lenguaje de la máquina:** generan directamente programas ejecutables (*.EXE) o bien (esto es mucho más frecuente) en un formato especial de código máquina (*.OBJ) que contiene información adicional, y que después será procesado por un programa enlazador (*linker*), que generará el programa ejecutable a partir de uno o más programas en formato OBJ, algunos de los cuales pueden estar contenidos en bibliotecas (*libraries*) que suelen proporcionarse junto con el compilador, y que contienen funciones y subrutinas prefabricadas de uso general.

En este capítulo se va a suponer que el compilador genera código simbólico (ensamblador), pues los ejemplos resultan mucho más legibles, pero todo lo que se diga podrá aplicarse a cualquier otro tipo de compilador. En los ejemplos se supondrá que el código generado es comprensible por un ensamblador típico aplicable a la familia 80x86 a partir del microprocesador 80386, en modo de funcionamiento de 32 bits (*véase* la Sección 10.1).

6.1 Generación directa de código ensamblador en un solo paso

Una instrucción del ensamblador genérico para el 80x86 que vamos a utilizar tiene la siguiente sintaxis:

```
etiqueta: código_instrucción operandos
```

- Los operandos de las instrucciones pueden ser registros, direcciones, constantes o expresiones.
- Existen distintos tipos de registros. En los ejemplos de esta sección se utilizarán los siguientes registros de 32 bits: EAX (acumulador), EBX, ECX, EDX, ESP (puntero a la pila), EBP, ESI y EDI.
- La dirección donde se almacena el valor de una variable se representará anteponiendo el símbolo de subrayado (_) al nombre de la variable.
- Para referirse al contenido de una posición de memoria, es necesario encerrar su dirección entre corchetes. Por ejemplo, la instrucción `mov eax,[_x]` carga el contenido de la variable `x` (de dirección `_x`) en el registro `EAX`.
- Cuando una instrucción, por ejemplo `mov`, hace referencia a una posición de memoria y a un registro, el tamaño de la posición de memoria se adapta por omisión al tamaño del registro. En cambio, si hace referencia a dos posiciones de memoria, es preciso especificar el tamaño de la zona de memoria afectada por la instrucción. Por ejemplo, si se trata de una doble palabra, se usará el indicador `dword`.
- La instrucción `mov op1,op2` copia el contenido del segundo operando en el primero. Por ejemplo, la instrucción `mov eax,ebx` copia el contenido del registro `EBX` en el registro `EAX`.
- La instrucción `fld operando`, donde el operando es una variable en punto flotante, introduce el contenido del operando en la primera posición de la pila de registros en punto flotante y empuja hacia abajo los contenidos anteriores de dicha pila.
- La instrucción `fstp operando`, donde el operando es una variable en punto flotante, extrae de la pila de registros en punto flotante el contenido de la primera posición de la pila (y lo elimina de ella) y lo almacena en el operando.
- La instrucción `fild operando`, donde el operando es una variable entera, convierte el valor del operando a punto flotante, introduce el resultado en la primera posición de la pila de registros en punto flotante y empuja hacia abajo los contenidos anteriores de dicha pila.
- La instrucción `fistp operando`, donde el operando es una variable entera, extrae de la pila de registros en punto flotante el contenido de la primera posición de la pila (y lo elimina de ella), convierte dicho valor al tipo entero y lo almacena en el operando.
- Instrucciones de manejo de la pila: usualmente, la pila de una aplicación de 32 bits gestiona datos con tamaño de dobles palabras. Al introducir datos nuevos, la pila se extiende hacia posiciones de memoria con direcciones más pequeñas, por lo que el valor del puntero a la pila (el registro `ESP`) disminuye en cuatro unidades cuando se introduce un dato en la pila y aumenta en cuatro unidades cuando se extrae un dato de la pila.
  - — La instrucción `push` *operando* resta 4 al contenido del registro `ESP` (puntero a la pila) y a continuación inserta el operando en la pila.

  - La instrucción `pop` *operando* copia el contenido que está situado en la cima de la pila (es decir, en la dirección más baja) sobre el operando, y a continuación suma 4 al contenido del registro `ESP`. Por ejemplo, la instrucción `pop eax` almacena el contenido de la cima de la pila en el registro `EAX`.

- Instrucciones aritméticas.

  - La instrucción `add op1,op2` suma los dos operandos enteros y almacena el resultado en el primero.
  - La instrucción `sub op1,op2` resta el segundo operando entero del primero y almacena en el resultado en el primero.
  - La instrucción `mul operando`, donde el operando es un entero con un tamaño de 32 bits, lo multiplica por el contenido de `EAX`. El resultado se almacena en la concatenación de los registros `EDX` y `EAX`.
  - La instrucción `div operando`, donde el operando es un entero con un tamaño de 32 bits, divide la concatenación de los registros `EDX` y `EAX` por el operando. El cociente se almacena en el registro `EAX`, y el resto de la división en el registro `EDX`.
  - Las instrucciones `fadd operando`, `fsub operando`, `fmul operando` y `fdiv operando`, donde el operando es una variable en punto flotante, suman, restan, multiplican o dividen (respectivamente) el operando con el contenido de la primera posición de la pila de registros en punto flotante y almacenan el resultado en la misma posición de la pila.
  - La instrucción `neg operando` sustituye el contenido del operando por el complemento a dos de su valor original (es decir, le cambia el signo).

- Instrucciones lógicas.

  - La instrucción `and op1,op2` lleva a cabo la operación lógica AND, bit a bit, entre los dos operandos, y almacena el resultado en el primero.
  - La instrucción `or op1,op2` lleva a cabo la operación lógica OR, bit a bit, entre los dos operandos, y almacena el resultado en el primero.
  - La instrucción `xor op1,op2` lleva a cabo la operación lógica XOR, bit a bit, entre los dos operandos, y almacena el resultado en el primero.

- La instrucción de comparación `cmp op1,op2` resta el segundo operando entero del primero, sin almacenar el resultado en ningún sitio. La operación afecta a los indicadores (*flags*) de la unidad aritmético-lógica, como si la operación se hubiera realizado realmente. El contenido de estos indicadores puede utilizarse posteriormente por instrucciones de salto.

- La instrucción `fcmp operando`, donde el operando es una variable en punto flotante, resta el operando del contenido de la primera posición de la pila de registros en punto flotante y modifica adecuadamente los indicadores, sin almacenar el resultado en ningún sitio.

- Instrucciones de salto.

  - La instrucción `jmp etiqueta` (salto incondicional) salta a la dirección especificada por la etiqueta.

— Después de una instrucción de comparación `cmp op1,op2` pueden aparecer las siguientes instrucciones de salto condicional:

`je etiqueta` salta a `etiqueta` si `op1` es igual a `op2`.
`jne etiqueta` salta a `etiqueta` si `op1` es distinto de `op2`.
`jl etiqueta` salta a `etiqueta` si `op1` es menor que `op2`.
`jle etiqueta` salta a `etiqueta` si `op1` es menor o igual que `op2`.
`jg etiqueta` salta a `etiqueta` si `op1` es mayor que `op2`.
`jge etiqueta` salta a `etiqueta` si `op1` es mayor o igual que `op2`.

— La instrucción `jz etiqueta` salta a la dirección especificada por la `etiqueta` si el indicador de resultado cero está encendido, es decir, si el resultado de la última operación realizada fue cero. De igual manera, la instrucción `jnz etiqueta` salta a la dirección especificada por la `etiqueta` si el indicador de resultado cero está apagado, es decir, si el resultado de la última operación realizada fue distinto de cero.

- Instrucciones de llamada y retorno de una subrutina.

— La instrucción `call etiqueta` invoca a la subrutina de nombre `etiqueta`. Para ello, primero almacena en la pila la dirección de la siguiente instrucción a ejecutar (la dirección de retorno) y después salta a la dirección de memoria correspondiente a `etiqueta`.
— La instrucción inversa a la anterior (`ret`) extrae de la pila el valor de la siguiente instrucción que se va a ejecutar y transfiere el control a dicha instrucción.

6.1.1. Gestión de los registros de la máquina

Cada tipo de computadora funciona con su propio lenguaje de la máquina y posee una unidad aritmético-lógica propia. Dicha unidad contiene cierto número de registros de trabajo de acceso muy rápido, en los que se puede almacenar información y operar con ella realizando sumas, restas, comparaciones, etc. Existen máquinas (más bien antiguas) que poseen un solo registro de trabajo especial, denominado *acumulador*. En otras máquinas la unidad aritmético-lógica contiene cierto número (por ejemplo, 32) de registros idénticos e intercambiables. Por último, algunas máquinas, como la serie INTEL 80x86, poseen un número reducido de registros de trabajo con propiedades no exactamente idénticas. Estos registros son diferentes según que el modo de trabajo de estas máquinas sea de 16 o de 32 bits. Como se ha indicado, en 32 bits los registros desempeñan papeles diferentes y se llaman EAX (acumulador), EBX (registro base de indexación), ECX (registro contador), EDX (registro complementario del acumulador), ESP (puntero a la pila), EBP (registro base para las variables automáticas en la pila, *véase* la Sección 10.1), ESI y EDI (registros de copia origen y destino). En 16 bits, los registros correspondientes se denominan AX, BX, CX, DX, SP, BP, SI y DI. Además, existen algunos nombres adicionales que se refieren a partes de los registros anteriores, como AH y AL (Bytes superior e inferior de AX), BH y BL, CH y CL, DH y DL.

Las instrucciones de la máquina pueden hacer uso de los registros, o bien trabajar directamente sobre la memoria direccionable. Sin embargo, numerosas operaciones (cálculo de expresiones, comparación de los valores de dos variables, etc.) exigen que al menos uno de los

operandos se encuentre copiado sobre uno de los registros de trabajo (a veces, como se ha visto al mencionar las instrucciones `mul` y `div`, debe encontrarse en un registro concreto). Por ello, una parte muy importante de todo generador de código tiene que ver con la carga o copia de los valores de las variables sobre los registros de trabajo, así como la gestión de los registros, pues al ser varios, en un momento dado podrían contener el valor de más de una variable.

Si la unidad aritmético-lógica estuviese provista de un solo registro acumulador (como ocurre en máquinas antiguas y, hasta cierto punto, en las máquinas INTEL, pues en éstas el registro EAX desempeña un papel distinguido en ciertos casos), es conveniente disponer en el compilador de una rutina que el generador de código puede utilizar para asegurarse de que una u otra de las variables que toman parte en un cálculo determinado se encuentra cargada en el acumulador, y en caso contrario realice la carga de una de ellas. En algunas operaciones conmutativas (como la suma o la comparación de la igualdad), no nos importa cuál de las dos variables está cargada en el acumulador, pues basta que sea una cualquiera de ellas. En otras operaciones no conmutativas, como la resta o la división, interesa, en cambio, especificar cuál de los dos operandos (normalmente el izquierdo) debe encontrarse en el acumulador. La función `CAC`, escrita en el lenguaje C, asegura todas estas condiciones:

```
int CAC (opd *x, opd *y)
{
  if (AC!=NULL && AC==y) return 1;
  if (AC!=x) {
    if (AC!=NULL) GEN ("MOV", AC, "EAX");
    GEN ("MOV", "EAX", x);
    AC=x;
    }
  return 0;
}
```

Esta rutina puede invocarse de dos maneras diferentes:

- `CAC (x,y)`: aplicable a las operaciones conmutativas, indica que se desea cargar el valor de la variable `x` o de la variable `y`, indistintamente.
- `CAC (x,NULL)`: aplicable a las operaciones no conmutativas, indica que se desea cargar el valor de `x`, exclusivamente.

La variable auxiliar `AC` contiene una estructura especial, llamada *plataforma*, que almacena información sobre la variable que está cargada en el acumulador en un momento dado. Dicha información (que coincide con la que se guarda en la pila semántica) indica cuál es el nombre de la variable, cuál es su tipo, la dirección que se le ha asignado, si se trata de un *vector* indexado y con qué subíndice, o si la variable es accesible a través de un puntero y con qué desplazamiento. Esta información podría sustituirse, toda o en parte, por un puntero al elemento de la tabla de símbolos correspondiente a la variable de que se trate.

Esencialmente, la rutina `CAC` realiza los siguientes pasos:

- Comprueba si el acumulador contiene ya la variable `y` (siempre que esta variable exista), en cuyo caso no hace nada y devuelve un 1.

- Comprueba si el acumulador contenía ya la variable `x`, en cuyo caso no hace nada y devuelve un 0.
- Si el acumulador estaba vacío (no contenía el valor de ninguna variable), se carga el valor de `x` en el acumulador y se devuelve un 0.
- En caso contrario, se genera una instrucción que guarde el valor actual del acumulador en la dirección de memoria asociada a la variable que contenía, se carga el valor de `x` en el acumulador y se devuelve un 0.

En cualquier caso, si `CAC` devuelve 0, significa que `x` está ahora cargado en el acumulador; si devuelve 1, que es el valor de `y` el que se encuentra allí.

La función auxiliar `GEN` añade una instrucción nueva al programa objeto. Esta función admite tres argumentos: el código de operación de la instrucción, el operando izquierdo y el operando derecho. Si el argumento es una cadena de caracteres, se copiará directamente sobre la instrucción generada. Si se trata de una plataforma, la función `GEN` generará el nombre apropiado para el operando. Por ejemplo:

- `GEN("MOV", "EAX", x)`, donde `x` es una plataforma que define el operando `A`, generará la instrucción `MOV EAX,A`.
- `GEN("MOV", "EAX", x)`, donde `x` es una plataforma que define el operando `B[4]`, generará la instrucción `MOV EAX,[_B+4*sizeof(tipo de B)]` (si el origen de índices en el lenguaje fuente es cero).

Si en vez de un solo acumulador existe un conjunto de registros intercambiables, la variable `AC` podría sustituirse por un `vector` de variables de tipo plataforma, cada uno de cuyos elementos contendrá información sobre el operando contenido en el registro correspondiente.

Será preciso distinguir los registros en punto fijo de los de punto flotante. Por otra parte, algunos registros podrían estar reservados para uso interno del compilador (por ejemplo, como índices de bucles).

Una rutina general de carga de registros deberá comenzar por seleccionar el registro que se va a utilizar entre todos los disponibles. Para ello hay que tener en cuenta el tipo del registro sobre el que se desea cargar (registros enteros o en punto flotante). Si no hay ninguno del tipo deseado, se elegirá uno de los que están ocupados, despreciando la información que contiene (si ya no es necesaria) o guardándola en la posición de memoria asociada, en caso contrario. Una vez seleccionado el registro, la carga propiamente dicha dependerá del tipo del objeto que hay que cargar. Para ver con claridad qué clase de operación se debe realizar en la carga de un operando sobre un registro, es conveniente que el diseñador del compilador construya una tabla parecida a la 6.1, que sólo contiene columnas para algunos de los tipos posibles. En esta tabla, el nombre `T` se aplica a una posición de la memoria del programa objeto que el compilador utilizaría como memoria auxiliar, para introducir en ella valores intermedios que no corresponden a ninguna variable del programa fuente. En este caso, sirve como etapa intermedia para la conversión de los datos de tipo entero a punto flotante.

Tabla 6.1. Generación de código ensamblador para la carga de un operando sobre un registro entero (RH-RL=RX) o en punto flotante.

Carga sobre un registro de tipo	Tipo del operando que hay que cargar			
	unsigned char	**int**	**constante entera**	**real**
entero	XOR RH,RH MOV RL,x	MOV RX,x	MOV RX,x	FLD x FISTP x MOV RX,x

punto flotante	XOR RH,RH
	MOV RL,x
	MOV T,x
	FLD T
		FIL D x	MOV T,x
			FLD T
				FLD x

6.1.2. Expresiones

A continuación se muestra una gramática de expresiones típica, como las que suelen encontrarse en muchos lenguajes de programación:

```
<exp>          ::= <exp> + <exp>
                 | <exp> – <exp>
                 | <exp> * <exp>
                 | <exp> / <exp>
                 | - <exp>
                 | id
                 | <constante>
                 | ( <exp> )
                 | ( <compare> )
                 | <dereference>
<compare>      ::= <exp> = <exp>
                 | <exp> != <exp>
                 | <exp> > <exp>
                 | <exp> >= <exp>
                 | <exp> < <exp>
                 | <exp> <= <exp>
                 | <compare> + <compare>
                 | <compare> * <compare>
                 | ¬ <compare>
<constant>     ::= <bool_const>
                 | int_const
                 | real_const
<bool_const>   ::= true
                 | false
```

En realidad, la gramática anterior es ambigua, por lo que el analizador sintáctico tendrá que emplear otra algo diferente, o bien utilizar algoritmos especiales de desambiguación (*véase* el Capítulo 4). En este capítulo se supondrá que los símbolos `id`, `int_const`, `real_const`, `true` y `false` son unidades sintácticas terminales, es decir, su construcción ha sido tratada previamente por el analizador morfológico.

Existen tipos muy diversos de expresiones, en función del conjunto de valores que se puede calcular. Por ejemplo, se podrían aceptar variables y expresiones de los tipos indicados en la Tabla 6.2, que también indica el tamaño que suelen tener los objetos de los tipos indicados.

Tabla 6.2. Tipos de datos y tamaño que ocupa cada elemento.

Tipo de dato	Tamaño de cada elemento
Boolean	1 bit, o 1, 2 o 4 Bytes
char	1 Byte
unsigned char	1 Byte
short	2 Bytes
unsigned short	2 Bytes
long	4 Bytes
unsigned long	4 Bytes
int	2 o 4 Bytes
unsigned int	2 o 4 Bytes
float	4 Bytes
double	8 Bytes

Dependiendo del lenguaje, un dato de tipo Boolean puede ocupar todos los tamaños indicados en la tabla. En APL, por ejemplo, los datos de este tipo se empaquetan a razón de 8 elementos por Byte, es decir, ocupan 1 bit. En C, los datos booleanos se tratan en realidad como si fuesen de tipo `int`: ocupan 2 o 4 Bytes (según que se esté usando un modelo de memoria de 16 o de 32 bits, respectivamente; *véase* la Sección 10.1). Si su valor es cero, se supone que representan el valor `false`; en caso contrario, representan el valor `true`.

Para simplificar, en este apartado se supondrá que sólo existen los siguientes tipos de expresiones: `Boolean`, `char`, `short` y `double`. Además, los datos de tipo `Boolean` no se podrán mezclar en las operaciones con los de los otros tipos, pero podrían obtenerse como resultado de operaciones de comparación realizados con dichos tipos.

La construcción de tablas de código generado, como la Tabla 6.1, a la que se hizo referencia en el tratamiento de la carga de un operando en un registro (*véase* la Sección 6.1.1), es también muy útil para generar el código asociado a las operaciones que aparecen en las expresiones, es-

Tabla 6.3. Generación de código ensamblador para la operación suma.

Tipo del operando izquierdo x	Tipo del operando derecho y					
	unsigned char	int	Registro entero	Constante entera	double	Registro double
unsigned char	Carga x Repite suma	Intercambio Repite suma	Intercambio Repite suma	Carga x Repite suma	Carga x Repite suma	Carga x Repite suma
int	Carga y Repite suma	Carga y Repite suma	ADD y,x	Carga y Repite suma	Carga x Repite suma	FIADD x
Registro entero	Carga Repite suma	ADD x,y	ADD x,y	ADD x,y	MOV T,x Repite suma	MOV T,x Repite suma
Constante entera	Intercambio Repite suma	Intercambio Repite suma	Intercambio Repite suma	—	Carga x Repite suma	FADD x
double	Carga y Repite suma	Carga y Repite suma	Intercambio Repite suma	Intercambio Repite suma	Carga y Repite suma	FADD x
Registro double	Intercambio Repite suma	Intercambio Repite suma	Intercambio Repite suma	Intercambio Repite suma	Intercambio Repite suma	FADD y

pecialmente para las diádicas, en las que la tabla será de doble entrada, en función de los tipos respectivos del argumento izquierdo y del derecho. A menudo, el cambio de tipo se puede realizar de forma más o menos directa a través de la operación de carga en registro definida anteriormente. La Tabla 6.3 muestra, como ejemplo, la tabla correspondiente a la operación suma, cuya regla es `<exp> ::= <exp> + <exp>`.

- Las operaciones `Carga x` y `Carga y` representan la aplicación de la Tabla 6.1 a la variable correspondiente. La realización de esta operación modifica la plataforma asociada a uno de los operandos, pues el tipo de la variable en cuestión pasa a ser *Registro entero* o *Registro flotante*, una vez realizada la operación. La `Carga` tendrá lugar sobre un registro entero si ambas variables (`x` e `y`) pertenecen a uno de los cuatro primeros tipos de la Tabla 6.3, y sobre un registro flotante (`double`) en caso contrario. Una vez realizada esta operación, se vuelve a aplicar la misma tabla sobre la nueva combinación de plataformas, lo que nos llevará automáticamente a una casilla diferente.

- La operación `Intercambio` consiste, como indica su nombre, en intercambiar las dos plataformas: la del operando izquierdo pasará a ser la del derecho, y viceversa. Este intercambio no afecta al resultado de la operación, pues la suma es conmutativa. Después de realizada esta operación, es preciso volver a aplicar la Tabla 6.3, lo que nos llevará a la casilla simétrica de la anterior respecto a la diagonal principal de la tabla.

- En la operación `MOV T,x`, que aparece en dos casillas de la Tabla 6.3, el nombre `T` se aplica a una posición de la memoria del programa objeto que el compilador utilizará como memoria auxiliar. Esta operación también modifica la plataforma, pues la correspondiente al operando izquierdo (`x`, cuyo tipo era registro entero) pasa a apuntar a la variable `T`, cuyo tipo es variable de tipo `int` contenida en la memoria.

La programación de la tabla de la suma se podría hacer como se indica en el seudocódigo siguiente:

```
Label Tabla[n][n] = {LCX, LXG, LXG, LCX, LCX, LCX}
                    {LCY, LCY, L1,  LCY, LCX, L2}
                    {LCY, L3,  L3,  L3,  L4,  L4}
                    {LXG, LXG, LXG, 0,   LCX, L5}
                    {LCY, LCY, LXG, LXG, LCY, L5}
                    {LXG, LXG, LXG, LXG, LXG, L6}
L: GOTO Tabla[tipox][tipoy];
LCX:  CARGA X;
      GOTO L;
LCY:  CARGA Y;
      GOTO L;
LXG:  Intercambio (X,Y);
      GOTO L;
L1:   GEN ("ADD", Y, X);
      return;
L2:   GEN ("FIADD", Y, NULL);
      return;
L3:   GEN ("ADD", X, Y);
      return;
L4:   GEN ("MOV", T, X);
      GOTO L;
L5:   GEN ("FADD", X, NULL);
      return;
L6:   GEN ("FADD", Y, NULL);
      return;
```

En las operaciones no conmutativas (como la resta) se construye una tabla semejante a la 6.3, pero al no poder intercambiar los operandos, el número de casillas de la tabla que generan código aumenta.

En las operaciones monádicas o unarias (con un solo argumento, normalmente situado a la derecha del operador en casi todos los lenguajes de programación) la tabla se reduce normalmente a una tabla de entrada simple. La Tabla 6.4 muestra, como ejemplo, la que correspondería a la operación cambio de signo (– monádico), que corresponde a la regla `<exp> ::= - <exp>`.

Tabla 6.4. Generación de código ensamblador para la operación de cambio de signo.

	Tipo del operando derecho y					
	unsigned char	**int**	**Registro entero**	**Constante entera**	**double**	**Registro double**
	Carga y Repite	Carga y Repite	NEG y	—	Carga y Repite	FCHS

Para una expresión de comparación correspondiente a las reglas de la gramática

```
<compare> ::= <exp> = <exp>
            | <exp> != <exp>
            | <exp> > <exp>
            | <exp> >= <exp>
            | <exp> < <exp>
            | <exp> <= <exp>
```

es posible construir también una tabla que será muy semejante a la de la operación resta, sustituyendo las instrucciones `SUB` y `FSUB` por las instrucciones de comparación `CMP` y `FCMP`.

En cuanto a las reglas

```
<compare> ::= <compare> + <compare>
            | <compare> * <compare>
            | ¬ <compare>
```

se ha optado en esta gramática por representar las operaciones OR y AND mediante los mismos símbolos + y * que se utilizan para la suma de números. Esto es posible, porque se ha dicho anteriormente que esta gramática no admite expresiones que mezclen datos Booleanos y numéricos, por lo que esta sintaxis no sería ambigua. En otros lenguajes (como C o APL) se utilizan símbolos diferentes para estos operadores, pues los datos pueden mezclarse, dado que el tipo booleano se reduce, en realidad, a una forma más de dato numérico.

6.1.3. Punteros

Como se ha explicado en la Sección 5.4, en muchos lenguajes existe un tipo especial de variables, llamadas punteros, que permiten acceder a la información contenida en las variables a las que apuntan por medio de reglas sintácticas más o menos parecidas a las siguientes:

```
<dereference> ::= deref <id>
                | deref <dereference>
```

En el lenguaje C, por ejemplo, la unidad sintáctica que aquí hemos representado con el símbolo terminal `deref` es un asterisco.

Cuando hay que desreferenciar un puntero, el código generado podría ser el siguiente:

```
mov ebx,id
```

Esta instrucción introduce en el registro de indexación `ebx` el contenido del puntero, es decir, la dirección de memoria de la variable a la que éste apunta. A continuación se genera una plataforma, en la que la dirección del operando correspondiente es `[ebx]`.

6.1.4. Asignación

La forma típica de las reglas sintácticas que regulan la asignación de un valor a una variable es la siguiente:

```
<asignacion> ::= <id> := <exp>
               | <dereference> := <exp>
```

La primera regla corresponde a la asignación de valor a una variable ordinaria; la segunda, a la asignación de valor (una dirección) a un puntero. Dependiendo del lenguaje de que se trate, pueden exigirse condiciones semánticas especiales a las variables afectadas por una asignación a un puntero.

Es posible (y conveniente) construir para la asignación una tabla parecida a la de la suma (*véase* la Tabla 6.3), que especifique el código que hay que generar en cada una de las combinaciones posibles de los tipos del identificador situado a la izquierda del símbolo de asignación y de la expresión situada a la derecha. Esto significa que la asignación puede considerarse como un operador diádico más, semejante a los operadores aritméticos (suma, resta, etc.), con la única salvedad de que el operando izquierdo se pasa por referencia, y no por valor, como ocurre con el derecho, y con ambos operandos en las operaciones aritméticas. Por consiguiente, entre los tipos que puede adoptar el operando izquierdo hay que incluir el tipo puntero, lo que significa añadir una línea más a la tabla.

6.1.5. Entrada y salida de datos

Las instrucciones de entrada y salida de datos desde memorias o dispositivos externos al programa ejecutable (como el teclado, la pantalla, archivos en disco, etc.) varían mucho con el lenguaje fuente, por lo que no vamos a considerarlas aquí. Baste decir respecto a ellas que normalmente se llevan a cabo mediante llamadas a subrutinas de biblioteca, con lo que el generador de código deberá generar, usualmente, una instrucción `CALL`.

6.1.6. Instrucciones condicionales

Una instrucción condicional típica que sólo tenga parte `then` podría tener una regla parecida a la siguiente:

```
<condicional> ::= if <exp> then (1) <instruccion> (2)
```

Como se vio en la Sección 6.1.2, el análisis del símbolo no terminal `<exp>` (que en este tipo de instrucciones se reducirá a una comparación) habrá generado el código apropiado para que los indicadores señalen adecuadamente el resultado de la operación. Hay dos maneras de generar el código correspondiente a la instrucción `if`:

- Que el código generado por la comparación almacene en los indicadores el valor `true` o `false`, según corresponda, encendiendo (*set*) o apagando (*reset*) uno de los indicadores

(por ejemplo, el de resultado cero). En tal caso, la instrucción condicional sólo tendría que generar la siguiente instrucción en la acción semántica (1) situada en la regla justo a continuación de la unidad sintáctica `then`:

```
jz fin_then#
```

donde `fin_then#` es una etiqueta interna generada por el compilador (diferente para cada instrucción condicional, por supuesto). Por otra parte, la acción semántica (2), situada al final de la regla, generaría el siguiente código:

```
fin_then#:
```

Es decir, colocaría la etiqueta después del código generado por el símbolo no terminal `<instruccion>`.

- Que el código generado por la comparación se limite a la instrucción de comparación, y que se añada a la información semántica asociada al resultado el tipo de operador de comparación que acaba de analizarse. La Tabla 6.5 indica el código que habría que generar en función de dicho operador. Esto correspondería a la acción semántica (1). La acción (2) sería idéntica al caso anterior. Este procedimiento genera código algo más optimizado que el otro.

Tabla 6.5. Generación de código condicional asociado a instrucciones de comparación.

Operación	Código generado
=	je fin_then
!=	jne fin_then
<	jl fin_then
<=	jle fin_then
>	jg fin_then
>=	jge fin_then

La Figura 6.1 proporciona un esquema gráfico de la generación de código para la instrucción condicional. En dicha figura y las sucesivas, los puntos de la regla donde aparece un número encerrado en un círculo indican acciones semánticas.

La Figura 6.2 muestra el código que habría que generar para la regla

```
<condicional> ::= if <exp> then (1) <instruccion1> (2)
                  else <instruccion2> (3)
```

La primera acción semántica genera un código exactamente igual al generado por la primera acción semántica de la Figura 6.1. La segunda genera una instrucción de salto incondicional

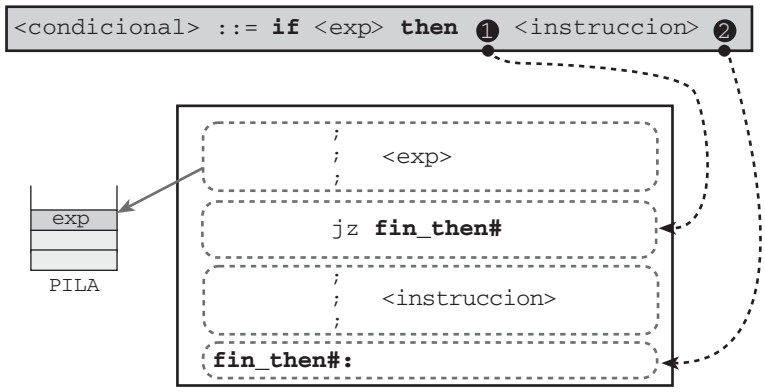


Figura 6.1. Generación de código para la instrucción `if-then`.

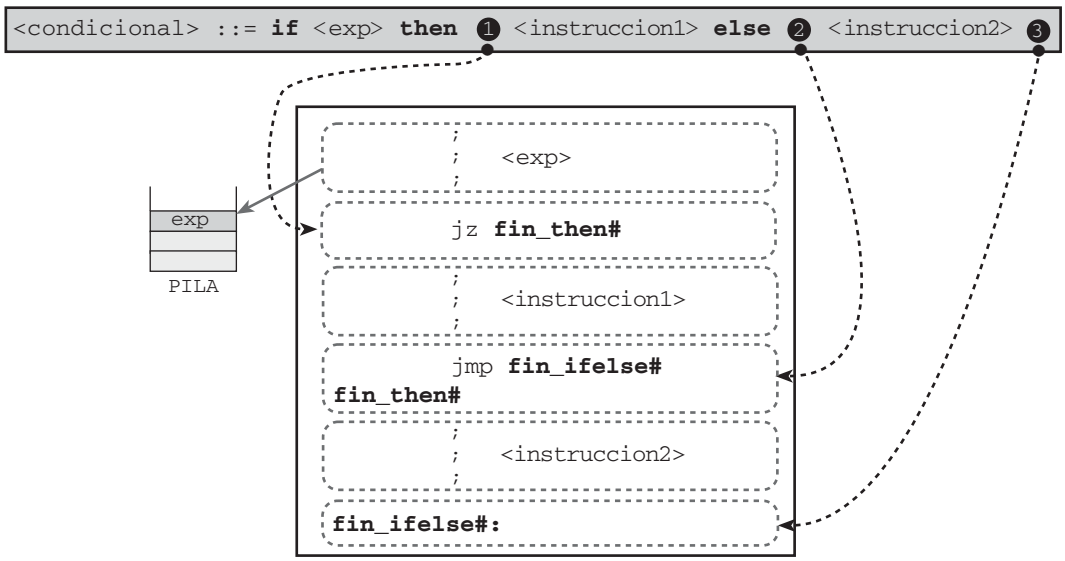


Figura 6.2. Generación de código para la instrucción `if-then-else`.

(`jmp`) a la etiqueta `fin_ifelse#` para evitar que se ejecuten las instrucciones de la parte `else` tras ejecutar las instrucciones de la parte `then`. Además se genera una línea que define la etiqueta `fin_then#`. Por último, la tercera acción semántica genera únicamente una línea que contiene la etiqueta `fin_ifelse#`.

6.1.7. Bucles

La Figura 6.3 muestra el código que habría que generar para la regla

```
<bucle> ::= while (1) <exp> do (2) <instruccion> end (3)
```

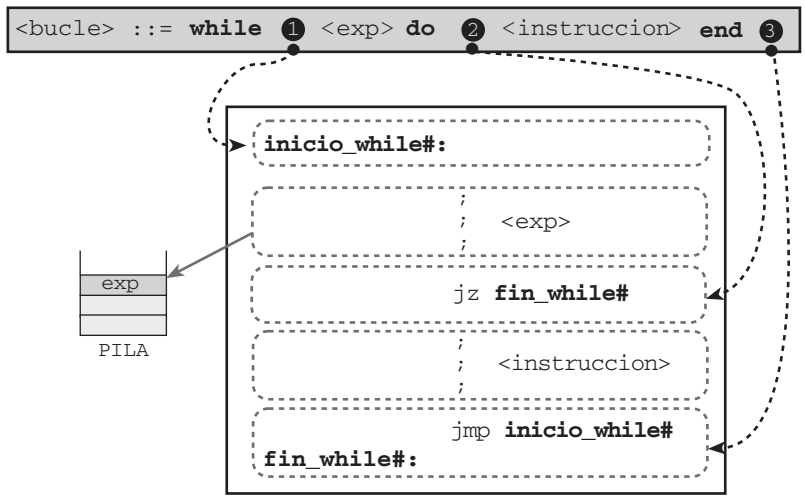


Figura 6.3. Generación de código para la instrucción `while`.

La primera acción semántica genera únicamente una línea que contiene la etiqueta `inicio_while#`. La segunda acción semántica genera un código exactamente igual al generado por la primera acción semántica de las Figuras 6.1 y 6.2. El efecto de este código es salir del bucle si el valor de la expresión es igual a 0, es decir, si la expresión es falsa. La tercera acción semántica genera una instrucción de salto incondicional (`jmp`) a la etiqueta `inicio_while#`, para continuar con la siguiente iteración del bucle. Además genera una línea que contiene la etiqueta `fin_while#`.

La Figura 6.4 muestra el código que habría que generar para la regla

```
<bucle> ::= repeat (1) <instruccion> until <exp> (2)
```

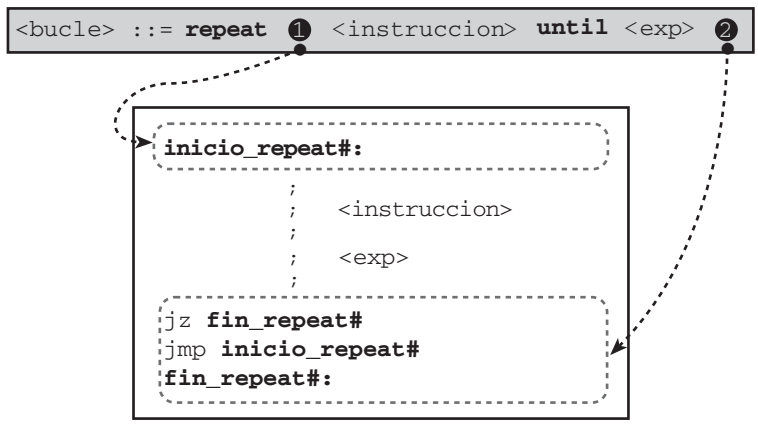


Figura 6.4. Generación de código para la instrucción `repeat`.

La primera acción semántica genera únicamente una línea que contiene la etiqueta `inicio_repeat#`. La segunda acción semántica genera un código similar al generado por la primera acción semántica de las Figuras 6.1 y 6.2. El efecto de este código es salir del bucle si el valor de la expresión es `true`, es decir, si la expresión es verdadera. En caso contrario, se ejecuta un salto incondicional (`jmp`) a la etiqueta `inicio_repeat#`, para continuar con la siguiente iteración del bucle. Además se genera una línea que contiene la etiqueta `fin_repeat#`.

6.1.8. Funciones

El diseño del manejo de funciones por un compilador implica dar respuesta a las siguientes preguntas:

- ¿Cómo se comunican los argumentos desde el programa que invoca a la función invocada?
- ¿Cómo se comunican los resultados desde la función invocada al programa que invoca?

En general, se utiliza una pila para almacenar las variables locales de la función (variables automáticas) y los argumentos de llamada a la función. En algunos lenguajes, como C y C++, los argumentos se guardan en la pila en orden inverso (de derecha a izquierda). En otros lenguajes, se hace al revés. Para pasar el resultado de la función al programa que la invocó, se suele utilizar el registro EAX, siempre que dicho resultado quepa en él. De lo contrario se puede usar la pila, o bien la memoria estática.

En la Sección 10.1, cuando se describa la gestión de memoria para las variables automáticas en un compilador, se puede encontrar un ejemplo del código generado en las llamadas a funciones.

6.2 Código intermedio

Como se describió en la Sección 5.1.3, en el primer paso de la compilación, en compiladores de dos o más pasos, el analizador semántico genera un código abstracto, denominado *código intermedio*. En un segundo paso se realiza la generación del código definitivo a partir del código intermedio. En esta sección se estudiarán dos formas de código intermedio: notación sufija y cuádruplas.

Los operadores diádicos (o binarios) pueden especificarse mediante tres notaciones principales:

- *prefija*: el operador diádico se analiza antes que sus operandos.
- *infija*: el operador diádico se analiza entre sus dos operandos.
- *sufija*: el operador diádico se analiza después que sus operandos.

En los lenguajes de programación clásicos, los operadores diádicos se representan usualmente en notación infija. La notación prefija permite al operador influir sobre la manera en que se pro-

cesan sus operandos, pero a cambio suele exigir mucha más memoria. La sufija no permite esa influencia, pero optimiza la gestión de memoria y permite eliminar el procesado de los paréntesis.

Los operadores monádicos sólo pueden presentarse en notación prefija o sufija. En casi todos los lenguajes, la mayor parte de estos operadores suelen utilizar la sintaxis prefija. En Smalltalk se usa siempre la notación sufija, que también puede utilizarse con algunos operadores en los lenguajes C y C++ (por ejemplo, el operador ++).

Además, un árbol sintáctico puede representarse en forma de tuplas de `n` elementos, de la forma `(operador, operando`$_1$`, ..., operando`$_n$`, resultado)`. Las tuplas pueden tener longitud variable o fija (con operandos nulos). Las más típicas son las cuádruplas, aunque éstas pueden representarse también en forma de tripletes.

6.2.1. Notación sufija

Llamada también *notación postfija* o *polaca inversa*, se usa para representar expresiones sin necesidad de paréntesis, eliminando también la necesidad de establecer precedencia entre los distintos operadores. La Tabla 6.6 muestra algunos ejemplos.

Tabla 6.6. Algunos ejemplos de expresiones en notación sufija.

Expresión	Notación sufija
`a*b`	`ab*`
`a*(b+c/d)`	`abcd/+*`
`a*(b+c*d)`	`ab*cd*+`

Como puede apreciarse en los ejemplos anteriores, en una expresión en notación sufija los identificadores aparecen en el mismo orden que en la forma usual de las expresiones, mientras que los operadores aparecen en el orden de su evaluación, de izquierda a derecha.

Un problema que se plantea en la notación sufija es cómo tratar los operadores monádicos o unarios cuyo símbolo coincide con el de algún operador binario, por ejemplo, el operador de cambio de signo (–). Existen dos posibilidades: transformarlos en operadores diádicos o binarios, o utilizar un símbolo distinto. Por ejemplo, la expresión `–a` puede convertirse en `0-a` o en `@a`. Si se elige la segunda opción, la expresión `a*(-b+c/d)` se representaría en notación sufija como `ab@cd/+*`.

Una vez descrita la notación sufija, es necesario explicar cómo realizará el compilador las siguientes tareas:

- Construcción de la notación sufija durante el análisis sintáctico
- Generación de código ensamblador a partir de la notación sufija

Construcción de la notación sufija durante el análisis sintáctico

En el análisis ascendente: Si el analizador sintáctico es ascendente, hacemos la siguiente suposición: cuando aparece un símbolo no terminal V en el asidero, la notación sufija correspondiente a la subcadena que se redujo a V ya ha sido generada.

Para generar la notación sufija, se utiliza una pila, inicialmente vacía, y se aprovechará el algoritmo de análisis ascendente descrito en el Capítulo 4, con las siguientes acciones adicionales:

- Cada vez que se realiza una operación de desplazamiento con un símbolo terminal, se asocia dicho símbolo a la información semántica del estado que se introduce en la pila de análisis.
- Cada vez que se realiza una operación de reducción, se pasa a la pila de notación sufija la información semántica asociada a los estados extraídos de la pila de análisis.

Ejemplo 6.1

Consideremos la gramática siguiente:

```
(1) E ::= E + T
(2) E ::= T
(3) T ::= i
```

En la Tabla 6.7 aparece la tabla de análisis para esta gramática.

Tabla 6.7. Tabla de análisis ascendente para la gramática del Ejemplo 6.1.

	E	T	i	+	$
0	d1	d2	d3		
1				d4	fin
2			r2	r2	r2
3			r3	r3	r3
4		d5	d3		
5			r1	r1	r1

La Figura 6.5 muestra el proceso de generación de la notación sufija para la expresión `a+b`, utilizando la tabla de análisis de la Tabla 6.7. La información semántica asociada a un estado aparece entre paréntesis, a continuación del número del estado.

En el análisis descendente: En el análisis descendente, es posible generar la notación sufija utilizando también una pila inicialmente vacía, que al final del análisis sintáctico contendrá la notación sufija resultante. Para ello, es necesario añadir a las funciones del analizador sintáctico algunas instrucciones que introducirán los valores adecuados en la pila.

Pila de análisis	Entrada	Notación sufija
0	a+b$	
03(a)	+b$	
0	T+b$	a
02	+b$	
0	E+b$	
01	+b$	
014(+)	b$	
014(+)3(b)	$	
014(+)	T$	ab
014(+)5	$	
0	E$	ab+

Figura 6.5. Generación de la notación sufija para la expresión `a+b` en el análisis ascendente.

Como ejemplo, consideremos la gramática del Ejemplo 4.6, que se reproduce aquí para mayor claridad.

```
E ::= T + E
E ::= T - E
E ::= T
T ::= F * T
T ::= F / T
T ::= F
F ::= i
F ::= (E)
```

Las funciones que componen el analizador sintáctico descendente para esta gramática aparecen en las Figuras 4.10 a 4.16. Las funciones correspondientes, modificadas para generar la notación sufija, pueden verse en las Figuras 6.6 a 6.12.

La Figura 6.13 muestra el proceso de la generación de la notación sufija para la expresión `a+b`, utilizando las funciones de las Figuras 6.6 a 6.12. Para mayor claridad, se representan mediante una pila las instrucciones pendientes de ejecución, es decir, las que se ejecutarán cuando la función invocada devuelva el control a la función que la invocó. Por ejemplo, cuando la función `V` llama a la función `E`, la instrucción `push(+)` queda pendiente, y se ejecutará cuando la función `E` termine y devuelva el control a la función `V`.

```
int E (char *cadena, int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case 'i':
        push(id);
        i++;
        i = V (cadena, i);
        break;
      case '(':
        i++;
        i = E (cadena, i);
        i = C (cadena, i);
        i = V (cadena, i);
        break;
      default: return -1;
    }
    return i;
  }
```

Figura 6.6. Generación de notación sufija: función para el símbolo no terminal E.

```
int V (char *cadena, int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case '*':
      case '/':
        i++;
        i = T (cadena, i);
        push(cadena[j]);
        i = X (cadena, i);
        break;
      case '+':
      case '-':
        i++;
        i = E (cadena, i);
        push(cadena[j]);
        break;
    }
    return i;
  }
```

Figura 6.7. Generación de notación sufija: función para el símbolo no terminal V.

```
int X (char *cadena, int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case '+':
      case '-':
        push + -
        i++;
        i = E (cadena, i);
        push(cadena[j]);
        break;
    }
    return i;
  }
```

Figura 6.8. Generación de notación sufija: función para el símbolo no terminal X.

```
int T (char *cadena, int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case 'i':
        push(id);
        i++;
        i = U (cadena, i);
        break;
      case '(':
        i++;
        i = E (cadena, i);
        i = C (cadena, i);
        i = U (cadena, i);
        break;
      default: return -2;
    }
    return i;
  }
```

Figura 6.9. Generación de notación sufija: función para el símbolo no terminal T.

```
int U (char *cadena, int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case '*':
      case '/':
        i++;
        i = T (cadena, i);
        push(cadena[j]);
        break;
    }
    return i;
  }
```

Figura 6.10. Generación de notación sufija: función para el símbolo no terminal U.

```
int F (char *cadena, int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case 'i':
        push(id);
        i++;
        break;
      case '(':
        i++;
        i = E (cadena, i);
        i = C (cadena, i);
        break;
      default: return -3;
    }
    return i;
  }
```

Figura 6.11. Generación de notación sufija: función para el símbolo no terminal F.

Generación de código ensamblador a partir de la notación sufija

En los ejemplos subsiguientes, supondremos que se está tratando con variables de tipo entero, de un tamaño igual al de los registros de trabajo. El algoritmo de generación de código ensamblador a partir de una expresión en notación sufija funciona de la siguiente forma:

- (Caso 1) Si el próximo símbolo es un identificador `id`, se genera la instrucción `push [_id]`.

```
int C (char *cadena, int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case ')':
        i++;
        break;
      default: return -4;
    }
    return i;
  }
```

Figura 6.12. Generación de notación sufija: función para el símbolo no terminal C.

Pasos de ejecución	Pila pendiente	Notación sufija
E("a+b",0)		a
V("a+b",1)	push(+)	
E("a+b",2)		ab
V("a+b",3)		
3		ab+

Figura 6.13. Generación de notación sufija para la expresión `a+b` en el análisis descendente.

- (Caso 2) Si el próximo símbolo es una constante `c`, se genera la instrucción `push c`.
- (Caso 3) Si el próximo símbolo es un operador diádico, por ejemplo, la suma, se generan las instrucciones:

```
pop edx
pop eax
add eax, edx
push eax
```

- (Caso 4) Si el próximo símbolo es un operador monádico, por ejemplo, el cambio de signo, se generan las instrucciones:

```
pop eax
neg eax
push dword eax
```

La Tabla 6.8 muestra la aplicación de este algoritmo para generar el código ensamblador para la expresión `ab@+`.

Tabla 6.8. Generación de código ensamblador para la expresión `ab@+`.

Caso	Entrada	Resultados intermedios
1	ab@+	`push [_a]`
1	b@+	`push [_b]`

4	@+	`pop eax`
		`neg eax`
		`push eax`

3	+	`pop edx`
		`pop eax`
		`add eax, edx`
		`push eax`

Otras instrucciones

La notación sufija se usa principalmente para representar expresiones aritméticas, pero puede extenderse para representar otro tipo de instrucciones, como las siguientes:

- La asignación puede tratarse como un operador binario, cuyos operandos son la parte izquierda y derecha de la asignación. La instrucción de asignación `a:=b*c+d` se representaría en notación sufija como `abc*d+:=`. Recuérdese que el operando izquierdo de la asignación debe pasarse por referencia, no por valor, como ocurre con la mayor parte de los otros operadores.
- Las etiquetas asociadas a determinadas instrucciones pueden representarse como `etiqueta:`.
- Un salto incondicional a otra instrucción con etiqueta `L`, puede representarse en notación sufija como `L TR`, donde `TR` significa *transferencia incondicional*.
- Un salto condicional a una etiqueta `L`, que debe realizarse únicamente si el resultado de la última operación aritmético-lógica efectuada fue igual a cero, puede representarse en notación sufija como `L TRZ`, donde `TRZ` significa *transferencia si cero*.
- Utilizando los operadores descritos anteriormente, la instrucción condicional `if p then inst1 else inst2` puede representarse en notación sufija como `nsp L1 TRZ nsinst1 L2 TR L1: nsinst2 L2:`, donde `nsp`, `nsints1` y `nsinst2` corresponden a la representación en notación sufija de `p`, `inst1` e `inst2`, respectivamente.
- Una expresión con subíndices, tal como `a[exp1; exp2; ...; expn]`, puede representarse en notación sufija como `a nsexp1 nsexp2 ... nsexpn SUBIN-n`, donde `nsexp1`, `nsexp2`, ..., `nsexpn` corresponden a la representación en notación sufija de `exp1`, `exp2`, ..., `expn`, respectivamente.

6.2.2. Cuádruplas

Una operación diádica se puede representar mediante la cuádrupla

```
(<operador>, <operando1>, <operando2>, <resultado>)
```

Un ejemplo de cuádrupla sería `(*,a,b,t)` que es equivalente a la expresión `a*b`.

Una expresión se puede representar mediante un conjunto de cuádruplas. Por ejemplo, la expresión `a*b+c*d` es equivalente a las siguientes cuádruplas:

```
(*,a,b,t1)
(*,c,d,t2)
(+,t1,t2,t3)
```

Como ejemplo adicional, se puede considerar la expresión con subíndices `c:=a[i;b[j]]` que es equivalente a las siguientes cuádruplas:

```
(*,i,d2,t1)
(+,t1,b[j],t2)
(:=,a[t2],,c)
```

donde `a` es una matriz con dimensiones `d1` (número de filas) y `d2` (número de columnas), y se supone que el origen de índices es cero.

Tripletes

Otro formato que se puede utilizar para generar código intermedio son los tripletes, que son similares a las cuádruplas, con la única diferencia de que los tripletes no dan nombre a su resultado, y cuando un triplete necesita hacer referencia al resultado de otro, se utiliza una referencia a dicho triplete. Por ejemplo, la expresión `a*b+c*d` equivale a los siguientes tripletes:

```
(1) (*,a,b)
(2) (*,c,d)
(3) (+,(1),(2))
```

mientras que `a*b+1` equivale a los tripletes:

```
(1) (*,a,b)
(2) (*,(1),1)
```

También puede utilizarse lo que se conoce como *tripletes indirectos*, que consiste en numerar arbitrariamente los tripletes y especificar aparte el orden de su ejecución. Por ejemplo, las instrucciones:

```
a := b*c
b := b*c
```

equivalen a los siguientes tripletes:

```
(1) (*,b,c)
(2) (:=,(1),a)
(3) (:=,(1),b)
```

y el orden de su ejecución es (1),(2),(1),(3).

Este formato es útil para preparar la optimización de código, porque es más fácil alterar el orden de las operaciones o eliminar alguna.

Generación de cuádruplas durante el análisis sintáctico

En el análisis ascendente: Para generar cuádruplas se utiliza el algoritmo de análisis ascendente descrito en el Capítulo 4, junto con una pila auxiliar y las siguientes acciones adicionales:

- Cada vez que se realiza una operación de desplazamiento con un símbolo terminal, se introduce dicho símbolo en la pila auxiliar.
- Cada vez que se realiza una operación de reducción con una regla que contiene un operador, se ejecuta la acción semántica asociada a la regla que se reduce, siendo las acciones semánticas de la siguiente forma:
  - — Para reglas del tipo `U ::= V operador W`, la acción semántica extrae de la pila auxiliar los operandos correspondientes `V (op1)` y `W (op2)`, analiza su compatibilidad, crea la cuádrupla `(operador,op1,op2,Ti)` e introduce `Ti` en la pila auxiliar.
  - — Para reglas del tipo `U ::= operador V`, la acción semántica extrae de la pila auxiliar el operando `v(op1)`, crea la cuádrupla `(operador,op1,,Ti)` e introduce `Ti` en la pila auxiliar.

Consideremos la gramática del Ejemplo 6.6 y la tabla del análisis de la Tabla 6.7. La Figura 6.14 muestra el proceso de generación de cuádruplas para la expresión `a+b`.

En el análisis descendente: De forma análoga a la generación de notación sufija, una posible forma de generar cuádruplas en el análisis descendente consiste en utilizar una pila auxiliar y modificar las funciones del analizador sintáctico para incluir instrucciones que introduzcan los valores necesarios en la pila y que generen las cuádruplas correspondientes.

Las funciones que componen el analizador sintáctico descendente para la gramática de expresiones del Ejemplo 4.6, modificadas para generar las cuádruplas correspondientes, aparecen en las Figuras 6.15 a 6.21.

La Figura 6.22 muestra el proceso de la generación de las cuádruplas correspondientes a la expresión `a+b*c`, utilizando las funciones de las Figuras 6.15 a 6.21.

Instrucciones condicionales

Cuando se generan cuádruplas para las instrucciones de control, puede ocurrir que en el momento en que se genera una cuádrupla de salto no se sepa la cuádrupla a la que hay que saltar, porque ésta no se ha generado todavía. Este problema se soluciona de la siguiente forma: se numeran

Pila de analisis	Entrada	Pila auxiliar	
0	a+b$		
03	+b$	a	
0	T+b$		
02	+b$		
0	E+b$		
01	+b$		
014	b$	ab	
0143	$		
014	T$		
0145	$		
0	E$		(+,a,b,t1)

Figura 6.14. Generación de cuádruplas para la expresión `a+b` en el análisis ascendente.

```
unsigned int E (char *cadena, unsigned int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case 'i':
        push(id);
        i++;
        i = V (cadena, i);
        break;
      case '(':
        i++;
        i = E (cadena, i);
        i = C (cadena, i);
        i = V (cadena, i);
        break;
      default: return -1;
    }
    return i;
  }
```

Figura 6.15. Generación de cuádruplas: función para el símbolo no terminal E.

```
unsigned int V (char *cadena, unsigned int i)
  {
    unsigned int j;
    if (i<0) return i;
    switch (cadena[i]) {
      case '*':
      case '/':
        j = i;
        i++;
        i = T (cadena, i);
        cuad(cadena[j], pop(), pop(), gen(Ti));
        push(Ti);
        i = X (cadena, i);
        break;
      case '+':
      case '-':
        j = i;
        i++;
        i = E (cadena, i);
        cuad(cadena[j], pop(), pop(), gen(Ti));
        push(Ti);
        break;
    }
    return i;
  }
```

Figura 6.16. Generación de cuádruplas: función para el símbolo no terminal V.

```
unsigned int X (char *cadena, unsigned int i)
  {
    unsigned int j;
    if (i<0) return i;
    switch (cadena[i]) {
      case '+':
      case '-':
        j = i;
        i++;
        i = E (cadena, i);
        cuad(cadena[j], pop(), pop(), gen(Ti));
        push(Ti);
        break;
    }
    return i;
  }
```

Figura 6.17. Generación de cuádruplas: función para el símbolo no terminal X.

```
unsigned int T (char *cadena, unsigned int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case 'i':
        push(id);
        i++;
        i = U (cadena, i);
        break;
      case '(':
        i++;
        i = E (cadena, i);
        i = C (cadena, i);
        i = U (cadena, i);
        break;
      default: return -2;
    }
    return i;
  }
```

Figura 6.18. Generación de cuádruplas: función para el símbolo no terminal T.

```
unsigned int U (char *cadena, unsigned int i)
  {
    if (i<0) return i;
    unsigned int j;
    switch (cadena[i]) {
      case '*':
      case '/':
        j = i;
        i++;
        i = T (cadena, i);
        cuad(cadena[j], pop(), pop(), gen(Ti));
        push(Ti);
        break;
    }
    return i;
  }
```

Figura 6.19. Generación de cuádruplas: función para el símbolo no terminal U.

```
unsigned int F (char *cadena, unsigned int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case 'i':
        push(id);
        i++;
        break;
      case '(':
        i++;
        i = E (cadena, i);
        i = C (cadena, i);
        break;
      default: return -3;
    }
    return i;
  }
```

Figura 6.20. Generación de cuádruplas: función para el símbolo no terminal F.

```
unsigned int C (char *cadena, unsigned int i)
  {
    if (i<0) return i;
    switch (cadena[i]) {
      case ')':
        i++;
        break;
      default: return -4;
    }
    return i;
  }
```

Figura 6.21. Generación de cuádruplas: función para el símbolo no terminal C.

las cuádruplas y se usa su número para identificarlas, se mantiene una variable global, `cl.sig`, cuyo valor es el número de la siguiente cuádrupla a generar, y se introducen en una pila los números de las cuádruplas pendientes de completar, es decir, aquellas para las que se desconocía el valor de alguno de sus componentes en el momento en que se generaron.

Pasos de ejecución	Pila pendiente	Pila aux.	Cuádruplas
E("a+b*c",0)		a	
V("a+b*c",1)			
E("a+b*c",2)	cuad(+,pop,pop,t1) push(t1)	ab	
V("a+b*c",3)			
T("a+b*c",4)	cuad(*,pop,pop,t2) push(t2) X("a+b*c",5) cuad(+,pop,pop,t1) push(t1)		
U("a+b*c",5)		abc	
		at2 t1	(*,b,c,t2) (+,a,t2,t1)

Figura 6.22. Generación de cuádruplas para la expresión `a+b` en análisis descendente.

Para llevar a cabo todas estas acciones, se introducen acciones semánticas en determinados puntos de las reglas correspondientes a la instrucción condicional. En concreto, son necesarias las tres acciones semánticas que aparecen entre paréntesis.

```
<condicional> ::= if <expr> (2) then <instr> (1)
                | if <expr> (2) then <instr1> else (3) <instr2> S(1)
```

La instrucción condicional

```
if <expr> then <instr1> else <instr2>
```

generaría la siguiente secuencia de cuádruplas:

```
(p-1)   (?,?,?,t1)          Cuádruplas correspondientes a <expr>
(p)     (TRZ,(q+1),t1,)     (2): Generar cuádrupla (p) | push p
        ...                 Cuádruplas correspondientes a <instr1>
(q)     (TR,(r),,)          (3): Generar cuádrupla (q)
                            Poner (cl.sig) en top | pop
                            push (q)
(q+1)   ...                 Cuádruplas correspondientes a <instr2>
                            (1) Poner (cl.sig) en top | pop
(r)
```

Cuando una componente de una cuádrupla aparece marcada en negrita, indica que esa componente queda vacía en el momento de generación de la cuádrupla y que su valor se rellenará posteriormente, cuando se conozca.

Al generar la cuádrupla (p) no conocemos el valor de (q+1), por lo que la acción semántica (2) mete en la pila el número de la cuádrupla que se acaba de generar (p) para que se rellene su segunda componente cuando se genere la cuádrupla q+1.

De la misma forma, al generar la cuádrupla (q) no conocemos todavía el valor de (r), por lo que la acción semántica (3) mete en la pila el número de la cuádrupla que se acaba de generar (q), para que se rellene su segunda componente cuando se genere la cuádrupla r.

La Figura 6.23 muestra el proceso de generación de cuádruplas para la siguiente instrucción:

```
if (a < b) then
   a:=2
else
   b:=3;
```

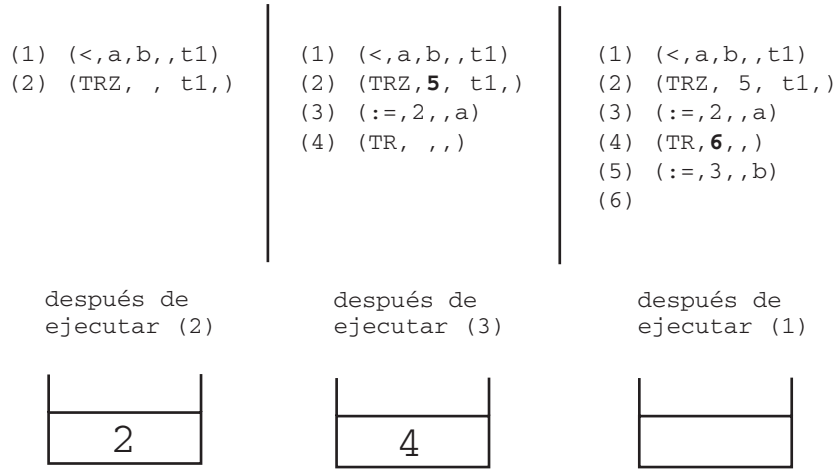


Figura 6.23. Generación de cuádruplas para una instrucción `if-then-else`.

La instrucción condicional

```
if <expr> then <instr>
```

generaría la siguiente secuencia de cuádruplas:

```
(p-1)  (?,?,?,t1)      Cuádruplas correspondientes a <expr>
(p)    (TRZ,(r),t1,)   (2): Generar cuádrupla (p) | push p
       ...             Cuádruplas correspondientes a <instr>
(r)                    (1): Poner (cl.sig.) en top | pop
```

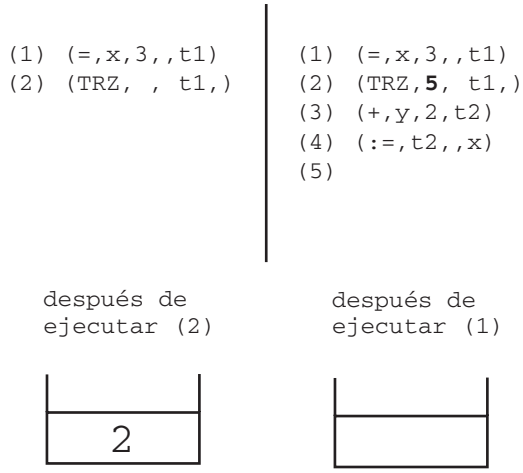


Figura 6.24. Generación de cuádruplas para una instrucción `if-then`.

Al generar la cuádrupla (`p`) no conocemos el valor de (`r`), por lo que la acción semántica `(2)` mete en la pila el número de la cuádrupla que se acaba de generar `(p)` para que se rellene su segunda componente cuando se genere la cuádrupla `r`.

La Figura 6.24 muestra el proceso de generación de cuádruplas para la siguiente instrucción:

```
if (x=3) then
   x:=y+2;
```

Etiquetas y GOTO

Aunque la programación estructurada que utilizan la mayor parte de los lenguajes de alto nivel excluye el uso de la instrucción `GOTO`, todos los compiladores la implementan, pues puede generarse automáticamente como consecuencia de algún preproceso previo del programa fuente, como ocurre, por ejemplo, con las instrucciones SQL embebidas. Por otra parte, en algunos lenguajes, como APL, `GOTO` es la única estructura disponible para el control de flujo del programa.

Para la generación de cuádruplas para las etiquetas asociadas a instrucciones y para las instrucciones `GOTO`, consideraremos que los identificadores que corresponden a etiquetas se reconocen porque, en el campo valor que se les asocia en la tabla de símbolos, el atributo `tipo` adopta el valor *etiqueta*. El campo valor contendrá, además, otros dos atributos:

- Atributo `localizada`, cuyos valores podrán ser `SI` o `NO`.
- Atributo `núm_cuádrupla`, cuyo valor será el número de la cuádrupla correspondiente a la etiqueta.

En la Figura 6.25 aparece la estructura de una tabla de símbolos con los atributos mencionados.

id	valor		
	tipo	localizada	núm_cuádrupla

Figura 6.25. Estructura de una tabla de símbolos con identificadores que corresponden a etiquetas.

La Figura 6.26 muestra el pseudocódigo para la generación de cuádruplas para la instrucción

```
<salto> ::= GOTO id
```

```
buscar id en la tabla de símbolos;
if (no está) {
   insertar(id,etiqueta,NO,cl.sig);
   generar cuádrupla (TR,,,);
}
else {
   if (tipo==etiqueta) {
      if (localizada==SI)
        generar cuádrupla (TR,núm_cuádrupla,,);
      else if (localizada==NO) {
        i=núm_cuádrupla;
        cambiar valor de id a (etiqueta,NO,cl.sig);
        generar cuádrupla (TR,i,,);
      }
   }
   else error();
}
```

Figura 6.26. Generación de cuádruplas para la instrucción `GOTO id`.

Con una instrucción `GOTO etiqueta` pueden darse tres casos:

- El identificador correspondiente a la etiqueta no está en la tabla de símbolos, porque todavía no se ha procesado la instrucción en la que aparece la etiqueta. En este caso, se inserta en la tabla de símbolos el identificador correspondiente a la etiqueta, y se genera una cuádrupla de salto incondicional con la segunda componente vacía, porque el número de la cuádrupla a la que hay que saltar no se conocerá hasta que se localice la etiqueta. Al insertar el identifica-

dor en la tabla de símbolos, en el campo `núm_cuádrupla` se almacena el número de la cuádrupla recién generada, correspondiente al salto incondicional. Es la forma de indicar que esa cuádrupla está incompleta y que debe completarse cuando se localice la etiqueta.

- El identificador correspondiente a la etiqueta está en la tabla de símbolos y el campo `localizada` contiene el valor `SI`; es decir, ya se ha procesado la instrucción en la que aparece la etiqueta. Éste es el caso más sencillo: sólo es necesario generar una cuádrupla de salto incondicional. El número de la cuádrupla a la que se debe saltar está en el campo `núm_cuádrupla` de la tabla de símbolos.
- El identificador correspondiente a la etiqueta está en la tabla de símbolos y el campo `localizada` contiene el valor `NO`; es decir, todavía no se ha procesado la instrucción en la que aparece la etiqueta, pero ya ha aparecido otra instrucción `GOTO` a la misma etiqueta. En este caso, el campo `núm_cuádrupla` del elemento correspondiente al identificador en la tabla de símbolos contiene el número de la cuádrupla pendiente de completar, correspondiente a la instrucción `GOTO` ya procesada. Puesto que pueden aparecer varias instrucciones `GOTO` a la misma etiqueta antes de que ésta sea localizada, el valor de dicho atributo no puede ser un número único, sino una lista de números de cuádrupla. El mecanismo utilizado para resolver este problema es el siguiente: en el campo `núm_cuádrupla` de la tabla de símbolos se almacena el número de la primera cuádrupla pendiente de completar; en la segunda componente de esta cuádrupla se almacena el número de la siguiente cuádrupla pendiente de completar; y así sucesivamente.

La Figura 6.27 muestra un ejemplo que ilustra el mecanismo de gestión de cuádruplas pendientes de completar para la instrucción `GOTO`. En dicho ejemplo, la lista de cuádruplas pendientes de completar para la etiqueta `et1` sería `r`, `q`, `p`.

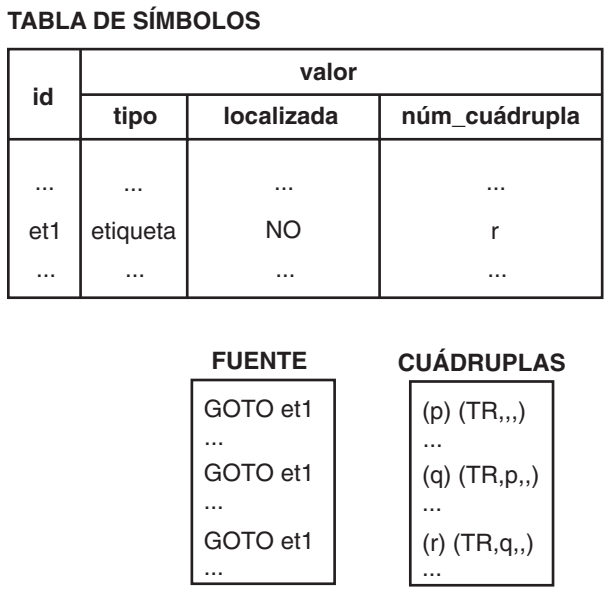


Figura 6.27. Gestión de cuádruplas pendientes de completar.

```
buscar id en la tabla de símbolos;
if (no está)
    insertar(id,etiqueta,SI,cl.sig);
else if (tipo==etiqueta && localizada==NO){
    i=núm_cuádrupla;
    while (i) {
        j=cuádrupla[i][2];
        cuádrupla[i][2]=cl.sig;
        i=j;
    }
    cambiar valor de id a (etiqueta,SI,cl.sig);
}
else error();
```

Figura 6.28. Generación de cuádruplas para la instrucción `etiqueta:`.

La Figura 6.28 muestra el pseudocódigo para la generación de cuádruplas para la instrucción

```
<etiqueta> ::= id : <instruccion>
```

Con una instrucción del tipo `etiqueta:`, pueden darse tres casos:

- El identificador correspondiente a la etiqueta no está en la tabla de símbolos, porque la etiqueta aparece antes de alguna instrucción `GOTO etiqueta`. En este caso, se inserta el identificador correspondiente a la etiqueta en la tabla de símbolos. El valor del campo `núm_cuádrupla` será el valor de la variable global `cl.sig`.
- El identificador correspondiente a la etiqueta está en la tabla de símbolos, pero no está definido como etiqueta o, si lo está, el campo `localizada` contiene el valor `SI`. En tal caso se ha detectado un error, porque la etiqueta ya había sido definida previamente, en el primer caso como variable, en el segundo como etiqueta (etiqueta duplicada).
- El identificador correspondiente a la etiqueta está en la tabla de símbolos, definido como etiqueta, y el campo `localizada` contiene el valor `NO`. Esto ocurre porque la definición de la etiqueta aparece después de una o más instrucciones del tipo `GOTO etiqueta`. En este caso, el bucle `while` que aparece en el pseudocódigo se encarga de completar las cuádruplas pendientes. Además, en la fila correspondiente a la etiqueta en la tabla de símbolos, se asigna el valor `SI` al campo `localizada` y el valor de la variable `cl.sig` al campo `núm_cuádrupla`.

La Figura 6.29 muestra el proceso de generación de cuádruplas para el siguiente esqueleto de código, que, aunque no tiene utilidad, sirve para ilustrar todos los casos descritos anteriormente.

```
GOTO L1;
a:=3;
GOTO L1;
a:=4;
L1: x:=5;
L2: y:=6;
GOTO L2;
```

(1) (TR,,,)

id	valor		
	tipo	localizada	núm_cuádrupla
L1	etiqueta	NO	1

(1) (TR,,,)
(2) (:=.3,aa)
(3) (TR,1,,)
(4) (:=,5,,x)

id	valor		
	tipo	localizada	núm_cuádrupla
L1	etiqueta	NO	3

(1) (TR,4,,)
(2) (:=.3,,a)
(3) (TR,4,,)
(4) (:=,5,,x)
(5) (:=,6,,y)

id	valor		
	tipo	localizada	núm_cuádrupla
L1	etiqueta	SÍ	4

(1) (TR,4,,)
(2) (:=.3,,a)
(3) (TR,4,,)
(4) (:=,5,,x)
(5) (:=,6,,y)
(6) (TR,5,,)

id	valor		
	tipo	localizada	núm_cuádrupla
L1	etiqueta	SÍ	4
L2	etiqueta	SÍ	5

Figura 6.29. Generación de cuádruplas para un ejemplo con etiquetas e instrucciones `GOTO`.

Si se permiten etiquetas locales a bloques, puede aparecer el siguiente caso:

```
L:    ...
      {
        ...
        GOTO L;
        ...
```

En un caso como éste, la instrucción `GOTO L` es ambigua, ya que `L` puede referirse a la etiqueta externa (que podría haber sido localizada previamente, como en este ejemplo, o tal vez no), o también puede referirse a una etiqueta local del bloque que contiene a la instrucción. Esta ambigüedad puede resolverse utilizando un compilador en dos pasos, o forzando a que las etiquetas se declaren como el resto de los identificadores. Una tercera forma de resolver la ambigüedad sería tratar la etiqueta `L` que aparece en el bloque como si tuviese que ser local. Si al final del bloque se descubre que no ha sido definida, pasará a considerarse como global. La lista de cuádruplas pendientes de completar debería entonces fundirse con la lista que corresponde a la etiqueta `L` global (si dicha etiqueta no ha sido localizada aún). En el caso de que la etiqueta `L` global ya haya sido localizada, deberán completarse las cuádruplas pendientes de completar correspondientes a la etiqueta `L` local. Si la etiqueta `L` global no estaba en la tabla de símbolos del bloque externo, debe crearse en ella, y su lista de cuádruplas pendientes de completar será la misma que la de la etiqueta `L` local.

Bucles

Para generar las cuádruplas correspondientes a un bucle `for` son necesarias las cinco acciones semánticas que aparecen entre paréntesis.

```
<bucle> ::= for <id> = <n1> (1)
            , <n2> (2)
            <CD1> do <instr> end S5
<CD1>   ::= , <n3> (3) | λ (4)
```

El contenido de las acciones semánticas es el siguiente:

- (1): generar cuádrupla (:=,n1,,id)
  i=cl_sig
- (2): generar cuádrupla (TRG,,id,n2)
  generar cuádrupla (TR,,,)
- (3): generar cuádrupla (+,id,n3,id)
  generar cuádrupla (TR,i,,)
  cuádrupla[i+1][2]=cl.sig
- (4): generar cuádrupla (+,id,1,id)
  generar cuádrupla (TR,i,,)
  cuádrupla[i+1][2]=cl.sig
- (5): generar cuádrupla (TR,(i+2),,)
  cuádrupla[i][2]=cl.sig

La Figura 6.30 muestra el proceso de generación de cuádruplas para el bucle

```
for x = n1,n2,n3 do a:=a+1; end
```

Como puede apreciarse en la Figura 6.39, la acción semántica `(4)` no se ejecuta en este caso, porque dicha acción sólo se ejecuta si no aparece el valor `n3`.

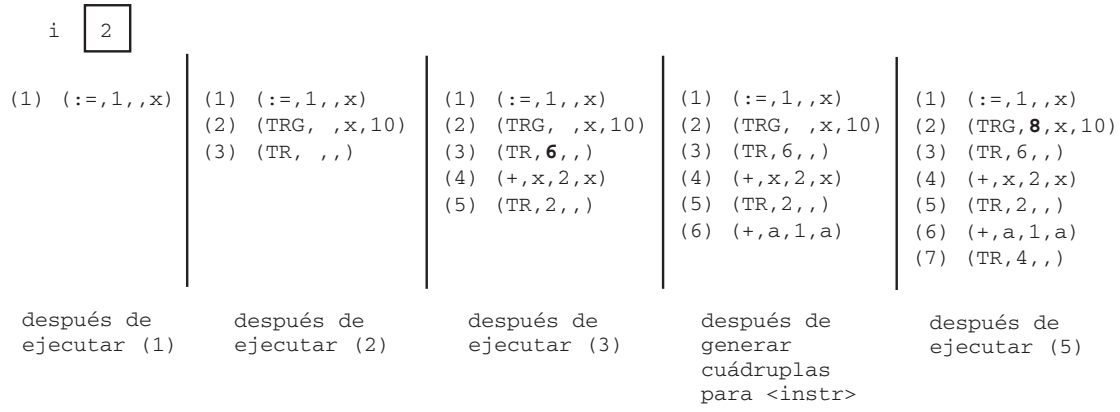


Figura 6.30. Generación de cuádruplas para la instrucción `for x=n1,n2,n3 do a:=a+1; end`.

Generación de código ensamblador a partir de cuádruplas

Una forma de generar código ensamblador a partir de cuádruplas consiste en implementar un procedimiento para cada uno de los operadores que pueden aparecer en la primera posición de una cuádrupla. Cada cuádrupla se traducirá en una llamada a uno de estos procedimientos.

Los procedimientos realizarán llamadas al procedimiento `GEN`, que escribirá en el fichero que contiene el código ensamblador las instrucciones correspondientes, y a la función `CAC`, que cargará el valor de una variable en el acumulador, tal como se indicó en la Sección 6.1.1.

Una cuádrupla del tipo `(+, O1, O2, R)` se traduce a una llamada al procedimiento `SUMA(O1, O2, R)`.

```
SUMA  (opd *x, opd *y, opd *z)
     {
       if (CAC (x, y)) GEN("ADD", "EAX", y)
       else GEN ("ADD", "EAX", x);
       AC=z;
     }
```

Este mismo procedimiento podría aplicarse a otras funciones diádicas conmutativas, como la multiplicación, en la que se sustituiría `ADD` por `MUL`.

Una cuádrupla del tipo `(-, O1, O2, R)` se traduce a una llamada al procedimiento `RESTA(O1, O2, R)`.

```
RESTA (opd *x, opd *y, opd *z)
     {
       CAC (x, NULL);
       GEN("SUB", "EAX", y)
       AC=z;
     }
```

Este mismo procedimiento podría aplicarse a otras funciones diádicas conmutativas, como la división, en la que se sustituiría `SUB` por `DIV`.

Para funciones monádicas, como el cambio de signo, podría aplicarse un procedimiento como el siguiente:

```
NEG (opd *x, opd *z)
     {
       CAC (x, NULL);
       GEN ("NEG EAX");
       AC=z;
     }
```

Una cuádrupla del tipo `(@, O1, , R)` se traduce a una llamada al procedimiento `NEG(O1, R)`.

Como ejemplo, en la Figura 6.31 aparecen las cuádruplas correspondientes a la expresión `a*((a*b+c)-c*d)`, así como el código ensamblador generado a partir de ellas.

Cuádrupla	Llamada a CAC	Se genera	Valor de AC
(*,a,b,t1)	CAC(a,b)	MOV AC,A MUL AC,B	a t1
(+,t1,c,t2)	CAC(t1,c)	ADD AC,C	t2

(*,c,d,t3)	CAC(c,d)	MOV T2,AC
		MOV AC,C
		MUL AC,D
			c
			t3

(-,t2,t3,t4)	CAC(t2,NULL)	MOV T3,AC
		MOV AC,T2
		SUB AC,T3
			t2
			t4

(*,a,t4,t5)	CAC(a,t4)	MUL AC,A	t5

Figura 6.31. Código ensamblador para la expresión `a*((a*b+c)-c*d)`.

6.3 Resumen

Este capítulo describe el módulo de generación de código de un compilador, cuyo objeto es generar el código equivalente al programa fuente, escrito en un lenguaje diferente. En primer lugar, se describe el proceso de la generación directa de código ensamblador en un solo paso. Se utiliza un ensamblador típico, aplicable a la familia 80x86 a partir del microprocesador 80386, en modo de funcionamiento de 32 bits. Se describe cómo se realiza la gestión de los registros de la máquina y cómo se genera código para las expresiones, tanto aritméticas como de comparación, y para la desreferenciación de punteros. Además, se analizan instrucciones de distintos tipos: asignaciones, entrada y salida de datos, condicionales, bucles y llamadas a funciones.

Se dedica otro apartado a dos formas de código intermedio: la notación sufija y las cuádruplas. Para ambas notaciones, se describe su generación en análisis ascendente y descendente. En relación con la notación de cuádruplas, se estudia también con detalle su generación para las instrucciones condicionales, las de salto a etiquetas y los bucles. En los compiladores de dos o más pasos, a partir del código intermedio se realiza la generación del código definitivo, por lo que en este capítulo se describe también el proceso de generación de dicho código a partir de cada una de las dos notaciones consideradas.

6.4 Ejercicios

1. Convertir en cuádruplas el programa C

```
fac=1; for (i=0; i<n; i++) fac*=i+1;
```

2. Poner en notación sufija la expresión

```
(-a+2*b)/a+(c/d-1)/a*a
```

3. Convertir en cuádruplas el programa C

```
fac=1; while (n>1) fac*=n—;
```

4. Poner en notación sufija la expresión:

```
(a-2/b)/a*a+(-c*d+3)/a
```

5. Convertir en cuádruplas el programa C

```
int a,b,c,d,i;
...
a = b+c;
for (i=0; i<a; i++) d+=(b+c)*i;
```

6. Dada la expresión

```
a=(a+b)*c+3/(a+b)
```

generar las cuádruplas equivalentes.

7. Poner en notación sufija la expresión

```
2+((-a-b)*c+d)/(a*(-a-1))
```

8. Construir las cuádruplas equivalentes a las instrucciones siguientes:

```
if (a=b) then
  do i:=1,n+1
      a:=(-b)-a*7
  end
else a:=a+1
```

9. Convertir a notación sufija la expresión siguiente:

$(-b)-a*(x+7)^2$

10. Dada la expresión

```
if ((a+b)<(c*d))
  a=a+b-(a+b)/(c*d)
else
  a=c*d-(a+b)/(c*d)
```

generar las cuádruplas equivalentes.

11. Poner en notación sufija la expresión

$a+((-2-b)^{c+d})/(3*(-b-a))$

12. Generar las cuádruplas equivalentes para el siguiente programa:

```
int f(int y) {
  int x,z;
  z=1;
   ......
  if (y>0)
    for (x=1; x<y; ) {
      x*=2;
      z*=2;
    }
  else x=0;
  z*=2;
  return x;
}
```

13. Generar las cuádruplas equivalentes para el siguiente programa:

```
int x, y, z, m, n, p;
   ......
m = y + z;
x = 1;
while (x < n) {
   p =(y+z)*x;
   x++;
}
```

14. Generar las cuádruplas equivalentes para el siguiente programa:

```
int a = 2, b = 8, c = 4, d;

for(i=0; i<5; i++){
    a = a * (i* (b/c));
    d = a * (i* (b/c));
}
```

15. Generar las cuádruplas equivalentes para el siguiente programa:

```
int a;
float b;
   ......
a = 4 + 3;
a = 5;
b = a + 0.7;
```

Capítulo 7

Optimización de código

La optimización de código es la fase cuyo objetivo consiste en modificar el código objeto generado por el generador de código, para mejorar su rendimiento. Esta fase puede realizarse, bien en un paso independiente, posterior a la generación de código, o bien mientras éste se genera.

Cuando los programadores escriben directamente código en el lenguaje objeto (sea éste un lenguaje simbólico o de alto nivel), pueden aplicar toda su experiencia y habilidad en la generación de un código suficientemente eficiente. La generación de código por parte de compiladores e intérpretes ha de ser automática y general y, por ello, es difícil que mantenga la pericia del experto humano. Los sistemas automáticos no llegan, en general, a realizar su trabajo con la misma calidad que los expertos humanos. Los beneficios de la automatización son distintos, y compensan con creces la disminución inherente en la calidad del resultado: aplicación del conocimiento en lugares y situaciones en las que no sería posible la presencia de un experto humano; incremento de la productividad; independencia respecto a factores subjetivos.

Se sabe que la optimización absoluta es indecidible, es decir, no puede saberse con certeza si una versión concreta de código objeto es la más eficiente posible. El objetivo de esta fase sólo puede ser, por tanto, proporcionar una versión que mejore en algo el código generado.

Otra peculiaridad de esta fase es que puede interferir en los objetivos de otras partes de los compiladores e intérpretes, como, por ejemplo, la depuración. Los procesadores de lenguaje que permiten realizar depuraciones muestran al programador las instrucciones del programa fuente mientras el programa objeto se está ejecutando, permiten observar y modificar los valores que toman las variables del programa, así como continuar la ejecución o detenerla de nuevo cuando se considere conveniente. Sin embargo, como resultado de la optimización, el código asociado con algunas secciones del programa fuente podría desaparecer, lo que haría imposible su depuración.

El objetivo de este capítulo es mostrar algunas técnicas e ideas cuya aplicación pueda dar lugar a alguna mejora en el código objeto generado. Las especificaciones de los compiladores e intérpretes reales son las que determinan el diseño final de la estrategia de optimización.

7.1 Tipos de optimizaciones

Las optimizaciones se pueden dividir en dos grandes grupos, en función de que se puedan aplicar únicamente en una máquina concreta, o en cualquiera.

7.1.1. Optimizaciones dependientes de la máquina

Para aplicarlas, la máquina debe proporcionar las herramientas necesarias. Se describirán los siguientes ejemplos de este tipo de optimizaciones:

- Minimización del uso de registros en máquinas en las que no se disponga de un conjunto de registros muy grande. Puede llegar a generarse código que utilice sólo un registro. Esta cuestión se ha analizado anteriormente, en la Sección 6.1.1.
- Uso de instrucciones especiales de la máquina, que supongan una optimización respecto al uso de construcciones más generales, presentes en todos los lenguajes de máquina.
- Reordenación de código: algunas arquitecturas son más eficientes cuando las operaciones se ejecutan en un orden determinado. Modificando el código para sacar provecho de ese orden se puede optimizar el programa objeto.

7.1.2. Optimizaciones independientes de la máquina

Mejoran la eficiencia sin depender de la máquina concreta utilizada. En este capítulo se explicarán con detalle los siguientes ejemplos de este tipo de optimización:

- Ejecución parcial del código por parte del compilador, en lugar de retrasar su ejecución al programa objeto.
- Eliminación de código que resulta redundante, porque previamente se ha ejecutado un código equivalente.
- Cambio de orden de algunas instrucciones, que puede dar lugar a un código más eficiente.
- Es frecuente que los bucles sean poco eficientes, porque se ejecuten en su cuerpo instrucciones que podrían estar fuera de él, o porque la reiteración inherente al bucle multiplique la ineficiencia causada por el uso de operaciones costosas, cuando podrían utilizarse otras menos costosas y equivalentes.

7.2 Instrucciones especiales

Algunas máquinas tienen instrucciones *especiales*, cuyo objetivo es facilitar la codificación y acelerar la ejecución, mediante el uso de operadores de mayor nivel de abstracción. Por ejemplo:

- Las instrucciones de la máquina `TRT` en la arquitectura IBM 390, y `XLAT` en la arquitectura INTEL, permiten realizar la traducción de un sistema de codificación (como, por ejemplo,

ASCII) a otro diferente en una sola instrucción de la máquina. Estas instrucciones pueden utilizarse también para buscar la primera aparición de un valor en una serie de datos.

- La instrucción `MOV` en la arquitectura IBM 390 permite copiar bloques de memoria de hasta 255 caracteres. De igual manera, la instrucción `REP` en la arquitectura INTEL permite copiar, comparar o introducir información en bloques de memoria, utilizando como registros índices `ESI` y `EDI`.
- La instrucción `TEST` en INTEL permite realizar fácilmente varias comparaciones booleanas simultáneas. Por ejemplo, la comparación `if (x&4 || x&8)`, escrita en el lenguaje C, se puede traducir así:

```
    TEST x,12
    JZ L
    ...
L:
```

7.3 Reordenación de código

En algunas circunstancias, la reordenación de las instrucciones del programa fuente permite reducir el tamaño o la complicación del código objeto. Esto pasa, por ejemplo, cuando hay que calcular varias veces el mismo resultado intermedio: generándolo una sola vez antes de utilizarlo, se puede obtener una versión optimizada.

Ejemplo 7.1 En muchas máquinas, la multiplicación en punto fijo de dos operandos enteros da como resultado un operando de longitud doble, mientras la división actúa sobre un operando de longitud doble y otro de longitud sencilla para generar un cociente y un resto de longitud sencilla. Una simple reordenación de las operaciones puede dar lugar a optimizaciones.

Supóngase un lenguaje fuente en el que la asignación se realice con el símbolo '=' y los operadores '*', '/' y '%' realicen, respectivamente, las operaciones multiplicación, cociente entero y resto de la división. Se supondrá que el objetivo de la traducción es un lenguaje de la máquina o simbólico del tipo INTEL, en el que:

- La instrucción `CDQ` extiende el signo del registro `EAX` al registro `EDX`, para que el operando pase a ocupar el par `EDX:EAX`.
- El operador `IMUL` realiza la multiplicación entera del registro `EAX` y una posición de memoria. El producto se almacena en el par de registros `EDX:EAX`.
- El operador `IDIV` realiza la división entera entre el par `EDX:EAX` y una posición de memoria. El cociente de la división se almacena en el registro `EAX` y el resto en `EDX`.
- El símbolo ';' inicia comentarios que terminan con el final de la línea.

Sea la expresión `a=b/c*d`. Un generador de código poco sofisticado podría generar el siguiente programa objeto equivalente:

```
MOV  EAX,B     ;1 EAX ← B
CDQ            ;2 EDX:EAX ← EAX (extensión de signo)
IDIV EAX,C     ;3 EAX ← B/C, EDX ← B%C
IMUL EAX,D     ;4 EAX←(B/C)*D
MOV  A,EAX     ;5 A ← EAX
```

Sin embargo, si la expresión anterior se reordena, aprovechando que la multiplicación y la división son asociativas, podríamos generar código para calcular `a=b*d/c`:

```
MOV  EAX,B     ;1 EAX ← B
IMUL EAX,D     ;4 EDX:EAX←B*D
IDIV EAX,C     ;3 EAX ← B*D/C
MOV  A,EAX     ;5 A ← EAX
```

Este código tiene una instrucción menos que el anterior.

Ejemplo 7.2

En el lenguaje del ejemplo anterior se podrían escribir las siguientes instrucciones:

```
a=b/c;
d=b%c;
```

El siguiente código, escrito en este lenguaje, es equivalente a dicho fragmento fuente:

```
MOV  EAX,B     ;1 EAX ← B
CDQ            ;2 EDX:EAX ← EAX (extensión de signo)
IDIV EAX,C     ;3 EAX ← B/C, EDX ← B%C
MOV  A,EAX     ;4 A←EAX(B/C)
MOV  EAX,B     ;5 EAX ← B
CDQ            ;6 EDX:EAX ← EAX (extensión de signo)
IDIV EAX,C     ;7 EAX ← B/C, EDX ← B%C
MOV  D,EDX     ;8 D ← EDX(B%C)
```

El análisis de este fragmento muestra que las tres primeras instrucciones hacen exactamente lo mismo que las instrucciones quinta, sexta y séptima. Además, tras la tercera instrucción ya está el resto de la división en el registro `EDX`. Podría aprovecharse esta situación para reducir el código de la siguiente manera:

```
MOV  EAX,B     ;1 EAX ← B
CDQ            ;2 EDX:EAX ← EAX (extensión de signo)
IDIV EAX,C     ;3 EAX ← B/C, EDX ← B%C
MOV  A,EAX     ;4 A←EAX(B/C)
MOV  D,EDX     ;8 D ← EDX(B%C)
```

7.4 Ejecución en tiempo de compilación

En algunas secciones del código, casi siempre relacionadas con las expresiones aritméticas y las conversiones de tipo, se puede elegir entre generar el código objeto que realizará todos los cálcu-

los o realizar en el compilador parte de ellos, de forma que el código generado tenga que realizar menos trabajo y resulte, por tanto, más eficiente.

Para realizar esta optimización, es necesario que el compilador lleve cuenta, de forma explícita y siempre que sea posible, del valor que toman los identificadores en cada momento. Esto puede hacerse directamente en la tabla de símbolos o en una estructura de datos al efecto. Para asegurar que los resultados son correctos, el compilador debe mantener la tabla permanentemente actualizada.

7.4.1. Algoritmo para la ejecución en tiempo de compilación

Para explicar esta optimización, se describirá un algoritmo aplicable a cuádruplas. Dado que las cuádruplas son una abstracción de los lenguajes ensambladores y de la máquina, podrán usarlo casi todos los compiladores, aunque no generen esta representación intermedia:

- Se supondrá que se dispone de una tabla (`T`), en la que se conservará la información de las variables del programa fuente: los identificadores y sus valores.
- Se selecciona el conjunto de cuádruplas objeto de la optimización. Casi siempre será el que corresponda a alguna expresión aritmética.
- Se tratan todas las cuádruplas en el orden en el que aparecen y se aplica reiteradas veces el siguiente tratamiento, según su tipo. Para aplicar el tratamiento, hay que utilizar la Tabla 7.1.

Tabla 7.1

Estructura de la cuádrupla	Tratamiento
1. `(op, op`$_1$`, op`$_2$`, res)`, `op`$_1$ es un identificador y (`op`$_1$, `v`$_1$) está en `T`	Se sustituye en la cuádrupla `op`$_1$ por `v`$_1$
2. `(op, op`$_1$`, op`$_2$`, res)`, `op`$_2$ es un identificador y (`op`$_2$, `v`) está en la tabla `T`	Se sustituye en la cuádrupla `op`$_2$ por `v`$_2$
3. `(op, v`$_1$`, v`$_2$`, res)`, donde `v`$_1$ y `v`$_2$ son valores constantes o nulos	Si al evaluar `v`$_1$ `op` `v`$_2$ se produce un error: • Se avisa del mismo[1] y se deja la cuádrupla original. En otro caso: • Se elimina la cuádrupla. • Se elimina de `T` el par (`res`, `v`), si existe. • Se añade a `T` el par (`res`, `v`$_1$ `op` `v`$_2$).
4. `(=, v`$_1$`, , res)`	• Se elimina de `T` el par (`res`, `v`), si existe. • Si `v`$_1$ es un valor constante, se añade a `T` el par (`res`, `v`$_1$).

[1] Lo único correcto es avisar, ya que puede ser que la cuádrupla que presenta el error en realidad nunca se ejecute. Por ejemplo, en la instrucción `if (false) a=1/0;`.

- El tipo de la cuádrupla determina el tratamiento adecuado.
- Hay que consultar la Tabla 7.1 en el orden en que aparecen sus filas y elegir el tratamiento cuya condición se satisface primero. Es decir, en caso de que sea aplicable más de un tratamiento, hay que elegir el que esté más arriba en la tabla.
- El resultado de cada cuádrupla se trata reiteradamente hasta que no se produce ningún cambio.

Ejemplo 7.3 Se va a aplicar la optimización de ejecución en tiempo de compilación al siguiente bloque de programa, escrito en el lenguaje C:

```
{    int i;
     float f;

     i=2+3;
     i=4;
     f=i+2.5;    }
```

La siguiente secuencia de cuádruplas es equivalente al bloque anterior. En ellas se utiliza el operador `CIF`, que significa ***c**onvertir* entero (***i**nteger*) en real (***f**loat*).

```
(+,      2,     3,   t1)
(=,     t1,      ,    i)
(=,      4,      ,    i)
(CIF,    i,      ,   t2)
(+,     t2,   2.5,   t3)
(=,     t3,      ,    f)
```

La Tabla 7.2 muestra los pasos del algoritmo para este caso:

Tabla 7.2

Cuádruplas	T	Tratamiento
	{}	
(+, 2, 3, t1)	{(t1,5)}	**Caso 3**: 2+3 se evalúa sin errores. Su resultado es 5. • Se ***elimina*** la cuádrupla. • No hay ningún par para t1 en T. • Se añade a T el par (t1, 5) en Q.
(=,t1, , i)	{(t1,5), (i,5)}	**Caso 2**: T contiene el par (t1, 5) • Se ***sustituye*** en la cuádrupla t1 por 5. ***(=, 5, , i)*** **Caso 4**: • T no contiene ningún par para i. • 5 es un valor constante, se añade (i, 5) a T.

Tabla 7.2 ***(continuación)***

Cuádruplas	T	Tratamiento
(=, 4, , i)	{(t1,5), (i,4)}	**Caso 4**: • Se elimina de T el par (i, 5). • 4 es un valor constante, se añade (i, 4) a T.
(CIF, i, , t2)	{(t1,5), (i,4), (t2,4.0)}	**Caso 1:** • Se ***sustituye*** en la cuádrupla i por 4. ***(CIF, 4, , t2)*** **Caso 3:** CIF 4 se evalúa sin errores, su resultado es 4.0. • Se ***elimina*** la cuádrupla. • No hay ningún par para t2 en T. • Se añade a T el par (t2, 4.0) en.
(+,t2, 2.5, t3)	{(t1,5), (i,4), (t2,4.0), (t3,6.5)}	**Caso 1:** • Se ***sustituye*** en la cuádrupla t3 por 4.0. ***(+, 4.0, 2.5, t3)*** **Caso 3:** 4.0+2.5 se evalúa sin errores, su resultado es 6.5. • ***Se elimina*** la cuádrupla. • No hay ningún par para t3 en T. Se añade a T el par (t3, 6.5) en.
(=,t3, , f)	{(t1,5), (i,4), (t2,4.0), (t3,6.5), (f,6.5)}	**Caso 1:** • Se ***sustituye*** en la cuádrupla t3 por 6.5. ***(=, 6.5, , f)*** **Caso 4**: • No hay ningún par para f en T. • 6.5 es un valor constante, se añade (f, 6.5) a T.

La Tabla 7.3 describe juntos los resultados de cada paso, que están resaltados en la columna *Tratamiento*:

Tabla 7.3

Cuádrupla original	Resultado
(+, 2, 3, t1)	Eliminada
(=, t1, , i)	(=, 5, , i)
(=, 4, , i)	(=, 4, , i)
(CIF, i, , t2)	Eliminada
(+, t2, 2.5, t1)	Eliminada
(=, t3, , f)	(=, 6.5, , f)

La comparación de la secuencia original de cuádruplas con el resultado del algoritmo muestra claramente la optimización obtenida.

7.5 Eliminación de redundancias

Esta optimización también se relaciona, casi siempre, con el código generado para las expresiones aritméticas. Como se ha visto en el Capítulo 6, para generar código para una expresión es necesario dividirla en una secuencia de muchos cálculos intermedios. El lugar donde se van almacenando los resultados parciales debe expresarse de forma explícita en cada uno de los pasos. Es frecuente que muchas operaciones intermedias resulten redundantes, por ejemplo, porque se calcule de nuevo algún dato intermedio que ya está disponible en una variable auxiliar.

Ejemplo 7.4

Considérese el siguiente fragmento de código escrito en el lenguaje de programación C:

```
int a,b,c,d;
a = a+b*c;
d = a+b*c;
b = a+b*c;
```

Este ejemplo es ideal para resaltar redundancias y estudiar técnicas para eliminarlas, ya que a las tres variables se les asigna el resultado de expresiones muy similares.

Con los algoritmos de generación de cuádruplas explicados en el Capítulo 6 podría obtenerse la secuencia de cuádruplas de la Tabla 7.4.

Tabla 7.4

`int a,b,c,d;`	

`a = a+b*c;`	`(*, b, c, t1)`
	`(+, a, t1, t2)`
	`(=, t2, , a)`

`d = a+b*c;`	`(*, b, c, t3)`
	`(+, a,t3, t4)`
	`(=, t4, , d)`

`b = a+b*c;`	`(*, b, c, t5)`
	`(+, a,t5, t6)`
	`(=, t6, , b)`

Obsérvese que a cada instrucción le corresponde un grupo de tres cuádruplas con la misma estructura:

- La primera almacena el valor de `b*c` en una nueva variable temporal.

- La segunda almacena en una nueva variable temporal el valor de la suma con la variable `a` de la variable temporal creada en la primera cuádrupla.
- La tercera asigna el resultado, contenido en la variable temporal creada en la segunda cuádrupla, a la variable correspondiente, según el código fuente.

En este ejemplo, es fácil identificar las redundancias mediante la simple observación del código: en la primera instrucción son necesarias las tres cuádruplas. En la segunda, puesto que `b` y `c` no han cambiado de valor (aunque `a` sí ha cambiado), en lugar de calcular de nuevo el valor de su producto, se puede tomar directamente de la variable auxiliar `t1`. Para la tercera cuádrupla, no ha cambiado el valor de ninguna de las tres variables, por lo que el valor de la expresión completa puede tomarse directamente de la variable `t4`.

La secuencia de cuádruplas tras esta optimización sería la que muestra la Tabla 7.5.

Tabla 7.5

(*, b, c, t1) (+, a,t1, t2) (=, t2, , a)
(+, a,t1, t4) (=, t4, , d)
(=, t4, , b)

El objetivo de esta sección es proporcionar un algoritmo que automatice la identificación y reducción de las redundancias.

7.5.1. Algoritmo para la eliminación de redundancias

Para ello se utiliza el concepto de *dependencia*: los identificadores dependen de la cuádrupla en la que se les asigna valor; las cuádruplas dependen de sus operadores. El concepto de dependencia relaciona, por tanto, las cuádruplas y los identificadores, teniendo en cuenta la existencia de las variables auxiliares para los resultados. Como se explicó en el Capítulo 6, la técnica para asegurar su gestión correcta consiste en llevar un contador de variables auxiliares, que se incrementa cada vez que se necesita una variable auxiliar nueva.

El algoritmo tiene los siguientes pasos:

1. Se asigna la dependencia inicial –1 a cada variable de la tabla de símbolos del compilador.
2. Se numeran las cuádruplas que van a ser tratadas por este algoritmo. A la primera de ellas le corresponde el número 0.

3. Desde i=0 y mientras i < número total de cuádruplas, se aplica el siguiente proceso:

   a. La dependencia de la cuádrupla número i se calcula sumando 1 al máximo de las dependencias de sus operandos. La dependencia de los operandos constantes se supondrá igual a -1.
   b. Si la cuádrupla número i tiene como resultado el identificador `id`, la dependencia del identificador `id` se hace coincidir con el número de la cuádrupla.
   c. Se estudia si la cuádrupla i es equivalente a otra j anterior (con j<i). Lo es si se cumplen las siguientes condiciones:

      - Para las cuádruplas que no sean asignaciones, todos sus campos coinciden, excepto el del resultado, y también coinciden sus dependencias.
      - Para las cuádruplas de asignaciones, la componente del resultado también tiene que coincidir.

   d. En ese caso se realizan los siguientes cambios en el conjunto de cuádruplas:

      - La cuádrupla i es sustituida por una nula que apunte a la cuádrupla anterior a la que es equivalente. Para ello se utiliza el operador "`COMO`". Estas cuádruplas sólo se utilizan durante el algoritmo, ya que posteriormente son eliminadas, pues no se necesita generar código para ellas. La nueva cuádrupla es `(COMO, j, , )`.
      - En adelante, en todas las cuádruplas en las que aparezca la variable del resultado de la cuádrupla i, se sustituye ésta por el de la cuádrupla j.

   e. Se incrementa en 1 el valor de i.

Obsérvese que, para la última sustitución, anterior al paso e, no es necesario realizar un recorrido por las cuádruplas siguientes a la i. Este cambio puede incorporarse al tratamiento de cada cuádrupla, siempre que se añada lo siguiente antes del paso a:

- Cualquiera de las variables que coincida con el resultado de una cuádrupla sustituida por otra de tipo `(COMO, j, , )` se reemplaza por el resultado de la cuádrupla `j`.

A continuación se aplica el algoritmo al Ejemplo 7.4. Hay cuatro variables en la tabla de símbolos: `a`, `b`, `c` y `d`. Se les asigna inicialmente una dependencia igual a –1. Se trata la cuádrupla número 0 (*véase* la Tabla 7.6).

Tabla 7.6

i	Operador	Operando	Operando	Resultado	Dependencia	Variable	Dependencia
0	*	b	c	t1		a	–1
						b	–1
						c	–1
						d	–1

Las dependencias de sus operandos, `b` y `c`, son iguales a –1, por lo que le corresponde una dependencia de –1+1=0. Se añade a la tabla el identificador resultado (`t1`) con una dependencia que coincide con el número de la cuádrupla en la que toma valor (0). *Véase* la Tabla 7.7.

Tabla 7.7

i	Operador	Operando	Operando	Resultado	Dependencia	Variable	Dependencia
0	*	b	c	t1		a	–1
						b	–1
						c	–1
						d	–1
						t1	0

Se trata ahora la cuádrupla número 1. Las dependencias de sus operandos, `a` y `t1`, son respectivamente –1 y 0, por lo que se asigna a la cuádrupla una dependencia de 0+1=1. Se añade a la tabla el identificador resultado (`t1`) con una dependencia igual al número de la cuádrupla (1). *Véase* la Tabla 7.8.

Tabla 7.8

i	Operador	Operando	Operando	Resultado	Dependencia	Variable	Dependencia
0	*	b	c	t1	0	a	–1
1	+	a	t1	t2	1	b	–1
						c	–1
						d	–1
						t1	0
						t2	1

Se trata la cuádrupla número 2. La dependencia de su operando, `t2`, es 1, por lo que se asigna a la cuádrupla una dependencia de 2. El resultado se asigna a la variable `a`, por lo que se modifica su dependencia con el número de la cuádrupla. *Véase* la Tabla 7.9.

Tabla 7.9

i	Operador	Operando	Operando	Resultado	Dependencia	Variable	Dependencia
0	*	b	c	t1	0	a	2
1	+	a	t1	t2	1	b	–1
2	=	t2		a	2	c	–1
						d	–1
						t1	0
						t2	1

Se trata la cuádrupla número 3. Las dependencias de `b` y `c` son iguales a –1, por lo que se asigna a la cuádrupla una dependencia de 0. Se añade a la tabla el identificador resultado (`t3`) con una dependencia igual al número de la cuádrupla (3). *Véase* la Tabla 7.10.

Tabla 7.10

i	Operador	Operando	Operando	Resultado	Dependencia	Variable	Dependencia
0	*	b	c	t1	0	a	2
1	+	a	t1	t2	1	b	–1
2	=	t2		a	2	c	–1
3	*	b	c	t3	0	d	–1
						t1	0
						t2	1
						t3	3

Antes de terminar con esta cuádrupla, se observa que todos sus datos, excepto el identificador del resultado, coinciden con los de la cuádrupla número 0. Se sustituye la cuádrupla por `(COMO, 0, , )`. *Véase* la Tabla 7.11.

Se conserva entre paréntesis la antigua variable resultado de la cuádrupla 3. En adelante, si alguna cuádrupla utiliza como operando la variable `t3`, la aparición de esta variable tendrá que ser reemplazada por `t1`, identificador del resultado de la cuádrupla 0, que aparece en la cuádrupla auxiliar `(COMO, 0, , )`. Obsérvese que esta información también podría deducirse de los datos sobre las dependencias de las variables, ya que la de `t3` coincide con el número de la cuá-

Tabla 7.11

i	Operador	Operando	Operando	Resultado	Dependencia	Variable	Dependencia
0	*	b	c	t1	0	a	2
1	+	a	t1	t2	1	b	–1
2	=	t2		a	2	c	–1
3	COMO	0		(t3)	0	d	–1
						t1	0
						t2	1
						t3	3

drupla donde tomó valor. Al acceder a esa cuádrupla, se constata que es necesario consultar la número 0 para usar su resultado en lugar de `t3`.

Al procesar la cuádrupla número 4, se observa que uno de sus operadores es `t3`. Ya se ha dicho anteriormente que esta variable debe sustituirse por `t1`. Las dependencias de `a` y `t1` son respectivamente 2 y 0, por lo que a la cuádrupla se le asigna 3 como dependencia. Se añade la variable del resultado (`t4`) con el número de la cuádrupla como dependencia. *Véase* la Tabla 7.12.

Tabla 7.12

i	Operador	Operando	Operando	Resultado	Dependencia	Variable	Dependencia
0	*	b	c	t*1*	0	a	2
1	+	a	t1	t2	1	b	–1
2	=	t2		a	2	c	–1
3	COMO	***0***		(*t3*)	0	d	–1
4	+	a	*t3*⇒t1	t4	3	t1	0
						t2	1
						t3	3
						t4	4

Se procesa la cuádrupla número 5. Su único operando tiene una dependencia igual a 4, por lo que se le asigna una dependencia de 5. Su resultado es la variable `d`, por lo que se cambia su dependencia por el número de la cuádrupla (5). *Véase* la Tabla 7.13.

Tabla 7.13

i	Operador	Operando	Operando	Resultado	Dependencia	Variable	Dependencia
0	*	b	c	t1	0	a	2
1	+	a	t1	t2	1	b	–1
2	=	t2		a	2	c	–1
3	COMO	0		(t3)	0	d	5
4	+	a	t1	t4	3	t1	0
5	=	t4		d	5	t2	1
						t3	3
						t4	4

A la cuádrupla 6 se le asigna una dependencia igual a 0 porque sus operandos tienen dependencia –1. Se añade a la tabla la variable de su resultado (`t5`) con una dependencia igual al número de la cuádrupla. *Véase* la Tabla 7.14.

Tabla 7.14

i	Operador	Operando	Operando	Resultado	Dependencia	Variable	Dependencia
0	*	b	c	t1	0	a	2
1	+	a	t1	t2	1	b	–1
2	=	t2		a	2	c	–1
3	COMO	0		(t3)	0	d	5
4	+	a	t1	t4	3	t1	0
5	=	t4		d	5	t2	1
6	*	b	c	t5	0	t3	3
						t4	4
						t5	6

Antes de terminar con su proceso, se observa que la cuádrupla 6 es *como* la 0, ya que coinciden todas sus informaciones excepto la variable del resultado. Se sustituye la cuádrupla 6 por `(COMO, 0, ,)`. En adelante, las apariciones de la variable `t5` serán reemplazadas por `t1` (el resultado de la cuádrupla 0). *Véase* la Tabla 7.15.

Tabla 7.15

i	Operador	Operando	Operando	Resultado	Dependencia	Variable	Dependencia
0	*	b	c	t1	0	a	2
1	+	a	t1	t2	1	b	–1
2	=	t2		a	2	c	–1
3	COMO	0		(t3)	0	d	5
4	+	a	t1	t4	3	t1	0
5	=	t4		d	5	t2	1
6	COMO	0		(t5)	0	t3	3
						t4	4
						t5	6

En la cuádrupla 7 es necesario realizar ese cambio. Sus operandos, que pasan a ser a y t_1, tienen una dependencia máxima de 2, por lo que se asigna a la cuádrupla una dependencia igual a 3. Se añade a la tabla de símbolos la variable resultado (t6) con el número de la cuádrupla como dependencia. *Véase* la Tabla 7.16.

Tabla 7.16

i	Operador	Operando	Operando	Resultado	Dependencia	Variable	Dependencia
0	*	b	c	***t1***	0	a	2
1	+	a	t1	t2	1	b	–1
2	=	t2		a	2	c	–1
3	COMO	0		(t3)	0	d	5
4	+	a	t1	t4	3	t1	0
5	=	t4		d	5	t2	1
6	COMO	***0***		(***t5***)	0	t3	3
7	+	a	***t5***⇒t1	t6	3	t4	4
						t5	6
						t6	7

Antes de terminar con ella, se observa que es *como* la cuádrupla 4. Se realiza el cambio. En la cuádrupla 8 aparece la variable resultado de la cuádrupla 7 original, que ha de ser cambiada por la de la cuádrupla 4 (`t4`) que tiene como dependencia 4, por lo que se asigna a la cuádrupla una dependencia de 5. Se cambia la dependencia de `b`, que es su variable resultado, asignándole el número de la cuádrupla. *Véase* la Tabla 7.17.

Tabla 7.17

i	Operador	Operando	Operando	Resultado	Dependencia	Variable	Dependencia
0	*	b	c	t1	0	a	2
1	+	a	t1	t2	1	b	8
2	=	t2		a	2	c	–1
3	COMO	0		(t3)	0	d	5
4	+	a	t1	***t4***	3	t1	0
5	=	t4		d	5	t2	1
6	COMO	0		(t5)	0	t3	3
7	COMO	***4***		(***t6***)	3	t4	4
8	=	***t6***⇒t4		b	5	t5	6
						t6	7

De esta forma, la secuencia de cuádruplas sin redundancias queda como se muestra a continuación:

```
(*,b,c,t1)
(+,a,t1,t2)
(=,t2,,a)
(+,a,t1,t4)
(=,t4,,d)
(=,t4,,b)
```

Que coincide con el resultado conseguido *a mano*.

7.6 Reordenación de operaciones

Se puede aumentar la eficiencia del código si se tiene en cuenta la conmutatividad y la asociatividad de las operaciones que aparecen en una expresión aritmética. Entre otras cosas, se puede

adoptar un orden preestablecido para los operandos de las expresiones, que permita identificar subexpresiones comunes; maximizar el uso de operaciones monádicas para aumentar la probabilidad de que aparezcan operaciones equivalentes; o reutilizar variables para los resultados intermedios, de forma que se precise el número mínimo de ellas.

7.6.1. Orden canónico entre los operandos de las expresiones aritméticas

Se puede seguir el siguiente orden al escribir las expresiones aritméticas:

1. Términos que no sean variables ni constantes.
2. Variables indexadas (*arrays*) en orden alfabético.
3. Variables sin indexar en orden alfabético.
4. Constantes.

Esto puede facilitar la localización de subexpresiones comunes y la aplicación de otras optimizaciones.

Ejemplo 7.5 Considérense, como ejemplo, las siguientes expresiones aritméticas:

`a=1+c+d+3;` es equivalente a `a=c+d+1+3;`
`b=d+c+2;` es equivalente a `b=c+d+2;`

Al considerar la segunda versión, se puede reducir el número de operaciones al observar que hay una subexpresión común: `c+d`.

Sin embargo, esta técnica no asegura siempre el éxito, como puede verse en el siguiente ejemplo.

Ejemplo 7.6 Como en el caso anterior, considérense las siguientes expresiones:

`a=1+c+d+3;` es equivalente a `a=c+d+1+3;`
`b=d+c+c+d;` es equivalente a `b=c+c+d+d;`

En este caso, existe una expresión común (`c+d`), pero el algoritmo no es capaz de identificarla. Por lo tanto, el alcance de estas optimizaciones es relativo.

7.6.2. Aumento del uso de operaciones monádicas

Esta optimización tiene también un alcance limitado. Se trata de identificar situaciones como la que muestra el siguiente ejemplo.

Ejemplo 7.7 Considérese la siguiente secuencia de instrucciones:

```
a=c-d;
b=d-c;
```

Podría obtenerse la siguiente secuencia equivalente de cuádruplas:

```
     /* a=c-d; */
(-, c, d, t1)
(=, t1, , a)
     /* b=d-c; */
(-, d, c, t2)
(=, t2, , b)
```

Si se analiza la segunda operación, su primera cuádrupla deja en la variable t_2 el mismo valor que contiene la variable t_1, salvo que tiene el signo contrario. Gracias a esto, se podría obtener la siguiente versión en la que se resalta el único cambio:

```
     /* a=c-d; */
(-, c, d, t1)
(=, t1, , a)
     /* b=d-c; */
(-, t1, , t2)
(=, t2, , b)
```

En este caso no se reduce el número de cuádruplas. La optimización consiste en que las operaciones monádicas son, en general, menos costosas que las binarias para la unidad aritmético-lógica.

7.6.3. Reducción del número de variables intermedias

Se ha explicado en el Capítulo 6 que las variables intermedias que aparecen en las cuádruplas acaban siendo, en el código objeto, asignadas a registros, posiciones en la pila del sistema o en la memoria (variables estáticas). Todos estos recursos son limitados. El sistema ofrece un número no muy grande de registros de propósito general de acceso muy rápido. El tamaño de la pila es mayor que el número de registros, aunque también está limitado, pero el acceso a ella es más lento. El espacio mayor es el de la memoria estática, como también es mayor el tiempo necesario para acceder a él. Lo ideal sería, por tanto, que el código objeto sólo utilizara registros. Esto no es siempre posible: al compilar expresiones aritméticas complicadas, puede necesitarse un número de variables intermedias mayor que el de registros. Restringir el número de variables intermedias puede permitir que la generación de código explote formas de almacenamiento de acceso más rápido.

Ejemplo 7.8 Considérense las dos versiones equivalentes de la siguiente expresión aritmética:

`(a*b)+(c+d)` y `((a*b)+c)+d`

Es fácil comprobar que la propiedad asociativa de la suma asegura la equivalencia. Esto posibilita dos secuencias de cuádruplas también equivalentes:

```
(*,a,b,t1)      y     (*,a,b,t1)
(+,c,d,t2)            (+,t1,t2,t1)
(+,t1,c,t1)           (+,t1,d,t1)
```

La segunda variante utiliza una variable menos que la primera.

Ejemplo 7.9 A continuación se presenta una situación similar. La propiedad conmutativa permite afirmar que las dos expresiones siguientes son equivalentes:

```
(a+b)+(c*d)     y     (c*d)+(a+b)
```

Como en el ejemplo anterior, se puede obtener, a partir de cada una de ellas, una secuencia distinta de cuádruplas:

```
(+,a,b,t1)      y     (*,c,d,t1)
(*,c,d,t2)            (+,a,t1,t1)
(+,t1,t2,t1)          (+,t1,b,t1)
```

Y la conclusión es la misma: la segunda versión utiliza una variable auxiliar menos que la primera.

- **Algoritmo para el cálculo del número mínimo de variables auxiliares que necesita una expresión**

La reflexión del apartado anterior justifica este algoritmo, que sólo determina el número mínimo de variables auxiliares que necesita la expresión, pero no describe cómo generar las cuádruplas que las usen.

El algoritmo tiene los siguientes pasos:

1. Construir el grafo (o el árbol) de la expresión. La complejidad y la estructura de la expresión determinarán si es necesario un grafo o basta con un árbol para representarla. Los árboles pueden considerarse casos particulares de los grafos, por lo que en el resto del algoritmo se mencionarán sólo los grafos.
2. Marcar las hojas del grafo con el valor 0.
3. Recorrer el grafo (desde las hojas hacia los padres) marcando los nodos:
   a. Si todos los hijos de un nodo tienen el mismo valor (j), se marca el nodo con el valor j+1.
   b. En otro caso, marcar el nodo padre con el valor máximo de sus hijos.
4. La etiqueta de la raíz del grafo es el número de variables auxiliares que necesita la expresión.

Ejemplo 7.10 La Figura 7.1 muestra el resultado del algoritmo para dos parejas de expresiones equivalentes:

```
(a*b)+(c+d)     y     ((a*b)+c)+d
(a+b)+(c*d)     y     a+(c*d)+b
```

La primera expresión de cada pareja necesita 2 variables auxiliares, la segunda 1.

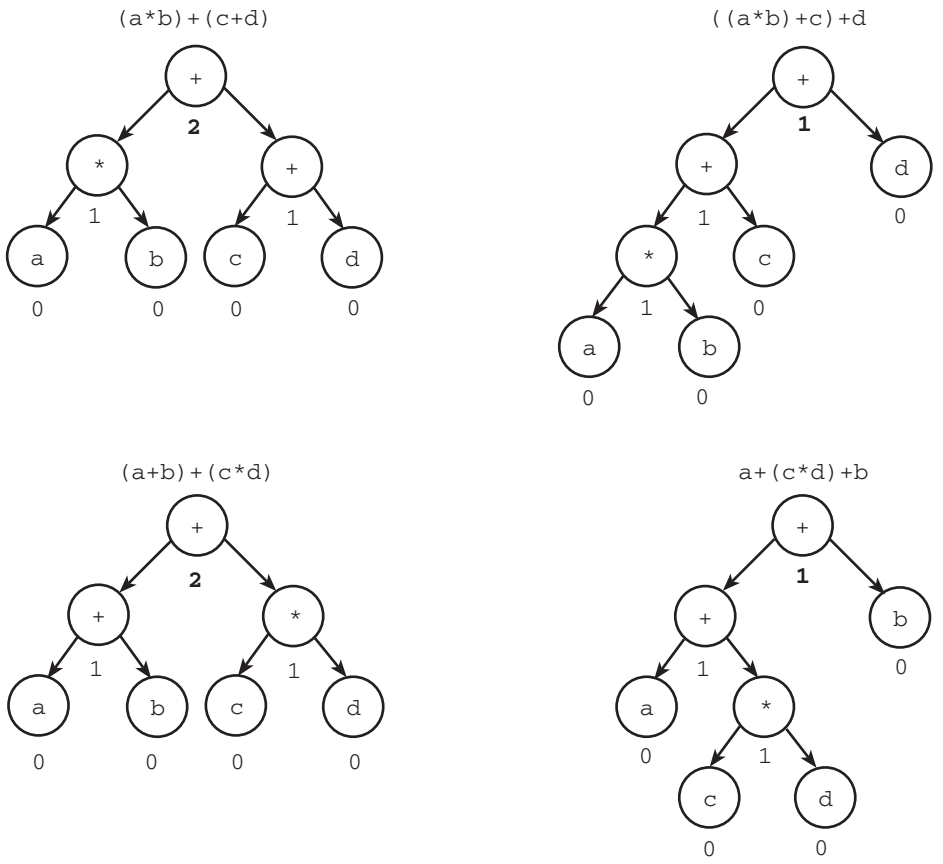


Figura 7.1. Ejemplos de aplicación del algoritmo de determinación del número mínimo de variables auxiliares.

7.7 Optimización de bucles

Los bucles son una parte del código muy propensa a la ineficiencia. Su naturaleza multiplica la diferencia de rendimiento entre dos versiones de código equivalentes, que fuera de un bucle no presentarían una mejora significativa. Por esta razón es importante generar el código de manera cuidadosa, para introducir en los cuerpos de los bucles sólo lo que sea estrictamente necesario, en versiones que minimicen el coste para la máquina. El primer enfoque identifica los *invariantes del bucle*; el segundo se llama *reducción de fuerza*. Una operación es invariante respecto a un bucle si ninguno de los operandos de los que depende cambia de valor durante la ejecución del bucle. La optimización consiste en sacar la operación fuera del bucle. Para estudiar la reducción de fuerza, será conveniente considerar un ejemplo previo.

Ejemplo 7.11 Considérese el siguiente bucle, escrito en el lenguaje C:

```
for (i=a; i<c; i+=b) {... d=i*k; ...}
```

Se supone que:

- `b` y `k` no se modifican dentro del cuerpo del bucle.
- `i`, variable del bucle, sólo se modifica para incrementarla.
- La única modificación del valor de la variable `d` dentro del cuerpo del bucle es la que se muestra.

Se puede comprobar que los valores que toma la variable `d` son los siguientes:

```
a*k
(a+b)*k
(a+2*b)*k
```

Es posible obtener una versión equivalente del bucle, que realice menos trabajo en cada iteración, reduciendo el cálculo del valor siguiente de la variable `d` a un incremento, que se le sumará como si se tratase de una variable del bucle (como la variable `i`):

```
d=a*k;
t1=b*k;
for (i=a; i<c; i+=b, d+=t1) {...}
```

Obsérvese:

- La aparición de una nueva variable (`t1`), que se hace cargo de parte del cálculo del valor de `d`.
- La conversión de `d` en una variable del bucle, que en lugar de calcularse por completo en el cuerpo del bucle simplemente se incrementa.
- Que los valores que recibe la variable `d` en cada ejecución del bucle coinciden con los indicados anteriormente.

La optimización consiste en que el cálculo que se repite dentro del bucle es una suma, en lugar de un producto, y los productos suelen ser más costosos para el sistema.

Esta técnica se puede generalizar, como muestra el siguiente algoritmo.

7.7.1. Algoritmo para la optimización de bucles mediante reducción de fuerza

El algoritmo distingue tres partes bien diferenciadas en el bucle:

- *Inicialización*, en la que se asigna el valor inicial a las variables que se utilizan dentro del bucle.
- *Incremento*, en la que se actualiza el valor de las variables del bucle.
- *Cuerpo*, en la que se realiza el resto de las acciones, que no tienen cabida en las dos partes anteriores.

Ejemplo 7.12 En el bucle obtenido aplicando la reducción de fuerza al Ejemplo 7.11:

```
d=a*k;
t1=b*k;
for (i=a; i<c; i+=b, d+=t1) {...}
```

- La *inicialización* contiene las siguientes instrucciones:

```
d=a*k;
t1=b*k;
i=a;
```

- El *incremento* contiene las dos instrucciones que aparecen en la sección correspondiente de la cabecera del bucle: `i+=b, d+=t1`.
- El *cuerpo* está representado por los puntos suspensivos entre las llaves: `{...}`.

El algoritmo describe ciertas transformaciones en las cuádruplas del cuerpo del bucle original, dependiendo de su tipo, y distribuye el trabajo correspondiente entre las secciones de *inicialización* y de *incremento*. Las transformaciones se basan en los siguientes conceptos:

- *Variable de bucle*: es aquella cuyo único tratamiento dentro del bucle consiste en recibir un valor en la *inicialización* y ser modificada en el *incremento*.
- *Invariante del bucle*: es la variable que no se modifica dentro del bucle.
- *Incremento*: es la expresión que se suma a una variable del bucle en la sección de *incremento*.

El algoritmo toma como entrada la secuencia de cuádruplas del cuerpo del bucle, y las procesa en el orden en el que aparecen, realizando el siguiente tratamiento:

- Si la cuádrupla tiene la estructura

```
(*,variable_bucle,invariante_bucle,variable_resultado)
```

  — Se elimina la cuádrupla del bucle.

  — Se añaden a la parte de *inicialización* las siguientes cuádruplas:

```
(*,variable_bucle,invariante_bucle,variable_resultado)
(*,incremento,invariante_bucle,var_temp_nueva)
```

  — Se añade a la parte de *incremento* la siguiente cuádrupla:

```
(+,variable_resultado,var_temp_nueva,variable_resultado)
```

- Si la cuádrupla tiene la estructura

```
(+,variable_bucle,invariante_bucle,variable_resultado)
```

  — Se elimina la cuádrupla del bucle.

  — Se añade a la parte de *inicialización* la siguiente cuádrupla:

```
(+,variable_bucle,invariante_bucle,variable_resultado)
```

— Se añade a la parte de *incremento* la siguiente cuádrupla:

```
(+,variable_resultado,incremento,variable_resultado)
```

Ejemplo 7.13 Considérese el siguiente bucle escrito en el lenguaje de programación C:

```
for (i=0; i<10; i++) {... a=(b+c*i)*d; ...}
```

Se supondrá que las variables `b`, `c` y `d` son invariantes al bucle. La siguiente secuencia de cuádruples es equivalente al bucle anterior:

```
INICIALIZACION:  (=,0,,i)
        CUERPO:  ...
                 (*,c,i,t1)
                 (+,b,t1,t2)
                 (*,t2,d,t3)
                 (=,t3,,a)
                 ...
    INCREMENTO:  (+,i,1,i)
```

Obsérvese que se han identificado en el bucle sus tres partes: inicialización, incremento y cuerpo. La clave del algoritmo consiste en identificar en cada cuádrupla las variables del bucle, las invariantes y el incremento.

Para identificar en la primera cuádrupla `(*,c,i,t1)` la variable del bucle (`i`) y el incremento (1) es suficiente consultar la parte *incremento* del bucle. El tratamiento especifica que se elimine la cuádrupla, que se añadan a la parte *inicialización* las cuádruplas `(*,c,i,`t_1`)` y `(*,c,1,`t_4`)` (t_4 es una variable temporal nueva) y en la parte *incremento* `(+,`t_1`,`t_4`,`t_1`)`. A continuación se muestra el resultado, en el que se resaltan los cambios:

```
INICIALIZACION:  (=,0,,i)
                 (*,c,i,t1)
                 (*,c,1,t4)
        CUERPO:  ...
                 (+,b,t1,t2)
                 (*,t2,d,t3)
                 (=,t3,,a)
                 ..S.
    INCREMENTO:  (+,i,1,i)
                 (+,t1,t4,t1)
```

Se repite el análisis para la cuádrupla `(+,b,t1,t2)`. La parte *incremento* indica que `t1` es variable de bucle y que su incremento es `t4`. Por tanto, hay que eliminar la cuádrupla, añadir a la parte *inicialización* la cuádrupla `(+,b,t1,t2)`, y a la parte *incremento* la cuádrupla `(+,t2,t4,t2)`.

```
INICIALIZACION:  (=,0,,i)
                 (*,c,i,t1)
                 (*,c,1,t4)
                 (+,b,t1,t2)
```

```
        CUERPO:  ...
                 (*,t2,d,t3)
                 (=,t3,,a)
                 ...
    INCREMENTO:  (+,i,1,i)
                 (+,t1,t4,t1)
                 (+,t2,t4,t2)
```

Es importante señalar que, al ir eliminando cuádruplas del cuerpo del bucle, algunas de las cuádruplas de las otras partes pueden llegar a ser innecesarias, porque se refieran a variables que ya no estén en el bucle. Esto pasa en la penúltima cuádrupla de la parte *incremento*. La variable `t1` ya no aparece en el cuerpo del bucle, por lo que la cuádrupla que la incrementa se puede eliminar. No se hace lo mismo con la primera cuádrupla `(+,i,1,i)`, porque se supone que en los fragmentos omitidos del cuerpo, representados con puntos suspensivos, sí puede aparecer la variable `i`.

```
INICIALIZACION:  (=,0,,,i)
                 (*,c,i,t1)
                 (*,c,1,t4)
                 (+,b,t1,t2)
        CUERPO:  ...
                 (*,t2,d,t3)
                 (=,t3,,a)
                 ...
    INCREMENTO:  (+,i,1,i)
                 (+,t2,t4,t2)
```

La consulta de la sección *incremento* indica que en la cuádrupla `(*,t2,d,t3)`, `t2` es variable del bucle y `t4` su incremento. El algoritmo indica que se elimine la cuádrupla y que se añadan a la parte *inicialización* las cuádruplas `(*,t2,d,t3)` y `(*,t4,d,t5)`, donde `t5` es una nueva variable temporal. También hay que añadir a la parte *incremento* la cuádrupla `(+, t3, t5, t3)`. Como en el paso anterior, con la cuádrupla eliminada desaparecen en el bucle las referencias a la variable `t2`, por lo que la cuádrupla `(+,t2,t4,t2)` ya no es necesaria y puede eliminarse de la parte *incremento*.

```
INICIALIZACION:  (=,0,,,i)
                 (*,c,i,t1)
                 (*,c,1,t4)
                 (+,b,t1,t2)
                 (*,t2,d,t3)
                 (*,t4,d,t5)
        CUERPO:  ...
                 (=,t3,,a)
                 ...
    INCREMENTO:  (+,i,1,i)
                 (+,t3,t5,t3)
```

La última cuádrupla del bucle no puede modificarse, ya que no tiene la estructura adecuada.

7.7.2. Algunas observaciones sobre la optimización de bucles por reducción de fuerza

Las llamadas a las funciones dentro de los bucles añaden dificultad a las optimizaciones, ya que pueden utilizar variables globales o incluso modificarlas. El compilador tiene que tener en cuenta estas circunstancias antes de generalizar a las llamadas a funciones el tratamiento propuesto en este algoritmo.

La existencia de bucles anidados, si en los cuerpos interiores se puede acceder a variables de los exteriores, también dificulta estas optimizaciones y tiene que ser tenido en cuenta por los programadores del compilador.

Si los bucles se repiten poco (0 o 1 veces), puede que el aumento de eficiencia no compense el esfuerzo realizado.

Estas consideraciones tienen como consecuencia que los compiladores que aplican este tipo de optimizaciones necesiten de dos etapas: en la primera se analiza el código fuente y se obtiene información para decidir el tipo de optimización que conviene; en una fase posterior se llevan a cabo las optimizaciones.

7.8 Optimización de regiones

La optimización de un programa completo suele realizarse en varias fases. Primero se aplican algunas optimizaciones parciales, que se aplican sólo a fragmentos concretos. Tras esto puede aplicarse un estudio global, que consiste en dividir el programa en regiones, entre las que discurre el flujo del control del programa. Se puede representar el programa completo mediante un grafo y aplicarle técnicas de simplificación procedentes del álgebra, que generen un programa más eficiente. Muchas de las optimizaciones descritas en las secciones anteriores conviene planificarlas de este modo global, porque esto asegura que los bloques internos se tratarán antes de los externos y que no se perderá ninguna optimización, debido al orden en que se hayan realizado.

Supongamos que se divide un programa en sus bloques básicos de control. Con ellos se puede formar un grafo en el que los nodos son los bloques, mientras los arcos indican que el bloque situado en el origen del arco puede ceder directamente el control de la ejecución al bloque situado al extremo del arco; es decir, la ejecución de los dos bloques puede ser sucesiva, aunque no siempre es obligatorio que lo sea, ya que cada bloque puede ser seguido por varios, en ejecuciones diferentes.

Se llama *región fuertemente conexa* (o simplemente región), del grafo que representa la división de un programa en bloques básicos, a un subgrafo en el que exista un camino desde cualquier nodo del subgrafo a cualquier otro nodo del subgrafo. Se llama *bloque de entrada* de una región al bloque de la región (puede haber más de uno) que recibe algún arco desde fuera de la región. El bloque en que se origina dicho arco, que no pertenece a la región, se llama *predecesor* de la región.

Una vez que se conocen las regiones del programa, se construye con ellas una o más listas de regiones, de la forma $R=\{R_1, R_2, \ldots, R_n\}$, tal que $R_i \neq R_j$ si $i \neq j$, e $i<j \Rightarrow R_i$ y R_j no tienen bloques en común, o bien R_i es un subconjunto propio de R_j.

A menudo puede haber varias listas posibles, con algunas regiones comunes y otras regiones diferentes. Entre todas ellas, es preciso elegir una sola. Conviene que en dicha lista estén todos los bucles, que normalmente tienen un solo nodo predecesor y un solo nodo de entrada. Cuando haya dos posibilidades, se preferirán las regiones con esta propiedad. También es bueno seleccionar la lista que contenga mayor número de regiones.

Ejemplo 7.14 La Figura 7.2 muestra cómo podría dividirse un programa ficticio en ocho bloques básicos. El bloque 1 es el inicio del programa y el bloque 8 su terminación. Mediante instrucciones de control tipo `if-then-else` o bucles, los bloques intermedios (del 2 al 7) permiten realizar flujos de ejecución como los indicados en la figura. En este grafo se pueden distinguir las cinco regiones o subgrafos fuertemente conexos siguientes: (6), (3,5), (2,3,5,6,7), (2,3,4,6,7), (2,3,4,5,6,7). El lector puede comprobar que, desde cualquier nodo de estas regiones, puede alcanzarse cualquier otro nodo de su región (incluido él mismo). Obsérvese que los bloques 1 y 8 no pertenecen a ninguna región.

En el programa de la Figura 7.2, la región (3,5) tiene un bloque de entrada, el nodo 3, que recibe un arco desde el nodo 2, que por tanto es un bloque predecesor de esta región. En cambio, la región (2,3,5,6,7) tiene dos bloques de entrada: el nodo 2, que recibe un arco desde el nodo 1, y el nodo 6, que recibe otro desde el nodo 4. Obsérvese que los nodos 1 y 4, los dos predecesores de la región, no pertenecen a ella.

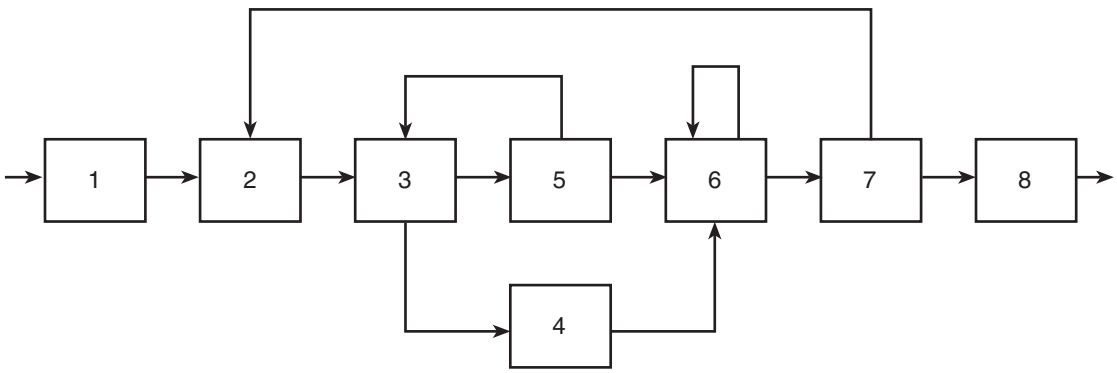


Figura 7.2. Ejemplo de grafo de regiones.

En el ejemplo de la Figura 7.2, una lista válida sería: (6), (3,5), (2,3,5,6,7), (2,3,4,5,6,7). Otra lista válida es: (6), (2,3,4,6,7), (2,3,4,5,6,7). Obsérvese que las dos regiones (2,3,4,6,7) y (2,3,5,6,7) no pueden estar juntas en una lista válida, pues tienen bloques en común, pero ninguna de las dos es subconjunto o subgrafo de la otra. Entre las dos listas, seleccionaremos la primera, que contiene cuatro regiones, mientras la segunda sólo contiene tres.

7.8.1. Algoritmo de planificación de optimizaciones utilizando regiones

Para cada bloque se definen los siguientes vectores booleanos, que tienen tantos elementos como variables forman parte del programa que se está planificando:

- `R[v]=1` si la variable `v` se utiliza dentro del bloque. Se le asignará 0 en caso contrario. `R` es la inicial de *Referencia* (*Reference*, en inglés).
- `A[v]=1` si la variable `v` recibe alguna asignación dentro del bloque. `A` es la inicial de *Asignación* (*Assignment*, en inglés).
- `B[v]=1` si la variable `v` se usa dentro del bloque antes de recibir alguna asignación. Se incluye el caso en que `v` es utilizada, pero no recibe asignación dentro del bloque. `B` es la inicial de *Before* (que significa *antes*, en inglés).

La disyunción lógica (O lógico) de `R[v]` para todos los bloques de una región, nos da el valor que toma `R[v]` para la región entera, e igual pasa con `A[v]`, pero no necesariamente con `B[v]`, pues en una región que contenga varios bloques, una variable que en uno de ellos sea utilizada antes de asignarle nada, podría no cumplir esta propiedad para la región entera.

Las optimizaciones se aplican sucesivamente a las distintas regiones de la lista, de la primera a la última. Una vez optimizada una región, pueden aparecer bloques nuevos, como consecuencia (por ejemplo) de la extracción de instrucciones del interior de los bucles, que pasan a la zona de inicialización, que se encuentra fuera del bucle (*véase* la Sección 7.7).

A continuación se resume el algoritmo completo.

1. Se selecciona una lista de regiones. Se hace `i=1` (`i` es un índice sobre la lista de regiones seleccionada).
2. Se toma la región R_i de la lista seleccionada. Se prepara un bloque `I` de iniciación vacío (sin instrucciones).
3. Se eliminan redundancias dentro de cada bloque de la región.
4. Se construyen los tres vectores booleanos para cada bloque de la región. Estos vectores se utilizan para comprobar si se cumplen las condiciones de optimización de bucles indicadas en la Sección 7.7. Por ejemplo, los invariantes del bucle son las variables para las que `A[v]=0`.
5. Se extraen fuera del bucle las instrucciones invariantes y se reduce la fuerza dentro de la región, con lo que se va llenando el bloque `I`. Se eliminan las asignaciones muertas (asignaciones que nunca se usan, *véase* la Sección 7.9). A medida que se realizan estas optimizaciones, se crean variables temporales y se van modificando los vectores booleanos, cuyos valores y cuyo tamaño van cambiando.
6. Se obtienen los valores de los tres vectores para la región entera.
7. Se insertan copias del bloque `I` entre cada bloque predecesor de la región y el bloque de entrada correspondiente.

8. Se colapsa la región, es decir, se sustituyen todos sus bloques por un bloque único, al que no hace falta aplicar más optimizaciones. A dicho bloque se le aplican los vectores calculados en el paso 6.
9. Se modifica la lista de regiones para tener en cuenta los cambios efectuados en los pasos 7 y 8.
10. `i++`; si `i<=n`, ir al paso 2.
11. Se realiza la optimización de cuádruplas y la eliminación de redundancias en los bloques básicos que no pertenecen a ninguna región.

Ejemplo 7.15 En el ejemplo de la Figura 7.2, la primera región a optimizar es la (6), que está formada por un solo bloque, que evidentemente es un bucle. Esta región tiene dos puntos de entrada externa (ambos dirigidos al bloque 6), cuyos nodos antecesores son 4 y 5. Al aplicar, en el punto 5, el algoritmo de la Sección 7.7, algunas de las instrucciones saldrán del bucle y pasarán a formar parte del nuevo bloque de inicialización. Habrá que colocar dos copias de este bloque (`I1` e `I2`) en todos los puntos de entrada de la región. (En la figura, en medio de los arcos que vienen de los bloques antecesores, 4 y 5). El resultado aparece en la Figura 7.3.

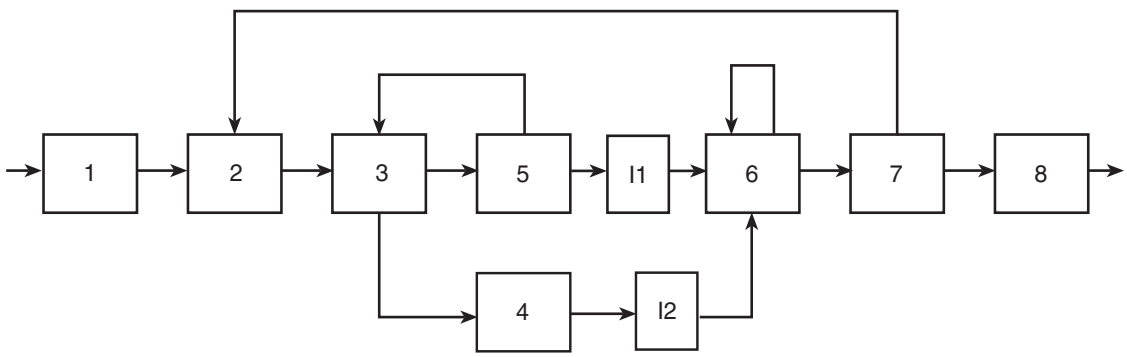


Figura 7.3. Optimización de regiones: primer paso.

A continuación, se colapsan todos los bloques de la región recién tratada, para formar un solo bloque, al que ya no hay que aplicar más optimizaciones. Al mismo tiempo, se calculará el valor de los vectores booleanos para el bloque colapsado. En el ejemplo, la región formada por el bloque (6) se colapsa para formar el bloque R1, y desaparece el bucle del bloque 6 sobre sí mismo (*véase* la Figura 7.4).

Una vez colapsada la región (6) e introducidos los bloques nuevos I1 e I2, la lista de regiones debe modificarse adecuadamente, quedando así: (R1), (3,5), (2,3,5,I1,R1,7), (2,3,4,5,I1,I2,R1,7).

A continuación, se pasa a la segunda región de la lista (3,5). Esta región, que evidentemente constituye otro bucle, tiene un solo arco de entrada, cuyo predecesor es el bloque 2. Se aplican sobre esta región las optimizaciones indicadas en el algoritmo. Como consecuencia de este proceso, aparecerá un bloque nuevo (`I3`), que contendrá las instrucciones extraídas desde el

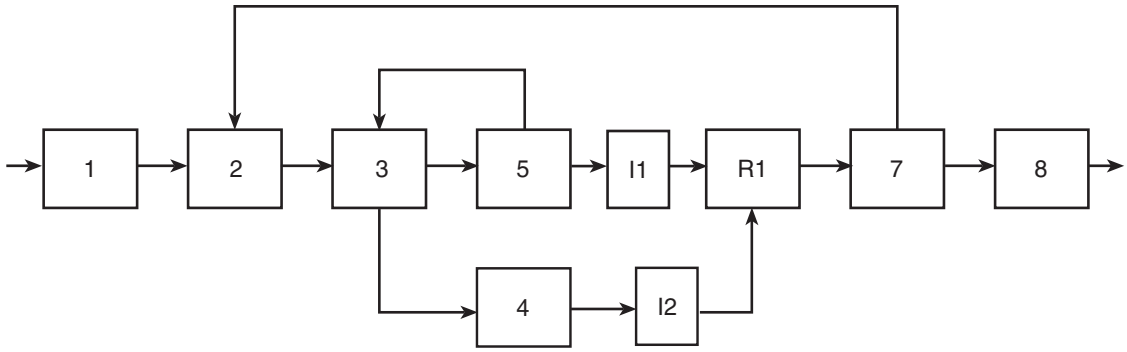


Figura 7.4. Optimización de regiones: segundo paso.

interior del bucle hasta su zona de inicialización. En este caso, habrá que insertar el bloque `I3` en un solo punto, ya que la región (3,5) tiene un solo nodo de entrada. También se colapsará la región, formando un bloque nuevo (R2). La Figura 7.5 muestra el resultado final de esta parte del algoritmo.

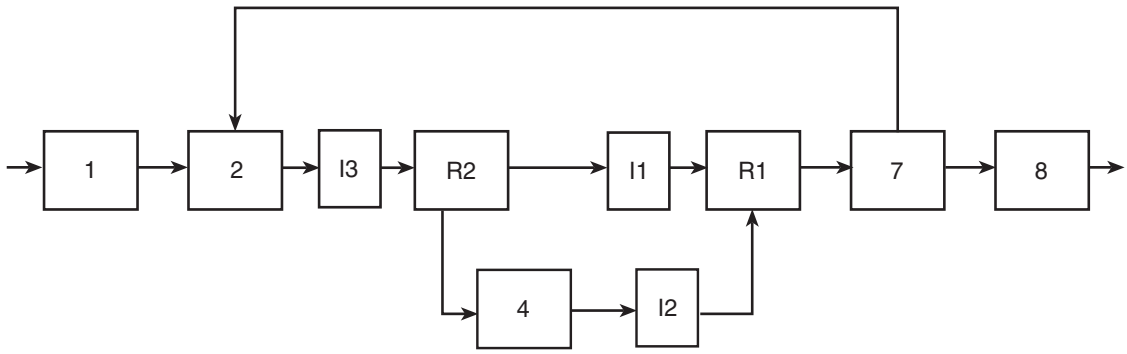


Figura 7.5. Optimización de regiones: tercer paso.

Después de estos cambios, la lista de regiones queda así: (R1), (R2), (2,I3,R2,I1,R1,7), (2,I3,R2,4,I1,I2,R1,7).

Las restantes iteraciones del algoritmo quedan como ejercicio para el lector.

7.9 Identificación y eliminación de las asignaciones muertas

El desarrollo de las aplicaciones, que a medida que se depuran los errores se ven sometidas a modificaciones frecuentes del código fuente, provoca a menudo la aparición de asignaciones con-

secutivas de valores distintos a la misma variable, sin que exista un uso intermedio de la primera asignación, que no aporta nada al programa. También pueden surgir asignaciones inútiles como consecuencia de la reducción de fuerza y de optimizaciones semejantes (como se vio en la Sección 7.7). Esta situación se formaliza en el concepto de *asignación muerta*, que sería conveniente eliminar.

A veces, las asignaciones muertas están camufladas bajo la apariencia de instrucciones útiles. Esto ocurre, por ejemplo, en las asignaciones recursivas a variables que sólo se utilizan en dichas definiciones recursivas. Véase, por ejemplo, la siguiente instrucción de C++:

```
for (int i=0; i<n; i++) {}
```

Obsérvese que, en esta instrucción, la variable `i` es local al bloque, cuyo cuerpo está vacío. Por lo tanto, esta variable recibe el valor inicial cero y se va incrementando hasta alcanzar el valor `n`, después de lo cual queda eliminada, por haberse alcanzado el límite de su rango de definición, sin haberse utilizado para nada. Por esta razón, las dos asignaciones a la variable `i` en esta instrucción se deberían considerar asignaciones muertas. (Recuérdese que la expresión `i++` representa una notación reducida de la instrucción `i=i+1`, y por tanto se trata de una asignación).

Para ver si una asignación está muerta, se puede utilizar el siguiente algoritmo:

1. A partir de una asignación sospechosa a la variable `v`, se sigue adelante con las instrucciones que forman parte del mismo bloque. Si aparece otra asignación a la misma variable sin un uso intermedio, la asignación está muerta. Si aparece un uso de esa variable, la asignación no está muerta. Si no ocurre ni una cosa ni otra, ir al paso 2.

2. Se siguen todas las ramificaciones del programa, a partir del bloque en que se encuentra la asignación sospechosa, y se comprueban los valores de los vectores `R`, `A` y `B` para dicha variable en cada bloque por el que se pase. Si se encuentra que `B[v]=1` en un bloque, la asignación no está muerta. Si `B[i]=0`, pero `A[i]=1`, se abandona ese camino. Si se acaban los caminos o se cierran todos los bucles de los bloques sucesivos, la asignación está muerta.

7.10 Resumen

Dado que la optimización perfecta no es decidible, este capítulo presenta un conjunto de técnicas de optimización que se pueden aplicar juntas o por separado, tanto sobre la representación intermedia en forma de cuádruplas, como sobre el código final generado, para incrementar la eficiencia del mismo en diferentes aspectos: reducción del número de instrucciones, del número de variables auxiliares utilizadas, del tiempo de proceso de las instrucciones, etc. El criterio más importante que se utiliza para agruparlas es su dependencia respecto a una máquina concreta.

7.11 Ejercicios

1. Dado el programa del Ejercicio 14 del Capítulo 6:

```
int a = 2, b = 8, c = 4, d;

for(i=0; i<5; i++){
   a = a * (i* (b/c));
   d = a * (i* (b/c));
}
```

Aplicar sucesivamente las siguientes optimizaciones a sus cuádruplas:

- Ejecución en tiempo de compilación.
- Eliminación de redundancias.
- Reducción de fuerza.

2. Dado el programa del Ejercicio 15 del Capítulo 6:

```
int a;
float b;

a = 4 + 3;

a = 5;

b = a + 0.7;
```

Suponiendo que inicialmente la tabla de símbolos está vacía, especificar su contenido después de aplicar el algoritmo de ejecución en tiempo de compilación a sus cuádruplas.

3. Dado el programa del Ejercicio 5 del Capítulo 6:

```
int a,b,c,d,i;
...
a = b+c;
for (i=0; i<a; i++) d+=(b+c)*i;
```

Aplicar el algoritmo de reducción de redundancias y de reducción de fuerza a sus cuádruplas.

Capítulo 8

Intérpretes

Como se vio en la Sección 1.13.2, un intérprete es un procesador de lenguaje que analiza un programa escrito en un lenguaje de alto nivel y, si es correcto, lo ejecuta directamente en el lenguaje de la máquina en que se está ejecutando el intérprete. Cada vez que se desea ejecutar el programa, es preciso interpretar el programa de nuevo.

En realidad, muchos de los intérpretes existentes son en realidad compiladores-intérpretes, cuyo funcionamiento tiene lugar en dos fases diferentes:

- La *fase de compilación, o de introducción del programa*: el programa de partida se compila y se traduce a un formato o lenguaje intermedio, que no suele coincidir con el lenguaje de ninguna máquina concreta, ni tampoco ser un lenguaje simbólico o de alto nivel, pues se acostumbra a diseñar un formato propio para cada caso. Esta operación se realiza usualmente una sola vez.

  El formato interno puede ser simplemente el resultado del análisis morfológico, o bien puede haberse realizado una parte del análisis sintáctico y semántico, como la traducción a notación sufija o a cuádruplas.

- La *fase de interpretación, o de ejecución del programa*: el programa compilado al formato o lenguaje intermedio se interpreta y se ejecuta. Esta operación se realiza tantas veces como se desee ejecutar el programa.

Las dos fases no tienen por qué ser consecutivas. Es posible que el programa se introduzca y compile mucho antes de ser ejecutado por el intérprete. En general, el compilador y el intérprete están integrados en un solo programa ejecutable. En cambio, en el lenguaje JAVA, las dos partes se han separado por completo y se tiene el compilador de JAVA, que traduce los programas escritos en JAVA a un formato especial llamado *bytecode*, y lo que se llama *intérprete o máquina virtual de JAVA* es, en realidad, un intérprete de *bytecode*.

8.1 Lenguajes interpretativos

Entre los lenguajes que usualmente se interpretan se pueden mencionar LISP, APL, PROLOG, SMALLTALK, Rexx, SNOBOL y JAVA. De algún lenguaje, como BASIC, existen a la vez compiladores e intérpretes.

Algunos de los lenguajes anteriores no pueden compilarse y exigen la utilización de un intérprete. Esto puede ocurrir por diversos motivos:

1. Porque el lenguaje contiene operadores muy difíciles o imposibles de compilar. El ejemplo típico de esto es una instrucción que ejecuta una cadena de caracteres como si se tratase de una instrucción ejecutable. LISP, por ejemplo, posee una instrucción `EVAL-QUOTE` que realiza precisamente esa función. El método `value`, aplicado a objetos de la clase `Block`, realiza la misma función en el lenguaje SMALLTALK. APL y PROLOG también disponen de funciones u operadores semejantes. Esta operación es imposible para un compilador, a menos que el código que genere incorpore un intérprete completo del lenguaje fuente. En cambio, para un intérprete es perfectamente posible, pues el intérprete está siempre presente durante la ejecución del programa, por lo que puede invocársele para ejecutar una cadena de caracteres. De hecho, lo que usualmente se llama *compilador de LISP, de PROLOG o de SMALLTALK* no genera realmente código máquina equivalente al programa fuente, sino un programa ejecutable que empaqueta dicho programa fuente, precompilado, junto con un intérprete del código o lenguaje intermedio.

2. Porque se ha eliminado del lenguaje la declaración de las variables, que pasa a ser implícita. En lenguajes como LISP, APL, PROLOG y SMALLTALK, una variable tiene siempre el último valor que se le asignó y no existe restricción alguna para asignarle a continuación otro valor de un tipo completamente diferente. En los lenguajes que se compilan totalmente, es difícil implementar esto, pues el compilador debe asignar espacio a las variables en el programa objeto (*véase* el Capítulo 10), y para ello debe saber a qué tipo pertenece su valor, ya que el tamaño de una variable depende de su tipo.

3. Porque se ha eliminado del lenguaje fuente la gestión dinámica de la memoria, confiándosela al intérprete y quitando así trabajo al programador. Esto se aplica, por ejemplo, a los lenguajes LISP, APL, PROLOG, SMALLTALK y JAVA.

4. Porque la presencia del intérprete durante la ejecución es necesaria por razones de seguridad o de independencia de la máquina. El ejemplo más claro de esto es el lenguaje JAVA, que fue diseñado para permitir incluir programas en las páginas de la red mundial (*World Wide Web*). Un programa JAVA debe poder ser ejecutado del mismo modo, cualquiera que sea la plataforma (máquina y sistema operativo) desde la que se invoque. No tendría sentido que las páginas de la red mundial sólo pudiesen verse desde Windows o desde Linux, o sólo desde máquinas dotadas de microprocesador INTEL, por ejemplo. La solución consistió en colocar un intérprete de *bytecode*, o máquina virtual de JAVA, en cada navegador, con lo que la dependencia de la plataforma se traslada al navegador, mientras el *bytecode* que se interpreta es idéntico en todos los entornos.

Por otra parte, había otro requisito indispensable para que la red mundial pudiera llegar a imponerse: la seguridad de que, al entrar en una página cualquiera, no se corre el riesgo de quedar infectado por un virus. Este peligro existió desde el momento en que las páginas web pudieron llevar programas ejecutables anejos. La solución, como en el caso anterior, consistió en el uso de un intérprete para ejecutar dichos programas. Los intérpretes tienen una gran ventaja respecto a los programas ejecutables generados por los compiladores: conservan el control durante la ejecución. Cuando un compilador genera un programa objeto ejecutable, éste puede ponerse en marcha incluso en otra máquina distinta, o si se ejecuta en la misma máquina, está fuera del control del compilador, cuyo trabajo terminó con la generación del código ejecutable. En cambio, puesto que los programas interpretados se ejecutan bajo control del intérprete, éste puede diseñarse de tal manera que asegure la seguridad frente a virus, gusanos, troyanos y otras pestes informáticas. Para ello, basta con que el intérprete o máquina virtual asociado al navegador se asegure de que no se pueden realizar ciertas operaciones peligrosas, como escribir en el disco duro local, o incluso a veces leer del mismo (un gusano que se dispersa utilizando el correo electrónico y la agenda de direcciones de la computadora local no precisa escribir en el disco duro, pero sí tiene que leer de él).

8.2 Comparación entre compiladores e intérpretes

El hecho de que los compiladores y los intérpretes coexistan, a veces incluso para el mismo lenguaje, como en el caso del BASIC, indica que ambos tipos de procesadores de lenguaje tienen que presentar alguna ventaja sobre el otro en determinadas condiciones. Esta sección describe algunas de las ventajas y desventajas de los intérpretes respecto a los compiladores.

8.2.1. Ventajas de los intérpretes

- **Flexibilidad:** Los lenguajes interpretativos suelen ser más flexibles y permiten realizar acciones más complejas, a menudo imposibles o muy difíciles de procesar para un compilador. En el apartado anterior se han visto algunas de ellas:
  - — Ejecución de cadenas de caracteres mediante operadores como *execute*, *interpret*, *evalquote* o *value*. Esto permite escribir programas muy potentes, capaces de modificarse a sí mismos.
  - — Cambiar sobre la marcha el significado de los símbolos, e incluso prescindir por completo de las declaraciones. Esto reduce la carga de trabajo del programador, que no tiene que preocuparse de declarar las variables, aunque también puede hacer más difícil la depuración de los programas.
  - — Simplificar la gestión dinámica de memoria en los programas fuente, poniéndola por completo bajo el control del intérprete. Al contrario de la propiedad anterior, esto facilita enormemente la depuración de los programas, pues una parte importante del tiempo de depuración de los programas escritos en lenguajes compilables, como C y C++, se dedica a la detección y corrección de errores en el manejo de la memoria dinámica.

— Obtener un enlace dinámico completo en los sistemas orientados a objetos. En los lenguajes de este tipo, los *mensajes* (equivalentes a las instrucciones en los lenguajes no orientados a objetos) se dirigen a un objeto determinado para indicarle que debe ejecutar cierto *método* (una función). En estos lenguajes existe una propiedad (el *polimorfismo*), que significa que el método concreto que se ejecute depende del objeto que recibe el mensaje: objetos de clases diferentes pueden tener métodos propios con el mismo nombre, que realicen acciones muy distintas. Piénsese, por ejemplo, en la posible existencia simultánea de un método llamado `abrir` en los objetos pertenecientes a las clases `Archivo` y `Ventana`, respectivamente.

Pues bien: en los lenguajes orientados a objetos procesados por un compilador, como C++, para conseguir cierto grado de enlace o ligamiento dinámico entre un mensaje y los métodos que invoca, hay que recurrir a declaraciones de métodos virtuales y otros procedimientos que complican la programación. Esto se debe a que el compilador debe resolver en tiempo de compilación todas las invocaciones de los mensajes, pues el programa ejecutable está escrito en el lenguaje de la máquina o en lenguaje simbólico, y no dispone de una tabla de símbolos que le proporcione información sobre la resolución de las llamadas a métodos polimórficos. De hecho, las declaraciones virtuales se procesan introduciendo en el programa objeto una minitabla que aporta información sobre las funciones polimórficas y traduce las invocaciones correspondientes en llamadas indirectas a funciones.

En cambio, en los lenguajes orientados a objetos procesados por un intérprete, como SMALLTALK y JAVA, este problema no se presenta, ya que el intérprete tiene acceso a la tabla de símbolos en el momento de la ejecución, y es capaz de resolver los mensajes polimórficos sin otra ayuda especial. Esto simplifica los programas (JAVA es un lenguaje más simple que C++, aunque deriva históricamente de él) y facilita su depuración.

- **Facilidad de depuración de programas:** Además de los comentarios anteriores a este respecto, durante la ejecución de un programa por un intérprete, dicha ejecución puede interrumpirse en cualquier momento, para examinar o modificar los valores de las variables, realizar saltos en la ejecución, abandonar la ejecución de una subrutina o alterar el entorno. Durante estas acciones, el hecho de que la tabla de símbolos esté disponible facilita mucho la depuración.

Es cierto que los lenguajes compilables vienen provistos de depuradores potentes, que también permiten observar y manipular el contenido de las variables y realizar acciones semejantes a las de los intérpretes, a costa de sobrecargar el programa objeto con tablas de símbolos y listas de direcciones de las instrucciones. Pero hay algo que ningún depurador de un lenguaje compilable permite hacer, y que resulta fácil con un intérprete: la habilidad de suspender la ejecución del programa en cierto punto, modificar el programa fuente y continuar la ejecución desde el mismo punto al que había llegado sin necesidad de volver al punto de partida. Con un programa compilable, cada vez que el depurador nos permite detectar un error en el programa, hay que interrumpir la ejecución para corregirlo, compilar el programa corregido, generar un nuevo programa ejecutable y empezar de nuevo. Con un intérprete, en cambio, si se detecta un error, éste puede corregirse, y al continuar la ejecución

es posible detectar más errores sin pérdida de tiempo, lo que acelera considerablemente la depuración.

- **Rapidez en el desarrollo:** Como consecuencia de lo anterior, los programadores que utilizan un lenguaje interpretable suelen conseguir mayor eficiencia de programación que los que programan en lenguajes compilables. Por esta razón, a veces se utilizan los primeros durante el desarrollo inicial de una aplicación muy grande, y una vez depurada se traduce a un lenguaje compilable. En principio, la aplicación no debería contener muchos errores nuevos, y la mejora de eficiencia de la primera fase suele compensar más que de sobra la pérdida de eficiencia debida a la traducción. De este modo se aprovecha lo mejor de ambos tipos de traductores, eludiendo las desventajas que se mencionan a continuación.

8.2.2. Desventajas de los intérpretes

- **Velocidad de los programas ejecutables:** A menudo son un orden de magnitud más lentos que programas compilados equivalentes, por lo menos. Esto se debe a que un compilador realiza los análisis morfológico, sintáctico y semántico una sola vez y genera código ejecutable en el lenguaje de la máquina de una vez para siempre. Este código puede después ejecutarse tantas veces como se desee, sin necesidad de volver a realizar análisis alguno.

  En cambio, con un intérprete, cada vez que se ejecuta un programa es preciso realizar con él algún tipo de análisis. Dicho análisis puede no ser tan complejo como el que realiza un compilador, si el intérprete está actuando sobre código parcialmente traducido (como ocurre en JAVA con *bytecode*), pero siempre supone cierta carga de trabajo, por muy pequeña que ésta sea. Lo peor es que dicha carga puede acumularse hasta alcanzar cifras enormes. Piénsese, por ejemplo, en un bucle que ha de ejecutarse 10 000 veces. Con un compilador, el bucle se traducirá una sola vez al código objeto, y es éste el que se ejecuta 10 000 veces. Con el intérprete, sin embargo, la pequeña pérdida debida al análisis que hay que realizar cada vez que se ejecute el bucle se multiplicará por 10 000.

- **Tamaño del programa objeto:** Se ha visto que lo que normalmente se llama *programa compilado* en entornos interpretables, en realidad no es tal, sino que se construye añadiendo una parte del intérprete al programa fuente (que habrá sido traducido a veces a algún formato intermedio). Esto significa que los programas ejecutables independientes escritos en lenguajes interpretativos suelen ser mucho más grandes que los programas compilados. La diferencia disminuye a medida que aumenta el tamaño de la aplicación, pero se nota mucho en programas triviales, del tipo *Hello, World*. En los lenguajes compilables, estos programas suelen dar lugar a ejecutables de 1 kilobyte como máximo, mientras que no es raro que el tamaño mínimo de un programa equivalente generado por un intérprete alcance un centenar de kilobytes, pues incluye dentro de sí al intérprete. Esta desventaja tiene menos importancia hoy día, cuando el bajo precio de las memorias ha dado lugar a que casi nadie se preocupe por la cantidad de memoria requerida por una aplicación.

8.3 Aplicaciones de los intérpretes

Resumiendo las consideraciones anteriores, se puede decir que los intérpretes se usan principalmente:

- Cuando el lenguaje presenta características que exigen un intérprete (LISP, APL, REXX, SMALLTALK, PROLOG).
- Para el desarrollo de prototipos.
- Para la enseñanza. Durante los años setenta, Seymour Pappert [1] diseñó un lenguaje interpretativo llamado LOGO, que en los ochenta fue muy bien acogido por los educadores como herramienta apropiada para enseñar los principios de la programación a los niños pequeños. Una de sus ideas en este contexto, los *gráficos tortuga*, ha encontrado aplicaciones inesperadas en otros campos, como la descripción de curvas fractales.
- Cuando el lenguaje dispone de operadores muy potentes. En tal caso, la pérdida de eficiencia debida al análisis de las instrucciones deja de tener tanta importancia, pues la mayor parte del tiempo los programas están ejecutando código rápido prefabricado, incluido en el intérprete (el código que implementa dichos operadores potentes), en lugar de analizar los programas fuente del programador. En APL, por ejemplo, existen funciones primitivas capaces de invertir o manipular matrices completas sin tener que escribir bucle alguno. SNOBOL es otro lenguaje de este tipo.
- Para obtener independencia de la máquina (*véase* la Sección 8.1, en referencia a JAVA).
- Para aumentar la seguridad (*véase* la Sección 8.1, en referencia a JAVA).

8.4 Estructura de un intérprete

Como se puede observar en la Figura 8.1, la estructura de un intérprete es muy semejante a la de un compilador, pues contiene los mismos analizadores (morfológico, sintáctico y semántico) y una tabla de símbolos o de identificadores. Hay también una sección de gestión de memoria y otra de proceso de errores, aunque éstas funcionan de una manera bastante diferente de las componentes respectivas de un compilador, como se verá en los dos capítulos siguientes.

La diferencia fundamental entre un compilador y un intérprete estriba en la ausencia de una etapa de generación de código (así como de un optimizador de código, como es lógico), que se sustituye por una componente de ejecución de código. Esto se debe a que un intérprete no genera código, sino que ejecuta directamente las instrucciones en cuanto detecta lo que tiene que hacer con ellas.

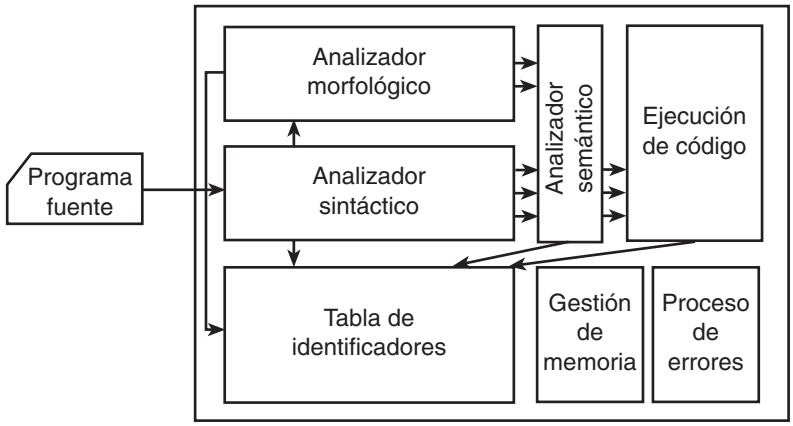


Figura 8.1. Estructura de un intérprete.

8.4.1. Diferencias entre un ejecutor y un generador de código

- La gestión de registros durante la ejecución es innecesaria. Un compilador debe recordar en cada momento qué variable ha colocado en cada registro de la máquina en la que se vaya a ejecutar el programa objeto (que incluso podría ser diferente de la máquina en que se está ejecutando el compilador). En cambio, un intérprete puede prescindir de este trabajo, pues tiene que cargar las variables en sus propios registros para realizar sobre ellas la operación correspondiente. Así, donde un compilador tuviese que generar el código siguiente, que corresponde a la instrucción `C:=A+B`:

  ```
  MOV EAX,A
  ADD EAX,B
  MOV C,EAX
  ```

  un intérprete tendría que ejecutar esas mismas instrucciones, lo que implica que ya estarán escritas en el código del intérprete (si no, no podría ejecutarlas), y la selección del registro a utilizar (en este caso `EAX`) ya habrá sido realizada previamente por el programador que ha construido el intérprete. Es evidente, por tanto, que un intérprete debe tener incluidas en su sección de ejecución todas las combinaciones de código que pudiese generar un compilador que traduzca programas escritos en el mismo lenguaje.

- Las conversiones de tipo pueden adelantarse o aplazarse hasta el último momento. Por ejemplo, si hay que realizar la suma de dos vectores de la misma longitud, uno de ellos entero y el otro en punto flotante, una opción sería convertir todo el vector entero al tipo flotante. Otra posibilidad sería esperar hasta el momento en que se realice la operación: a medida que le llega el turno, cada elemento del vector entero puede convertirse al tipo flotante para sumarlo con el elemento correspondiente del vector cuyos valores son números reales. Ambos procedimientos pueden tener ventajas e inconvenientes: normalmente habrá que buscar un equilibrio entre la ocupación de memoria y el tiempo de ejecución.

- Durante el análisis sintáctico, la información semántica asociada a los operandos de las expresiones puede generarse sobre *plataformas de operandos*, es decir, vectores de estructuras que contienen toda la información asociada al operando izquierdo, el operando derecho y el resultado de la operación. Esta información se pone al día cada vez que el intérprete realiza una operación, lo que permite obtener mejoras de eficiencia significativas. Las plataformas de operandos están relacionadas con la pila semántica, pues la información que contienen puede tener que almacenarse en dicha pila para utilizarla posteriormente, como ocurriría, por ejemplo, cuando el analizador detecte que se ha abierto un paréntesis durante la ejecución de una instrucción.

8.4.2. Distintos tipos de tabla de símbolos en un intérprete

Aunque la gestión de memoria en los intérpretes se verá con detalle en el Capítulo 10, aquí se va a mencionar la parte de dicha gestión que afecta a la estructura de las tablas de símbolos, puesto que éstas constituyen una parte independiente de los procesadores de lenguaje.

- Algunos intérpretes utilizan una tabla de símbolos de tamaño fijo, cuyos elementos contienen punteros a la memoria asignada a las variables. En estos casos, la tabla de símbolos tendrá normalmente estructura de *array*. Esto es especialmente útil cuando las variables pueden tener valores que ocupan mucho espacio, como ocurre con vectores, matrices y otras estructuras de datos. Por otra parte, si el valor de una variable es pequeño (como ocurre cuando se trata de un número entero), a veces puede introducirse dicho valor en la propia tabla de símbolos para mejorar la eficiencia del acceso al valor a través del nombre de la variable.

- Otras veces, los intérpretes disponen de tablas de símbolos cuyo tamaño puede modificarse de forma dinámica, y la tabla de símbolos podría estructurarse como una lista encadenada. En estos casos también pueden utilizarse los tipos de tabla *hash* descritos en el Capítulo 2, con los algoritmos adaptados para tener en cuenta la estructura dinámica de las tablas.

- En algunos intérpretes, la tabla de símbolos no apunta directamente a la memoria asignada a las variables, sino que lo hace a través de una tabla intermedia de referencias, que lleva la cuenta del número de punteros que apuntan en un momento dado al objeto de que se trate. Esto simplifica la recolección de desechos y la gestión de la memoria, a costa de aumentar el tiempo de acceso a los valores de las variables, pues hay que atravesar un direccionamiento indirecto más. En cambio, la memoria ocupada por los programas durante su ejecución puede ser significativamente menor, pues una instrucción de asignación tal como `x:=y` no supondría trasvase alguno de datos (*véase* la Figura 8.2): bastaría apuntar desde el elemento de la tabla de símbolos correspondiente a la variable `x` a la misma referencia a la que apunta la variable `y`, incrementando al mismo tiempo el número de referencias que apuntan a dicho elemento. De este modo, si el valor de la variable `x` cambiase posteriormente, el de la variable `y` no se perderá, pues dicho valor no se declarará basura hasta que el número de referencias que le apuntan se reduzca a cero. Naturalmente, si se modificase parcialmente el valor de la variable `x`, por ejemplo, con la instrucción `x[3]:=5`, sería pre-

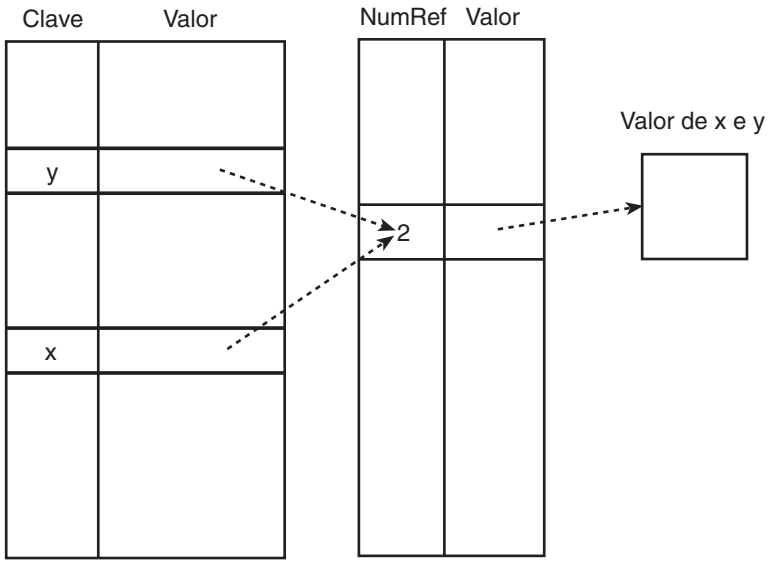


Figura 8.2. Uso de la tabla de referencias para minimizar el uso de memoria en las asignaciones.

ciso obtener una referencia independiente para esta variable, copiar su valor a una zona nueva de memoria, y sólo entonces modificar el valor del elemento especificado (*véase* la Figura 8.3).

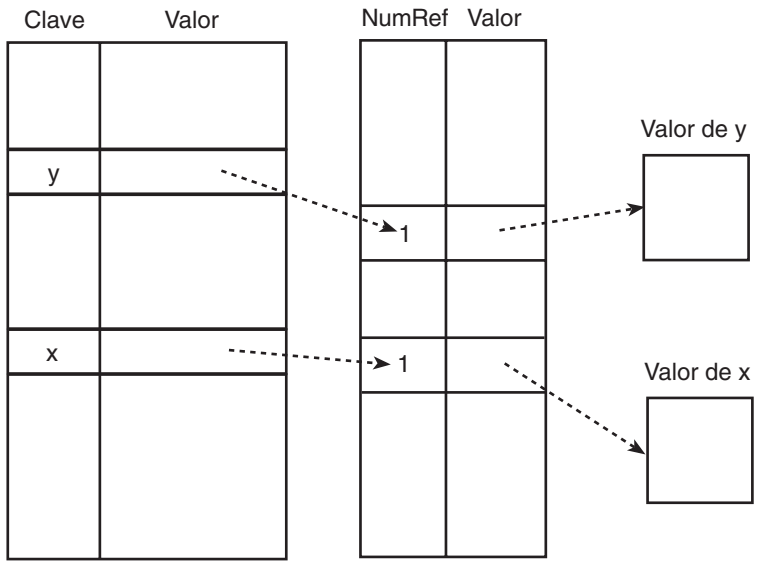


Figura 8.3. La tabla de referencias después de la ejecución de la instrucción `x[3]:=5`.

8.5 Resumen

Este capítulo revisa las principales diferencias entre los compiladores y los intérpretes. Ambos tipos de procesadores de lenguaje tienen una gran parte en común, pero se diferencian esencialmente en la generación de código, que en los intérpretes es sustituida por una fase de ejecución directa del código fuente interpretado.

Se analizan los principales lenguajes que usualmente exigen la utilización de un intérprete y las razones por las que conviene diseñarlos así. Se comparan los compiladores y los intérpretes desde el punto de vista de las ventajas y los inconvenientes del uso de unos y otros. Finalmente, se estudia con más detalle la estructura interna de un intérprete, con especial énfasis en la ejecución de código y en las diferencias que se pueden detectar en el uso de la tabla de símbolos.

8.6 Bibliografía

[1] Papert, S. (1980): *Mindstorms: children, computers and powerful ideas,* The Harvester Press, ISBN: 0855271639.

Capítulo 9

Tratamiento de errores

Para un compilador o un intérprete, la detección de errores es indispensable, pues todos los programas erróneos deben ser rechazados automáticamente. Un compilador dispondría de un procesador de errores perfecto si realiza correctamente las tres acciones siguientes:

- Detectar todos los errores que contiene el programa que se está analizando.
- No generar ningún mensaje que señale errores donde no los hay.
- No generar mensajes de error innecesarios.

9.1 Detección de todos los errores verdaderos

El primer objetivo es casi imposible de conseguir para un compilador, pero no para un intérprete. Esto se debe a que un programa contiene ordinariamente errores de varios tipos:

- Errores morfológicos, como identificadores mal formados, constantes mal construidas, símbolos no pertenecientes al lenguaje, comentarios incorrectos, etc.
- Errores sintácticos, es decir, cadenas de unidades sintácticas que no se adaptan a la sintaxis del lenguaje fuente.
- Errores semánticos, como operaciones realizadas sobre tipos incompatibles. Entre éstos se incluyen los errores relacionados con el uso de la tabla de símbolos, como uso de identificadores no declarados o declaración doble de un identificador en la misma región de alcance.
- Errores detectables únicamente en tiempo de ejecución, como punteros con valor nulo o cuyo valor se sale de los límites permitidos, o indexación de vectores con índices inapropiados.

Los tres primeros tipos de error deberían ser localizados por cualquier compilador. Por esta razón, se llaman *errores de compilación*, en oposición a los del último tipo (los *errores de ejecución*). Sin embargo, no todos los compiladores son capaces de detectar a la primera todos los errores de compilación. Considérese, por ejemplo, el siguiente programa escrito en el lenguaje C:

```
void main () {
     int i,j,k;
     i=0; /* Asigno 0 a i //
     j=;
     =k;
}
```

Este programa contiene tres errores: un comentario mal escrito (no está bien señalado el punto donde termina) y las dos instrucciones siguientes, cuya sintaxis es incorrecta. Pues bien, al no detectar el final del comentario en la primera línea errónea, algunos compiladores supondrán que las tres líneas siguientes forman parte del comentario, no realizarán su análisis sintáctico y no detectarán los errores que contienen. En cambio, además del primer error correcto (`comentario mal terminado`) seguramente se señalará un error inexistente (`la función main no termina correctamente`), que indica que el compilador echa de menos la presencia de la llave final de bloque (`}`), que sin embargo sí está presente.

En un caso como el que nos ocupa, el programador corregirá el primer error señalado (cerrando bien el comentario, con los símbolos `*/`), pero no habrá recibido información que le permita detectar los dos errores sintácticos (piénsese que puede haber decenas de instrucciones entre el primer error y el segundo). En cambio, el segundo mensaje recibido le parecerá absurdo, pues puede ver a simple vista que la llave que señala el final de la función `main` sí está presente. En consecuencia, después de corregir el primer error, volverá a compilar el programa fuente, y sólo entonces el compilador señalará los dos errores pendientes, al mismo tiempo que desaparece el error espurio referente a la llave final. En definitiva, más pronto o más tarde se detectan todos los errores, pero a costa de perder el tiempo, teniendo que realizar varias compilaciones innecesarias.

Un compilador con un procesador de errores más perfecto podría detectar que el comentario mal terminado se debe al cambio del carácter `*` por el carácter `/`, daría por terminado el comentario al final de la línea y continuaría el análisis sintáctico a partir de ahí, detectando las dos instrucciones erróneas y eliminando el mensaje de error espurio.

Por otra parte, los errores en tiempo de ejecución suelen estar siempre fuera del alcance de un compilador, que no tiene control alguno sobre la ejecución de los programas que ha generado. Para un intérprete, en cambio, no existe ningún problema, pues mantiene control total sobre la ejecución del programa objeto, que se lleva a cabo como parte del propio intérprete (*véase* el Capítulo 8). Por ejemplo, en el momento de indexar un vector de datos con una variable, el intérprete puede comprobar si el valor de la variable se encuentra dentro del margen permitido por el tamaño del vector, pues este dato le es accesible a través de la tabla de símbolos.

9.2 Detección incorrecta de errores falsos

El segundo objetivo puede parecer tan obvio que resulte innecesario mencionarlo. Sin embargo, a lo largo de la historia de la Informática, más de un compilador comercial de prestigio no ha cumplido este requisito. Esto ocurre cuando la presencia de un error desequilibra al compilador hasta tal punto que a partir de entonces detecta errores en instrucciones que no los tienen. A veces, el efecto se propaga hasta el final del análisis, con lo que el compilador genera una serie de mensajes de error en cadena, que afectan a todas las instrucciones a partir de la que provocó el desequilibrio. Se ha visto un ejemplo sencillo en la sección anterior, donde la detección de un error provocó la aparición de otro espurio (señalando que la función `main` no terminaba con una llave `}`). Veamos otro ejemplo, aún más espectacular. Considérese el siguiente programa correcto en lenguaje C o C++:

```
void main () {          // Línea 1
     int i, j;          // Línea 2
     i=1;               // Línea 3
     while (i) {        // Línea 4
       int j;           // Línea 5
          …
       }                // Línea 10
     j=2;               // Línea 11
     if (i<j) j++;      // Línea 12
     …
}                       // Línea 65
```

Supóngase que el programador olvidó escribir la llave situada en la línea 4, que abre el bloque `while`. En tal caso, es muy probable que el compilador genere los siguientes mensajes de error:

```
Línea 5: doble definición del identificador j.
Línea 11: instrucción situada fuera de una función.
Línea 12: instrucción situada fuera de una función.
...
Línea 65: instrucción situada fuera de una función.
```

Al faltar la llave inicial del bloque `while`, el compilador interpreta que las instrucciones 5 a 9 pertenecen al bloque principal de la función `main`, por lo que la declaración de la variable `j` en la línea 6 sería incorrecta (ya había sido declarada en dicho bloque en la línea 2). Además, la llave final de bloque de la línea 10 será interpretada como el cierre de la función `main`, con lo que todas las instrucciones sucesivas (de la 11 a la 65) serán marcadas como erróneas, por encontrarse fuera de un bloque. Cualquier error adicional situado en alguna de estas instrucciones no sería detectado. El compilador habrá generado así 56 mensajes de error, de los que sólo el primero da alguna información al programador, después de pensar un poco, permitiéndole detectar la ausen-

cia de la llave en la línea 4. Una vez corregido este error, en una nueva compilación habrán desaparecido los 55 mensajes espurios, detectándose entonces otros posibles errores situados en las líneas posteriores.

Cuando un compilador es capaz de evitar estos errores en cadena, se dice que posee un procesador de errores con buena capacidad de recuperación. Es muy difícil que un compilador se recupere del problema mencionado en el ejemplo. Una forma de conseguirlo sería la siguiente: al detectar que se están generando demasiados errores, el compilador podría intentar introducir alguna llave adicional en algún punto que parezca razonable (quizá haciendo uso de la información proporcionada por el indentado de las instrucciones del programa fuente) hasta conseguir que los errores espurios desaparezcan.

9.3 Generación de mensajes de error innecesarios

Un mensaje de error es innecesario si ya ha sido señalado previamente (en la misma o en otra instrucción) o si no añade información nueva a la proporcionada por mensajes de error anteriores. La razón de que se desee prescindir de los mensajes innecesarios es parecida a la de la eliminación de los mensajes espurios: facilitar la labor del programador, evitando que tenga que localizar los mensajes de error *buenos* en medio de un fárrago inabordable de mensajes, que proporcionan información nula (los mensajes repetidos) o contraproducente (los mensajes espurios). A continuación se presenta un análisis por separado de los dos casos mencionados:

- Evitar que un solo error dé lugar a la aparición de más de un mensaje. Por ejemplo, sea la expresión `a[i1;i2;i3]`, donde `a` no ha sido declarado como *array*. Al abrir el corchete, el compilador señalará el siguiente error:

  ```
  a no es un array y no se le puede indexar
  ```

  Al cerrar el corchete, un compilador con un procesador de errores poco sofisticado podría señalar otro error:

  ```
  El número de índices no coincide con el rango de a.
  ```

  Una vez señalado el primero, el segundo es innecesario. Para solucionar este problema, una vez detectado el primer error, podría sustituirse la referencia a la variable `a` en la instrucción que se está analizando por una referencia a un identificador interno, generado automáticamente por el compilador e introducido en la tabla de símbolos. La rutina de tratamiento de errores podría interceptar, a partir de entonces, todos los mensajes de error que hagan referencia al identificador interno.

- Evitar que un error idéntico repetido dé lugar a varios mensajes. Por ejemplo, considérese el siguiente programa correcto en lenguaje C/C++:

```
void main () {      // Línea 1
   int i;           // Línea 2
   i=1;             // Línea 3
```

```
    while (i) {     // Línea 4
      int j;        // Línea 5
         …
      if (i<j) {    // Línea 10
        …
      }             // Línea 20
      for (j=0; j<n; j++) { // Línea 21
         a=j-i+1;   // Línea 22
         b=2*a+j;   // Línea 23
      }             // Línea 24
    ...
}                   // Línea 65
```

Supóngase que el programador olvida escribir la llave de la línea 10. En tal caso, la llave de la línea 20 cerraría el bloque que se abrió en la línea 4, con lo que se daría por terminado el alcance de la variable `j`. Pero dicha variable se utiliza cinco veces en las líneas 21, 22 y 23. Por lo tanto, un compilador con un procesador de errores poco sofisticado generaría cinco mensajes de error idénticos en dichas líneas:

```
La variable j no ha sido declarada
```

Esta situación puede solucionarse de varias maneras:

— Cuando se detecta que el identificador `j` no ha sido declarado, se puede introducir en la tabla de símbolos un identificador con ese nombre y los atributos que se pueda deducir que tiene, con el objeto de que los siguientes mensajes `identificador no declarado` que podrían generarse no lleguen a ser detectados. Esta solución tiene el peligro de que un error real subsiguiente no llegue a ser detectado, como consecuencia de una asignación errónea de atributos a la variable fantasma.

— Cuando se detectan mensajes de este tipo, se puede aplazar su generación hasta que se esté seguro de que ya no pueden detectarse más casos, generando entonces un solo mensaje como el siguiente:

```
La variable j no ha sido declarada en las líneas 21, 22 y 23
```

9.4 Corrección automática de errores

En un compilador, un procesador de errores óptimo sería capaz no sólo de detectar los errores, sino también de corregirlos, con lo que el programa ejecutable podría generarse ya en la primera pasada de compilación, lo que ahorraría mucho tiempo. Sin embargo, la corrección automática presenta el peligro de introducir errores difíciles de detectar, si las suposiciones realizadas por el compilador para corregir los errores fuesen incorrectas, por lo que los compiladores comerciales no suelen realizar corrección automática de errores, excepto algunos de propósito muy específico, que usualmente generan directamente programas ejecutables (`.EXE`), en los que este

procedimiento puede ahorrar tiempo de desarrollo, pues la mayor parte de los errores de compilación habrán sido correctamente corregidos, y los que no lo hayan sido podrán detectarse, con un depurador conveniente, a la vez que se buscan los errores de ejecución. En todo caso, las decisiones que tome el compilador para corregir los errores detectados deben estar bien documentadas, para que el programador pueda comprobarlas *a posteriori* y no se pierda tratando de depurar un programa ejecutable que no corresponde exactamente a lo que había programado.

Existen varios tipos de correcciones automáticas:

- **Corrección ortográfica.** Corresponde al analizador morfológico y a la tabla de símbolos, aunque el analizador sintáctico y el semántico también pueden desempeñar algún papel. Su funcionamiento es muy similar al de los correctores ortográficos de los procesadores de textos: cuando se detecta un identificador no declarado, el error podría ser simplemente ortográfico. Se trata de considerar los errores más típicos que podrían haberse producido y detectar el identificador correcto, utilizando como término de comparación la lista de palabras reservadas del lenguaje y los identificadores correctamente definidos. Una forma de acelerar el proceso consistiría en comprobar únicamente los errores ortográficos típicos, que son:

  — Cambio de un carácter por otro (usualmente situado en el teclado cerca del carácter correcto).
  — Eliminación de un carácter.
  — Introducción de un carácter espurio.
  — Intercambio de la posición de dos caracteres consecutivos.

  Un identificador declarado, pero no utilizado, puede ser candidato para una corrección ortográfica. Para detectar estos casos, podría ser útil añadir en la tabla de símbolos, entre la información asociada a cada identificador, un contador de usos y referencias. Sin embargo, debe recordarse que este caso también puede darse cuando un identificador utilizado en versiones anteriores del programa ha dejado de usarse, sin que el programador se acordase de eliminar su declaración.

  El analizador sintáctico puede ayudar en este proceso. Por ejemplo, si se espera en este punto una palabra reservada, y lo que aparece es un identificador no declarado, se puede buscar la palabra reservada más próxima al identificador erróneo, teniendo en cuenta los errores ortográficos típicos mencionados.

  El analizador sintáctico puede ser también útil para detectar los errores de concatenación (en realidad un caso particular de la eliminación de un carácter, cuando éste es el espacio en blanco). Por ejemplo, cuando la cadena de entrada contiene `begina` y lo que se espera encontrar es `begin a`.

  En cuanto al analizador semántico, también puede ayudar en estas correcciones. Por ejemplo, si aparece un identificador declarado correctamente en un contexto incompatible con su tipo, el error podría consistir en que se ha utilizado un identificador parecido en lugar del deseado, que sí tiene tipo compatible.

- **Corrección sintáctica.** Cuando se detecta un error al analizar la cadena `xUy`, donde $x, y \in \Sigma^*$ y $U \in \Sigma$ es el próximo símbolo (terminal o no terminal) que se va a analizar, se puede intentar una de las dos acciones siguientes:

— Borrar `U` e intentar de nuevo el análisis. Esta opción tiene en cuenta la posibilidad de que se haya introducido en el programa una unidad sintáctica espuria, que conviene eliminar.

Ejemplo: sea la instrucción

```
else x=0;
```

donde no existe ninguna instrucción `if` previa pendiente. El analizador sintáctico detectaría un error al tratar de analizar la palabra reservada `else`. Una posible acción podría ser eliminar dicha unidad sintáctica y tratar esta instrucción como si se tratase de:

```
x=0;
```

Hay que insistir de nuevo en la necesidad de que todas estas correcciones automáticas del compilador estén muy bien documentadas.

— Insertar una cadena de símbolos terminales `z` entre `x` y `U` y continuar el análisis a partir de `z`.

Ejemplo: sea la instrucción

```
if (i<j) {x=1; y=2; else x=0;
```

Éste parece un caso muy semejante al ejemplo anterior, en el que también se podría intentar eliminar la palabra reservada `else`. Sin embargo, obsérvese que es más probable que el error, en esta instrucción `if-then-else`, consista en la falta de una llave que cierre el bloque de instrucciones correspondiente a la parte `then`. En este caso, el analizador sintáctico, al detectar un error en la unidad sintáctica `else`, haría mejor en tratar de insertar la llave `}` delante de dicha unidad sintáctica y volver a intentar el análisis, que probablemente resultaría correcto.

— Una tercera opción posible (borrar símbolos del final de `x`) no es aconsejable, pues supondría echar marcha atrás en el análisis y desharía la información semántica asociada.

9.5 Recuperación de errores en un intérprete

En el caso de un intérprete, como se mencionó en el Capítulo 8, el tratamiento de errores es un poco diferente al de un compilador.

En primer lugar, una cosa es la ejecución de un programa en condiciones normales, bajo control total del intérprete, y otra el programa ejecutable generado por algunos intérpretes, que empaqueta el propio intérprete junto con un programa fuente, o bien con dicho programa precompilado y traducido a un formato intermedio. En el segundo caso se supone que, antes de generar el programa empaquetado, la aplicación ya habrá sido depurada previamente, bajo control total del intérprete y en la forma que vamos a ver a continuación, por lo que no es probable que dé lugar a nuevos errores, aunque ya se sabe que toda aplicación informática grande contiene errores ocultos.

Sin embargo, y dado que es imposible asegurar que una aplicación informática está totalmente libre de errores (tal demostración es un problema NP-completo), el programa empaquetado de-

bería tener previsto algún tipo de actuación en el caso de que se detecte un error. Normalmente, lo mejor es generar un mensaje de error apropiado y dar por terminada la ejecución del programa, que en todo caso terminaría de una forma más ordenada y elegante que lo que suele ocurrir cuando un programa generado por un compilador topa con un error de ejecución que no ha sido previsto y detectado por el programador. Esto significa que el intérprete que se empaqueta con el programa para generar una aplicación independiente llevará usualmente un procesador de errores más simple y restringido que el que corresponde al intérprete completo.

Si un intérprete detecta un error durante el análisis y ejecución de un programa fuente bajo el control total del intérprete (recuérdese que ambas fases están entrelazadas, pues cada instrucción se analiza y se ejecuta sucesivamente), lo mejor es detener la ejecución, señalar el error detectado con un mensaje apropiado, y permitir al programador que tome alguna acción adecuada, eligiendo entre las siguientes:

- Revisar los valores de las variables.
- Revisar el código que se está ejecutando.
- Modificar el código para corregir el error sobre la marcha. En cuanto sea posible, hacerlo sin afectar a la ejecución interrumpida.
- Reanudar la ejecución, continuando con el programa corregido a partir del punto al que se había llegado. Téngase en cuenta que, si el cambio realizado es muy drástico (como una reorganización total del programa), quizá no sea posible reanudar la ejecución.
- Reanudar la ejecución en un punto diferente del programa que se ha interrumpido, situado en la misma función o procedimiento que estaba activo en el momento de la detección del error. Esto puede significar saltarse algunas líneas, reejecutar otras (echar marcha atrás) o pasar a una zona totalmente diferente de la función o procedimiento.
- Reanudar la ejecución en un punto diferente del programa que se ha interrumpido, pero abandonando la función o procedimiento donde se detectó el error. Normalmente esto significa que el intérprete tendrá que manipular las pilas sintáctica y semántica para asegurar que el proceso se reanuda de una forma ordenada.
- Abandonar totalmente la ejecución para empezar de nuevo, al estilo de lo que se hace durante la depuración de un programa compilado, aunque en este caso no hace falta distinguir entre las dos fases de compilación y depuración separadas, que corresponden al uso de un compilador. Dicho de otro modo, no es preciso alternar entre dos aplicaciones informáticas diferentes (el compilador y el depurador), siendo posible volver a comenzar la ejecución desde el principio, sin que el intérprete tenga que ceder control a una aplicación diferente.

9.6 Resumen

En este capítulo se revisa el tema de la detección y corrección de errores, con especial énfasis en los siguientes puntos:

- Cómo detectar todos los errores de compilación que contiene un programa.

- Cómo evitar que el procesador señale errores que no lo son.
- Cómo evitar que la detección de un error provoque que el compilador genere una cascada de errores espurios.
- Cómo interceptar los mensajes de error innecesarios.
- ¿Debe el compilador corregir un error automáticamente? ¿En qué circunstancias?
- Qué diferencias separan la detección de errores en un compilador y en un intérprete.

Capítulo 10

Gestión de memoria

El módulo de gestión de la memoria es muy diferente en los compiladores y en los intérpretes: en los primeros tiene por objeto gestionar la memoria del programa objeto generado por el compilador, mientras que en los segundos debe gestionar su propia memoria, parte de la cuál va a ser asignada a las variables propias del programa objeto, que se ejecuta bajo control directo del intérprete.

10.1 Gestión de la memoria en un compilador

Al generar el programa objeto, un compilador debe tener en cuenta que éste está formado por dos partes, claramente distintas en las computadoras que siguen el modelo de John von Neumann (actualmente, todos):

- El código ejecutable o instrucciones (que es generado, como indica su nombre, por el generador de código).
- Los datos.

En los primeros microprocesadores de INTEL de los años ochenta (8086 y 8088), la distinción entre el código ejecutable y los datos estaba muy clara: aquellas máquinas utilizaban direccionamiento de 16 bits, que daba acceso únicamente a 64 kBytes de memoria, un tamaño demasiado pequeño para muchas aplicaciones informáticas. Por ello, dichos procesadores utilizaban un procedimiento especial para acceder a memorias más grandes, para lo cuál su unidad aritmético-lógica venía provista de ciertos registros especiales (registros de segmentación) que permitían acceder a una memoria de 12 bits, es decir, un máximo de 1 MByte.

Los registros de segmentación son cuatro: CS, DS, ES y SS. El primero (*Code Segment*) permite acceder a una zona reservada para el código ejecutable; los dos siguientes (*Data Segment* y *Extended data Segment*), a dos zonas reservadas para datos; el cuarto (*Stack Segment*) a la pila de almacenamiento temporal, utilizada, por ejemplo, en las llamadas a subrutinas. Una dirección

concreta de cualquiera de las cuatro zonas se obtiene multiplicando por 16 el valor del registro de segmentación correspondiente y sumando un desplazamiento de 16 bits (que en el caso del código ejecutable está guardado en el registro IP). Así se consigue acceder a $16 \times 65536 = 1$ MByte de memoria.

Los compiladores que generaban código para máquinas INTEL de este tipo tenían que tener en cuenta la existencia de los registros de segmentación y permitían al usuario elegir entre varios modelos de gestión de memoria completamente diferentes:

- **Modelo *tiny* (diminuto).** En este modelo CS=DS=ES=SS, constante durante la ejecución del programa objeto. Es decir, tanto el código, como los datos, como la pila, comparten una sola zona de memoria de 64 kBytes.
- **Modelo *small* (pequeño).** En este modelo DS=ES=SS y distinto de CS, ambos constantes durante la ejecución del programa objeto. Es decir, los datos y la pila comparten una zona de 64 kBytes, pero el código ejecutable se almacena en otra. El tamaño máximo de un programa más los datos que utiliza es, por tanto, igual a 128 kBytes.
- **Modelo *medium* (intermedio).** En este modelo CS es constante durante la ejecución del programa, pero DS, ES y SS pueden variar a lo largo de la misma. Es decir, el código ejecutable cabe en 64 kBytes, pero los datos pueden distribuirse entre varios segmentos, sin rebasar el máximo teórico total de 1 MByte. El generador de código debe tener esto en cuenta para asignar a los registros de segmentación de datos los valores apropiados a lo largo de la ejecución.
- **Modelo *compact* (compacto).** En este modelo DS=ES=SS, constantes durante la ejecución del programa, pero CS puede variar a lo largo de ella. Es decir, los datos caben en 64 kBytes, pero el código ejecutable puede distribuirse entre varios segmentos, sin rebasar el máximo teórico total de 1 MByte. El generador de código debe utilizar instrucciones con código de operación diferente para llamar a una subrutina y volver al punto en que se la llamó, ya que la dirección de retorno debe recuperar no sólo el valor del registro de desplazamiento IP, sino también el registro de segmentación CS.
- **Modelo *large* (grande).** En este modelo CS, DS, ES y SS pueden variar independientemente a lo largo de la ejecución del programa objeto. Es decir, la memoria total máxima de 1 MByte puede distribuirse arbitrariamente en zonas de datos y zonas de código ejecutable. Este modelo viene a ser una combinación de los dos anteriores.
- **Modelo *huge* (gigantesco).** En todos los modelos anteriores, una sola variable no puede rebasar el tamaño máximo de 64 kBytes. En el modelo *huge* este límite puede rebasarse. Al generar código con este modelo para indexar esas variables, el compilador debe modificar adecuadamente el valor del registro de segmentación durante la propia operación de indexación.

En los modelos *medium*, *large* y *huge*, la memoria de datos suele dividirse en dos zonas diferentes, llamadas *near heap* (montón próximo, con un tamaño de 64 kBytes) y *far heap* (montón lejano, con el resto de la memoria disponible). Inicialmente, los compiladores hacen que los registros de segmentación de los datos y la pila del programa objeto apunten al *near heap*, donde se colocan la pila de llamadas de subrutina (normalmente al final de la región) y las variables

estáticas del programa. El *far heap* se reserva para la memoria dinámica (véase más adelante). La mayor parte del tiempo, los registros de segmentación de datos permanecen invariables. El compilador sólo tiene que cambiar su contenido cuando es preciso apuntar a la memoria dinámica, y recupera el valor usual en cuanto deja de ser necesario hacerlo. Para guardar y recuperar dicho valor, se utiliza la pila de llamadas a subrutina, mediante las instrucciones `PUSH` y `POP`.

Uno de los errores de ejecución típicos en los programas que usaban este modelo de memoria se produce cuando la memoria asignada a la pila de llamadas de subrutina (que, a medida que se requiere memoria, se extiende desde las direcciones más altas del *near heap* hacia abajo) se superpone con la memoria asignada a las variables estáticas, destruyendo los valores de éstas. Esta situación se llama rebosamiento de pila (*stack overflow*, en inglés) y, aunque los compiladores podían generar código que comprobara la presencia de esta situación, era posible inhabilitarlo mediante una opción de compilación que casi todos los programadores utilizaban para ahorrar memoria y tiempo de ejecución en los programas objetos generados.

A mediados de los años ochenta, INTEL desarrolló un nuevo procesador, 80286, que, aunque mantenía el funcionamiento anterior por compatibilidad con los programas desarrollados para máquinas más antiguas, poseía un nuevo modo de funcionamiento que aumentaba la memoria disponible a un direccionamiento de 24 bits, es decir, 16 MBytes.

A finales de los ochenta apareció un tercer procesador de INTEL, 80386, que mantenía la compatibilidad con los dos modos anteriores, pero añadía un tercer modo de funcionamiento con direcciones de 32 bits, lo que permitía alcanzar memorias de 4 Gbytes. Éste es el modo más utilizado hoy, pues ha seguido siendo aplicable a todos los procesadores posteriores de INTEL: 80486, Pentium, Pentium II, Pentium III y Pentium IV.

A continuación se va a estudiar con más detalle el tratamiento que debe dar un compilador a los distintos tipos de variables de datos que suele haber en un programa objeto.

- **Variables estáticas:** En algunos lenguajes (como FORTRAN) todas las variables pertenecen a este grupo. En otros lenguajes (como C o C++) sólo son estáticas las variables declaradas como tales (con la palabra reservada `static`) o como externas (con la palabra reservada `extern`), además de las constantes que precisen que se les asigne memoria (como las cadenas de caracteres y algunas de las constantes que aparecen como parámetro en las llamadas a subrutinas). Además, las variables que se han declarado fuera del alcance de una función son estáticas, por omisión.

  La gestión de las variables estáticas por el compilador es relativamente fácil. Como se ha indicado más arriba, suelen colocarse en el *near heap* o zona de memoria equivalente, situada dentro del campo de acción más usual de los registros de segmentación de datos (DS y ES). Usualmente, la dirección de memoria asignada a estas variables en el programa objeto forma parte de la información asociada a la variable en la tabla de símbolos, aunque, si el compilador genera código simbólico o ensamblador, dicha información puede omitirse, sustituyéndola por etiquetas, que el ensamblador se encargará de convertir en las direcciones adecuadas.

  La dirección asignada a una variable puede ser absoluta o relativa. Las direcciones relativas se expresan mediante un *offset* (desplazamiento) respecto a una dirección base, que sue-

le coincidir con el principio de la zona donde se almacenan todas las variables estáticas. Dado que muchos programas deben ejecutarse aunque se instalen en direcciones de memoria diferentes (se dice entonces que son *reubicables*), en general es mejor trabajar con direcciones relativas que con direcciones absolutas. Cuando se utilizan registros de segmentación de datos, todas las direcciones son, por definición, relativas al contenido del registro de segmentación.

Otra información útil asociada a las variables en la tabla de símbolos es su longitud, que es función del tipo de la variable y del número de elementos que contiene si se trata de un *array* (vector, matriz, etc.), o de su composición, si se trata de una estructura (`struct` en C, `class` o `struct` en C++, `record` en Pascal, etc.). La Tabla 10.1 muestra el tamaño usual de los distintos tipos atómicos que pueden tener las variables en el lenguaje C.

Tabla 10.1. Tamaños de algunos tipos atómicos de datos.

Tipo de dato	Tamaño de cada elemento
Boolean	1 bit
char	1 Byte
short	2 Bytes
long	4 Bytes
int	2 o 4 Bytes
float	4 Bytes
double	8 Bytes

A veces, un compilador puede compactar las variables estáticas para optimizar el uso de la memoria. Sean, por ejemplo, las siguientes variables estáticas en el lenguaje C:

```
const int meses[] = {31, 28, 31, 30, 31, 30,
                     31, 31, 30, 31, 30, 31};
const int base = 31;
```

El compilador podría asignarle a la variable `base` la misma dirección que a la variable `meses[0]`, pues ambas tienen el mismo valor. Lo mismo podría hacerse con cadenas de caracteres: supongamos que a lo largo de un programa se utilizan las cadenas de caracteres constantes `"abcd"` y `"d"`; para ahorrar memoria, el compilador podría asignar a la segunda la dirección del cuarto elemento de la primera, como indica la Figura 10.1.

Obsérvese que esto sólo se puede hacer cuando la segunda cadena de caracteres es subcadena final de la primera, pues el símbolo que indica el final de la cadena en el lenguaje C (`'\0'`) debe estar incluido en ambas cadenas. También hay que tener mucho cuidado de que este tipo de optimizaciones se haga únicamente cuando las dos variables que se super-

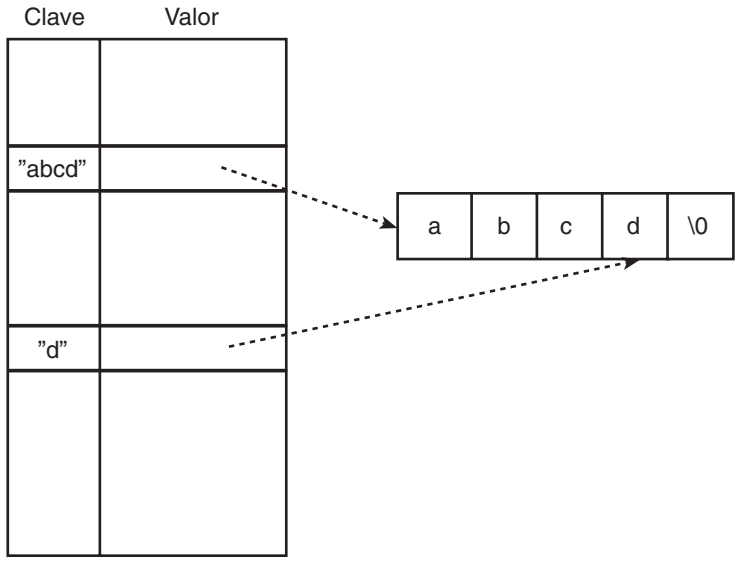


Figura 10.1. Asignación optimizada de memoria a dos cadenas de caracteres.

ponen sean constantes (es decir, que su valor no pueda cambiar durante la ejecución del programa objeto), ya que, en caso contrario, al cambiar una de ellas cambiaría la otra, lo que no sería correcto.

- **Variables automáticas:** Son las variables locales de las subrutinas o de los bloques de datos, así como los parámetros que se pasan a las subrutinas (a menos que sean muy grandes, en cuyo caso se les puede asignar memoria estática, pasando su dirección como variable automática). Usualmente, a estas variables se les asigna espacio en la pila de llamadas de subrutinas.

  El conjunto de todas las variables automáticas asociadas a una subrutina se llama *activation record* (bloque o registro de activación) y a veces se maneja en bloque, reservando la memoria asociada a todas las variables en una sola operación, cuando se invoca la subrutina, y liberándola asimismo toda a la vez al terminar su ejecución. Véase como ejemplo el código que podría generarse en la invocación de una subrutina, tal como la expresión `rutina(a, b)`, donde `a` y `b` son variables enteras:

```
PUSH b
PUSH a
CALL rutina
ADD SP,4
```

  Las dos primeras instrucciones introducen en la pila los dos argumentos de la subrutina (en orden inverso, como exigen las reglas del lenguaje C). A continuación se produce la llamada a la subrutina. La última instrucción libera de un golpe el espacio ocupado en la pila por los dos argumentos, sin necesidad de ejecutar dos instrucciones `POP`.

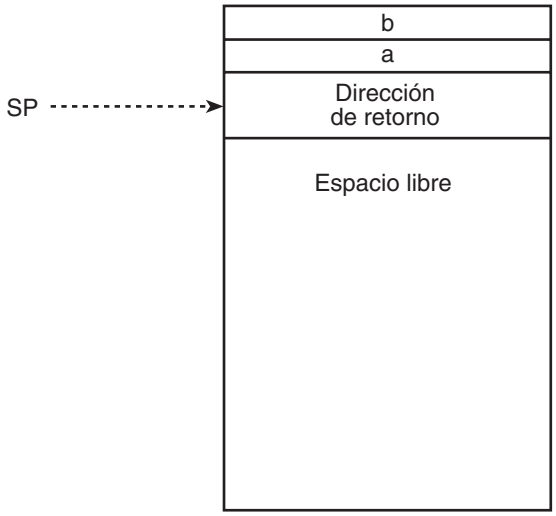


Figura 10.2. La pila de llamadas a subrutinas después de llamar a `rutina(a,b)`.

Inmediatamente después de la ejecución de la instrucción `CALL rutina` por el programa objeto, el aspecto de la pila sería semejante al indicado por la Figura 10.2, donde se ha supuesto que la instrucción `CALL` guarda en la pila la dirección de retorno (la dirección de la instrucción `ADD SP,4`) y que ésta ocupa 4 Bytes (cada espacio pequeño señalado en la figura ocupa 2 Bytes).

Véase ahora el código generado para la definición de la subrutina, suponiendo que el código fuente correspondiente fuese el siguiente:

```
int rutina (int a, char *b)
{
    int i, j, k;
    double r;
    ...
    return k;
}
```

El código generado por el compilador para esta subrutina podría ser:

```
rutina:
    PUSH BP
    MOV BP,SP
    SUB SP,14
    ...
    MOV SP,BP
    POP BP
    RET
```

El registro BP desempeña el papel de registro apuntador del bloque de activación de la subrutina. Por eso, la segunda instrucción generada (`MOV BP,SP`) le asigna el valor actual del

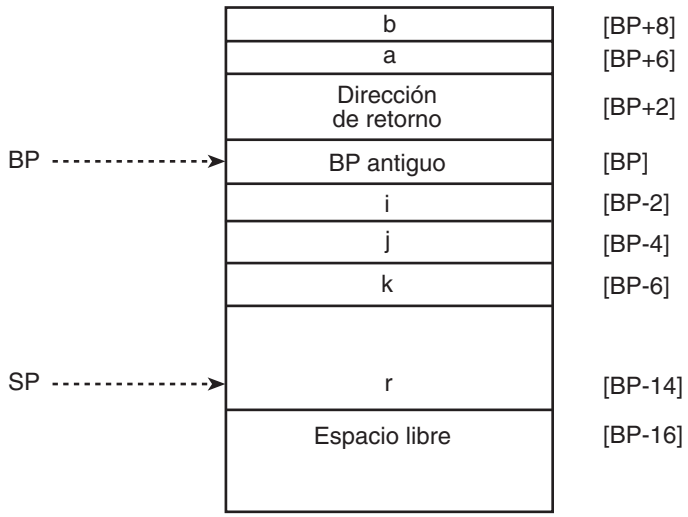


Figura 10.3. La pila de llamadas a subrutinas después de ejecutar el principio de `rutina(a,b)`.

registro SP, que apunta a la cabecera de la pila, tal como está en ese momento. Antes de hacerlo, sin embargo, el valor antiguo de BP se guarda en la propia pila, para poder recuperarlo al final de esta subrutina, ya que dicho valor antiguo de BP apuntaba al bloque de activación de la subrutina que ha llamado a ésta.

A continuación, la tercera instrucción generada (`SUB SP,14`) reserva memoria en la pila para el resto del bloque de activación de la subrutina, que está formado por las variables locales declaradas dentro de ésta (`i`, `j`, `k` y `r`), cuyo tamaño total es de 14 Bytes.

Al llegar la ejecución del programa objeto a los puntos suspensivos, el estado de la pila será el que indica la Figura 10.3.

Obsérvese que el registro BP (apuntador del bloque de activación de la subrutina) apunta a la dirección en que se ha guardado en la pila el valor anterior del propio registro BP. Por encima de esa dirección (recuérdese que la pila se expande hacia abajo) están la dirección de retorno a la subrutina que invocó a ésta, y los valores de los argumentos que se le han pasado, situados en las posiciones BP+6 (`a`) y BP+8 (`b`). En cuanto al resto del bloque de activación (las variables locales), el espacio reservado para ellas se encuentra situado justo por debajo del valor actual de BP. El compilador ha asignado a dichas variables las direcciones indicadas en la Tabla 10.2.

Al final de la ejecución de la subrutina, en el momento de realizar el retorno al punto desde el que se la invocó, es preciso deshacer todo este trabajo inicial, para devolver la pila a las condiciones en que se encontraba antes de la invocación de la subrutina, con objeto de que el programa anterior pueda seguir funcionando correctamente (en particular, el registro BP debe volver a apuntar al bloque de activación de dicho programa). Para ello, basta con que el compilador genere las tres instrucciones siguientes:

Tabla 10.2. Direcciones asignadas a las variables en el bloque de activación del programa `rutina`.

Variable	Dirección asignada
`b`	`[BP+8]`
`a`	`[BP+6]`
`dirección de retorno`	`[BP+2]`
`BP antiguo`	`[BP]`
`i`	`[BP-2]`
`j`	`[BP-4]`
`k`	`[BP-6]`
`r`	`[BP-14]`

— `MOV SP,BP`, que deshace la reserva de espacio para las variables locales de la subrutina. Obsérvese que se habría conseguido lo mismo ejecutando la instrucción `ADD SP,14`, pero ésta es menos eficiente, por incluir una operación de suma, mientras que la instrucción elegida sólo tiene que copiar el contenido de un registro en otro.

— `POP BP`, que devuelve al registro BP el valor que tenía en la rutina que invocó a la subrutina actual (con lo que pasa a apuntar al bloque de activación de aquélla). Después de ejecutar esta instrucción, el registro SP pasa a apuntar a la dirección de retorno.

— `RET`, que pasa el control a la dirección apuntada por SP (es decir, devuelve la ejecución a la instrucción siguiente a `CALL rutina`) en el código generado para el programa que invocó a la subrutina. En ese punto, la instrucción siguiente que se debe ejecutar es `ADD SP,4`, que libera el espacio ocupado por los argumentos de la subrutina, devolviendo la pila (y el registro apuntador SP) al estado en que se encontraba antes de la invocación de la subrutina.

La Figura 10.4 resume la ejecución de la llamada a la subrutina y de la ejecución de ésta.

Las variables automáticas que aparecen definidas en bloques internos se tratan exactamente igual, pero excluyendo la llamada de subrutina, el tratamiento del registro apuntador BP y el retorno. Véase un ejemplo en la instrucción siguiente:

```
for (i=0; i<n; i++) {
   int j, k;
   . . .
   }
```

El código generado por el compilador para este bloque será:

```
SUB SP,4
...
ADD SP,4
```

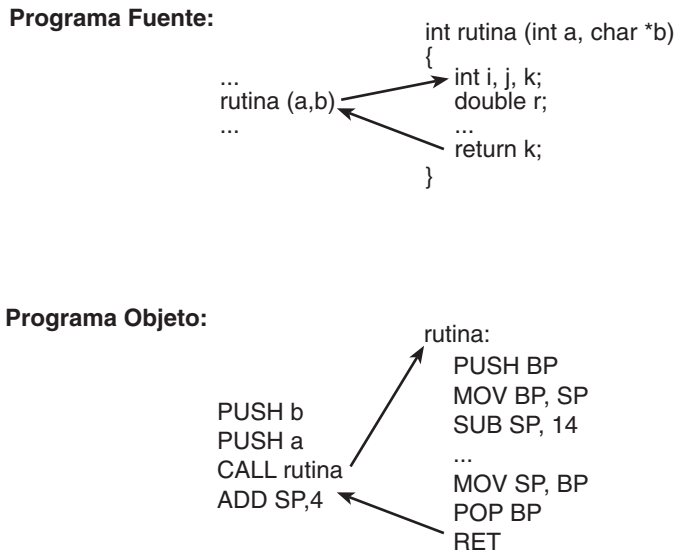


Figura 10.4. Resumen del código generado para la ejecución de una llamada de subrutina.

La primera instrucción reserva espacio para las variables locales del bloque, `j` y `k`, al final del bloque de activación de la subrutina que contiene a esta instrucción. Si se tratase de la subrutina del ejemplo anterior, esto significa que a las dos variables locales al bloque se les asignarían las direcciones `[BP-16]` y `[BP-18]`, respectivamente. Obsérvese que, en este caso, el compilador no puede utilizar la instrucción `MOV SP,BP`, en lugar de `ADD SP,4`, pues en tal caso liberaría todo el bloque de activación de la subrutina, en lugar del espacio local a este bloque.

- **Variables dinámicas:** El propio programa objeto pide espacio para ellas durante su ejecución. Usualmente esto se hace mediante una llamada a alguna función u operador adecuados (como `malloc` en el lenguaje C, o `new` en C++). El código objeto suele venir prefabricado, dentro de alguna biblioteca (*library*) de subrutinas asociada al compilador, que se proporciona con éste para que el programador o el compilador puedan utilizarlas, cuyas llamadas serán resueltas más tarde por el programa enlazador (*linker*). La única responsabilidad del compilador, en este caso, es generar la llamada a dicha subrutina de biblioteca.

  Es importante que las subrutinas asociadas a la gestión de la memoria dinámica estén bien diseñadas. A este respecto, se puede mencionar un caso concreto. En un compilador para el lenguaje C comercializado por Microsoft a principios de los años noventa, la subrutina predefinida `malloc` utilizaba el siguiente criterio para la reasignación de la memoria dinámica. Buscaba espacio en primer lugar en el bloque inicial (que al principio abarcaba toda la memoria dinámica), y sólo si no encontraba espacio en él buscaba en los bloques reutilizables (liberados anteriormente por el programa objeto). Esto tenía la consecuencia de que la memoria dinámica se fragmentaba progresivamente, aunque también se obtenía a cambio alguna mejora en el tiempo de ejecución, pues era más probable (al principio, al menos) en-

contrar espacio en dicho bloque, cuyo tamaño solía ser el más grande de todos, hasta que su fragmentación progresiva lo iba reduciendo.

El problema se presentaba en aplicaciones de funcionamiento permanente (como, por ejemplo, programas de gestión del negocio en una tienda) que están continuamente requiriendo espacio dinámico y liberándolo y que deben funcionar 24 horas al día y siete días por semana. Al cabo de algún tiempo, el programa fallaba porque la memoria se había fragmentado hasta tal punto que la siguiente petición de espacio ya no encontraba un bloque de memoria del tamaño suficiente. Para evitarlo, hubo que sustituir la versión prefabricada de `malloc` contenida en la biblioteca asociada al compilador por una versión propia, que invertía el criterio de asignación de espacio, reutilizando los bloques liberados antes de intentar buscarlo en el bloque inicial. Con este cambio se resolvió totalmente el problema, y la aplicación pudo funcionar indefinidamente sin que se le acabase la memoria.

Aunque las variables dinámicas presentan menos dificultades para un compilador, en realidad la carga de trabajo asociada a su utilización correcta se traslada al programador. En particular, es importante que el programador tenga cuidado de liberar de forma ordenada toda la memoria dinámica que ha ido reservando a lo largo del programa. Si no lo hace, pueden producirse errores de ejecución muy difíciles de detectar, pues no suelen hacer efecto hasta mucho después de haberse producido, lo que dificulta su depuración.

Ante el tratamiento de la liberación de la memoria dinámica, existen varias posibilidades:

— No liberar nada. La memoria dinámica que va reservando el programa no puede reutilizarse, aunque el programa haya dejado de necesitarla. Cuando se acabe totalmente la memoria disponible, el programa fallará. Antes de que esto ocurra, si el sistema operativo utiliza el disco duro como extensión de la memoria principal (paginación o *swap*, en inglés), la eficiencia del programa puede deteriorarse.

— Liberación explícita, que se realiza mediante llamadas a subrutinas u operadores especiales (`free` en el lenguaje C, `delete` en C++, `dispose` en PASCAL). En este caso hay que tener un cuidado especial, pues un mal uso de la liberación puede dar lugar a errores catastróficos, como los siguientes:

  ➤ Liberación de una memoria que no había sido asignada.

  ➤ Liberación incorrecta de una variable, antes de que el programa objeto haya terminado de utilizarla. Si dicha memoria es reasignada a otra variable por la subrutina u operador de asignación de memoria dinámica, el programa no funcionará bien, pues una de sus variables habrá recibido un valor incorrecto, que podría ser incluso incompatible con su tipo. Este error se detectará cuando se vuelva a usar dicha variable, lo que puede ocurrir mucho después de la liberación incorrecta, a veces después de transcurrir horas de ejecución del programa.

  ➤ Doble liberación de la misma memoria dinámica asignada a una variable: se reduce al caso anterior.

— Liberación implícita. En los compiladores, sólo suele aplicarse a las variables automáticas, como se explicó anteriormente. En los intérpretes desempeña un papel muy importante, como se verá en el apartado siguiente.

10.2 Gestión de la memoria en un intérprete

En los intérpretes, la gestión de la memoria es muy distinta a la de un compilador. En primer lugar, toda la memoria asignada al programa objeto suele tratarse como memoria dinámica (excepto las variables que ocupan poco espacio, cuyo valor puede incluirse en la propia tabla de símbolos), recabándose espacio para cada variable cuando comienza a ser necesaria, y liberándolo automáticamente en cuanto deja de serlo. En este caso, la gestión de la memoria dinámica la lleva el propio intérprete, lo que quita mucho trabajo al programador y permite simplificar considerablemente el lenguaje fuente, que gracias a ello puede prescindir de declaraciones de variables y de todos los problemas asociados a la gestión y liberación de la memoria dinámica, que se acaban de comentar en el apartado anterior.

La liberación automática de memoria dinámica por el intérprete puede provocar la fragmentación de dicha memoria, por lo que más pronto o más tarde es preciso realizar su compactación, para poder reutilizarla correctamente. La gestión de la memoria dinámica se denomina **recolección de basuras o de desechos** (*garbage collection*, en inglés).

Recordando lo que se dijo en la Sección 8.4, al hablar de la estructura de la tabla de símbolos en un intérprete, existen cuatro métodos principales para liberar la memoria asociada a una variable:

- **Liberación explícita o definición estática de qué es basura:** El programador decide cuándo debe liberarse una variable, con un método semejante al que se usa en un compilador.
- **Sin declaración explícita de basura:** La liberación de espacio innecesario no se produce cuando la memoria asignada a una variable deja de ser necesaria, sino que se retarda hasta el momento en que la memoria dinámica se agota y es indispensable realizar una recolección de desechos. Este método parece más rápido a primera vista, pero esa ventaja es engañosa, pues, cuando se produce el agotamiento de la memoria, la recolección de desechos suele ser difícil y lleva mucho tiempo, con lo que interrumpe durante períodos considerables la ejecución de los programas. En algunos intérpretes que se comercializaron durante los años ochenta, a menudo ocurría que aparecía un mensaje en la pantalla advirtiendo que la recolección de basuras estaba en proceso, y a veces se tardaba media hora en reanudar la ejecución del programa objeto. En ciertos entornos (como en el control de instrumentos de laboratorio o en sistemas en tiempo real) este funcionamiento es inadmisible, por lo que este método de asignación dinámica de memoria ha dejado prácticamente de utilizarse, excepto en casos muy concretos y especializados.
- **Referencia única:** Cuando a un puntero se le asigna un nuevo valor, la zona de memoria a la que apuntaba previamente se declara basura. Esto tiene la restricción de que todo espacio reservado en la memoria dinámica sólo puede estar apuntado por un único puntero: asignar un puntero a otro estaría prohibido, pues en caso contrario, al declarar basura el primero, el segundo quedaría apuntando a una región inutilizada, lo que podría dar lugar a errores peligrosos. Este procedimiento, que se emplea cuando la tabla de símbolos del intérprete no utiliza una tabla auxiliar de referencias, obliga a duplicar la asignación de la memoria cuando se desea disponer de dos punteros que apunten a los mismos valores, como

ocurre, por ejemplo, cuando se asigna el valor de una variable a otra, o cuando se introduce una variable como argumento de una subrutina, con la modalidad de paso por valor. Aunque este procedimiento ocupa mucha memoria, es algo más rápido que el que vamos a ver a continuación, por lo que, en la práctica, se utilizan ambos en diversos intérpretes.

- **Referencia múltiple:** Cuando los punteros de la tabla de símbolos no apuntan directamente a la memoria dinámica asignada a las variables, sino a través de una tabla de referencias, como se explicó en la Sección 8.4. En este caso, pueden existir varios punteros apuntando a la misma zona de datos dinámicos. La tabla de referencias contiene información sobre el número de punteros que apuntan al mismo espacio, que no se declarará basura hasta que dicho número se reduzca a cero, lo que indica que el espacio ha dejado de ser necesario.

10.2.1. Algoritmos de recolección automática de basura

Existen diversos algoritmos para la recolección de desechos, que se utilizan en diversos intérpretes. A veces pueden combinarse con éxito varios de ellos en un solo intérprete. Mencionaremos los siguientes:

- **Asignación de memoria por un extremo del bloque libre inicial y recolección global de basuras cuando no queda espacio en dicho bloque:** Este procedimiento tiene la ventaja de que los bloques más usados acabarán ocupando espacio en direcciones bajas de memoria y ya no se moverán más. En cambio, lo que va quedando del bloque libre inicial a lo largo de la ejecución se encuentra siempre en las direcciones más altas, y su localización será factible con un solo puntero. La desventaja principal de este método es que, cuando un bloque antiguo queda libre, para reutilizar ese espacio hay que mover casi todo lo demás. Su problema principal es que el tiempo necesario para llevar a cabo la ejecución de un programa concreto no es predecible con exactitud, pues en la mitad de dicha ejecución puede interponerse una recolección de basuras, de duración impredecible. En una de estas operaciones, para un espacio total de 20 MBytes, puede ser necesario cambiar de sitio de 2 a 4 MBytes. Si el sistema está paginado (*swap*), la cosa puede ser peor si el sistema operativo disminuye automáticamente la prioridad de las zonas de memoria afectadas.

  Cada vez que se mueve la posición de un bloque, es preciso modificar al mismo tiempo todos los apuntadores que contenían su dirección. Con los sistemas de referencia única y de tabla de referencias vistos anteriormente, esto significa cambiar el valor de un solo puntero. A veces se incluye en cada bloque un puntero inverso que apunta a su propio apuntador, lo que reduce el tiempo necesario para la recolección de basuras.

  Para reducir el número de veces que se realiza un desplazamiento completo, retrasando la aparición de la necesidad de hacerlo, lo que ocurre cuando una petición de memoria dinámica no puede satisfacerse, porque no existe ningún bloque del tamaño adecuado, suelen utilizarse diversos mecanismos paliativos, como los siguientes:

  — **Fusión automática del último bloque:** Si el bloque liberado es el último, y a continuación comienza lo que queda de la zona libre inicial, en lugar de someter dicho blo-

que al sistema general de recolección de basuras, bastaría fusionarlo con la zona libre inicial, modificando el valor del puntero que apunta a ésta.

— **Fusión de bloques libres:** Siempre que dos bloques contiguos quedan libres, se fusionan para formar un solo bloque más grande.

— **Desplazamiento de bloques ocupados para permitir la fusión de bloques libres no contiguos:** Es un paso adicional que facilita la realización del paso anterior y retrasa aún más la necesidad de la recolección de basuras global.

- **Asignación de memoria alternativa por los dos extremos del bloque libre inicial y recolección global de basuras cuando no queda espacio en dicho bloque (algoritmo ping-pong):** En este caso, lo que va quedando del bloque libre inicial a lo largo de la ejecución se encuentra más o menos centrado en la memoria dinámica disponible, por lo que será preciso localizar su posición mediante dos punteros. Este procedimiento puede reducir considerablemente el número de recolecciones de basuras que ha de realizarse durante la ejecución de un programa. Para ver cómo puede ocurrir esto, y cómo difiere este método del anterior, considérese el siguiente programa escrito en el lenguaje APL:

```
A ← 1 3 2.5
A ← 2 + 3 × ⌊ A
```

La primera instrucción asigna a la variable `A` el vector formado por los tres valores reales (`1 3 2.5`). La segunda instrucción calcula el resultado de sumar `2` al triple de la parte entera de `A`. La expresión `⌊A` significa *la parte entera de* `A`. En APL, las operaciones aritméticas se aplican automáticamente a vectores, matrices y estructuras completas más complejas, de modo que las operaciones indicadas en la segunda instrucción necesitarían de la asignación de memoria dinámica para almacenar los distintos resultados parciales, de los que sólo el último será asignado como nuevo valor de la variable `A`.

Las Figuras 10.5 y 10.6 muestran cómo se ejecutarían estas dos instrucciones con el procedimiento de la asignación de memoria por un solo extremo y por los dos extremos del bloque inicial, respectivamente.

Caso de la asignación de memoria por un solo extremo:

a) En la situación inicial, la memoria está vacía. El puntero apunta al principio de dicha memoria.

b) Durante la ejecución de la instrucción `A ← 1 3 2.5`, se obtiene de la memoria libre el espacio dinámico necesario para introducir un vector de tres elementos. Dicho espacio queda asignado a la variable `A`, mientras el puntero a la memoria libre se ha desplazado adecuadamente.

c) En la segunda instrucción, la primera operación que hay que realizar es el cálculo de la parte entera de (`⌊A`), cuyo valor resultará ser el vector de tres elementos (`1 3 2`). Para calcularlo, se necesita otro bloque de memoria dinámica, que se extrae del principio de la zona libre. El puntero se desplaza adecuadamente.

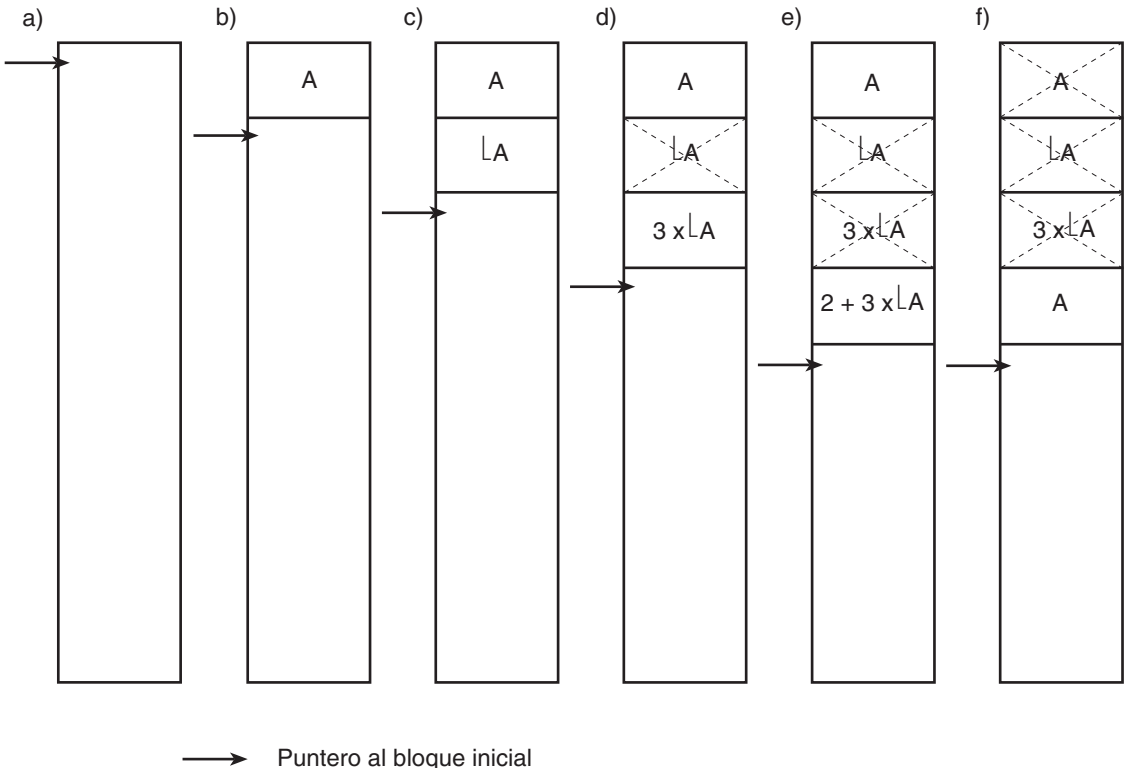


Figura 10.5. Ejecución de dos instrucciones APL con asignación dinámica de memoria por un solo extremo del bloque inicial.

d) La segunda operación que hay que realizar es el producto de la constante `3` por el resultado de la operación anterior. Para ello, hay que pedir espacio para otro vector de tres elementos, al que se asignarán los valores (`3 9 6`). Mientras no se haya calculado ese producto, no se puede prescindir del resultado de `⌊A`. Pero una vez calculado `3 × ⌊ A` ya se puede declarar basura el espacio asignado a `⌊A`.

e) A continuación, hay que sumar la constante `2` al resultado de la operación anterior. Para ello, hay que pedir nuevo espacio. Con el algoritmo que estamos considerando, dicho espacio se obtiene del bloque libre inicial, sin reutilizar el que fue declarado basura en el paso anterior, que no estará directamente accesible hasta que se realice una recolección de desechos. En el bloque nuevo se calcula la operación `2 + 3 × ⌊ A`, cuyo resultado es (`5 11 8`). Sólo en este punto se puede declarar basura el bloque correspondiente al resultado de la operación anterior (`3 × ⌊ A`).

f) Finalmente, se ejecuta la asignación del último resultado a la variable `A`. El valor antiguo se declara basura. Obsérvese que el bloque libre inicial ha quedado fragmentado en dos zonas libres, separadas por una zona utilizada por el valor actual de la variable `A`.

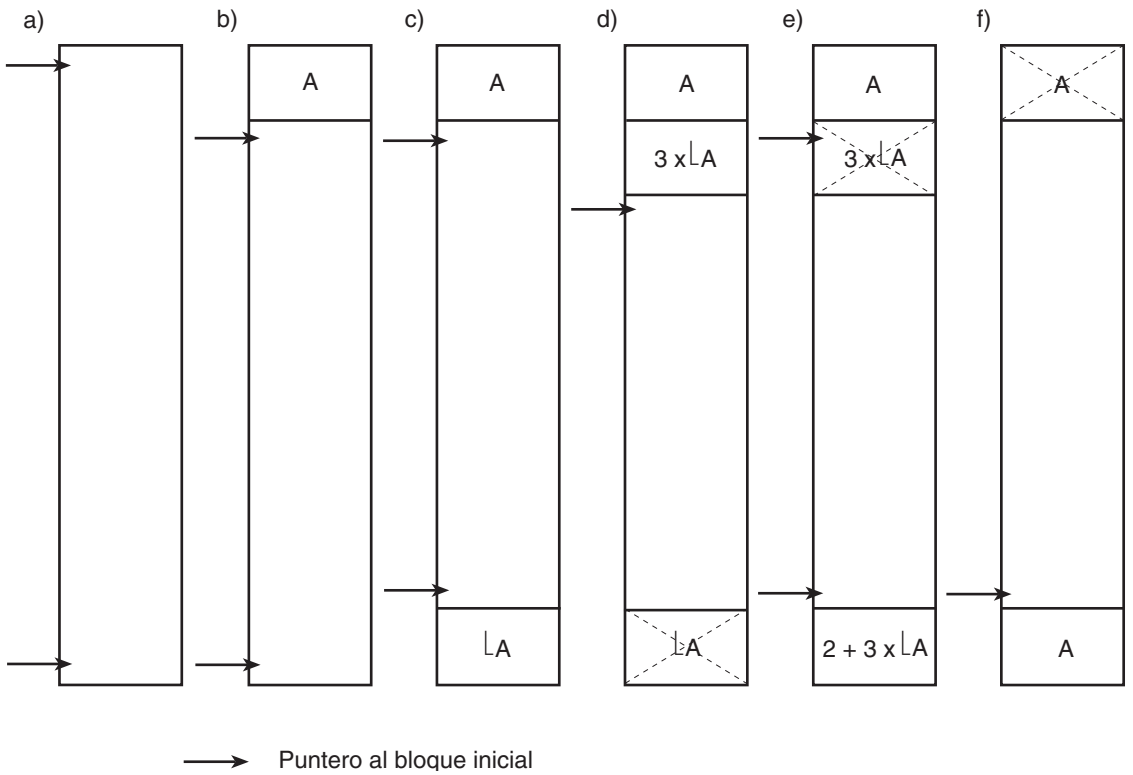


Figura 10.6. Ejecución de dos instrucciones APL con asignación dinámica de memoria por los dos extremos del bloque inicial.

Caso de la asignación de memoria alternativamente por los dos extremos:

a) En la situación inicial, la memoria está vacía. Los dos punteros apuntan al principio y al fin de dicha memoria.

b) Durante la ejecución de la instrucción `A ← 1 3 2.5`, se obtiene de la parte superior de la memoria libre el espacio dinámico necesario para introducir un vector de tres elementos. Dicho espacio queda asignado a la variable `A`, mientras el primer puntero a la memoria libre se ha desplazado adecuadamente.

c) En la segunda instrucción, la primera operación que hay que realizar es el cálculo de la parte entera de (`⌊A`), cuyo valor resultará ser el vector de tres elementos (`1 3 2`). Para calcularlo, se necesita otro bloque de memoria dinámica, que se extrae del final de la zona libre. El segundo puntero se desplaza adecuadamente.

d) La segunda operación que hay que realizar es el producto de la constante `3` por el resultado de la operación anterior. Para ello, hay que pedir espacio para otro vector de tres elementos, que se obtendrá del extremo superior de la memoria libre, y al que se asignarán los valores (`3 9 6`). Una vez calculado el producto `3 × ⌊ A` ya se puede declarar basura el espacio asignado a `⌊A`. Como este espacio es frontero con el final

actual de la memoria libre, puede fusionarse directamente con ésta, modificando el valor de dicho puntero.

e) A continuación, hay que sumar la constante `2` al resultado de la operación anterior. Para ello, hay que pedir nuevo espacio, que se obtendrá del extremo inferior del bloque libre inicial, con lo que se reutilizará automáticamente el bloque que quedó libre en el paso anterior. En el bloque nuevo se calcula la operación `2 + 3 ×⌊ A`, cuyo resultado es (`5 11 8`). En este punto se puede declarar basura el bloque correspondiente al resultado de la operación anterior (`3 ×⌊ A`). Como dicho bloque es fronterizo con el extremo superior de la memoria libre, el algoritmo que estamos utilizando lo fusionará automáticamente con ésta, modificando el valor del puntero correspondiente.

f) Finalmente, se ejecuta la asignación del último resultado a la variable `A`. El valor antiguo se declara basura. Dado que dicho valor es fronterizo con el bloque libre, puede fusionarse con él inmediatamente. Obsérvese que el bloque libre inicial no ha quedado fragmentado, y que el valor asignado a la variable `A` se encuentra ahora en el otro extremo de la memoria. Es evidente que, con este algoritmo, el número de recolecciones de basura necesarias disminuye considerablemente.

- **Algoritmo basado en la utilización de células colega (*buddy cells*):** Este algoritmo, que se debe a Knuth, Markowitz y Knowlton, reparte el espacio disponible en celdas cuyo tamaño es una potencia de 2. Si se necesita un bloque de un tamaño determinado, dicho tamaño se extiende a la potencia de 2 inmediata superior. Existen tantas listas de zonas libres como tamaños posibles. Si una lista está vacía y se precisa un bloque de ese tamaño, se extrae un bloque de la lista de tamaño superior y se divide en dos. Uno queda asignado como respuesta a la petición, y el otro pasa a la lista de bloques libres. El procedimiento es recursivo. Estos dos bloques están ligados entre sí permanentemente (se los llama colegas).

  Cuando se declara basura un bloque, se examina el estado en que se encuentra su compañero. Si está libre también, ambos bloques se fusionan para formar un solo bloque de tamaño doble. Este procedimiento también es recursivo.

  Veamos un ejemplo. Supongamos que la memoria libre está formada por un solo bloque de 4 kBytes y que éste es el tamaño máximo posible. Los tamaños menores que vamos a considerar son 128, 256 y 512 Bytes, 1 kByte y 2 kBytes. Existirán, por tanto, seis listas de memoria libre, cinco de las cuales estarán inicialmente vacías [*véase* la Figura 10.7.a)].

  Se pide espacio para 80 Bytes. La petición se redondea a la potencia de 2 inmediata superior (128 Bytes). Dado que la lista correspondiente está vacía, se pasa a la lista siguiente (256), que también está vacía, y así sucesivamente hasta llegar a la única lista con elementos libres (la de 4 kBytes). Se divide el bloque de 4 kBytes en dos zonas iguales de 2 kBytes. Se pone una en la lista de bloques correspondiente, mientras la otra se divide en dos zonas de 1 kByte. Se pone una en la lista de bloques de 1 kByte y se divide la otra en dos de 512 Bytes, y así sucesivamente hasta que obtenemos dos zonas de 128 Bytes, una de las cuales

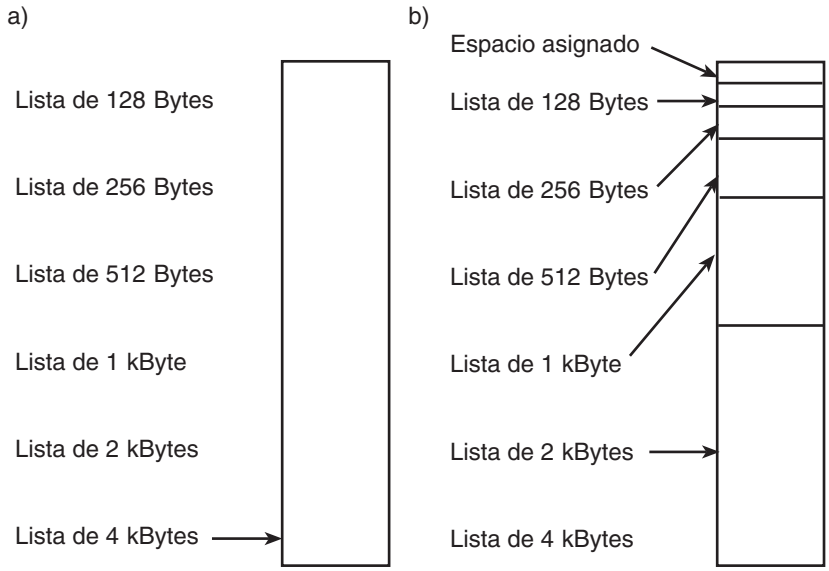


Figura 10.7. Ejemplo de recolección de basura utilizando células colega.

irá a la memoria libre y la otra será devuelta como respuesta a la petición inicial [*véase* la Figura 10.7.b)].

Si la memoria libre de 4 kBytes empieza originalmente en una dirección múltiplo de 4 kBytes, cada bloque parcial de 2^n Bytes tendrá una dirección cuyos n bits inferiores son iguales a cero. Esto significa que la dirección de la celda colega de una celda dada se puede obtener fácilmente haciendo una operación *O exclusivo* de su dirección con su tamaño. Cada zona debe estar asociada con dos datos: un bit que indique si está libre y un campo que indique su tamaño. Con eso, se puede localizar la celda colega. Cuado se libera un bloque, se mira a ver si su pareja tiene el mismo tamaño y si está libre. En tal caso, ambas pueden fusionarse en un bloque de tamaño doble.

Para evitar fusiones y divisiones consecutivas, se puede dejar un número mínimo de bloques en cada cadena o retardar la fusión hasta que se requiera un bloque grande. Se calcula que la utilización media de cada bloque no pasa del 75% (en la práctica es algo menor, pues los bloques menores son más frecuentes que los mayores).

10.3 Resumen

Este capítulo aborda el tema de la gestión de memoria en los procesadores de lenguaje. Se trata de una componente que funciona de modo completamente diferente para los compiladores y para los intérpretes, por lo que el capítulo se divide de forma natural en las dos secciones correspondientes.

En la primera sección, se analiza el modo en que un compilador debe gestionar la utilización de la memoria por el programa objeto que está generando. Se analizan distintos modos del uso de la memoria por los programas objetos en las arquitecturas INTEL, y se distingue entre el tratamiento que se puede aplicar a las variables estáticas, automáticas y dinámicas.

En la segunda sección, se revisan distintos métodos para la gestión de la memoria de un intérprete (memoria que utilizan en común el propio intérprete y el programa objeto). El problema principal que hay que resolver en este contexto es la eliminación de la memoria innecesaria, que tuvo utilidad en algún momento pero posteriormente ha dejado de tenerla. Se comparan distintos procedimientos para la compactación de la memoria utilizable y la eliminación de basuras o desechos.

Bibliografía

[1] Aho, A. V.; Sthi, R. y Ullman, J. D. (1986): *Compiler: Principles, techniques and Tools,* Reading, MA, Addison-Wesley Publishing Company.

[2] Alfonseca, N.; Sancho, J. y Martínez, M. (1990): *Teoría de lenguajes, gramáticas y Autómatas,* Madrid, Ed. Universidad y Cultura.

[3] Fischer, C. N. y LeBlanc Jr., R. J. (1991): *Crafting a compiler with C. Redwood City,* Benjamin/Cummings.

[4] Gries, D. (1971): *Compiler construction for Digital Computers,* New York, John Wiley and Sons, Inc. Existe traducción española de F. J. Sanchís Llorca, 1975.

[5] Grune, D. (2000): *Modern Compiler Design,* Wiley.

[6] Hopcroft, J. E.; Motwani, R. y Ullman, J. D. (2001): *Introduction to Automata Thory, Languages, and Computation,* Addison Wesley. Existe versión española: Hopcroff, J. E., Motwani, R. y Ullman, J. D. (2002): *Introducción a la Teoría de Autómatas, Lenguajes y Computación,* Madrid, Addison Wesley.

[7] Ed. Koskimies, K. (1998), *Compiler construction,* Proc. 7th Int. Conf. CC'98, Springer.

[8] Linz, P. (1990): *An introduction to Formal Languages and Automata,* Lexington, D. C., Heath and Co.

[9] Louden, K. C. (2004): *Construcción de compiladores. Principios y práctica,* Thomson.

[10] Marcotty, M.; Ledgard, H. F. y Bochmann, G. V. (1976): *A sampler of Formal Definitions,* Computing Surveys, 8:2, pp. 181-276.

[11] Wirth, N. (1996): *Compiler Construction,* Addison-Wesley.

Índice analítico

B

C

R

S