Apuntes de compiladores

Miguel Toro Bonilla
Rafael Corchuelo Gil
José A. Troyano Jiménez

Departamento de Lenguajes y Sistemas Informáticos
Facultad de Informática y Estadística, Universidad Hispalense
Avda. Reina Mercedes s/n, 41.012, Sevilla, España
Teléfono/Fax: (95) 455.71.39
E-mail: {mtoro, corchu, troyano}@lsi.us.es

Curso 98/99

Índice

I Lenguajes y Máquinas Formales **1**

1 Analizadores LR(k) **3**
- 1.1 Analizadores LR y autómatas de pila no deterministas . . . 3
- 1.2 Problemas generales de implementación . . . 6
- 1.3 Tablas de análisis LR . . . 8
  - 1.3.1 Interpretación . . . 8
  - 1.3.2 El lenguaje de la pila de análisis . . . 9
  - 1.3.3 Ítems LR(0) . . . 10
  - 1.3.4 Autómata finito no determinista para el lenguaje de la pila . . . 10
  - 1.3.5 Autómata finito determinista para el lenguaje de la pila . . . 12
  - 1.3.6 Construcción de las tablas de análisis a partir del autómata para el lenguaje de la pila . . . 15
  - 1.3.7 Ejemplo: Las sentencias `if...then...else` . . . 16
  - 1.3.8 Ejemplo: Una lista de elementos . . . 18
- 1.4 Tablas de análisis SLR(1) . . . 20
- 1.5 Tablas de análisis LR(1) . . . 21
  - 1.5.1 Ítems LR(1) . . . 21
  - 1.5.2 Construcción del autómata basado en ítems LR(1) para el lenguaje de la pila . . . 21
  - 1.5.3 Construcción de la tabla de análisis . . . 21
  - 1.5.4 Ejemplo: Las sentencias `if...then...else` . . . 22
  - 1.5.5 Ejemplo: Concatenación de dos listas . . . 23
  - 1.5.6 Ejemplo: Concatenación de múltiples listas . . . 24
  - 1.5.7 ¿Por qué el símbolo de lectura adelantada del método LR(1) resuelve más conflictos que el del método SLR(1)? . . . 25
- 1.6 Tablas de análisis LR(k), $k \geq 2$ . . . 27
- 1.7 Tablas de análisis LALR(1) . . . 30
- 1.8 Bibliografía . . . 31

2 Tratamiento de errores en analizadores LR **33**
- 2.1 Conceptos generales . . . 33
  - 2.1.1 ¿Qué es un error sintáctico? . . . 33
  - 2.1.2 Respuesta ante los errores . . . 34
- 2.2 Métodos *ad hoc* . . . 35
- 2.3 Método de Aho y Johnson . . . 35
  - 2.3.1 χ como símbolo ficticio . . . 37
  - 2.3.2 χ para ignorar parte de la entrada . . . 37
- 2.4 Método de análisis de transiciones . . . 38
- 2.5 Otros métodos de recuperación . . . 41

2.5.1 Método de Graham y Rhodes . . . 42
2.5.2 Método de McKenzie, Weatman y De Vere . . . 43
2.5.3 La segunda oportunidad . . . 43
2.6 Bibliografía . . . 43

II Léxico y Sintaxis 45

3 Analizadores léxicos 47
3.1 Papel del Analizador Léxico . . . 47
3.1.1 Interfaces del analizador léxico . . . 48
3.2 Especificación del Analizador Léxico . . . 48
3.2.1 Atributos de los tokens . . . 49
3.3 Problemas con el diseño . . . 51
3.3.1 Problemas con los identificadores . . . 51

4 La herramienta `lex` 55
4.1 Esquema de un fuente en `lex` . . . 55
4.1.1 Zona de definiciones . . . 55
4.1.2 Zona de reglas . . . 56
4.1.3 Zona de rutinas de usuario . . . 58
4.2 Cómo se compila un fuente `lex` . . . 58
4.2.1 Opciones estándar . . . 58
4.2.2 Depuración de un fuente en `lex` . . . 59
4.2.3 Prueba rápida de un fuente . . . 59
4.3 Expresiones regulares en `lex` . . . 59
4.3.1 Caracteres . . . 60
4.3.2 Clases de caracteres . . . 60
4.3.3 Cómo reconocer cualquier carácter . . . 61
4.3.4 Unión . . . 61
4.3.5 Clausuras . . . 62
4.3.6 Repetición acotada . . . 62
4.3.7 Opción . . . 62
4.3.8 Paréntesis . . . 63
4.4 Tratamiento de las ambigüedades . . . 63
4.5 Dependencias del contexto . . . 64
4.5.1 Dependencias simples . . . 64
4.5.2 Dependencias complejas . . . 65
4.6 Variables predefinidas por `lex` . . . 67
4.7 Acciones predefinidas . . . 68
4.7.1 yyless(n) . . . 68
4.7.2 yymore() . . . 69
4.7.3 REJECT . . . 69
4.7.4 yywrap() . . . 70
4.8 Algunas implementaciones comerciales de `lex` . . . 71
4.8.1 PC `lex` . . . 71
4.8.2 Sun `lex` . . . 71
4.8.3 Ejemplos . . . 71
4.8.4 Ejemplo 1 . . . 72
4.8.5 Ejemplo 2 . . . 72

4.8.6 Ejemplo 3 . . . 72
4.8.7 Ejemplo 4 . . . 73
4.8.8 Ejemplo 5 . . . 74
4.8.9 Ejemplo 6 . . . 74
4.8.10 Ejemplo 7 . . . 77
4.8.11 Ejemplo 8 . . . 78

5 Análisis sintáctico 81
5.1 Introducción . . . 81
5.2 Especificación del Analizador Sintáctico . . . 81
5.2.1 Especificación de Secuencias . . . 85
5.3 Transformación de Especificaciones . . . 86
5.3.1 Incorporación de las reglas de ruptura de la ambiguedad a la Gramática 86
5.3.2 Transformación de BNFE a BNF . . . 87

6 La herramienta yacc 89
6.1 Introducción . . . 89
6.2 Especificaciones Básicas . . . 91
6.3 Acciones . . . 92
6.4 Relación con el analizador léxico . . . 95
6.5 Ambigüedados y conflictos . . . 96
6.6 Precedencia . . . 99

7 Tratamiento de errores sintácticos en YACC 103
7.1 Conceptos generales . . . 103
7.1.1 Los mensajes de error . . . 103
7.1.2 Herramientas para la corrección de errores . . . 104
7.2 Detección, recuperación y reparación de errores en analizadores LR . . . 104
7.2.1 La rutina estándar de errores . . . 104
7.2.2 Comportamiento predefinido ante los errores . . . 105
7.2.3 Un mecanismo elemental de tratamiento de errores . . . 105
7.2.4 Mecanismo elaborado de tratamiento de errores . . . 106
7.2.5 Ejemplo: Una calculadora avanzada . . . 112
7.2.6 Otra forma de conseguir la misma calculadora . . . 117

8 Árboles con atributos 119
8.1 Introducción . . . 119
8.1.1 Notación orientada a la manipulación de árboles . . . 119
8.1.2 Formato de entrada . . . 121
8.2 Taxonomía de las reglas: Constructores y Tipos . . . 122
8.3 Atributos de los árboles . . . 124
8.4 Otras operaciones de manipulación de árboles . . . 126
8.4.1 Manipulación de agregados . . . 126
8.4.2 Manipulación de secuencias . . . 126
8.4.3 Manipulación de elecciones . . . 127
8.4.4 Consideraciones Generales sobre las diferentes operaciones . . . 127
8.5 Funciones sobre árboles . . . 128
8.5.1 Concordancia de patrones sobre árboles . . . 128
8.5.2 Descripción metódica de recorridos . . . 131
8.5.3 Ejemplo . . . 133
8.6 Automatización de la implementación . . . 134

8.6.1 Formato de salida . . . 134
8.6.2 Implementación de la estructura segun . . . 136

9 Implementación de Gramáticas con atributos 139
9.1 Tipos de gramáticas con Atributos . . . 139
9.2 Evaluación sobre reconocedores sintácticos . . . 140
9.2.1 Reconocedores ascendentes . . . 141
9.2.2 Evaluación en YACC de gramáticas s-atribuidas: Construcción de árboles de SA en el reconocimiento sintáctico . . . 141
9.2.3 Evaluación en YACC de gramáticas lc-atribuidas . . . 143
9.3 Evaluación sobre árboles con atributos . . . 143
9.3.1 Especificación . . . 144
9.3.2 Implementación . . . 144

III Semántica Estática 147

10 Sintaxis Abstracta y gramáticas con atributos 149
10.1 Introducción . . . 149
10.1.1 Detalles de sintaxis concreta . . . 149
10.1.2 Ambigüedad y árboles . . . 150
10.2 Gramáticas abstractas . . . 150
10.3 Aplicaciones de la sintaxis abstracta . . . 151
10.4 Gramáticas con atributos . . . 152
10.4.1 Un ejemplo simple . . . 152
10.4.2 ¿Gramáticas concretas o gramáticas abstractas? . . . 153
10.5 Definición y especificación de gramáticas con atributos . . . 153
10.5.1 Notación . . . 154
10.5.2 Atributos heredados y sintetizados . . . 156
10.5.3 Especificación . . . 156

11 Tablas de Símbolos 159
11.1 Las declaraciones en los lenguajes de programación . . . 159
11.2 La tabla de símbolos . . . 160
11.3 Tablas de símbolos con estructura de bloques . . . 162
11.4 Soluciones a problemas particulares . . . 164
11.4.1 Campos y registros . . . 164
11.4.2 Reglas de importación y exportación . . . 164
11.4.3 Sobrecarga . . . 165
11.5 Relación entre el analizador léxico, el sintáctico y la tabla de símbolos . . . 166
11.5.1 ¿Debe el analizador léxico insertar símbolos en la tabla de símbolos? 166
11.5.2 ¿Debe el analizador léxico consultar la tabla de símbolos . . . 167
11.5.3 Entonces, ¿dónde se usa entonces la tabla de símbolos? . . . 169

12 La comprobación de tipos 173
12.1 Los sistemas de tipos . . . 173
12.2 Taxonomía de los tipos de datos . . . 174
12.2.1 Tipos Básicos . . . 174
12.2.2 Constructores de tipos . . . 176
12.2.3 Nombres de tipo . . . 179
12.3 Igualdad de tipos . . . 179

12.3.1 Igualdad de tipos y tablas . . . 180
12.3.2 Algunos comentarios respecto a la implementación de la igualdad de tipos . . . 180
12.4 Compatibilidad de tipos . . . 181
12.4.1 Implementación de la compatibilidad de tipos . . . 181
12.4.2 Un ejemplo sencillo . . . 181
12.5 Sobrecarga de operadores y rutinas . . . 182
12.5.1 Posibles tipos de una subexpresión . . . 182
12.5.2 Reducción del conjunto de tipos posibles . . . 185
12.5.3 Unos ejemplos más complicados . . . 187
12.6 Polimorfismo . . . 190
12.6.1 Igualdad de tipos polimorfos . . . 190
12.6.2 Inferencia de Tipos . . . 192
12.7 Valores–l y valores–r . . . 193
12.7.1 Comprobación de l–values y r–values en presencia de un comprobador de tipos para operadores y rutinas sobrecargadas . . . 195
12.8 Comprobación de tipos para las sentencias . . . 196
12.9 Problemas . . . 196

IV Semántica Dinámica 197

13 Código intermedio. Generación 199
13.1 Introducción . . . 199
13.2 Notaciones prefija y postfija . . . 199
13.2.1 Notacion cuartetos . . . 200
13.2.2 Notacion Tercetos . . . 201
13.2.3 Notacion tercetos indirectos . . . 201
13.3 Generación de código intermedio . . . 202
13.3.1 Traducción de expresiones y referencias a estructuras de datos. . . . 202
13.3.2 Traducción de las estructuras de control . . . 208
13.3.3 Traducción de procedimientos y funciones . . . 210

14 Máquinas Virtuales 213
14.1 Máquinas Virtuales . . . 213
14.2 Implementación de los tipos en la Máquina Virtual . . . 214
14.3 Registros de activación asociados a los procedimientos . . . 216
14.4 Asignación de Memoria . . . 216
14.4.1 Asignación estática . . . 217
14.4.2 Asignación dinámica . . . 217
14.4.3 Anidamiento estático . . . 217
14.5 Maquina C . . . 218
14.5.1 Memorias de la máquina C . . . 218
14.5.2 Manejo de la Pila . . . 222
14.5.3 Subrutinas . . . 222
14.5.4 Estructura del código . . . 223
14.6 Cambios en la máquina C para tratar agregados . . . 226
14.6.1 Clasificación de los agregados de datos . . . 226
14.6.2 Métodos de representación . . . 227
14.6.3 Representación contigua de agregados de datos . . . 227

14.6.4 Representación dispersa de agregados de datos 231

15 Optimización local de código **235**
15.1 Normalización de expresiones 235
15.2 Representación de bloques básicos 235
15.2.1 Construcción de un GDA 235

V Apéndices **243**

A El preprocesador **245**
A.1 Introducción 245
A.2 El preprocesador de C 246
A.2.1 Inclusión de ficheros 246
A.2.2 Definición y expansión de macros 247
A.2.3 Compilación condicional 248
A.2.4 Otras directivas del preprocesador 250
A.3 Implementación de un preprocesador de C 253
A.3.1 Esquema general 253
A.3.2 Inclusión de ficheros 254
A.3.3 Las macros 255
A.3.4 Compilación condicional 258
A.3.5 Otras directivas 259
A.3.6 Implementación de `FicheroPP` 259
A.4 Enlace con el análisis léxico 260
A.5 Revisión del preprocesador 260
A.6 El léxico del preprocesador 261
A.6.1 Dependencias del contexto en el léxico 261
A.6.2 Los espacios en blanco 262
A.6.3 Expansión de las macros 263
A.7 La sintaxis del preprocesador 264
A.7.1 Estructura general 264
A.7.2 La directiva `#include` 265
A.7.3 Las directivas `#define` y `#undef` 265
A.7.4 Las directivas `#if`, `#ifdef`, etcétera 266
A.7.5 Directivas `#line`, `#error` y `#pragma` 267

B Implementación de tablas de símbolos **269**
B.1 Técnicas básicas de implementación 271
B.1.1 Secuencias no ordenadas 271
B.1.2 Secuencias ordenadas 271
B.1.3 Arboles binarios ordenados 271
B.1.4 Tablas *hash* 272
B.1.5 Espacio de cadenas 272

C Bibliografía recomendada **273**

Parte I

Lenguajes y Máquinas Formales

Capítulo 1

Analizadores LR(k)

En este capítulo vamos a introducir un nuevo concepto de analizador sintáctico que evita muchos de los problemas asociados a los de tipo LL(k).

LR(k) significa que la cadena de entrada se recorre siempre de izquierda a derecha (*Left to right*) y que cuando una palabra es reconocida, en la cadena de derivaciones que nos conduce a ella a partir del axioma siempre se desarrolla el símbolo no terminal más a la derecha (*Rightmost symbol*). El valor k indica el número de símbolos de lectura adelantada que necesita el reconocedor para funcionar de manera completamente determinista.

Estos analizadores son del tipo *bottom–up*, pues parten de la cadena de entrada e intentan reducirla al axioma de la gramática aplicando las reglas de producción en sentido inverso. Recordemos que los analizadores LL(k) toman el axioma y a partir de él intentan derivar la cadena de entrada, lo que restringe en algunas ocasiones su utilidad desde el punto de vista práctico.

Todos ellos están basados en el mismo autómata de pila y se diferencian en la fórma en que lo simulan. El esquema de funcionamiento de este autómata es el siguiente:

1. Se mira si lo que hay en las últimas casillas de la pila concuerda con la parte derecha de alguna de las reglas de producción de la gramática analizada.

2. Si existe concordancia, se elimina de la cima de la pila esa cadena y se cambia por la parte izquierda de la correspondiente producción. A esta acción se le llama reducción (*reduction* en la terminología inglesa).

3. Si no existe concordancia alguna, se lee un carácter más de la entrada y se apila. A esta acción se le suele llamar desplazamiento (*shift* en la terminología inglesa).

Si usando este proceso conseguimos agotar el contenido de la cinta de entrada y en la cima de la pila queda el axioma de la gramática, la palabra es reconocida. En otro caso no lo es.

Como ejemplo examinaremos la gramática $\mathtt{G} = (\{\mathtt{a}\}, \{\mathtt{S}\}, \mathtt{S}, \{\mathtt{S} ::= \mathtt{Sa}|\mathtt{a}\})$ y cómo se comportaría su autómata de reconocimiento LR ante la cadena de entrada `aa`. La traza de funcionamiento del autómata se muestra en la tabla 1.1.

Pila	Entrada	Acción
	aa$	shift a
a	a$	reduce S ::= a
S	a$	shift a
Sa	$	reduce S ::= Sa
S	$	accept

Tabla 1.1: Ejemplo de funcionamiento de un reconocedor LR.

En este capítulo estudiaremos varias técnicas para la construcción de este tipo de analizadores, valoraremos su utilidad desde el punto de vista práctico y las complementaremos con numerosos ejemplos.

1.1 Analizadores LR y autómatas de pila no deterministas

En esta sección estudiaremos un método para construir un autómata de pila no determinista que implemente el reconocedor LR de cualquier gramática independiente del contexto. Su principal desventaja es que el resultado es un autómata cuya simulación resulta muy ineficiente, por lo que será preciso estudiar más adelante algunos métodos para corregir este defecto. Fíjese en que sea cual sea el número de caracteres de lectura adelantada que utilicemos, el autómata base es siempre el mismo. Las técnicas que en adelante estudiaremos tienen como único objetivo optimar la simulación de este autómata base.

Para construir el autómata de reconocimiento LR de una gramática independiente del contexto $\mathtt{G} = (\Sigma_\mathtt{T}, \Sigma_\mathtt{N}, \mathtt{S}, \mathtt{P})$, se deben seguir los siguientes pasos:

1. Construir los cuatro estados del autómata: `i` el inicial, `f` el de aceptación y `p` y `q` dos estados intermedios.
2. Introducir las transiciones estándar para marcar la cima de la pila, para determinar cuándo aparece en ella el axioma y para determinar cuándo queda vacía: $(\mathtt{i}, \lambda, \lambda; \mathtt{p}, \#)$, $(\mathtt{p}, \lambda, \mathtt{S}; \mathtt{q}, \lambda)$ y $(\mathtt{q}, \lambda, \#; \mathtt{f}, \lambda)$.
3. Por cada símbolo terminal $\mathtt{a} \in \Sigma_\mathtt{T}$, introducir una transición de la forma $(\mathtt{p}, \mathtt{a}, \lambda; \mathtt{p}, \mathtt{a})$. Esta transición se corresponde con las acciones `shift` de desplazamiento de caracteres desde la entrada hasta la cima de la pila que estudiábamos en la introducción del tema.
4. Por cada regla de la forma $\mathtt{A} ::= \alpha \in \mathtt{P}$, añadimos la transición $(\mathtt{p}, \lambda, \alpha; \mathtt{p}, \mathtt{A})$. Estas transiciones se corresponden con las acciones de reducción.

Si seguimos estos pasos, el aspecto de los autómatas que obtendremos será el que se muestra en la figura 1.1.

Es conveniente destacar que en el paso número 4 hemos modificado el concepto de transición para permitir desapilar simultáneamente varios símbolos. Según este paso, si en nuestra gramática existe una regla de producción de la forma $\mathtt{A} ::= \mathtt{abA}$, tenemos que

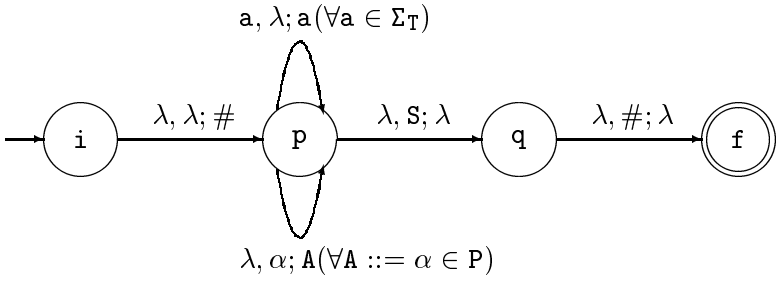


Figura 1.1: (pics/egalr.pic) Esquema general de un autómata LR.

añadir al autómata la transición $(\mathtt{p}, \lambda, \mathtt{abA}; \mathtt{p}, \mathtt{A})$ que indica que cada vez que en la cima de la pila se encuentre la cadena `abA` se debe desapilar y sustituir por el símbolo `A`. De forma general, se considera que la transición $(\mathtt{p}, \mathtt{x}, \mathtt{x_1x_2}\ldots\mathtt{x_n}; \mathtt{q}, \mathtt{y})$ es equivalente a las siguientes `n` transiciones encadenadas:

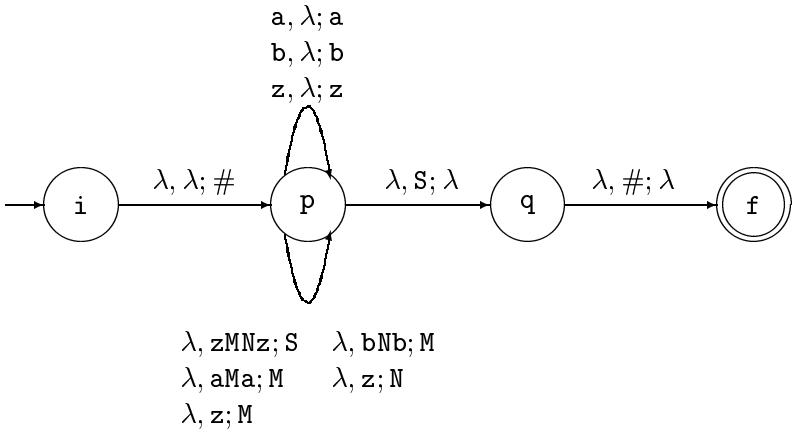
$$(\mathtt{p}, \mathtt{x}, \mathtt{x_n}; \mathtt{p_1}, \lambda) \rightarrow (\mathtt{p_1}, \lambda, \mathtt{x_{n-1}}; \mathtt{p_2}, \lambda) \rightarrow \cdots \rightarrow (\mathtt{p_{n-1}}, \lambda, \mathtt{x_1}; \mathtt{q}, \mathtt{y})$$

en donde $\mathtt{p_1}, \mathtt{p_2}, \ldots, \mathtt{p_{n-1}}$ son estados transitorios nuevos, no alcanzables más que a través de las transiciones que hemos indicado.

Como ejemplo examinaremos la siguiente gramática:

$$\mathtt{G} = (\{\mathtt{a}, \mathtt{b}, \mathtt{z}\}, \{\mathtt{S}, \mathtt{M}, \mathtt{N}\}, \mathtt{S}, \{\mathtt{S} ::= \mathtt{zMNz}, \mathtt{M} ::= \mathtt{aMa}|\mathtt{z}, \mathtt{N} ::= \mathtt{bNb}|\mathtt{z}\}$$

Si seguimos los pasos que hemos indicado, resulta sencillo comprobar que el autómata que se obtiene como resultado es el que se muestra en la siguiente figura:



Para hacernos una idea de lo complejo que puede llegar a ser simular este autómata, vamos a realizar la traza de su funcionamiento cuando lo enfrentamos a la cadena de entrada `zazabzbz$`.

Estado	Pila	Entrada	Acción
i		zazabzbz$	push #
p	#	zazabzbz$	shift z
*p	#z	azabzbz$	...

En este punto nos encontramos con un caso de indeterminación, pues en el estado de nombre p teniendo el símbolo z en la cima de la pila y una a en la cinta de entrada, son posibles cualquiera de las siguientes transiciones: $(p, a, \lambda; p, a)$, $(p, \lambda, z; p, M)$ o $(p, \lambda, z; p, N)$. Para determinar si la palabra pertenece o no al lenguaje descrito por la gramática, el autómata debería tomar estas tres acciones en paralelo y determinar si alguna de ellas nos conduce al estado de aceptación.

En lo que resta de la traza, marcaremos con un asterisco todos aquellos casos en los que se presentan varias elecciones y nos decantaremos por una u otra arbitrariamente señalándola entre corchetes.

Estado	Pila	Entrada	Acción
*p	#z	azabzbz$	[shift a] reduce M ::= z reduce N ::= z
p	#za	zabzbz$	shift z
*p	#zaz	abzbz$	shift a [reduce M ::= z] reduce N ::= z
p	#zaM	abzbz$	shift a
*p	#zaMa	bzbz$	shift b [reduce M ::= aMa]
p	#zM	bzbz$	shift b
p	#zMb	zbz$	shift z
*p	#zMbz	bz$	shift z reduce M ::= z [reduce N ::= z]
p	#zMbN	bz$	shift b
*p	#zMbNb	z$	shift z [reduce N ::= bNb]
p	#zMN	z$	shift z
p	#zMNz	$	[reduce S ::= zMNz] reduce M ::= z reduce N ::= z
p	#S	$	pop S
q	#	$	pop #
f		$	accept

Si analiza con detenimiento la forma en que se han resuelto los casos de indeterminación, descubrirá que en cada uno hemos utilizado la reducción más adecuada para llegar a la aceptación de la palabra. Un simulador probablemente optaría por otras alternativas que conducirían a situaciones de error, pero tendría que probar con todas ellas antes de poder rechazar una palabra como no perteneciente al lenguaje para el que se ha creado el autómata. El problema es que el número de alternativas suele ser enorme, produciéndose

un fenómeno conocido como explosión combinatoria que dificulta enormemente la simulación de este autómata.

De todas formas, si aceptamos una cadena usando este autómata podremos reconstruir con facilidad la secuencia de derivaciones que nos conduce a ella a partir del axioma `S`. Para conseguirlo tan sólo hay que estudiar las acciones reduce en orden inverso:

$$\texttt{S} \longrightarrow \texttt{zMNz} \longrightarrow \texttt{zMbNbz} \longrightarrow \texttt{zMbzbz} \longrightarrow \texttt{zaMabzbz} \longrightarrow \texttt{zazabzbz}$$

En esta secuencia de derivaciones se ve claramente que, en efecto, el no terminal que se desarrolla siempre es el que se encuentra más a la derecha.

1.2 Problemas generales de implementación

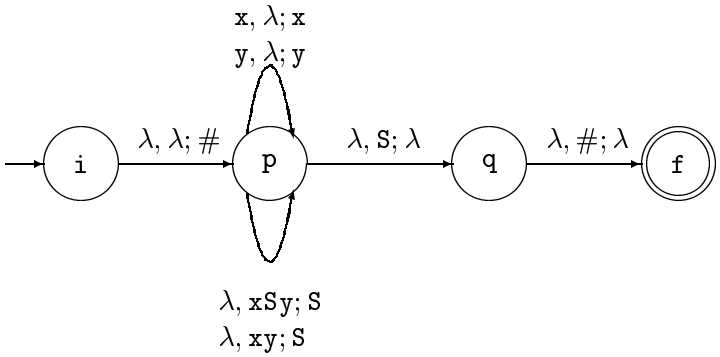
El método que hemos analizado adolece de problemas de eficiencia muy importantes que podrían hacer infactible el uso de los reconocedores LR. Son los siguientes:

- El carácter no determinista del autómata que hemos construido, lo que da lugar a explosiones combinatorias durante su simulación.

- El manejo de la pila, puesto que para que este autómata funcione es preciso comparar continuamente las partes derechas de todas las reglas de producción con las últimas casillas de la pila.

El primero se ha solucionado utilizando símbolos de lectura adelantada (*lookaheads* en la literatura inglesa). Cada vez que nos enfrentamos a una indeterminación leemos varios símbolos de la entrada, tantos como sea necesario para poder deshacer la ambigüedad. Esto no siempre es posible, pero en la mayoría de los problemas con interés práctico suele ser suficiente con uno o dos símbolos de lectura adelantada.

El segundo problema es bastante más difícil de resolver. Inicialmente se propusieron variantes del algoritmo de Boyer-Moore para la concordancia de patrones sobre la pila, pero aún así el resultado era muy ineficiente. Afortunadamente a alguien se le ocurrió una idea muy buena: introducir en la pila ciertas marcas que nos indiquen qué cadena hay debajo de ellas. De esta forma tan sólo consultado la cima podríamos saber qué hay en sus últimas casillas. Para ilustrar este procedimiento vamos a considerar la siguiente gramática: $\texttt{G} = (\{\texttt{x}, \texttt{y}\}, \{\texttt{S}\}, \texttt{S}, \{\texttt{S} ::= \texttt{xSy}|\texttt{xy}\})$. Si construimos su autómata de pila utilizando el método de la sección anterior, se obtiene el resultado que muestra la siguiente figura:



Vamos a seguir la traza de la pila al analizar la cadena de entrada `xxxyyy` e iremos asignando marcas de forma simultánea (de momento no se preocupe de cuál es la manera de elegirlas).

Estado	**Pila**	**Marca**	**Entrada**	**Acción**
i		0	xxxyyy$	push #
p	#	1	xxxyyy$	shift x
p	#x	2	xxyyy$	shift x
p	#xx	2	xyyy$	shift x
p	#xxx	2	yyy$	shift y
*p	#xxxy	3	yy$	reduce S ::= xy
p	#xxS	4	yy$	shift y
*p	#xxSy	5	y$	reduce S ::= xSy
p	#xS	4	y$	shift y
p	#xSy	5	$	reduce S ::= xSy
p	#S	4	$	pop S
q	#	1	$	pop #
f		0	$	accept

Es interesante darse cuenta de que cada uno de los estados en que se puede encontrar la pila ha sido caracterizado por una marca que se puede definir utilizando una expresión regular, tal y como se muestra a continuación:

$0 = \lambda$ $1 = \#$ $2 = \#xx^*$
$3 = \#xx^*y$ $4 = \#xx^*S$ $5 = \#xx^*Sy$

De esta forma, si en la pila introducimos estas marcas además de los símbolos de la gramática, cada vez que en la cima encontremos la marca `5` sabremos que debajo está la cadena `xSy`. El aumento de eficiencia aparece en el momento en el que asociamos a cada una de las marcas una acción (`reduce S ::= xSy`, si se trata de la marca `5`), pues en este caso podríamos reunir esa información en una tabla y la simulación del autómata podría llevarse a cabo sin realizar ni una sola concordancia de patrones. Esta idea permite ganar muchísima eficiencia, sobre todo en gramáticas reales que pueden contar con unas 250 reglas de producción con 5 símbolos en la parte derecha por término medio.

Más adelante estudiaremos el método que nos permite obtener de forma sistemática las marcas para la pila y la forma de usarlas. De momento tan sólo debe quedarle claro cuál es su significado y de qué forma se pueden aprovechar.

1.3 Tablas de análisis LR

En esta sección estudiaremos unas tablas especiales que nos sirven para representar los autómatas de reconocimiento LR. El aspecto más general de una de estas tablas es el que se muestra a continuación:

	a_1	$\cdots$	a_n	\$	A_1	$\cdots$	A_m
0	act_{0,a_1}	$\cdots$	act_{0,a_n}	$act_{0,\$}$	$goto_{0,A_1}$	$\cdots$	$goto_{0,A_m}$
1	act_{1,a_1}	$\cdots$	act_{1,a_n}	$act_{1,\$}$	$goto_{1,A_1}$	$\cdots$	$goto_{1,A_m}$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
k	act_{k,a_1}	$\cdots$	act_{k,a_n}	$act_{k,\$}$	$goto_{k,A_1}$	$\cdots$	$goto_{k,A_m}$

La tabla se divide verticalmente en dos zonas llamadas de acción y de saltos. Las filas se etiquetan con las marcas de la pila, las columnas de la zona de acción con los símbolos terminales de la gramática, incluido el terminador \$, y las de la zona de saltos con los símbolos no terminales. Las casillas de la zona de acciones pueden contener una o varias de las acciones que pueden realizar los autómatas de reconocimiento (`shift`, `reduce` o `accept`) y las de la zona de saltos contienen nombres de marcas. También es posible encontrar casillas vacías que indicarán situaciones de error.

1.3.1 Interpretación

El análisis de cualquier cadena de entrada comenzará siempre introduciendo en la pila del autómata la marca inicial, a la que se suele llamar `0`. A partir de este momento, cada vez que introduzcamos un símbolo en la pila, debemos meter a continuación un símbolo de estado o marca. Por lo tanto, en cualquier instante, la pila contendrá alternativamente símbolos de la gramática y marcas.

La forma de consultar la tabla es la siguiente: La fila a estudiar vendrá siempre dada por la marca que aparezca en la cima de la pila y la columna dependerá del carácter sobre el que se encuentra el puntero de lectura. Una vez seleccionada una casilla por este procedimiento, actuamos así:

1. Si la casilla tiene la palabra `accept`, entonces paramos, pues la cadena de entrada pertenece al lenguaje estudiado.
2. Si en la casilla no hay nada, entonces paramos y la cadena de entrada no pertenece al lenguaje estudiado.
3. Si en la casilla sólo pone `shift i`, entonces se lee el carácter actual de la entrada y se apila junto a la marca `i`.
4. Si en la casilla sólo pone `reduce i` entonces hay que estudiar la `i`-ésima regla de producción de la gramática. Si ésta es de la forma $\mathtt{A} ::= \alpha$, entonces se desapilan $2|\alpha|$ símbolos, con lo que en la cima quedará una marca temporal `m`. A continuación se introduce en la pila el símbolo `A` y se apila la marca de la casilla $goto_{m,A}$.
5. Si en la casilla existe más de una acción deberíamos realizarlas todas en paralelo, pero en la práctica este problema suele resolverse usando métodos heurísticos que comentaremos más adelante en la sección 1.3.7.

Antes de ver la forma en que se construyen estas tablas, vamos a estudiar un ejemplo detallado de su funcionamiento. Sea la gramática siguiente:

$$\mathtt{G} = (\{\mathtt{x}, \mathtt{y}\}, \{\mathtt{S}', \mathtt{S}\}, \mathtt{S}', \{\mathtt{S}' ::= \mathtt{S}, \mathtt{S} ::= \mathtt{xSy} | \mathtt{xy}\})$$

Las razones de por qué es preciso ampliar la gramática con el símbolo $\mathtt{S}'$ y la regla $\mathtt{S}' ::= \mathtt{S}$ las comprenderemos más adelante. La tabla LR para esta gramática es la siguiente:

#	Regla
1	$\mathtt{S}' ::= \mathtt{S}$
2	$\mathtt{S} ::= \mathtt{xSy}$
3	$\mathtt{S} ::= \mathtt{xy}$

	x	y	$	S′	S
0	shift 2				1
1			accept		
2	shift 2	shift 3			4
3	reduce 3	reduce 3	reduce 3		
4		shift 5			
5	reduce 2	reduce 2	reduce 2		

Usando esta tabla veremos a continuación la evolución del autómata al reconocer la cadena de entrada xxyy$.

Pila	Entrada	Acción
0	xxyy$	shift 2
0x2	xyy$	shift 2
0x2x2	yy$	shift 3
0x2x2y3	y$	reduce S ::= xy
0x2S4	y$	shift 5
0x2S4y5	$	reduce S ::= xSy
0S1	$	accept

1.3.2 El lenguaje de la pila de análisis

Para crear una tabla de análisis LR es necesario conocer bien el siguiente concepto fundamental: si nos fijamos con detenimiento en la forma en que se llena y se vacía la pila del autómata de reconocimiento, nos daremos cuenta de que todas las secuencias de símbolos que aparecen en ella constituyen un lenguaje regular. Por lo tanto, cada marca i se puede caracterizar utilizando una expresión regular α_i, como ya hemos visto intuitivamente en un ejemplo anterior, con lo que el lenguaje completo de la pila del autómata viene definido por la expresión regular:

$$\alpha = \alpha_1 + \alpha_2 + \cdots + \alpha_n$$

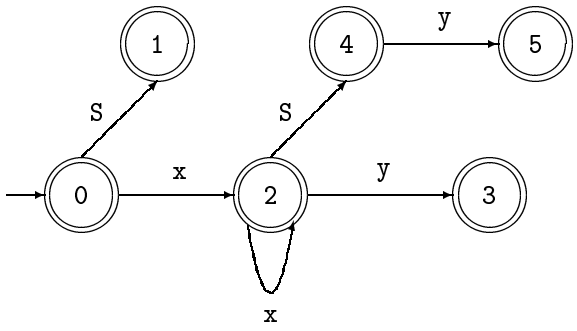
En el ejemplo anterior, resulta que:

$$\alpha_0 = \lambda \qquad \alpha_1 = \mathtt{S} \qquad \alpha_2 = \mathtt{xx}^*$$
$$\alpha_3 = \mathtt{xx}^*\mathtt{y} \quad \alpha_4 = \mathtt{xx}^*\mathtt{S} \quad \alpha_5 = \mathtt{xx}^*\mathtt{Sy}$$

Por lo que la expresión que describe el lenguaje de la pila es:

$$\alpha = \lambda + \mathtt{S} + \mathtt{x}\mathtt{x}^* + \mathtt{x}\mathtt{x}^*\mathtt{y} + \mathtt{x}\mathtt{x}^*\mathtt{S} + \mathtt{x}\mathtt{x}^*\mathtt{S}\mathtt{y}$$

Aplicando alguno de los algoritmos que ya hemos estudiado en un tema anterior se demuestra que uno de los autómatas finitos que reconoce este lenguaje es el que se muestra en la siguiente figura:



En estos autómatas todos los estados son siempre finales. En las siguientes secciones veremos de qué forma nos ayudan a construir el autómata para reconocer un lenguaje.

1.3.3 Ítems LR(0)

Este es un concepto previo necesario para construir el autómata para el lenguaje de la pila. Un ítem LR(0) es simplemente una regla gramatical a la que hemos añadido como información adicional un punto que puede aparecer en cualquier lugar de la parte derecha. Por ejemplo, los ítems LR(0) de la regla $\mathtt{S} ::= \mathtt{xSy}$ son: $\mathtt{S} ::= \bullet\mathtt{xSy}$, $\mathtt{S} ::= \mathtt{x}\bullet\mathtt{Sy}$, $\mathtt{S} ::= \mathtt{xS}\bullet\mathtt{y}$ y $\mathtt{S} ::= \mathtt{xSy}\bullet$.

Las reglas de la forma $\mathtt{A} ::= \lambda$ tan sólo tienen un ítem LR(0) que se puede escribir de tres formas distintas: $\mathtt{A} ::= \bullet\lambda$, $\mathtt{A} ::= \lambda\bullet$ o bien $\mathtt{A} ::= \bullet$. Los tres ítems son equivalentes puesto que λ es un símbolo ficticio que tan sólo sirve para representar la ausencia de símbolos en la parte derecha de la producción.

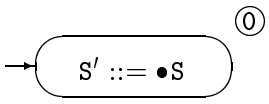
Para construir el autómata marcaremos sus estados con conjuntos de ítems LR(0). De esta forma, si en uno de los estados aparece $\mathtt{A} ::= \alpha \bullet \beta$ debemos interpretarlo diciendo que nos encontramos en un estado en el que en la cima de la pila tenemos una cadena que encaja en el patrón α y nos falta aún una cadena que encaje en el patrón β para poder reducir toda la regla. Por ejemplo, si el ítem es $\mathtt{S} ::= \mathtt{x}\bullet\mathtt{Sy}$, significa que en la cima de la pila hay una `x` y nos falta reducir una cadena que encaje en alguno de los patrones de `S` y leer de la entrada una `y` para poder reducir la regla completa.

1.3.4 Autómata finito no determinista para el lenguaje de la pila

Analizaremos primero un método sencillo que nos permite obtener con facilidad un autómata finito no determinista para reconocer el lenguaje de la pila. El método se desarrolla en los pasos que detallamos a continuación:

1. Dada la gramática $\mathtt{G} = (\Sigma_\mathtt{T}, \Sigma_\mathtt{N}, \mathtt{S}, \mathtt{P})$, la convertimos en otra equivalente de la forma $\mathtt{G}' = (\Sigma_\mathtt{T}, \Sigma_\mathtt{N} \cup \{\mathtt{S}'\}, \mathtt{S}', \mathtt{P} \cup \{\mathtt{S}' ::= \mathtt{S}\})$, en donde $\mathtt{S}'$ es un nuevo símbolo no terminal que hemos convertido en el axioma. La gramática se amplía de esta forma con el único objetivo de simplificar el algoritmo.

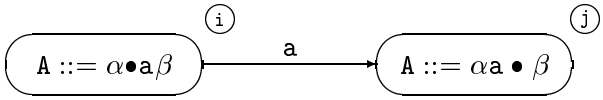
2. Crear el estado inicial con la estructura que se muestra en la siguiente figura:



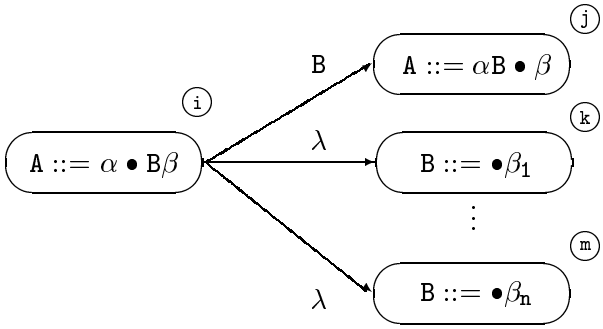
   Como en estos autómatas todos los estados son finales, no es necesario marcarlos con un doble círculo. Cualquier numeración que hagamos de ellos será válida, aunque es habitual reservar el cero para el estado inicial y numerar el resto en el mismo orden en que van apareciendo.

3. Repetir los pasos restantes del algoritmo hasta que sea imposible aplicarlos en más ocasiones.

4. En cada estado etiquetado con un ítem de la forma $\mathtt{A} ::= \alpha \bullet \mathtt{a}\beta$, siendo $\mathtt{a} \in \Sigma_\mathtt{T}$, introducimos una transición etiquetada con el carácter $\mathtt{a}$ hacia el estado etiquetado con $\mathtt{A} ::= \alpha\mathtt{a} \bullet \beta$. Si no existiese aún, sería preciso crearlo tal y como se muestra en la siguiente figura:



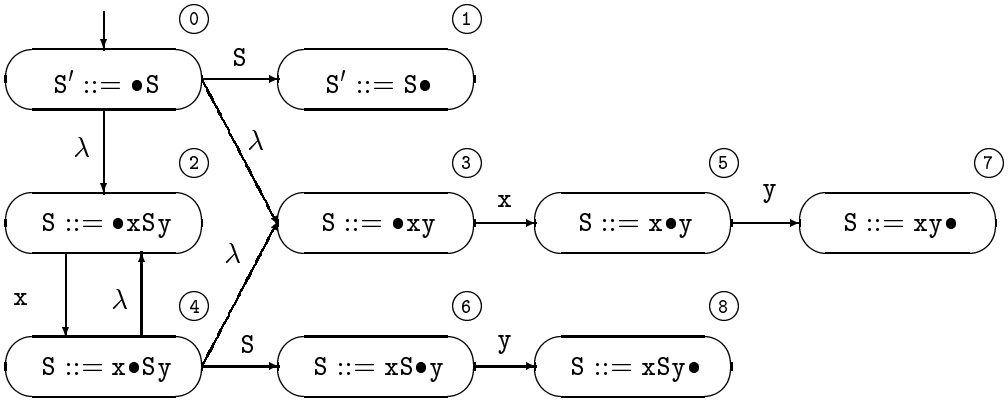
5. Por cada estado etiquetado con un ítem de la forma $\mathtt{A} ::= \alpha \bullet \mathtt{B}\beta$, siendo $\mathtt{B} \in \Sigma_\mathtt{N} \cup \{\mathtt{S}'\}$ y todas las $\mathtt{B}$-producciones de la forma $\mathtt{B} ::= \gamma_1 | \gamma_2 | \ldots | \gamma_\mathtt{n}$, introduciremos las transiciones que se muestran a continuación:



   Un estado como el que tratamos en este paso indica que en la cima de la pila tenemos una cadena de símbolos que encaja en el patrón α y nos falta el símbolo $\mathtt{B}$ y una cadena que encaje en β para poder reducir la regla $\mathtt{A} ::= \alpha\mathtt{B}\beta$. ¿De qué forma puede aparecer el símbolo $\mathtt{B}$ en la pila? Resulta evidente que tan sólo puede aparecer al realizar una operación `reduce`, y estas acciones tan sólo se pueden ejecutar cuando se reconoce una cadena que encaje en alguno de los patrones γ_1, γ_2, ..., γ_n. Por lo

tanto, lo más sensato es introducir λ-transiciones a partir del estado `i` hacia todos aquellos en los que comienza a reconocerse alguna de las `B`-producciones. Como en otras ocasiones, si estos estados no existen aún, deberán ser creados ahora.

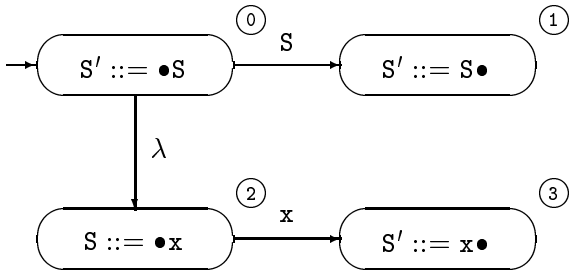
A modo de ejemplo, vamos a construir el autómata que reconoce el lenguaje asociado a las palabras de la pila durante el reconocimiento LR del lenguaje descrito por la gramática ampliada $\mathtt{G} = (\{\mathtt{x},\mathtt{y}\},\{\mathtt{S}',\mathtt{S}\},\mathtt{S}',\{\mathtt{S}' ::= \mathtt{S}, \mathtt{S} ::= \mathtt{xSy}|\mathtt{xy}\})$. Si seguimos cuidadosamente todos los pasos anteriores se obtiene el autómata de la siguiente figura:



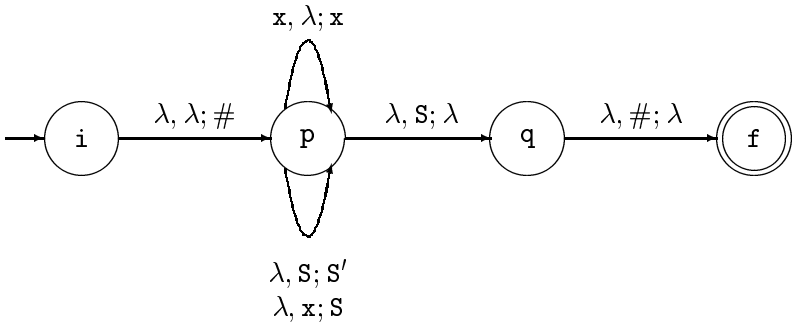
Observemos por ejemplo el estado `4` con el ítem LR(0) `S ::= x•Sy`, lo que significa que cuando el autómata llega a ese estado en la pila tenemos una `x` y para poder reducir `xSy` necesitamos una `S` y una `y`. ¿De qué manera se puede conseguir que en la cima aparezca una `S`? Evidentemente esto ocurrirá si logramos reducir `S ::= xSy` o `S ::= xy`. Por lo tanto, lo más sencillo es incluir λ-transiciones hacia todos aquellos estados en los que se empieza a reconocer una `S`-producción.

Adelantaremos que los números que hemos dado a los estados, se convertirán más adelante en las marcas para la pila. De esta forma, cuando la pila tenga la marca `7`, esto es, cuando el autómata de reconocimiento de su lenguaje se encuentra en ese estado, lo que deberíamos hacer sería un reduce `S ::= xy`, pues el ítem LR(0) de ese estado nos indica que ya tenemos en la cima de la pila todos los símbolos necesarios para ejecutar tal acción. En cambio, cuando la pila tenga la marca `3`, no podremos realizar ninguna reducción. En este caso, la única acción posible será `shift` si el carácter `x` aparece en la cinta de entrada.

Una vez introducidos estos conceptos de forma intuitiva, nos encontramos en situación de ofrecer una justificación práctica del quito paso del algoritmo. Para ello vamos a considerar la gramática: $\mathtt{G} = (\{\mathtt{x}\},\{\mathtt{S}',\mathtt{S}\},\mathtt{S}',\{\mathtt{S}' ::= \mathtt{S}, \mathtt{S} ::= \mathtt{x}\})$. Si seguimos los pasos del algoritmo, podremos comprobar que el autómata finito que se obtiene para reconocer el lenguaje de la pila de análisis LR es el que aparece en la siguiente figura:



También necesitaremos el autómata de pila para el reconocimiento LR:



Para realizar la justificación, vamos a realizar la traza de estos autómatas cuando la cadena de entrada contiene la palabra x$. Por el hecho de tratarse de autómatas no deterministas, sabemos que en un momento dado pueden encontrarse en varios estados simultáneamente y que cualquiera de las transiciones que parten de ellos es posible si en la entrada aparece el carácter adecuado. Hemos dividido la traza en dos columnas: la de la izquierda nos informa sobre el funcionamiento del autómata de pila y la de la derecha sobre la evolución paralela del autómata finito.

Traza del autómata de pila			**Traza del autómata finito**		
Pila	**Entrada**	**Acción**	**Entrada**	**Transición**	**Estados**
	x$	shift x		$0 \xrightarrow{\lambda} 2$	$\{0, 2\}$
x	$	reduce S := x	x	$2 \xrightarrow{x} 3$	$\{0, 3\}$
S	$	accept	S	$0 \xrightarrow{S} 1$	$\{1, 3\}$

Observe con detenimiento la forma en que evoluciona el autómata finito conforme van apareciendo nuevos caracteres en la pila que constituye su entrada. Inicialmente este autómata se encuentra en los estados número 0 y 2. Cuando el autómata de pila lleva a cabo la acción shift x, aparece una x en la pila, por lo que pasa a los estados 0 y 3 siguiendo la transición indicada. A continuación el autómata de pila realiza la acción reduce S ::= x, por lo que coloca en la cima de la pila el símbolo S. Dado que el autómata finito se encuentra aún en el estado 0, podrá llevarse a cabo la transición que lo conduce de este estado al número 1, en el que se acepta finalmente la cadena de entrada.

De una manera informal, podríamos decir que el autómata finito no puede salir del estado 0 hasta que no se produzca una reducción de alguna de las S-producciones. Para

detectar cuándo ocurre esto, lanza dos λ-transiciones hacia todos aquellos estados en los que se empiezan a reconocer estas producciones y "queda agazapado" hasta que a partir de alguno de ellos se llegue a uno con un ítem LR(0) terminal para alguna de las S-producciones.

1.3.5 Autómata finito determinista para el lenguaje de la pila

El método anterior es bueno, puesto que podemos construir un AFND, después convertirlo en determinista y minimizarlo. Pero en las gramáticas de interés práctico el número de estados que aparece puede ser muy grande, lo que dificulta el uso de esta técnica.

En cualquier caso es posible construir un autómata determinista directamente si tenemos en cuenta que tan sólo salen λ-transiciones de aquellos estados cuyo ítem LR(0) es de la forma $\mathtt{A} ::= \alpha \bullet \mathtt{B}\beta$, o lo que es lo mismo, aquellos estados en los que el punto está situado a la izquierda de un símbolo no terminal. Conviene recordar que la idea del algoritmo de conversión de autómata no determinista a determinista era compactar todos aquellos estados a los que se llega desde otro mediante λ-transiciones. En el caso de los autómatas construidos con el método de la sección anterior, dado un estado cualquiera es posible determinar si de él parten o no λ-transiciones y, en caso afirmativo, hacia qué estados están dirigidas.

A continuación se muestra el pseudocódigo de un algoritmo que implementa esta idea:

```
Entrada:
  G = (Σ_T, Σ_N, S, P)

Proceso:
  q_0 := Clausura({S' ::= •S}, P)
  EM := ∅  -- Estados marcados (estudiados)
  ENM := {q_0}   -- Estados no marcados (no estudiados)
  f := ∅   -- Función de transición

  mientras ENM ≠ ∅
    sacar un estado T de ENM
    para cada A ::= α•xβ ∈ T
      R_x^T := Clausura({A ::= αx•β}, P)
      si R_x^T ∉ (EM ∪ ENM) entonces
        ENM := ENM ∪ {R_x^T}
      fin si
      f(T, x) = R_x^T
    fin para
  fin mientras

Salida:
  A = (Q, Σ, f, F, q_0), donde
    Q = EM, F = EM, Σ = Σ_T ∪ Σ_N
```

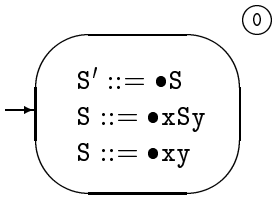
La parte fundamental del algoritmo es la función `Clausura` que toma una etiqueta de un estado `I` (un conjunto de ítems LR(0)) y le añade ítems de la forma $\mathtt{B} ::= \bullet\gamma$

para todos aquellos ítems que aparecen en I y son de la forma $\mathtt{A} ::= \alpha \bullet \mathtt{B}\beta$. Por ejemplo, si en un momento dado resulta que $\mathtt{I} = \{\mathtt{S} ::= \bullet\mathtt{AB}\}$ y el conjunto de producciones $\mathtt{P} = \{\mathtt{S} ::= \mathtt{AB}, \mathtt{A} ::= \mathtt{a}|\mathtt{B}, \mathtt{B} ::= \mathtt{b}\}$, entonces se obtiene que $\mathtt{Clausura(I)} = \{\mathtt{S} ::= \bullet\mathtt{AB}, \mathtt{A} ::= \bullet\mathtt{a}, \mathtt{A} ::= \bullet\mathtt{B}$ Fíjese en que al introducir el ítem $\mathtt{A} ::= \bullet\mathtt{B}$, nos encontramos en una situación en la que es preciso añadir todas las B-producciones a la clausura. En el caso general, es necesario repetir este proceso hasta que resulte imposible añadir más ítems al conjunto clausura.

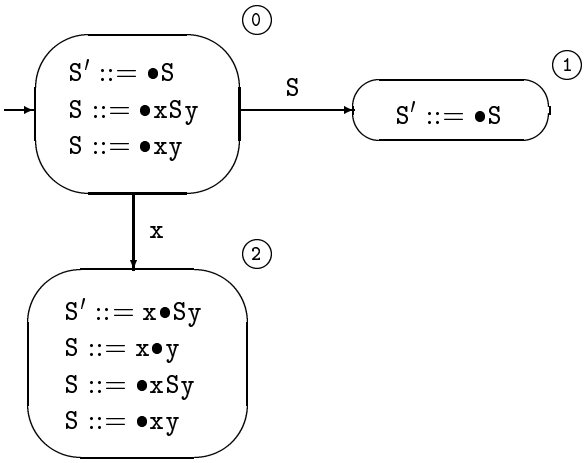
El pseudocódigo del algoritmo para calcular Clausura(I) es el siguiente:

```
Entrada:
  I -- Estado al que calcular la clausura
  P -- Conjunto de reglas de producción

Proceso:
  J := I
  repetir
    para cada ítem A ::= α•Bβ ∈ J
      para cada producción B ::= γ ∈ P tal que B ::= γ ∉ J
        J := J ∪ {B ::= •γ}
      fin para
    fin para
  hasta que sea imposible añadir más ítems LR(0) a J

Salida:
  J
```

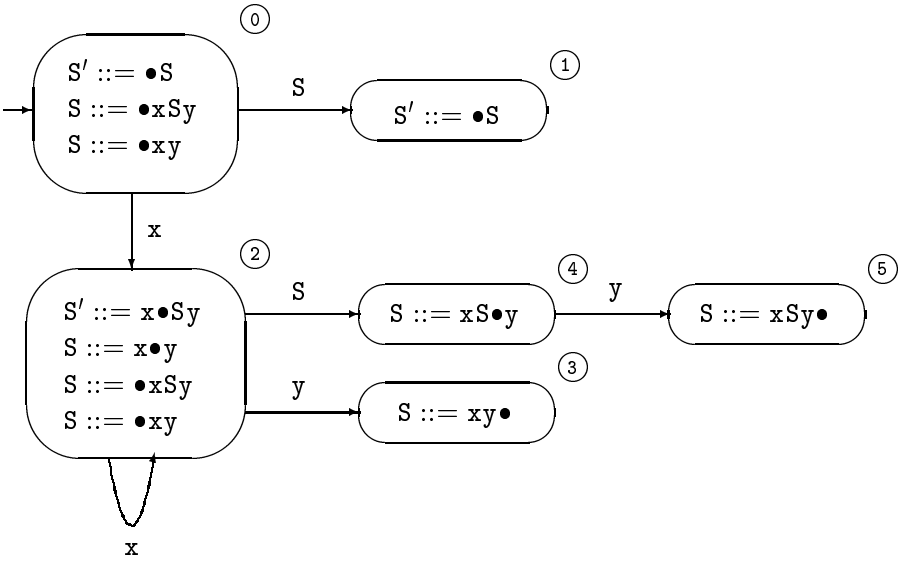
Como ejemplo aplicaremos el método a la misma gramática que vimos antes e iremos comentando cada paso que damos con detenimiento. Como estado inicial del autómata usaremos la clausura del ítem LR(0) $\mathtt{S}' ::= \bullet\mathtt{S}$. Ésta se calcula añadiendo al estado todos los ítems LR(0) iniciales de todas las S-producciones de la gramática, con lo que obtenemos el resultado de la siguiente figura:



Dado que en este estado el punto de los ítems LR(0) está a la izquierda del símbolo S y del símbolo x, son posibles dos transiciones etiquetadas con estos caracteres. La primera nos conduce hacia un estado etiquetado con $\mathtt{S}' ::= \mathtt{S}\bullet$. La segunda es más compleja, puesto que inicialmente nos conduciría hacia un estado etiquetado con los ítems $\{\mathtt{S} ::= \mathtt{x}\bullet\mathtt{Sy}, \mathtt{S} ::= \mathtt{x}\bullet\mathtt{y}\}$, pero debemos darnos cuenta de que en esta situación, el punto ha quedado a la izquierda del símbolo no terminal S, por lo que siguiendo los pasos del algoritmo deberíamos añadir a este estado los ítems LR(0) iniciales de todas las S-producciones. Si lo hacemos se obtiene el resultado de la siguiente figura:



Si continuamos estudiando el estado 2 y todos los que a partir de éste aparezcan, se obtiene el autómata completo para el lenguaje de la pila que aparece en la figura que se muestra a continuación:



A partir de este autómata se puede obtener la expresión regular de las palabras de la pila usando alguno de los algoritmos que estudiamos en un tema anterior. En este caso resulta ser:

$$\alpha = \lambda + \mathtt{S} + \mathtt{x}\mathtt{x}^* + \mathtt{x}\mathtt{x}^*\mathtt{S} + \mathtt{x}\mathtt{x}^*\mathtt{y} + \mathtt{x}\mathtt{x}^*\mathtt{S}\mathtt{y}$$

1.3.6 Construcción de las tablas de análisis a partir del autómata para el lenguaje de la pila

Una vez construido el AFD para el lenguaje de las palabras de la pila, ya tenemos toda la información necesaria para construir la tabla de análisis. Veremos primero la forma de construir la tabla de saltos:

1. Los números de los estados constituirán las marcas de la pila. Cualquier numeración que realicemos será válida.

2. Por cada transición entre los estados `i` y `j` etiquetada con el símbolo `A`, $\mathtt{A} \in \Sigma_N$, asignamos a la entrada $\mathtt{goto}_{\mathtt{i,A}}$ el valor `j`.

La construcción de la tabla de acciones no es difícil, pero involucra un número mayor de pasos. Fíjese con detenimiento en que las casillas de la tabla de saltos tan sólo pueden contener un valor, pero las de la tabla de acciones pueden contener varias acciones al mismo tiempo.

1. Por cada transición entre los estados `i` y `j` etiquetada con el símbolo `a`, $\mathtt{a} \in \Sigma_T$, añadimos a $\mathtt{act}_{\mathtt{i,a}}$, la acción `shift j`.

2. Si en el estado marcado con `i` existe un ítem LR(0) de la forma $\mathtt{A} ::= \alpha\bullet$ distinto a $\mathtt{S}' ::= \mathtt{S}\bullet$, entonces añadimos a cada casilla de la fila `i`, independientemente del carácter de la entrada, la acción `reduce k`, siendo `k` el número de la regla $\mathtt{A} ::= \alpha$ en el conjunto de reglas de producción. Cualquier numeración que asigne a cada regla un número diferente será válida.

3. Si el estado `i` contiene el ítem $\mathtt{S}' ::= \mathtt{S}\bullet$, entonces añadimos a la casilla $\mathtt{act}_{\mathtt{i},\$}$ la acción `accept`.

Es muy importante destacar en la aplicación de este método que en cada paso se añade a las casillas nuevas acciones, por lo que podrían producirse conflictos. Los más habituales son `shift i/reduce j` y `reduce i/reduce j` y nos referiremos a ellos como s/r y r/r respectivamente[1]. Cuando se producen se suele decir que la gramática que estamos estudiando no es LR, pero no se confunda con esto. El autómata LR base puede decidir ante cualquier cadena de entrada si pertenece o no al lenguaje para el que ha sido construido. El problema es que la tabla de análisis LR no es capaz de simular el funcionamiento de dicho autómata de forma completamente determinista y la mayoría de las implementaciones que se han realizado de este tipo de analizadores resuelven los casos de indeterminación de una forma heurística que puede llevar a rechazar palabras que realmente pertenecen al lenguaje. En cualquier caso es habitual utilizar la expresión "esta gramática no es LR" cuando se producen este tipo de conflictos.

A continuación veremos la tabla de la gramática que nos ha servido de ejemplo hasta ahora:

#	**Regla**
1	$\mathtt{S}' ::= \mathtt{S}$
2	$\mathtt{S} ::= \mathtt{xSy}$
3	$\mathtt{S} ::= \mathtt{xy}$

	x	y	$	S′	S
0	shift 2				1
1			accept		
2	shift 2	shift 3			4
3	reduce 3	reduce 3	reduce 3		
4		shift 5			
5	reduce 2	reduce 2	reduce 2		

[1] Es posible encontrar conflictos de reducción múltiples, pero no son en absoluto comunes.

1.3.7 Ejemplo: Las sentencias `if...then...else`

Veremos de qué forma se construye el autómata para reconocer la sentencia `if...then...else` que proporciona la mayoría de los lenguajes de programación. Su estructura sintáctica suele ser la siguiente:

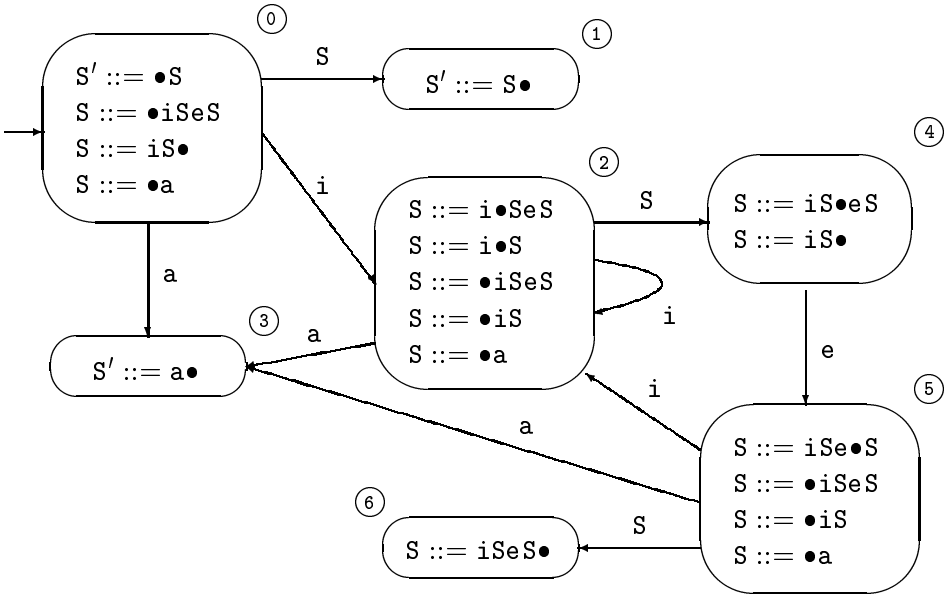
```
sent ::= if exp then sent else sent
     |   if exp then sent
     |   otra_sent
```

De forma reducida, representaremos esta gramática así:

$$\mathtt{G} = (\{\mathtt{i}, \mathtt{e}, \mathtt{a}\}, \{\mathtt{S}', \mathtt{S}\}, \mathtt{S}', \{\mathtt{S} ::= \mathtt{S}, \mathtt{S} ::= \mathtt{iSeS}|\mathtt{iS}|\mathtt{a}\})$$

`i` es un símbolo que engloba `if exp then`, `e` representa la palabra `else` y `a` el resto de las sentencias del lenguaje. Hemos realizado esta simplificación puesto que el autómata para la gramática completa es muy similar al que vamos a obtener para la simplificada, pero en él resulta más difícil apreciar los detalles en los que estamos interesados en este ejercicio práctico.

Lo primero es construir el AFD para el lenguaje de la pila y a partir de él la tabla de análisis que se muestran a continuación:



	i	e	a	$	S′	S
0	shift 2		shift 3			1
1				accept		
2	shift 2		shift 3			4
3	reduce 4	reduce 4	reduce 4	reduce 4		
4	reduce 3	shift 5/reduce 3	reduce 3	reduce 3		
5	shift 2		shift 3			6
6	reduce 2	reduce 2	reduce 2	reduce 2		

#	**Regla**
1	$S' ::= S$
2	$S ::= iSeS$
3	$S ::= iS$
4	$S ::= a$

Es interesante observar que en la casilla $act_{4,e}$ existe un conflicto s/r. Esto significa que cuando en la cima de la pila está la marca 4 y en la entrada el carácter e, el analizador LR no sabe qué hacer. Desde un punto de vista teórico, ante una cierta cadena de entrada, el analizador tan sólo podría rechazar una palabra después de haber intentado todas las opciones posibles, aunque en la práctica esto no se hace nunca. Lo normal es aplicar las siguientes reglas heurísticas:

1. Si $act_{i,a}$ = `shift i/reduce j`, entonces se lleva a cabo la acción `shift i`.

2. Si $act_{i,a}$ a contiene un conflicto `reduce i/reduce j`, se reduce la regla a la que se haya asignado un número más bajo. Esto es, si $i < j$ se reduce la regla número i y en otro caso la regla número j.

Es muy importante tener claro que estas dos reglas son de tipo heurístico, esto es, generalmente conducen a una buena solución, pero no tiene por qué. Si aplicamos la heurística y no logramos reconocer la entrada, antes de no aceptarla tendríamos que explorar todas las alternativas que hemos dejado atrás. De todas formas, por motivos técnicos, casi ninguna implementación de estos analizadores lleva a cabo la vuelta atrás, por lo que puede ocurrir que rechacen algunas palabras que pertenecen al lenguaje. Las gramáticas LR libres de conflictos constituyen un pequeño conjunto de las gramáticas independientes del contexto, por lo que este tipo de analizadores, a los que de ahora en adelante llamaremos LR(0), se utiliza bastante poco.

A continuación analizaremos la traza del autómata guiado por la tabla que acabamos de obtener al enfrentarlo a la cadena de entrada iiaeaea$. Esta cadena se corresponde con la sentencia:

```
if exp then
  if exp then
    sent
  else
    sent
else
    sent
```

El sangrado que hemos utilizado recoge el anidamiento deseado para esta sentencia, pero como veremos más adelante, no siempre corresponde con el que realiza el analizador.

Pila	Entrada	Acción
0	iiaeaea$	shift 2
0i2	iaeaea$	shift 2
0i2i2	aeaea$	shift 3
0i2i2a3	eaea$	reduce S ::= a
0i2i2S4	eaea$	[shift 5] reduce S ::= iS
0i2i2S4e5	aea$	shift 3
0i2i2S4e5a3	ea$	reduce S ::= a
0i2i2S4e5S6	ea$	reduce S ::= iSeS
0i2S4	ea$	[shift 5] reduce S ::= iS
0i2S4e5	a$	shift 2
0i2S4e5a3	$	reduce S ::= iSeS
0S1	$	accept

Si analizamos con detenimiento la traza, podremos observar que al decantarnos por la acción `shift` el analizador ha agrupado cada `else` con la cabecera `if` más próxima, como es habitual en los lenguajes de programación. En cambio, si hubiésemos optado por la acción `reduce` habría realizado una asociación incorrecta que nos habría llevado a rechazar la cadena de entrada, tal y como muestra la siguiente traza:

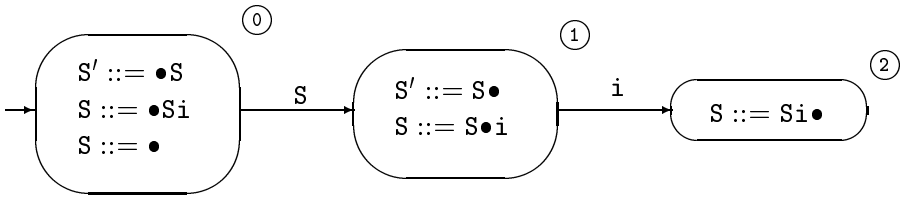
Pila	Entrada	Acción
0	iiaeaea$	shift 2
0i2	iaeaea$	shift 2
0i2i2	aeaea$	shift 3
0i2i2a3	eaea$	reduce S ::= a
0i2i2S4	eaea$	shift 5 [reduce S ::= iS]
0i2S4	eaea$	shift 5 [reduce S ::= iS]
0S1	eaea$	error

1.3.8 Ejemplo: Una lista de elementos

Terminaremos el tema de la construcción de analizadores LR(0) examinando un ejemplo sencillo que se presenta con frecuencia en la práctica. Se trata de obtener un analizador para listas de elementos que pueden estar vacías. La estructura sintáctica es:

```
lista  ::= lista ítem
       |    λ
```

De forma reducida, representaremos esta gramática de la siguiente forma: $\mathtt{G} = (\{\mathtt{i}\}, \{\mathtt{S}', \mathtt{S}\}, \mathtt{S}', \{\mathtt{S}' ::= \mathtt{S}$
A partir de ella se obtiene el autómata de la siguiente figura:



Es muy sencillo y en él lo único destacable es la forma en que se ha tratado la producción $\mathtt{S} ::= \lambda$, que da lugar al ítem LR(0) $\mathtt{S} ::= \bullet$. Esto supone una reducción automática cada vez que en la cima de la pila se encuentre la marca `0`. Siempre que en la gramática existe una λ-producción, se trata de esta forma. La tabla LR(0) se obtiene fácilmente:

#	Regla
1	$\mathtt{S}' ::= \mathtt{S}$
2	$\mathtt{S} ::= \mathtt{Si}$
3	$\mathtt{S} ::= \lambda$

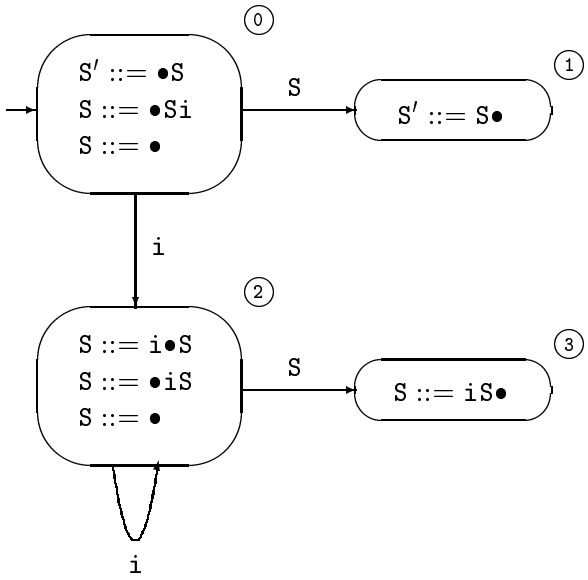
	i	$	S′	S
0	reduce 3	reduce 3		1
1	shift 2	accept		
2	reduce 2	reduce 2		

A continuación veremos la traza del autómata al enfrentarlo a la entrada `iii$`.

Pila	Entrada	Acción
0	iii$	reduce S ::= λ
0S1	iii$	shift 2
0S1i2	ii$	reduce S ::= Si
0S1	ii$	shift 2
0S1i2	i$	reduce S ::= Si
0S1	i$	shift 2
0S1i2	$	reduce S ::= Si
0S1	$	accept

Lo más destacable de esta traza es el hecho de que la pila nunca crece demasiado, pues en el momento en el que se encuentra sobre ella los símbolos `S` e `i` puede realizarse una reducción.

Veremos ahora qué ocurriría si el mismo lenguaje los describiéramos con una gramática recursiva por la derecha: $\mathtt{G}' = (\{\mathtt{i}\}, \{\mathtt{S}', \mathtt{S}\}, \mathtt{S}', \{\mathtt{S}' ::= \mathtt{S}, \mathtt{S} ::= \mathtt{iS} | \lambda\})$. A continuación se recogen el autómata para el lenguaje de la pila, la tabla de análisis y la traza del autómata al enfrentarlo a la misma cadena de entrada de antes.



	i	$	S′	S
0	shift 2/reduce 3	reduce 3		1
1		accept		
2	shift 2/reduce 3	reduce 3		3
3	reduce 2	reduce 2		

Pila	**Entrada**	**Acción**
0	iii$	shift 2
0i2	ii$	shift 2
0i2i2	i$	shift 2
0i2i2i2	$	reduce S ::= λ
0i2i2i2S3	$	reduce S ::= iS
0i2i2S3	$	reduce S ::= iS
0i2S3	$	reduce S ::= iS
0S1	$	accept

Al analizarla con cuidado, vemos que primero se apilan todos los símbolos i que aparezcan en la entrada y sólo al final comienzan a producirse las reducciones. De forma general, cuando el lenguaje viene descrito por una gramática recursiva por la izquierda, el autómata logra reconocer las cadenas de entrada eficientemente sin consumir demasiadas casillas de la pila. Si por el contrario lo describimos usando una gramática recursiva por la derecha, es preciso introducir toda la entrada en la pila antes de llevar a cabo la primera reducción, lo que resulta ineficiente y puede dar lugar a desbordamientos y otros tipos de problemas de implementación.

1.4 Tablas de análisis SLR(1)

El principal problema del método LR(0) son los conflictos s/r, por lo que la primera idea para solucionar estas ambigüedades fue realizar la reducción de una regla como A ::= α

tan sólo cuando el carácter que se encuentra en la cinta de entrada pertenezca al conjunto de los caracteres que pueden aparecer tras el símbolo `A`. Esto es, cuando el carácter de la entrada pertenece al conjunto `siguiente(A)`. Este es el método SLR(1), puesto que estamos tomando un carácter de lectura adelantada para decidir si llevar a cabo la reducción o no, pero es posible tomar más símbolos de lectura adelantada para construir la tabla.

Por lo tanto, la tabla SLR(1) se construye como la LR(0) y la única diferencia está en el segundo paso de la construcción de la tabla de acciones:

1. Ídem.
2. Si en el estado `i` existe un ítem de la forma $\mathtt{A} ::= \alpha\bullet$ distinto a $\mathtt{S}' ::= \mathtt{S}\bullet$, entonces para cualquier $\mathtt{a} \in \mathtt{siguiente(A)}$ añadimos a la casilla $\mathtt{act_{i,a}}$ la acción `reduce k`, siendo `k` el número de la regla $\mathtt{A} ::= \alpha$ en el conjunto de reglas de producción.
3. Ídem.

Como ejemplo examinaremos la gramática para la sentencia `if...then...else` que estudiábamos en la sección 1.3.7. En este caso, $\mathtt{siguiente(S')} = \{\$\}$ y $\mathtt{siguiente(S)} = \{\$, \mathtt{e}\}$ por lo que se obtiene la siguiente tabla SLR(1):

	i	e	a	$	S'	S
0	shift 2		shift 3			1
1				accept		
2	shift 2		shift 3			4
3		reduce 4		reduce 4		
4		shift 5/reduce 3		reduce 3		
5	shift 2		shift 3			6
6		reduce 2		reduce 2		

#	**Regla**
1	$\mathtt{S}' ::= \mathtt{S}$
2	$\mathtt{S} ::= \mathtt{iSeS}$
3	$\mathtt{S} ::= \mathtt{iS}$
4	$\mathtt{S} ::= \mathtt{a}$

Como se puede apreciar, en esta tabla existen más situaciones de error que en el caso LR(0). Por desgracia, no hemos podido solucionar el conflicto s/r, y la verdad es que en la mayoría de los casos el método SLR(1) no nos resuelve el problema.

Por lo tanto, no estudiaremos con más detalle estos analizadores y pasaremos a unos mucho más potentes que permiten reconocer de forma determinista la mayoría de los lenguajes de programación actuales.

1.5 Tablas de análisis LR(1)

La idea de este método es deshacer las ambigüedades utilizando para ello un símbolo de lectura adelantada o *lookahead* que usado de forma inteligente suele resolver muchos

más conflictos que el método SLR(1). En cualquier caso, como veremos en los ejemplos, tampoco es un método que cubra todas las gramáticas independientes del contexto sin la aparición de conflictos.

1.5.1 Ítems LR(1)

Un ítem LR(1) es un ítem LR(0) ampliado al que hemos añadido un carácter de lectura adelantada. Se representan como $\mathtt{A} ::= \alpha \bullet \beta, \mathtt{a}$, en donde $\mathtt{a}$ representa cualquier símbolo terminal de la gramática o el indicador de fin de cadena \$.

Son de especial interés los ítems LR(1) que se corresponden con reducciones, es decir, aquellos de la forma $\mathtt{A} ::= \alpha\bullet, \mathtt{a}$. Dado un estado con este ítem, la reducción de la cadena α por el no terminal $\mathtt{A}$ se llevará a cabo tan sólo en aquellos casos en que el carácter de entrada sea $\mathtt{a}$.

1.5.2 Construcción del autómata basado en ítems LR(1) para el lenguaje de la pila

El método de construcción es muy similar al que vimos para ítems LR(0), pues tan sólo se diferencia en en el cálculo de los caracteres de lectura adelantada. Los pasos del método son los siguientes:

1. Crear un estado inicial con el ítem LR(1) $\mathtt{S'} ::= \bullet\mathtt{S}, \$$.
2. Repetir los pasos siguientes hasta que sea imposible aplicarlos más veces.
3. En cada estado en el que exista una regla de la forma $\mathtt{A} ::= \alpha \bullet \mathtt{B}\beta, \mathtt{c}$, siendo todas las $\mathtt{B}$-producciones de la forma $\mathtt{B} ::= \gamma_1|\gamma_2|\ldots|\gamma_\mathtt{n}$, añadir a ese estado los ítems LR(1) siguientes:

   $\mathtt{B} ::= \bullet\gamma_1, \mathtt{a} \quad \forall \mathtt{a} \in \mathtt{primero}(\beta\mathtt{c})$
   $\mathtt{B} ::= \bullet\gamma_2, \mathtt{a} \quad \forall \mathtt{a} \in \mathtt{primero}(\beta\mathtt{c})$
   $\ldots \qquad\qquad \ldots$
   $\mathtt{B} ::= \bullet\gamma_\mathtt{n}, \mathtt{a} \quad \forall \mathtt{a} \in \mathtt{primero}(\beta\mathtt{c})$
4. De cada estado etiquetado con un ítem de la forma $\mathtt{A} ::= \alpha\bullet\mathtt{x}\beta, \mathtt{c}$ $(\mathtt{x} \in \Sigma_\mathtt{T} \cup \Sigma_\mathtt{N})$, añadir una transición etiquetada con el símbolo $\mathtt{x}$ a un estado etiquetado con el ítem $\mathtt{A} ::= \alpha\mathtt{x} \bullet \beta, \mathtt{c}$.

1.5.3 Construcción de la tabla de análisis

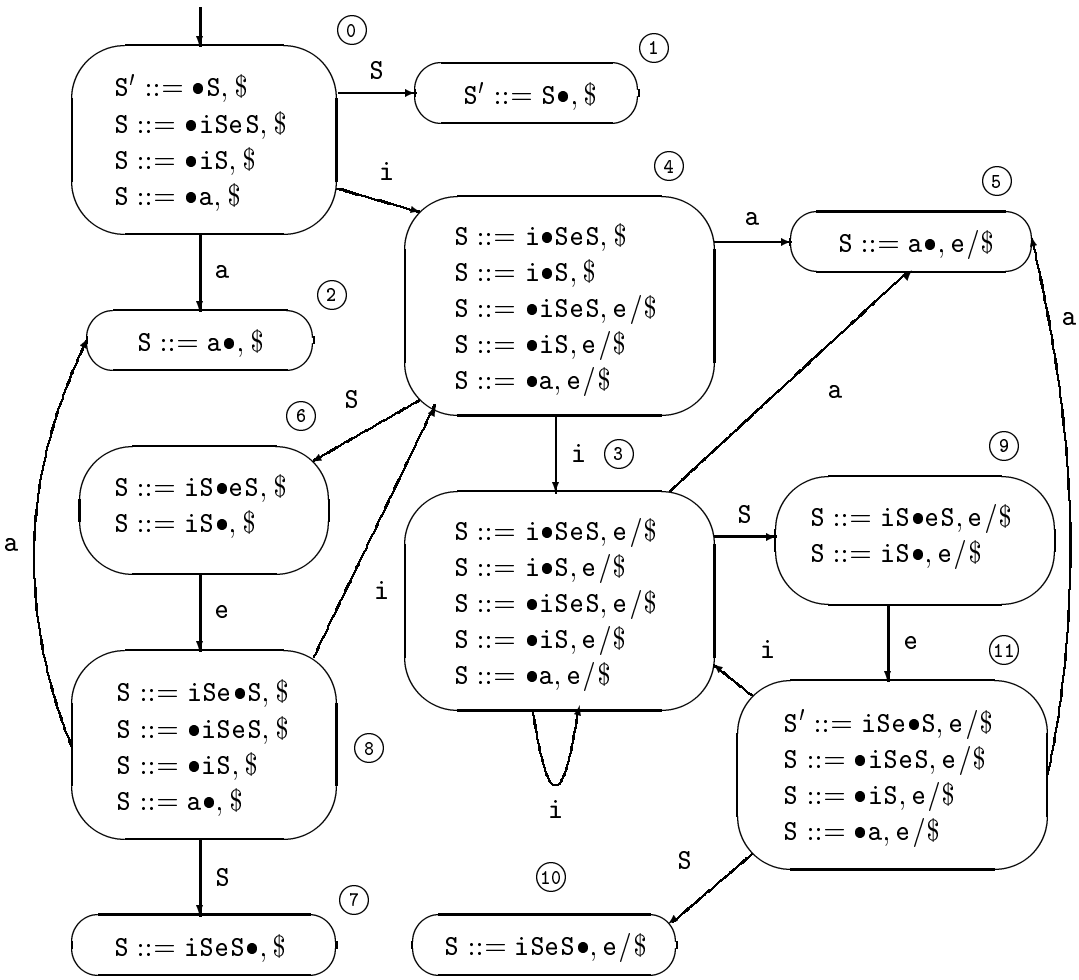
A partir del autómata obtenido podemos construir la tabla LR(1). El método es muy parecido al de las tablas LR(0), y tan sólo se diferencia en el segundo paso de la construcción de la tabla de acciones.

1. Ídem.
2. Si en el estado $\mathtt{i}$ existe un ítem LR(1) de la forma $\mathtt{A} ::= \alpha\bullet, \mathtt{c}$ distinto a $\mathtt{S'} ::= \mathtt{S}\bullet, \mathtt{c}$, entonces añadimos a la entrada $\mathtt{act}_{\mathtt{i},\mathtt{c}}$ la acción `reduce k`, siendo $\mathtt{k}$ el número de la regla $\mathtt{A} ::= \alpha$ en la gramática.

3. Ídem.

1.5.4 Ejemplo: Las sentencias `if...then...else`

Revisaremos ahora el mismo ejemplo de la sección 1.3.7. El autómata que se obtiene para el lenguaje de la pila se muestra en la siguiente figura. En él, la notación $\mathtt{A} ::= \alpha \bullet \beta, \mathtt{a_1}/\mathtt{a_2}/\ldots/\mathtt{a_n}$ se considera una forma compacta de escribir $\{\mathtt{A} ::= \alpha \bullet \beta, \mathtt{a_1}, \mathtt{A} ::= \alpha \bullet \beta, \mathtt{a_2}, \ldots, \mathtt{A} ::= \alpha \bullet \beta, \mathtt{a_n}\}$. Además, como podremos comprobar, sigue apareciendo un conflicto s/r en el estado 9 que ha resultado imposible de eliminar por el método LR(1).



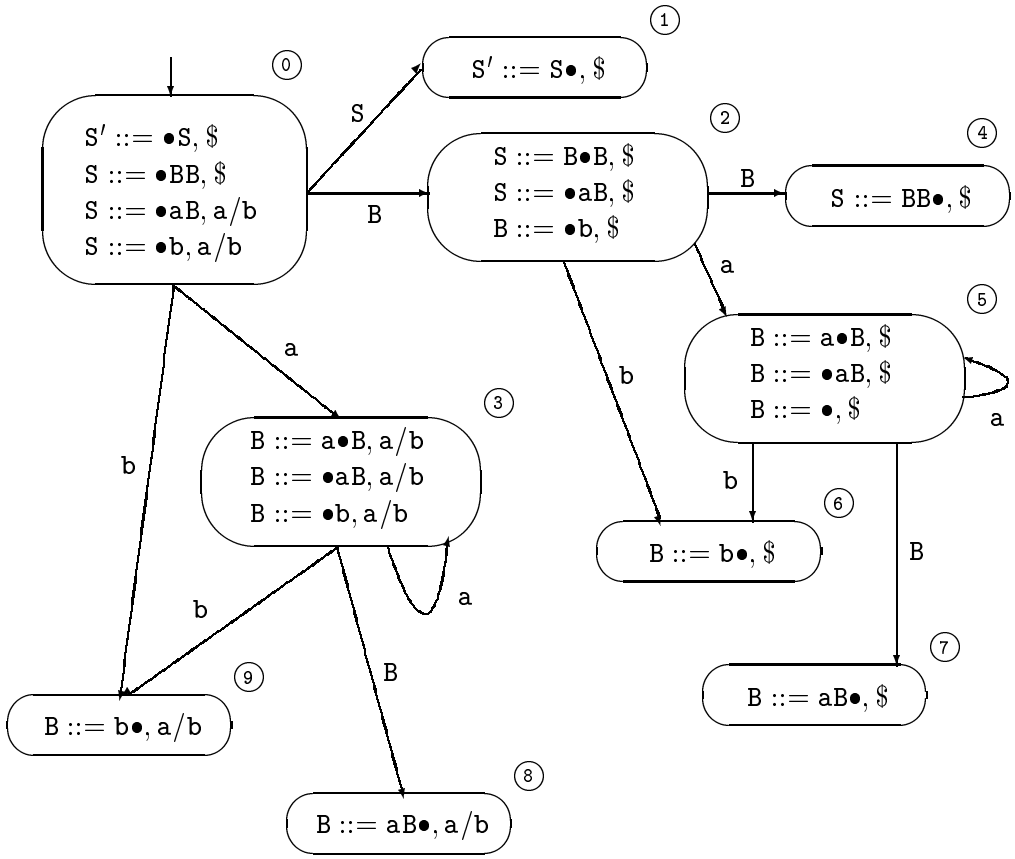
	i	e	a	$	S′	S
0	shift 4		shift 2			1
1				accept		
2				reduce 4		
3	shift 3		shift 5			9
4	shift 3		shift 5			6
5		reduce 4		reduce 4		
6		shift 8		reduce 3		
7				reduce 2		
8	shift 4		shift 2			7
9		shift 11/reduce 3		reduce 3		
10		reduce 2		reduce 2		
11	shift 3		shift 5			10

#	**Regla**
1	S′ ::= S
2	S ::= iSeS
3	S ::= iS
4	S ::= a

1.5.5 Ejemplo: Concatenación de dos listas

En esta sección examinaresmos otro ejemplo sencillo en el construiremos la tabla de análisis para una gramática que describe concatenaciones de dos listas de elementos: $\mathtt{G} = (\mathtt{b}\}, \{\mathtt{S}', \mathtt{S}, \mathtt{B}\}, \mathtt{S}', \{\mathtt{S}' ::= \mathtt{S}, \mathtt{S} ::= \mathtt{BB}, \mathtt{B} ::= \mathtt{aB}|\mathtt{b}\})$.

Lo primero es calcular los conjuntos primero para todos los símbolos no terminales de la gramática, pues serán necesarios para el cálculo de los *lookaheads*: $\mathtt{siguiente}(\mathtt{S}') = \{\$\}$, $\mathtt{siguiente}(\mathtt{S}) = \mathtt{siguiente}(\mathtt{B}) = \{\mathtt{a}, \mathtt{b}\}$. Con esta información se obtiene el autómata y la tabla de análisis siguientes:



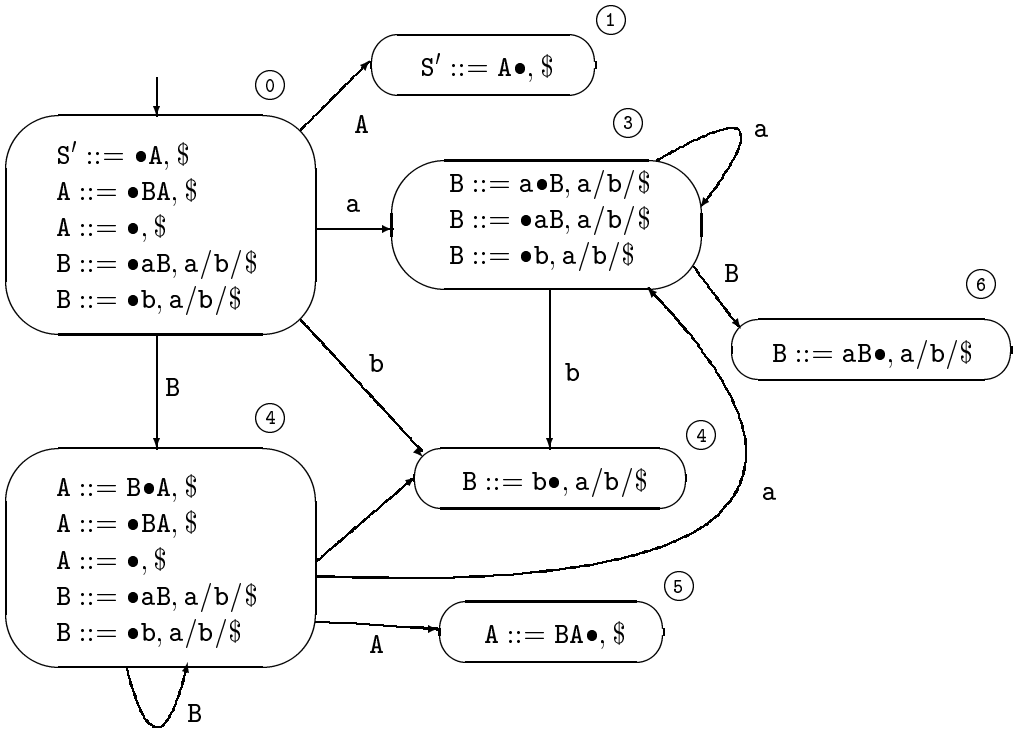
#	Regla
1	S′ ::= S
2	S ::= BB
3	B ::= aB
4	B ::= b

	a	b	$	S′	S	B
0	shift 3	shift 9			1	2
1			accept			
2	shift 5	shift 6				4
3	shift 3	shift 9				8
4			reduce 2			
5	shift 5	shift 6				7
6			reduce 4			
7			reduce 3			
8	reduce 3	reduce 3				
9	redeuce 4	reduce 4				

1.5.6 Ejemplo: Concatenación de múltiples listas

Terminaremos esta sección con un último ejemplo en el que mostramos cómo se crea un reconocedor LR(1) para una gramática que incluye λ-producciones. Se trata de $\mathtt{G} = (\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{S}', \mathtt{A}, \mathtt{B}\}, \mathtt{S}',$

Si seguimos los pasos de método se obtiene el autómata y la tabla de análisis que se muestran a continuación:



#	Regla
1	S′ ::= A
2	A ::= BA
3	A ::= λ
4	B ::= aB
5	B ::= b

	a	b	$	S′	A	B
0	shift 3	shift 4	reduce 3		1	2
1			accept			
2	shift 3	shift 4	reduce 3		5	2
3	shift 3	shift 4				6
4	reduce 5	reduce 5	reduce 5			
5			reduce 2			
6	reduce 4	reduce 4	reduce 4			

A continuación estudiaremos la traza del autómata al analizar la cadena de entrada aabb$.

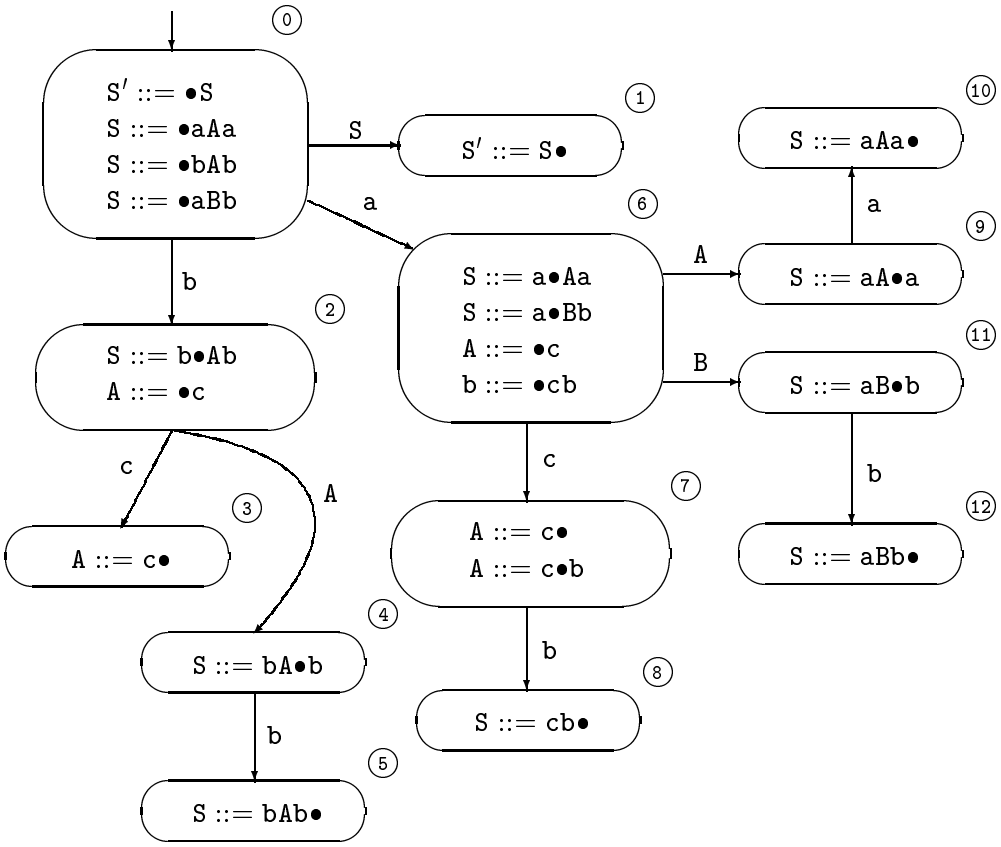
Pila	Entrada	Acción
0	aabb$	shift 3
0a3	abb$	shift 3
0a3a3	bb$	shift 4
0a3a3b4	b$	reduce B := b
0a3a3B6	b$	reduce B ::= aB
0a3B6	b$	reduce B ::= aB
0B2	b$	shift 4
0B2b4	$	reduce B ::= b
0B2B2	$	reduce A ::= λ
0B2B2A5	$	reduce A ::= BA
0B2A5	$	reduce A ::= BA
0A1	$	accept

1.5.7 ¿Por qué el símbolo de lectura adelantada del método LR(1) resuelve más conflictos que el del método SLR(1)?

Anteriormente hemos comentado que el método de análisis SLR(1) prácticamente no se utiliza puesto que en las gramáticas habituales no suele eliminar demasiados conflictos s/r o r/r. En cambio hemos comentado que el método LR(1), o algunas de las variantes que estudiaremos en breve, es muy utilizado puesto que en las situaciones reales suele eliminar muchos conflictos que aparecen en las tablas LR(0).

Lo sorprendente es que ambos se basan en llevar un símbolo de lectura adelantada que nos permite determinar en qué situaciones es realmente necesario reducir alguna regla sintáctica. La diferencia está en la forma en que se utiliza ese caracter de lectura adelantada. En el método SLR(1) una regla como $\mathtt{A} ::= \alpha$ se reduce tan sólo cuando el caracter de lectura adelantada es uno de los caracteres que pueden aparecer a continuación del símbolo $\mathtt{A}$ en la gramática. En otros casos, no es necesario llevar a cabo la reducción puesto que de hacerlo llegaríamos a un estado para el que no hay ninguna transición definida con el caracter de lectura adelantada, puesto que de otra manera, ese caracter aparecería en el conjunto $\mathtt{siguiente}(\mathtt{A})$. El problema es que el símbolo $\mathtt{A}$ puede aparecer en la parte derecha de varias reglas, por ejemplo, $\mathtt{B} ::= \gamma_1\mathtt{A}\gamma_2|\gamma_3\mathtt{A}\gamma_4$ y el método SLR(1) no distingue realmente cuándo estamos reduciendo $\mathtt{A}$ para incorporarlo en la reducción de $\gamma_1\mathtt{A}\gamma_2$ o en la de $\gamma_3\mathtt{A}\gamma_4$. Por el contrario, el método LR(1) sí que se distingue mediante el uso del caracter de lectura adelantada que se añade a cada uno de los ítems LR(1). Dicho en otras palabras, el método SLR(1) maneja información global, mientras que el método LR(1) maneja información local a cada una de las reglas. Evidentemente este es uno de los problemas fundamentales del método LR(1), puesto que manejar información acerca de cada regla incrementa notablemente el número de entradas de la tabla de análisis, tal y como hemos podido comprobar.

Para ilustrar de una forma más gráfica lo que acabamos de decir vamos a estudiar la gramática $\mathtt{G} = (\{\mathtt{a},\mathtt{b},\mathtt{c}\},\{\mathtt{S}',\mathtt{S},\mathtt{A},\mathtt{B}\},\mathtt{S}',\mathtt{P})$ en la que el conjunto de reglas es $\mathtt{P} = \{\mathtt{S} ::= \mathtt{aAa}|\mathtt{bAb}|\mathtt{aBb},\mathtt{A}$ y presenta un conflicto s/r cuando intentamos reconocer su lenguaje utilizando el método SLR(1). Si aplicamos los pasos de este método se obtiene el siguiente autómata para el lenguaje de la pila de análisis:



En esta gramática se verifica que $\texttt{siguiente}(\texttt{S}) = \{\$\}$, $\texttt{siguiente}(\texttt{A}) = \{\texttt{a}, \texttt{b}\}$ y $\texttt{siguiente}(\texttt{B}) = \{\texttt{b}\}$, por lo que si construímos la tabla de análisis SLR(1) encontraremos un conflicto en el estado 7:

	a	b	c	$	S′	S	A	B
0	shift 6	shift 2				1		
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
7	reduce 4	shift 8/reduce 4						
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
12				reduce 13				

#	**Regla**
1	S ::= aAa
2	S ::= bAb
3	S ::= aBb
4	A ::= c
5	B ::= cb

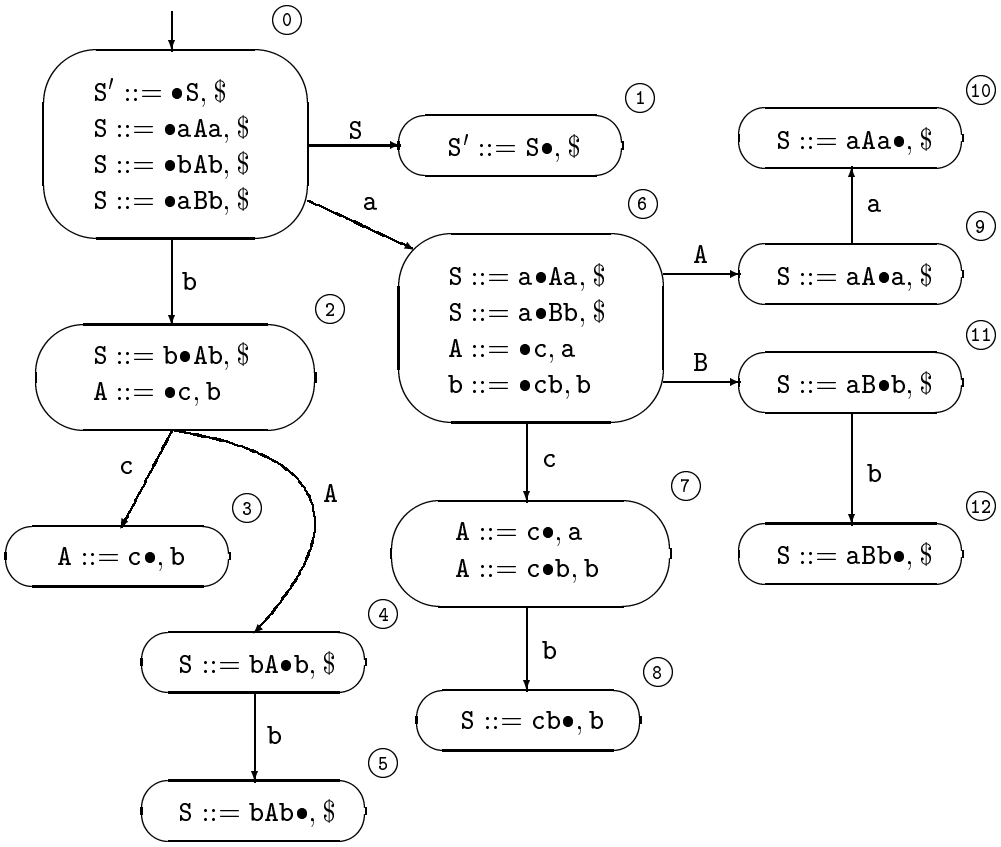
El método SLR(1) ha eliminado eliminado respecto al método LR(0) las acciones `reduce` de las casillas $\texttt{act}_{7,\texttt{c}}$ y $\texttt{act}_{7,\$}$, pues ninguno de estos símbolos puede aparecer detrás del símbolo `A` en esta gramática. Pero no ha tenido en cuenta que cuando nos encontramos en el estado 7 tenemos sobre la cima de la pila una `a` y una `c`, por lo que la

reducción de la regla $\mathtt{A} ::= \mathtt{c}$ tan sólo puede llevarse a cabo en el caso en que el siguiente caracter de la entrada sea una `a`. En caso de que sea una `b` debemos apilarlo para así poder completar la regla $\mathtt{B} ::= \mathtt{cb}$.

El método SLR(1) no ha sido capaz de resolver el conflicto puesto que ha usado información global acerca de cuáles son los símbolos que pueden aparecer detrás de `A` en toda la gramática, cuando realmente lo más adecuado sería utilizar información local a la regla que estamos intentado reducir en el estado en el que se ha producido el conflicto. Esto es precisamente lo que hace el método LR(1) al ir calculando el caracter de lectura adelantada en cada uno de los ítems LR(1). Sabemos que si en un estado aparece un ítem de la forma $\mathtt{A} ::= \alpha \bullet \mathtt{B} \beta, \mathtt{c}$, entonces debemos crear una transición hacia un estado etiquetado con ítems de la forma $\mathtt{B} ::= \bullet \gamma_{\mathtt{i}}, \mathtt{a}$ para todo caracter $\mathtt{a} \in \mathtt{primero}(\beta \mathtt{c})$. De esta forma, los caracteres de lectura adelantada indican siempre cuáles son los caracteres que pueden aparecer detrás del símbolo `B` en la regla en la que pretendemos integrarlo cuando reduzcamos en la pila alguna de las cadenas $\gamma_{\mathtt{i}}$.

En la siguiente figura aparece el autóma para el lenguaje de la pila utilizando para su construcción ítems LR(1). En él podemos ver cómo ahora en el estado `7` la reducción tan sólo se lleva a cabo cuando el caracter de lectura adelantada es `a`. En cambio en el estado `3` la reducción tan sólo se lleva a cabo si este caracter es la `b`. Dicho en otras palabras, el método LR(1) es capaz de distinguir cada vez que en la pila se reduce la cadena `c` por `A` cuál es la regla de producción de la gramática en la que se va a integrar este símbolo.

El problema de mantener esta información sobre los símbolos que localmente pueden aparecer a continaución de un no terminal en cada regla de la gramática es que el número de filas de la tabla de análisis puede crecer considerablemente con respecto al método SLR(1) o el LR(0). En la sección 1.7 estudiaremos un tipo especial de tablas, conocidas como LALR(1), que permiten subsanar este problema.



1.6 Tablas de análisis LR(k), $k \geq 2$

Una vez hemos estudiado en profundidad varios métodos para construir tablas de análisis con cero o un caracter de lectura adelantada, es el momento de conocer tablas que utilizan más de un caracter de lectura adelantada para disminuir el número de casillas en las que aparecen acciones `reduce`.

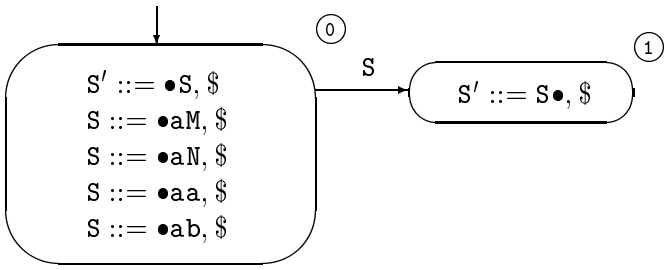
La única diferencia del método LR(k), $k \geq 2$, con respecto al método LR(1) es que en vez de tomar un caracter de lectura adelantada se toman entre 2 y k símbolos. Esto significa que los ítems LR(k) son de la forma $\mathtt{A} ::= \alpha \bullet \beta, \mathtt{s}$, en donde $\mathtt{s}$ es una cadena de entre 1 y k caracteres posiblemente terminada en un \$. Evidentemente, si se llega a un estado marcado con un ítem LR(k) de la forma $\mathtt{A} ::= \alpha\beta\bullet, \mathtt{s}$, la reducción de la cadena $\alpha\beta$ tan sólo se llevará a cabo si los siguientes caracteres de entrada coinciden con los de la cadena $\mathtt{s}$.

Para construir el autómata LR(k) para el lenguaje de la pila tan sólo debemos tener en cuenta que si un estado está marcado con el ítem LR(2) $\mathtt{A} ::= \alpha\bullet\mathtt{B}\beta, \mathtt{s}$ y todas las $\mathtt{B}$–producciones son de la forma $\mathtt{B} ::= \gamma_1|\gamma_2|\ldots|\gamma_\mathtt{n}$, entonces debemos introducir a partir de él una transición etiquetada con $\mathtt{B}$ hacia el estado etiquetado con $\mathtt{B} ::= \bullet\gamma_\mathtt{i}, \mathtt{r}$ para toda cadena $\mathtt{r} \in \mathtt{primero_k}(\beta\mathtt{s})$. Tal y como comentabamos en la sección 1.5.7 esta forma de calcular la cadena de lectura adelantada permite obtener información local acerca de cuales son los caracteres que siguen a $\mathtt{B}$ en la producción que corresponde al ítem $\mathtt{B} ::= \bullet\gamma_\mathtt{i}, \mathtt{r}$.

A modo de ejemplo estudiaremos la tabla LR(2) para la siguiente gramática:

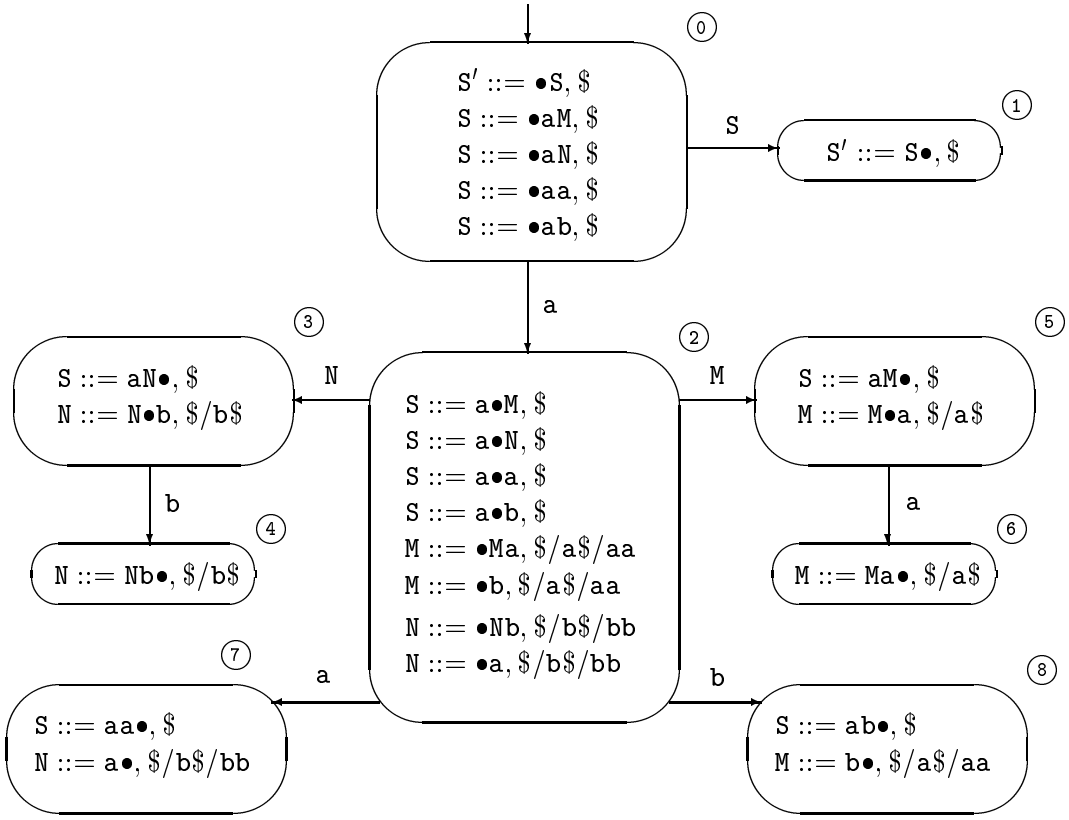
$$(\{\mathtt{a},\mathtt{b}\},\{\mathtt{S}',\mathtt{S},\mathtt{M},\mathtt{N}\},\mathtt{S}',\{\mathtt{S}' ::= \mathtt{S},\mathtt{S} ::= \mathtt{aM}|\mathtt{aN}|\mathtt{aa}|\mathtt{ab},\mathtt{M} ::= \mathtt{Ma}|\mathtt{b},\mathtt{N} ::= \mathtt{Nb}|\mathtt{a}\})$$

Comenzaremos por obtener el autómata LR(2) para el lenguaje de la pila de análisis. Como en el caso de los autómatas LR(1), el estado inicial está etiquetado con la clausura del ítem LR(2) $\mathtt{S}' ::= \bullet\mathtt{S}, \$$. Fíjese en que el ítem inicial es siempre el mismo, con independencia al número de caracteres de lectura adelantada que vayamos a utilizar. La razón es que una cadena de entrada debe aceptarse tan sólo cuando en la cima de la pila se logra reducir el axioma de la gramática y se han consumido todos los caracteres de la entrada. La siguiente figura muestra los dos primeros estados del autómata:



A partir de este estado, ya que el punto queda a la izquierda del símbolo terminal a, es posible una transición con este símbolo hacia un estado etiquetado con la clausura de $\mathtt{I} = \{\mathtt{S} ::= \mathtt{a}\bullet\mathtt{M}, \$, \mathtt{S} ::= \mathtt{a}\bullet\mathtt{N}, \$, \mathtt{S} ::= \mathtt{a}\bullet\mathtt{a}, \$, \mathtt{S} ::= \mathtt{a}\bullet\mathtt{b}, \$\}$. El cálculo de esta clausura puede ser complicado, por lo que nos detendremos un instante en él. Dado que el punto ha quedado a la izquierda del símbolo no terminal M es necesario añadir a la clausura de I los ítems LR(2) iniciales de todas las M–producciones usando como cadena de lectura adelantada cualquier $\mathtt{r} \in \mathtt{primero}_2(\lambda\$) = \{\$\}$. Esto es, debemos añadir $\mathtt{M} ::= \bullet\mathtt{Ma}, \$$ y $\mathtt{M} ::= \bullet\mathtt{b}, \$$. Pero fíjese en que al añadir estos ítems, hemos introducido uno en el que el punto vuelve a quedar a la izquierda del símbolo no terminal M. Por lo tanto debemos añadir a la clausura los ítems LR(2) iniciales de todas las M–producciones usando como cadena de lectura adelantada cualquier $\mathtt{r} \in \mathtt{primero}_2(\mathtt{a}\$) = \{\mathtt{a}\$\}$, esto es, $\mathtt{M} ::= \bullet\mathtt{Ma}, \mathtt{a}\$$ y $\mathtt{M} ::= \bullet\mathtt{b}, \mathtt{a}\$$. Pero fíjese en que ahora hemos incluido un nuevo ítem con el punto a la izquierda del símbolo M, por lo que es necesario añadir de nuevo los ítems LR(2) iniciales de todas las M–producciones usando como cadena de lectura adelantada cualquier $\mathtt{r} \in \mathtt{primero}_2(\mathtt{aa}\$) = \{\mathtt{aa}\}$, esto es, $\mathtt{M} ::= \bullet\mathtt{Ma}, \mathtt{aa}$ y $\mathtt{M} ::= \bullet\mathtt{b}, \mathtt{aa}$. De nuevo vuelve a quedar el punto a la izquierda de M por lo que tenemos que incluir de nuevo todos los ítems LR(2) iniciales de todas las M–producciones utilizando como cadena de lectura adelantada cualquier $\mathtt{r} \in \mathtt{primero}_2(\mathtt{aaa}\$) = \{\mathtt{aa}\}$. Por lo tanto en esta ocasión no es preciso añadir ningún nuevo ítem para las M–producciones.

Si repetimos el mismo proceso con el ítem $\mathtt{S} ::= \mathtt{a}\bullet\mathtt{N}, \$$ se acaba obteniendo el autómata que aparece en la siguiente figura:



A partir de él es fácil obtener la tabla de análisis LR(2) para la gramática que estamos estudiando. La única consideración de interés es que las columnas de la zona de acción se etiquetan ahora con cadenas de dos caracteres, tal y como se muestra a continuación:

	a$	aa	ab	b$	ba	bb	$	S′	S	M	N
0	sh 2	sh 2	sh 2						1		
1							accept				
2	sh 7	sh 7	sh 7	sh 8	sh 8	sh 8	sh 8			5	3
3				sh 3	sh 3	sh 3	rd 3				
4				rd 8			rd 8				
5	sh 6	sh 6	sh 6				rd 2				
6	rd 6						rd 6				
7				rd 9		rd 9	rd 4/rd 9				
8	rd 7	rd 7					rd 5/rd 7				

#	**Regla**	#	**Regla**	#	**Regla**
1	S′ ::= S	2	S ::= aM	3	S ::= aN
4	A := aa	5	A ::= ab	6	M ::= Ma
7	M ::= b	8	N ::= Nb	9	N ::= a

Tal y como podemos apreciar en los estados marcados como 7 y 8 han aparecido dos conflictos r/r, que si bien no son habituales, si que son posibles tal y como acabamos de comprobar. El problema es que cuando en la cima de la pila aparece la marca 7 debajo se encuentra la cadena aa, y en esa situación, si en la entrada aparece el terminador de

cadenas \$, la tabla no puede decidir si debe reducirse la regla `S` ::= `aa` o la regla `N` ::= `a`. En el estado `8` se produce un problema similar pero en este caso con las reglas `S` ::= `ab` y `M` ::= `b`.

Pudiera parecer que si la tabla de análisis LR(k) para una gramática contiene conflictos, la solución es elegir un número mayor de caracteres de lectura adelantada. Pero esto no es cierto y la gramática que hemos presentado es un ejemplo bastante claro. La razón es que en el estado `2` siempre aparecerán los items LR(k) `S` ::= `a`•`a`, \$, `N` ::= •`a`, \$, `S` ::= `a`•`b`, \$, `M` ::= •`b`, \$, por lo que siempre serán posibles dos transiciones etiquetadas con `a` y `b` hacia estados conflictivos.

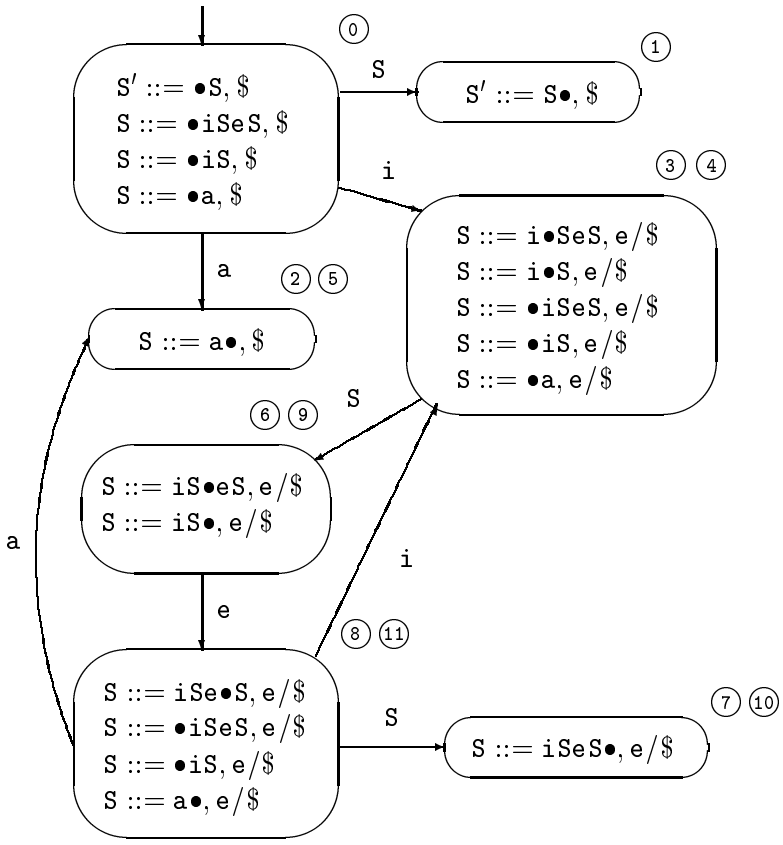
La mayor parte de los lenguajes de programación tienen gramáticas que se pueden reconocer de forma absolutamente determinista utilizando tablas de análisis LR(1), por lo que en la práctica los analizadores LR(k), $k \geq 2$, no suelen utilizarse.

1.7 Tablas de análisis LALR(1)

Como hemos visto en uno de los ejemplos, los autómatas LR(1) tampoco son capaces de reconocer los lenguajes descritos por todas las gramáticas independientes del contexto. Pero su mayor inconveniente es el gran número de marcas que necesitan para la pila, lo que da lugar a tablas de análisis muy grandes.

Las tablas de análisis LALR(1) se obtienen a partir de las tablas LR(1) mezclando todos aquellos estados que tienen el mismo núcleo o conjunto de ítems LR(0). Lo que realmente se mezcla son los números de los estados y los caracteres de lectura adelantada para aquellos ítems LR(0) que coinciden. De esta forma, para una gramática dada, la tabla final tiene el mismo tamaño que la tabla SLR(1) pero aún es capaz de eliminar muchos conflictos puesto que conserva los caracteres de lectura adelantada de las LR(1).

Como ejemplo podemos examinar el mismo de la sección 1.5.4. En él construímos la tabla de análisis LR(1) para la gramática `if...then...else`. Si nos fijamos en el autómata veremos que podemos unir los estados `2` y `5`, `3` y `4`, `6` y `9`, `7` y `10` y el `8` y el `11` puesto que todos ellos contienen el mismo núcleo. Si lo hacemos se obtiene el autómata LALR(1) que se muestra en la siguiente figura:



A partir de este autómata se puede obtener con gran facilidad la tabla de análisis LALR(1), por lo que no la mostraremos. En cualquier caso es interesante señalar que sigue presentando un conflicto s/r en el estado combinado 6–9.

El método LALR(1) es en la actualidad el más empleado en los generadores de analizadores sintacticos que se encuentran disponibles en el mercado, puesto que combina la eficiencia con una potencia intermedia entre las tablas de análisis SLR(1) y LR(1). No piense en ningún momento que si una tabla de análisis LR(1) no presenta conflictos entonces su tabla LALR(1) tampoco los presenta. Para convencerse de ello le dejamos como ejercicio que obtenga las tablas de anásis SLR(1), LR(1) y LALR(1) de la siguiente gramática:

$$(\{\mathtt{a},\mathtt{b},\mathtt{c}\},\{\mathtt{S}',\mathtt{S},\mathtt{A},\mathtt{B}\},\mathtt{S}',\{\mathtt{S}' ::= \mathtt{S},\mathtt{S} ::= \mathtt{aAa}|\mathtt{aBb}|\mathtt{bAb}|\mathtt{bBa},\mathtt{A} ::= \mathtt{c},\mathtt{B} ::= \mathtt{c}\})$$

Obtendrá que la gramática es LR(1), pero ni SLR(1) ni tampoco LALR(1). La razón es que al mezclar estados, estamos mezclando ítems terminales que dan lugar a acciones de reducción que en la tabla LR(1) no son conflictivas, pero sí en los nuevos estados combinados.

1.8 Bibliografía

El método de análisis LR fue introducido por Donald E. Knuth en el trabajo [Knuth 1965]. El principal problema es que las tablas de análisis LR(0) no son adecuadas para muchos

lenguajes de programación reales y las LR(1) suelen ser demasiado grandes. [DeRemer 1969] introdujo el método SLR(k) para aumentar la potencia de los analizadores LR y sugirió la posibilidad de mezclar algunas filas de las tablas LR para reducir su tamaño. En cualquier caso, [Korenjak 1969] fue el primero que logró reducir el tamaño de las tablas de análisis introduciendo el método LALR. Bastante tiempo después, en [DeRemer y Pennello 1982] se presentó un algoritmo muy eficiente para calcular los conjuntos de caracteres de lectura adelantada de los analizadores LALR(1) que se usa muchísimo en los generadores de analizadores sintácticos actuales.

Para completar sus conocimientos de análisis LR le recomendamos la lectura de [Aho y Ullman 197 que incluye extensísimos capítulos dedicados a la teoría de análisis sintáctico, y [Aho *et al.* 1986] y [Tremblay y Sorenson 1985] que describen métodos eficientes de almacenamiento de las tablas de análisis.

Capítulo 2

Tratamiento de errores en analizadores LR

Algún autor ha indicado que, atendiendo a su uso habitual, un compilador se debería definir como una herramienta informática cuyo objetivo es encontrar errores. Y es cierto, pues la mayor parte de las veces los compiladores deben enfrentarse a textos de entrada que son incorrectos y ante los cuales deben comportarse de una manera predecible y razonable.

En la actualidad se conocen multitud de métodos de análisis sintáctico y cada uno de ellos tiene características que lo hacen único, por lo que hablar de corrección o recuperación de errores sintácticos desde un punto de vista muy general resulta prácticamente imposible. En este capítulo aprenderemos a construir analizadores LR robustos que tratan adecuadamente los textos con errores.

En el capítulo 1 estudiamos con detalle la técnica de reconocimiento LR(k). En este tipo de analizadores los errores sintácticos se detectan cuando a partir del estado que actualmente se encuentra en la cima de la pila no existe ninguna transición definida con el carácter actual de la entrada. En estas situaciones el contenido de la pila representa el contexto a la izquierda del error y el resto de la cadena de entrada el contexto a su derecha. La recuperación de errores se lleva a cabo modificando la pila y/o la cadena de entrada de forma que el analizador sintáctico pueda llegar a un estado adecuado en el que continuar de una forma segura el análisis de la cadena de entrada.

2.1 Conceptos generales

Como hemos indicado en la introducción, el tratamiento de los errores sintácticos debe estar muy cuidado en cualquier compilador, puesto que la mayoría de las veces estas herramientas se usan precisamente para eso: para encontrar errores en vez de para generar un código eficiente.

En esta sección estudiaremos con detalle qué es un error sintáctico y cuáles son las respuestas posibles de un compilador ante ellos.

2.1.1 ¿Qué es un error sintáctico?

En todos los capítulos de este libro realizaremos una distinción explícita entre los errores de carácter sintáctico y los de carácter semántico. En cualquier caso conviene hacer hincapié en que todos los errores que se pueden encontrar en el fuente de un programa *son de carácter sintáctico*, por sorprendente que pueda parecer. Los reconocedores de lenguajes con los que nosotros trabajamos permiten tratar un subconjunto de aquellos que se pueden describir utilizando gramáticas independientes del contexto. Este tipo de gramáticas suele ser adecuado para lenguajes de juguete, pero claramente insuficientes para describir lenguajes de programación reales. Ni C, ni Eiffel, ni Pascal, ni tan siquiera los ensambladores se pueden describir usando gramáticas independientes del contexto. Lo que se suele hacer es obtener una gramática independiente del contexto para un superconjunto del lenguaje y utilizar rutinas auxiliares para determinar qué cadenas de ese superconjunto no pertenecen al lenguaje que estamos estudiando.

Por lo tanto, desde un punto de vista absolutamente estricto, el error de tipos que en Pascal se produce al intentar sumar un entero con un carácter se puede considerar como un error sintáctico. La única diferencia con el error que se produce al insertar una palabra reservada en un lugar inadecuado, es que este último es detectado por los analizadores que se pueden construir con la tecnología actual, mientras que los primeros son muy difíciles o imposibles de detectar de esta forma. Por lo tanto la diferencia entre ambos tipos de errores está claramente condicionada por la tecnología disponible.

En este capítulo estudiaremos tan sólo aquellos errores que un analizador LR actual puede detectar. Los errores de tipos y otros serán materia de un capítulo posterior.

2.1.2 Respuesta ante los errores

Las formas en las que un compilador puede reaccionar cada vez que encuentra un error en el texto fuente pueden ser variadas. A continuación se muestran algunas de las posibles:

1. Respuestas inaceptables.
   (a) Respuestas incorrectas.
      i. En el compilador se produce un error interno y termina su ejecución.
      ii. El compilador entra en un ciclo.
      iii. El compilador continua sin mostrar el error y produciendo código objeto incorrecto.
   (b) Respuestas correctas pero de poca utilidad.
      i. El compilador detecta el primer error y termina inmediatamente su ejecución.
      ii. El compilador detecta varios errores, pero muy separados entre sí en el fuente.
2. Respuestas aceptables.
   (a) Respuestas posibles con la tecnología actual.
      i. El compilador descubre un error, informa de él, lo ignora y continua procesando el fuente a la búsqueda de más errores.

ii. El compilador descubre un error, informa de él, lo repara y continua procesando el fuente.

(b) Respuestas imposibles en la actualidad.

i. El compilador detecta los errores, los muestra y los corrige de forma que el código objeto final sea la traducción de aquello que el programador pretendía expresar en su fuente.

Las respuestas del primer grupo son inaceptables, bien porque los errores llevan a nuestro compilador a funcionar mal o bien porque no se detecta en cada pasada por el fuente todos sus errores. Aunque este tipo de respuesta no se debería presentar nunca, algunos compiladores comerciales de amplia difusión las producen en determinadas situaciones, lo que en definitiva nos da una idea clara de la enorme complejidad del problema que estamos tratando.

Las respuestas del segundo grupo son aceptables puesto que el compilador encuentra todos los errores que puede y además no entra en ciclos ni se producen en él errores internos. Las dos formas más habituales de comportarse ante los errores son ignorarlos o repararlos. En el primer caso lo que se hace es eliminar de la entrada aquellos símbolos que son conflictivos. Por ejemplo, en el lenguaje de una calculadora, la siguiente entrada sería incorrecta:

```
1 + ERROR 3
```

La forma más sencilla de tratar este error consiste en mostrar un mensaje al usuario e ignorar la palabra `ERROR`. De esta manera, el analizador se comportará como se realmente hubiese leído la frase $1 + 3$. Este comportamiento es sensato, pero en algunas ocasiones puede ser incorrecto. Pensemos, por ejemplo en un programa para la calculadora como el siguiente:

```
1 * * 3
```

En este caso, parece que al usuario se le ha olvidado escribir un número detrás del primer asterisco. Por lo tanto, en esta ocasión, lo mejor no sería ignorar ese símbolo, sino reparar el error introduciendo un número cualquiera ambos asteriscos de forma que el analizador realmente lea la cadena $1 * \mathtt{n} * 3$, donde `n` es cualquier número.

En la práctica, los analizadores reales deben distinguir entre una y otra situación y decantarse por ignorar símbolos o insertarlos según sea más adecuado en cada ocasión. En este tema estudiaremos de qué forma se puede llevar a cabo esto de una forma local. La modalidad de reparación local se diferencia de la reparación global en que intenta corregir los errores ignorando o insertando símbolos a partir del punto en el que se han, sin volver nunca atrás en la cadena de entrada para insertar o eliminar de ella ningún símbolo. Un ejemplo en el que es preciso reparación global de errores es el siguiente trozo de programa en lenguaje C:

```
  (alfa = sin(omega) / sqrt(tan(beta / gamma)))
  delta = -alfa;
else
  delta = 1 / alfa;
```

Es evidente que para reparar este error, la mejor solución sería insertar el lexema `if` al comienzo de la primera línea, pues parece que lo que ha ocurrido es que el programador ha olvidado escribirlo. Pero en la actualidad esto es muy costoso, pues entre la posición en que hay que insertar el símbolo `if` y aquella en que se detecta el error (cuando aparece el lexema `delta` por primera vez) hay demasiados símbolos intermedios.

El problema de la reparación global es muy complejo y en la actualidad tan solo algunos analizadores experimentales implementan estas técnicas. La mayoría de los analizadores habituales tan sólo implementan técnicas locales, menos potentes, pero bastante más asequibles desde el punto de vista práctico. Lo más probable es que uno de estos analizadores corrigiese el error en el anterior texto insertando un punto y coman antes del identificador `delta` y eliminando la palabra reservada `else`, convirtiendo de esta forma el texto incorrecto en una secuencia de tres asignaciones.

En las secciones siguientes estudiaremos algunos de los métodos de reparación local más conocidos.

2.2 Métodos *ad hoc*

La mayor parte de los compiladores comerciales utilizan un método de recuperación de errores específico que no se puede extrapolar con facilidad al caso de gramáticas para otros lenguajes. Generalmente cuando el analizador sintáctico encuentra un error en la cadena de entrada llama a una rutina de manejo específico de ese error. De esta forma, aquellos que escriben el compilador son completamente libres de decidir las acciones más oportunas a llevar a cabo en cada situación y se puede conseguir una recuperación de ciertos errores bastante buena, aunque difícilmente reutilizable en otros proyectos.

La recuperación de errores *ad hoc* puede resultar bastante adecuada cuando los errores que se presentan encajan en alguna de las categorías de errores que los creadores del compilador han anticipado. Pero en la práctica es muy difícil anticipar todas las clases de errores sintácticos que se pueden cometer cuando se escribe el fuente de un programa.

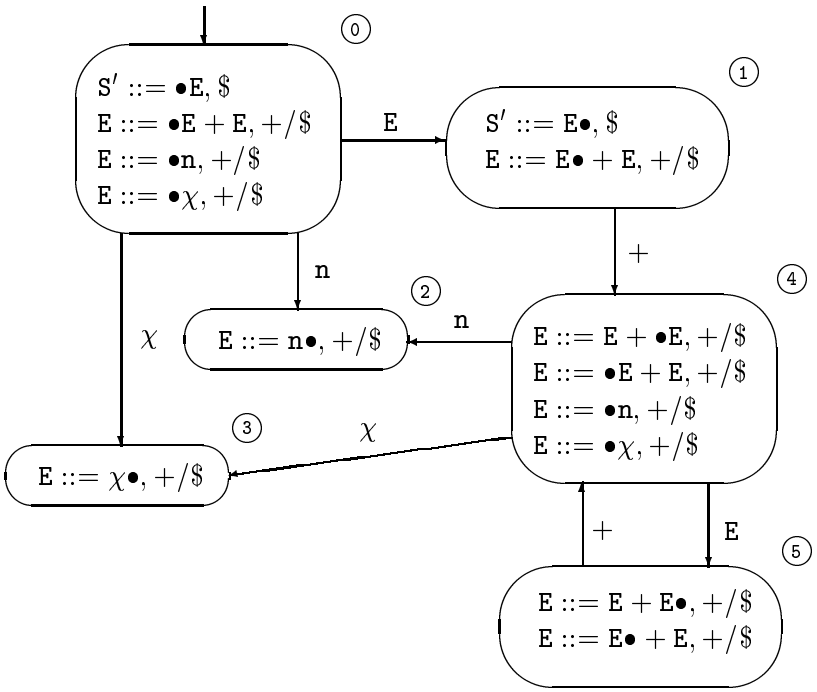
Como hemos indicado, la mayor parte de las técnicas de análisis sintáctico se basan en el uso de una tabla que indica en cada momento cuál es la acción que hay que llevar a cabo. Añadir este tipo de recuperación de errores al analizado consiste en codificar las rutinas adecuadas en las entradas apropiadas de esta tabla. El problema de estas técnicas es que requieren de un riguroso conocimiento de la tabla de análisis y se adapta con dificultad a los cambios que se realicen en la gramática, puesto que estos siempre implican la modificación de la tabla.

2.3 Método de Aho y Johnson

Este método se basa en la incorporación en la gramática del lenguaje de lo que se conoce con el nombre de producciones de error. Desde el punto de vista formal, se trata de producciones normales en las que se ha incorporado un símbolo terminal especial que indica al algoritmo de análisis dónde deseamos realizar una recuperación específica de errores. En adelante representaremos es símbolo especial como χ y lo trataremos como cualquier otro símbolo terminal.

Mientras que la entrada sea correcta no se tienen en cuenta las producciones de error, pero en el momento en el que se detecta uno se inserta el símbolo χ en la entrada y se descartan tantos elementos de la cima de la pila como sea necesario hasta encontrar en ella un estado para el que esté definida una transición con el símbolo χ. Es por lo tanto un símbolo terminal ficticio que el analizador sintáctico inserta en la entrada cada vez que se detecta un error previsto por la persona que ha escrito la gramática. El problema de este tipo de producciones es que puede llevar un tiempo considerable colocarlas y comprobar su funcionamiento, sin contar los conflictos que pueden introducir en las tablas de análisis.

Como ejemplo, vamos a estudiar una sencilla gramática que describe, de forma ambigua, sumas: $(\{\mathtt{n}, +, \chi\}, \{\mathtt{S}', \mathtt{E}\}, \mathtt{S}', \{\mathtt{S}' ::= \mathtt{E}, \mathtt{E} ::= \mathtt{E} + \mathtt{E} | \mathtt{n} | \chi\})$. Para entender bien cómo funciona este mecanismo, debemos obtener el autómata LR y la tabla de análisis asociada. Puesto que el método LR más habitual es el LALR(1), será el que emplearemos en todo el capítulo, pero los conceptos se adaptan con enorme sencillez a cualquier otro método de análisis LR.



	+	n	χ	$	S′	E
0		shift 2	shift 3			1
1	shift 4			accept		
2	reduce 3			reduce 3		
3	reduce 4			reduce 4		
4		shift 2	shift 3			5
5	shift 4/reduce 2			reduce 2		

#	**Regla**
1	S′ ::= E
2	E ::= E + E
3	E ::= n
4	E ::= χ

Fíjese en que el símbolo χ se trata como si fuese cualquier otro terminal. De hecho forma parte del alfabeto Σ_T de esta gramática. La única diferencia es que el analizador tan sólo lo tiene en cuenta cuando en el estado actual se encuentra en la entrada con un símbolo para el que no hay definida ninguna acción. En este caso lo que hace es insertar en la entrada el símbolo χ y si en el estado actual hay definida alguna acción con este símbolo la lleva a cabo y continua con el análisis. En caso contrario, comienza a eliminar símbolos de la pila hasta encontrar un estado para el que sí que exista una transición definida con este símbolo. En caso de vaciar la pila sin haber encontrado ningún estado adecuado, termina indicando que la cadena errónea de entrada no ha podido ser analizada.

A continuación veremos ejemplos que ilustran cada una de las situaciones que acabamos de describir.

2.3.1 χ como símbolo ficticio

Estudiaremos de qué forma reacciona un analizador con recuperación de errores ente la cadena errónea $\mathtt{n}+\,+\mathtt{n\$}$. En la siguiente tabla recogeremos la evolución de la pila , la entrada y la acción que se lleva a cabo en cada momento.

Pila	**Entrada**	**Acción**
0	$\mathtt{n}+\,+\mathtt{n\$}$	`shift 2`
0n2	$+\,+\mathtt{n\$}$	`reduce E ::= n`
0E1	$+\,+\mathtt{n\$}$	`shift 4`
0E1 + 4	$+\mathtt{n\$}$	`error`

Como vemos en la traza, tras llegar al estado 4 no existe ninguna transición válida con el carácter +, por lo que la cadena de entrada no pertenece al lenguaje. En esta situación, el analizador introduciría el símbolo χ en la entrada y como en el estado `4` hay una transición para este carácter, llevaría a cabo la acción `shift 3` y continuaría con el análisis tal y como se indica a continuación.

Pila	**Entrada**	**Acción**
0E1 + 4	$\chi+\mathtt{n\$}$	`shift 3`
0E1 + 4χ3	$+\mathtt{n\$}$	`reduce E ::=` χ
0E1 + 4E5	$+\mathtt{n\$}$	[`shift 4`]/`reduce E ::=` χ
0E1 + 4E5 + 4	$\mathtt{n\$}$	`shift 2`
0E1 + 4E5 + 4n2	\$	`reduce E ::= n`
0E1 + 4E5 + 4E5	\$	`reduce E ::= E + E`
0E1 + 4E5	\$	`reduce E ::= E + E`
0E1	\$	`accept`

Por lo tanto, el analizador se ha recuperado del error introduciendo en la cadena de entrada el símbolo ficticio χ. Es decir, que en este caso el carácter χ ha sido insertado por el analizador entre los dos operadores de suma, de forma que la expresión que realmente se ha reconocido ha sido $\mathtt{n}+\chi+\mathtt{n\$}$.

2.3.2 χ para ignorar parte de la entrada

Como vemos, el mecanismo de corrección de errores se ha comportado bastante bien, pero aún se le puede obtener más juego ya que en ocasiones también es capaz de ignorar símbolos. En estos casos se suele decir que el analizador "tira" parte de la entrada antes de continuar el reconocimiento. Podremos ver este comportamiento cuando le suministramos la cadena $\mathtt{nn+n\$}$. Como en el caso anterior, empezaremos por realizar la traza del analizador hasta que se produzca el primer error.

Pila	Entrada	Acción
0	nn + n$	shift 2
0n2	n + n$	reduce E ::= n
0E1	n + n$	error

En este punto nos encontramos con que no existe ninguna transición a partir del estado 1 con el carácter n, pero que tampoco hay ninguna para chi. En esta situación lo que hace el analizador es substituir el carácter conflictivo de la entrada por χ y eliminar símbolos de la pila hasta encontrar en ella un estado que admita una transición con este símbolo de error. En este caso sería necesario desapilar hasta que el estado 0 quedase en la cima. A partir de ahí, la evolución sería la siguiente:

Pila	Entrada	Acción
0	χ + n$	shift 3
0χ3	+n$	reduce E ::= χ
0E1	+n$	shift 4
0E1 + 4	n$	shift 2
0E1 + 4n2	$	reduce E ::= n
0E1 + 4E5	$	reduce E ::= E + E
0E1	$	accept

Como es habitual podemos reconstruir la cadena de derivaciones que nos conducen del axioma a la cadena de entrada siguiendo en la traza las acciones reduce en orden inverso: accept, reduce E ::= E + E, reduce E ::= n, reduce E ::= χ, reduce E ::= n, esto es, $\mathtt{S'} \longrightarrow \mathtt{E} \longrightarrow \mathtt{E+E} \longrightarrow \mathtt{E+n} \longrightarrow \chi + \mathtt{n}$. Pero llegados a este punto aún nos falta por aplicar la reducción de E ::= n, lo que ocurre es que en la cadena de derivaciones ya no aparece ningún símbolo E para substituir.

Aunque parezca sorprendente, el analizador ha ignorado la segunda n que aparecía en la entrada sustituyéndola por el símbolo χ, pero realmente lo que ha hecho ha sido ignorar la primera reducción que aplicó.

Este sencillo ejercicio nos demuestra que esta técnica de recuperación es compleja de utilizar y que siempre que la usemos debemos estudiar con detenimiento la tabla de análisis subyacente, pues de otra forma los resultados pueden ser bastante inesperados. Fíjese en que en el ejemplo de la sección anterior la cadena de entrada era n + +n, e, intuitivamente, también podría haberse corregido ignorando el segundo signo +. Pero no ha sido así debido a la tabla de análisis que hemos obtenido para esta gramática.

2.4 Método de análisis de transiciones

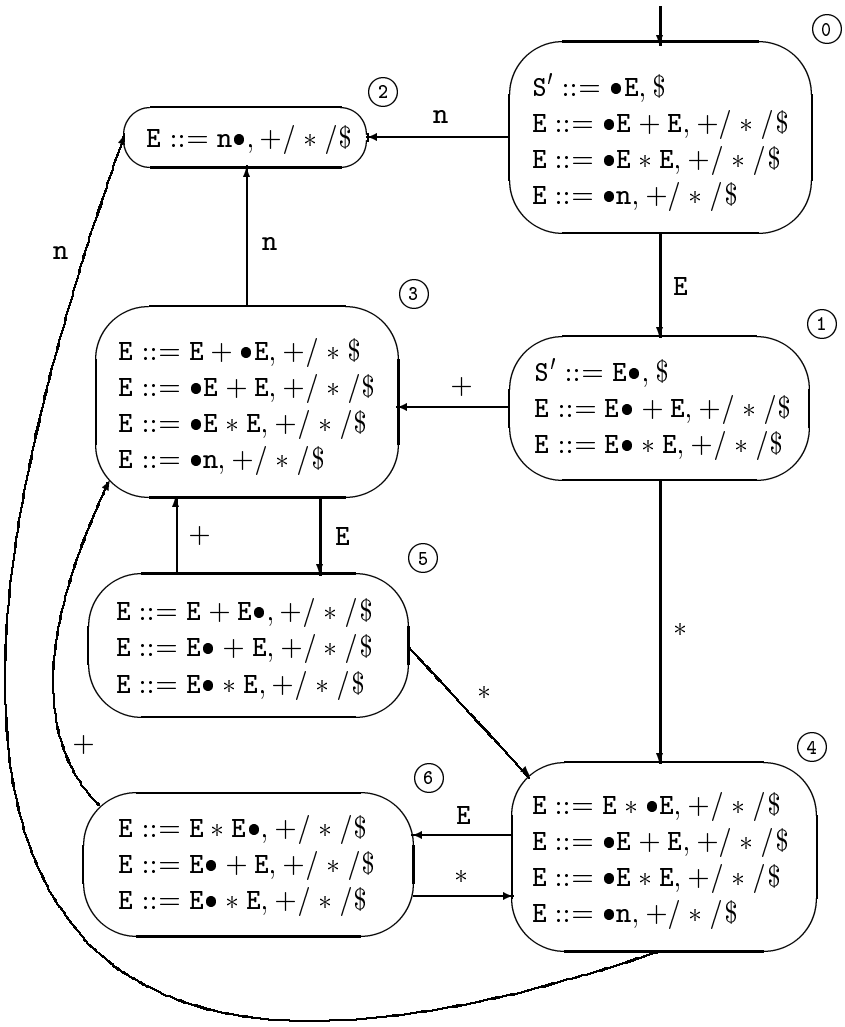
El principal problema de la técnica de recuperación de Aho y Johnson es que deja en manos del diseñador de la gramática decidir en qué lugares desea llevar a cabo una recuperación local de errores, y esto, tal y como hemos podido comprobar, implica conocer de forma exacta la tabla de análisis, algo que en absoluto es común cuando trabajamos con las gramáticas de lenguajes reales. En esta sección estudiaremos un método propuesto por Pedro Carrillo y Rafael Corchuelo que tiene como principal ventaja el hecho de que la reparación se lleva a cabo sin que el usuario deba escribir producciones de error especiales.

Este método intenta recuperarse de los errores insertando o eliminado símbolos en la cadena de entrada de forma que esta quede en una situación a partir de la cual sea posible continuar con el análisis. Para ello asume que el diseñador de la gramática ha definido dos funciones que determinan cual es el coste de insertar un símbolo terminal en la posición del error ($\mathcal{I}(\mathtt{t}, \Omega)$) o de borrarlo ($\mathcal{B}(\mathtt{t}, \Omega)$). Ambas toman un parámetro adicional Ω que representa el contexto en que se ha producido el error. La definición de este contexto queda completamente abierta y puede consistir, por ejemplo, en una porción de la pila y de la cadena de entrada que nos ayude a calcular el coste relativo de insertar o borrar un símbolo en ese contexto. Si el coste es independiente del contexto podemos simplificar estas dos funciones eliminando su segundo parámetro.

Cuando se detecta un error, este método investiga a partir del estado que se encuentra en la cima de la pila cuáles son los caracteres del alfabeto terminal para los que existen transiciones y cuál sería el coste de insertar esos símbolos en la posición del error. También investiga si de borrar el símbolo conflictivo, podría continuarse con el análisis a partir del símbolo siguiente. Una vez realizado este análisis, el método selecciona la alternativa viable de mínimo coste y continua con el análisis de la entrada.

Como ejemplo examinaremos la siguiente gramática ambigua para sumas y multiplicaciones: $(\{\mathtt{n}, +, *, \chi\}, \{\mathtt{S}', \mathtt{E}\}, \mathtt{S}', \{\mathtt{S}' ::= \mathtt{E}, \mathtt{E} ::= \mathtt{E} + \mathtt{E} | \mathtt{E} * \mathtt{E} | \mathtt{n}\})$. Tal y como indicamos no es necesario introducir ninguna producción de error, pero sí que es necesario proporcionar los costes de insertar o borrar cada uno de los símbolos terminales de la gramática. El problema es que no existe ninguna regla o heurística que permita seleccionar de forma óptima estos costes y en gran medida su ponderación depende por completo de la experiencia que tenga el diseñador de la gramática con el método y de los experimentos prácticos que realice. En cualquier caso, debemos pensar en que aquellos símbolos que involucren información importante deben ser los más costosos de borrar o insertar, mientras que los símbolos de puntuación pueden tener un coste relativamente bajo. La razón es que si un símbolo representa información de gran valor al eliminarlo podemos perderla y al insertarlo puede ocurrir que estemos introduciendo esa información en un lugar en el que no es relevante. En este caso, vamos a utilizar la siguiente ponderación independiente del contexto: $\mathcal{I}(+) = \mathcal{I}(*) = \mathcal{B}(+) = \mathcal{B}(*) = \mathtt{1}$, $\mathcal{I}(\mathtt{n}) = \mathtt{2}$ y $\mathcal{B}(\mathtt{n})$. En esta gramática estamos suponiendo que `n` representa los números que sumamos o multiplicamos, por lo que tienen información relevante que no nos interesa ni perder ni insertar con valores inadecuados. En cambio borrar o insertar un operador tiene un coste relativamente bajo. ¿Por qué hemos elegido estos costes y no otros? Tal y como indicábamos anteriormente, no hay ninguna razón especial para ello. Es la experiencia la que nos ayuda a valorar de forma subjetiva el coste de inserción o borrado de cada símbolo terminal.

A continuación se muestra el autómata y la tabla de análisis LALR(1):



	+	∗	n	$	S′	E
0			shift 2			1
1	shift 3	shift 4		accept		
2	reduce 4	reduce 4		reduce 4		
3			shift 2			5
4			shift 2			6
5	shift 3/reduce 2	shift 4/reduce 2		reduce 2		
6	shift 3/reduce 3	shift 4/reduce 3		reduce 3		

#	**Regla**
1	S′ ::= E
2	E ::= E + E
3	E ::= E ∗ E
4	E ::= n

Para ver de qué manera funciona este método realizaremos una traza de ejecución tomando como cadena de entrada nn + n$.

Pila	Entrada	Acción
0	nn + n$	shift 2
0n2	n + n$	error

En este punto se ha detectado un error, por lo que el procedimiento de análisis examinaría el estado en la cima de la pila para determinar de todas las transiciones válidas a partir de él cuál es la que resulta menos costosa. En este caso aparecen dos posibilidades de coste 1 si insertamos los símbolos + o ∗. El método también explora la posibilidad de eliminar de la entrada el carácter conflictivo, pero antes de considerar esta alternativa debe determinar si es viable. Para ellos tan sólo tenemos que examinar si el carácter que aparece a continuación del conflictivo es una continuación válida de la cadena que llevamos analizada, es decir, si existe con él definida alguna transición a partir del estado en la cima de la pila. En este caso, eliminar el carácter n es viable puesto que el siguiente símbolo es un + y a partir del estado 2 existe una transición válida para ese carácter. Lo que ocurre es que el coste de borrar n es de 3 unidades, por lo que el analizador se decidiría arbitrariamente por insertar un + o un ∗.

Supongamos que se decide por insertar un +, entonces el análisis de la cadena de entrada continuaría tal y como se indica a continuación:

Pila	Entrada	Acción
0n2	+n + n$	reduce E ::= n
0E1	+n + n$	shift 3
0E1 + 3	n + n$	shift 2
0E1 + 3n2	+n$	reduce E ::= n
0E1 + 3E5	+n$	[shift 3]/reduce E ::= E + E
0E1 + 3E5 + 3	n$	shift 2
0E1 + 3E5 + 3n2	$	reduce E ::= n
0E1 + 3E5 + 3E5	$	reduce E ::= E + E
0E1 + 3E5	$	reduce E ::= E + E
0E1	$	accept

Por lo tanto el método lo que ha hecho ha sido corregir la cadena de entrada convirtiéndola en n + n + n$, pero sin que el usuario tenga que conocer en absoluto cuáles son las características de la tabla de análisis subyacente. Lo único que debe hacer es ponderar basándose en su experiencia el coste relativo de insertar o borrar cada uno de los símbolos terminales de la gramática. Fíjese en que otra posible corrección de la cadena de entrada podría ser n + n$, es decir, eliminando la segunda n, pero el diseñador de la gramática a considerado que borrar el símbolo n puede llevar consigo pérdida de información importante y por lo tanto lo ha penalizado con un alto coste.

Para terminar mostraremos un nuevo ejemplo en el que intentaremos reconocer la cadena de entrada n + ∗n$. De nuevo se plantearán dos alternativas: insertar una n entre los operadores o eliminar el operador ∗. Dada la ponderación que hemos utilizado, el método optará por la segunda opción.

Pila	Entrada	Acción
0	n + ∗n$	shift 2
0n2	+ ∗ n$	reduce E ::= n
0E1	+ ∗ n$	shift 3
0E1 + 3	∗n$	error
0E1 + 3	n$	shift 2
0E1 + 3n2	$	reduce E ::= n
0E1 + 3E5	$	reduce E ::= E + E
0E1	$	accept

De todas formas no crea que una opción de coste elevado nunca se va a llevar a cabo por esta razón. Para convencerse de ello podemos realizar la traza cuando se le suministra al analizador la cadena n + ∗ ∗ n$.

Pila	Entrada	Acción
0	n + ∗ ∗ n$	shift 2
0n2	+ ∗ ∗n$	reduce E := n
0E1	+ ∗ ∗n$	shift 3
0E1 + 3	∗ ∗ n$	error

En este punto nos enfrentamos a una situación de error, por lo que ahora con el estado 3 en la cima de la pila la única alternativa posible es insertar en la entrada una n. En esta ocasión no es viable eliminar el asterisco puesto que detrás de él aparece otro y en el estado 3 no hay ninguna transición definida con este símbolo. Por lo tanto, la traza continua de la siguiente forma:

Pila	Entrada	Acción
0E1 + 3	n ∗ ∗n$	shift 2
0E1 + 3n2	∗ ∗ n$	reduce E ::= n
0E1 + 3E5	∗ ∗ n$	shift 4
0E1 + 3E5 ∗ 4	∗n$	error

De nuevo nos encontramos en una situación de error en la que podemos insertar en la entrada una n con coste 2 o eliminar el carácter ∗ con coste 1. En esta ocasión la diferencia es que si eliminamos el ∗ en la entrada queda una n, que es uno de los caracteres admisibles a partir del estado 4. Por lo tanto se opta por eliminar el carácter conflictivo de la entrada.

Pila	Entrada	Acción
0E1 + 3E5 ∗ 4	n$	shift 2
0E1 + 3E5 ∗ 4n2	$	reduce E ::= n
0E1 + 3E5 ∗ 4E6	$	reduce E ::= E ∗ E
0E1 + 3E5	$	reduce E ::= E + E
0E1	$	accept

Por lo tanto, en esta ocasión el analizador ha corregido la entrada convirtiéndola en la cadena n + n ∗ n$.

2.5 Otros métodos de recuperación

La principal ventaja de los métodos que hemos estudiado en las secciones anteriores es que son fáciles de implementar y la recuperación ante la presencia de un error puede llevarse a cabo de una forma eficiente. Los métodos que estudiaremos a continuación suelen producir mejores correcciones de la entrada que los que hemos expuesto y también prescinden de las producciones de error, pero su implementación práctica resulta muchísmo más complicada e ineficiente, lo que limita en norme medida su aplicación práctica.

2.5.1 Método de Graham y Rhodes

Este método lleva a cabo la corrección de los errores en dos etapas conocidas como condensación y corrección. La primera de ellas intenta modificar el contenido de la pila y la cadena de entrada de tal forma que en la segunda se pueda utilizar la mayor porción posible del contexto en que se ha producido el error para corregirlo.

La fase de condensación comienza realizando un movimiento hacia atrás en la pila en el que el analizador intenta reducir tantas cadenas como sea posible. A continuación se realiza un movimiento hacia adelante durante el cual se intenta analizar la porción de texto que se encuentra inmediatamente delante del error. El movimiento hacia adelante termina cuando se encuentra un segundo error o en el momento en que resulta imposible seguir llevando a cabo su análisis sin tener en cuenta el contexto previo a la aparición del error, esto es, en el momento en que el análisis de la entrada requiere una reducción que involucra algún símbolo a la izquierda de la posición en la que se detectó el error. El movimiento hacia adelante del analizador sintáctico puede consumir una porción importante de la entrada, pero no se ignora ninguno de los símbolos que contiene.

Una vez terminada la fase de condensación se analiza la pila y se realiza un proceso de concordancia de patrones para ver cuál o cuáles son las reglas de producción la gramática cuya parte derecha es más parecida a los símbolos que se encuentran en la pila. Una vez elegida esa regla se insertan o eliminan símbolos de la pila de forma que se pueda llevar a cabo la reducción de la regla seleccionada y se continua con el análisis de forma normal.

Como se puede ver, es un método bastante complejo de implementar, por lo que no entraremos en más detalles técnicos. Tan sólo lo ilustraremos a grandes con la siguiente gramática que representa, de forma simplificada, una lista de dos tipos de sentencias separadas por puntos y comas:

$$(\{\texttt{s}, ;\}, \{\texttt{S}', \texttt{L}, \texttt{S}\}, \texttt{S}', \{\texttt{S}' ::= \texttt{L}, \texttt{L} ::= \texttt{S};\texttt{L}|\texttt{S}, \texttt{S} ::= \texttt{a}|\texttt{b}\})$$

Si enfrentamos nuestro analizador a la cadena de entrada `ab;a` en la que falta un punto y coma detrás de la primera `a`, cuando se detecte el error la pila, prescindiendo de las marcas de estado, contendrá tan sólo una `a`. En este punto entra en funcionamiento el procedimiento de recuperación de Graham y Rhodes, que intenta reducir en la pila tantas reglas de producción como sea posible. En este caso, la única reducción aplicable sería $\texttt{S} ::= \texttt{a}$, por lo que en la pila tan sólo quedaría una `S`. A continuación apilaría el símbolo χ para indicar dónde se ha producido el error y continuaría con el análisis de la entrada hasta encontrar un nuevo error o hasta que sea necesario llevar una reducción que involucre el contexto a la izquierda de la posición del error, esto es, el símbolo χ. En nuestro ejemplo

esto significaría leer todo el resto de la entrada hasta que la pila quedase en la situación `S`χ`L` después de haber reducido la cadena `b;a` por `L`.

Una vez acabada la fase de condensación se lleva a cabo el proceso de concordancia de patrones entre el contenido de la pila y la parte derecha de todas las reglas de producción de la gramática para ver cuál es la regla que más se parece a lo que tenemos en la pila. En este caso es evidente que la más parecida es `L ::= S; L` por lo que en esta fase se cambiaría el símbolo χ por el ; y a partir de ahí continuaría el análisis normal de la cadena de entrada.

En este ejemplo la corrección ha resultado sencilla, pero de forma general, este método presenta muchos problemas importantes de implementación. El primero es que en el movimiento hacia atrás es necesario llevar a cabo reducciones en la pila, pero en los métodos que llevan algún carácter de lectura adelantada esto no es posible si el carácter de la entrada es erróneo. Recuerde que la acción a llevar a cabo viene siempre determinada por la marca que se encuentra en la cima de la pila y el símbolo que se encuentra en la entrada, y si este es erróneo entonces determinar qué reducción llevar a cabo puede ser una tarea realmente complicada. En el ejemplo que hemos mostrado anteriormente cuando en la pila tan sólo se encuentra el símbolo `a` hemos decidido reducir la regla `S ::= a` y nos hemos parado en ese punto. Pero fíjese en que también podríamos haber reducido la regla `L ::= S` a continuación y en esta ocasión la recuperación no habría tenido éxito. En los analizadores LR(0), también se producen problemas similares.

El movimiento hacia adelante también plantea problemas serios, puesto que una vez terminado el movimiento hacia atrás hay que dejar en la cima de la pila una marca de estado a partir de la cual sea posible continuar con el análisis de la cadena de entrada a partir de la posición del error, y esto requiere de un análisis muy riguroso y complejo del autómata para el lenguaje de la pila.

Otra característica de este método es que permite al diseñador de la gramática mostrar el conocimiento que tiene acerca de los errores que se pueden presentar con cierta frecuencia y sus causas. Para ello utiliza un criterio de mínimo coste a la hora de elegir entre los distintos cambios que se pueden llevar a cabo en la pila durante la fase de corrección. El diseñador puede emplear su conocimiento acerca del lenguaje proporcionando al analizador dos vectores con los costes de insertar en la pila o borrar de ella cada uno de los símbolos de la gramática. La rutina de recuperación intenta minimizar el coste total de todos los cambios que realiza en la pila.

Pese a todos estos problemas, se han obtenido implementaciones y mejoras del método de Graham y Rhodes, aunque ninguna de ellas es de uso habitual.

2.5.2 Método de McKenzie, Weatman y De Vere

Este método parte de la misma idea que el de análisis de transiciones, pero la mejora notablemente incluyendo la posibilidad de utilizar además producciones de error al estilo de las del método de Aho y Johnson y con un sencillo método heurístico de validación de reparaciones.

Al igual que en el método de análisis de transiciones, en el momento en que se detecta un error en la entrada, se analizan aquellos caracteres para los que sí está definida una transición a partir del estado actual y la posibilidad de eliminar aquél que es conflictivo de la entrada. La diferencia es que una vez elegida la alternativa de mínimo coste se explora

para ver si a partir de ella sería posible introducir en la pila al menos tres símbolos sin que se produzca ningún error. En caso de que sea posible, el error se da por corregido usando esa alternativa de mínimo coste. En otro caso se analiza la siguiente alternativa de mínimo coste y se repite el proceso hasta encontrar una que se viable o hasta que se agoten todas las posibilidades. En este caso, si el diseñador de la gramática ha escrito producciones de error, se aplica el método de Aho y Johnson.

Esta técnica ha sido implementada con éxito y se puede considerar una alternativa a medio camino entre el método de análisis de transiciones y el de Graham y Rhodes, pues se comporta bastante bien en la mayoría de los casos sin presentar problemas de implementación tan complejos como los del método de Graham y Rhodes.

2.5.3 La segunda oportunidad

Algunas estrategias de recuperación de errores no siempre tienen éxito a la hora de localizar o recuperarse de un error, y puede ocurrir que consuman un tiempo demasiado grande analizando el texto de entrada. En estos casos suele ser conveniente dotar al analizador sintáctico de algún método de recuperación de segunda oportunidad que entra en funcionamiento cuando los otros han fallado o no han sido capaz de encontrar la fuente del error en un tiempo razonable.

Los métodos habituales de segunda oportunidad son los conocidos como pánico y borrado unitario. El primero de ellos consiste en descartar de la entrada cierto número de símbolos hasta encontrar uno que sirva de delimitador. Un ejemplo suele ser el punto y como que sirve de terminación de las sentencias en la mayoría de los lenguajes de programación. El método de borrado unitario no descarta símbolos del fuente hasta encontrar un delimitador, sino que elimina del mismo toda la estructura sintáctica en la que ha aparecido el error. La característica principal del borrado unitario es que preserva la corrección sintáctica de los programas, aunque puede ser bastante más costosa y difícil de implementar que el método pánico.

2.6 Bibliografía

El problema de analizar sintácticamente cadenas con errores ha recibido una enorme atención en la literatura especializada, fundamentalmente en la década de los 70. En cualquier caso, hoy, cuando ya existen métodos de amplia difusión implementados en la mayoría de generadores de analizadores sintácticos, siguen apareciendo en ocasiones artículos dedicados a este tema.

El método de Aho y Johnson se describe con detalle en [Aho y Johnson 1974], el de Graham y Rhodes en [Rhodes 1973, Graham y Rhodes 1975] y el de Mckenzie, Yeatman y De Vere en [McKenzie *et al.* 1995]. También recomendamos la lectura de [Tremblay y Sorenson 1985] que recoge un compendio bastante extenso acerca de las técnicas más comunes de recuperación y reparación de errores.

La idea de realizar reparaciones de mínimo coste en base a dos funciones definidas por el diseñador de la gramática fue propuesta en [Aho y Peterson 1972] y después aplicada por diversos autores a sus técnicas particulares de recuperación, entre ellos el de análisis de transiciones o el de Graham y Rhodes. Este último es, como hemos comentado, uno de los

que presentan mayores problemas de implementación práctica, pero algunos autores como [Mickanus y Modry 1978, Pennello y DeRemer 1978] han propuesto mejoras que recortan su capacidad de corrección haciéndolo mucho más fácil de implementar. El origen de estas mejoras se encuentra en el artículo [Druseikis y Ripley 1976] que describe una idea sencilla para la recuperación ante errores para analizadores de tipo SLR(1).

La clasificación de las respuestas que un compilador puede ofrecer cuando encuentra errores en la entrada ha sido adaptada a partir de [Horning 1976].

Parte II

Léxico y Sintaxis

Capítulo 3

Analizadores léxicos

Uno de los objetivos de este tema es desarrollar habilidades en la especificación de la parte léxica de un lenguaje. Entre estas habilidades podriamos destacar la especificación de los números reales, los identificadores, las palabras reservadas del lenguaje, etc. Es importante resaltar que el analizador léxico debe comunicarse con el analizador sintáctico y debe transferirle una determinada información. De las especificaciones escritas siempre debemos destacar el interface de las mismas con el analizador sintáctico. En concreto los token definidos, sus atributos. De esa forma consideramos el analizador léxico como un procedimiento que cada vez que es llamado produce un token con un conjunto de atributos asociados.

A partir de las especificaciones usaremos la herramienta Lex para generar un analizador léxico automáticamente o lo escribiremos manualmente. Hemos de valorar las ventajas e incovenientes de las dos posibilidades en cuanto a eficiencia y trabajo invertido.

El tema puede prepararse a partir del capítulo tres de [ASU86] y el capítulo tres de [FiL88].

3.1 Papel del Analizador Léxico

Un Analizador Léxico se podría considerar como una caja negra con un conjunto de entradas y de salidas:

```
Fichero  ---->     Analizador   ----->    <token,atributos>
Fuente              Lexico
```

La entrada sería el Fichero Fuente y la salida son listas de tuplas, compuestas por Tokens y los atributos asociados a cada uno. Desde un punto de vista funcional el Analizador Léxico es una rutina de la forma

```
<token,atributos> = analex(fiche)
```

La rutina *analex* toma como parámetros un fichero de entrada (en una determinada posición de lectura), y devuelve el siguiente token encontrado y sus atributos.

Asociado a cada token hay un conjunto de secuencias de caracteres. Cada una de estas secuencias de caracteres las denominaremos un *lexema*. El conjunto de lexemas asociados a un token forman un lenguaje. Este lenguaje suele ser bastante simple y generalmente se describe con gramáticas regulares (por la derecha o por la izquierda) o mas usualmente con *expresiones regulares*. Especificar un token es dar la expresión regular que describe el conjunto de lexemas asociado, sus atributos y los algoritmos para calcularlos.

Especificar un analizador léxico o la parte léxica de un lenguaje es dar la especificación de todos los token del lenguaje. Asi una serie de llamadas sucesivas al analizador léxico transforma el fichero de entrada en una lista de token junto con sus atributos.

De ahora en adelante vamos a suponer que el tipo Token es un tipo enumerado que toma valores en el conjunto

$$Token = \{t_1, t_2, ..., t_n\}$$

Asociado a cada token t_j describiremos un conjunto de atributos at_j. Pero puede haber varios token que tengan la misma expresión regular para describir sus lexemas. Por eso la descripción de una analizador léxico se lleva a cabo especificando un conjunto de expresiones regulares

$$\{e_1, ..., e_r\}$$

y asociando a cada una de ellas un algoritmo a_j que tomando el lexema encontrado en cada caso, calcule el token asociado y sus atributos.

Asumimos que la rutina *analex* tiene una variable local que *text* que contiene el lexema particular encontrado al reconocer un determinado patrón. Cada una de los algoritmos a_j que debemos describir tienen la estructura:

```
<token,atributos> =a(text)
```

Una vez especificado el analizador léxico su funcionamiento es el siguiente: cuando se llama a analex se van leyendo caracteres hasta que se encuentra una hilera que pertenece a uno de los lenguajes descritos mediante las diferentes expresiones regulares. Sea e_j la expresión regular a la que pertenece el lexema encontrado, entonces se ejecuta la acción a_j.

3.1.1 Interfaces del analizador léxico

El analizador léxico tiene interfaces con el analizador sintáctico, fichero de entrada. Estos interfaces son:

El analizador sintáctico llama a la rutina analex. Esta devuelve el token encontrado y sus atributos.

El interface del fichero de entrada es a traves de funciones del tipo:

```
es_ff(f); getc(f): ungetc(f,c);
```

Veamos cuál sería el programa principal para chequear si analex funciona. Suponemos que para probarlo diseñamos las acciones para que no interactuen con el contexto ni con la tabla de símbolos. El programa de chequeo.

```
inicializacion;
mientras no(es_ff(f_entrada);
    <token,atributo> = analex(text);
    mostrar <token,atributo>;
fin_mientras
```

Después comprobaremos que la lista de atributos mostrados es realmente la que queríamos.

3.2 Especificación del Analizador Léxico

El diseño del A.L. influirá en el A.S. Pretendemos hacer una especificación del analizador léxico independiente de una herramienta específica como LEX, FLEX, etc. La especificación se compondrá de una conjunto de declaraciones regulares que daran nombre a determinadas expresiones regulares de uso frecuente. Una lista de tokens junto con sus atributos y una lista de expresiones regulares con sus acciones asociadas. Una especificación tendría la forma:

Declaraciones Regulares:

```
letra     [a-zA-Z_]
digito    [0-9]
id        {letra}+
```

Lista de Tokens y sus atributos

```
Token       Atributo
------      --------
ID          at1
C_ENT       at2
---         ---
```

Expresiones Regulares y Algoritmos Asociadas

```
 Expres. Regulares        Accion
 -----                   --------
 {letra}{alfanum}*         a1
 {digito}+                 a2
 "<-"                      a3
 "<"                       a4
 {string}                  a5
 --------                  --
```

Los algoritmos ai que se especifican en la tabla anterior toman como entrada el lexema actual y producen como salida el token asociado, sus atributos:

```
<token,atributos> =a(text)
```

3.2.1 Atributos de los tokens

Es muy importante tener en cuenta que una cuidadosa elección de los atributos de los tokens puede facilitar o entorpecer el desarrollo de los módulos restantes del compilador. Por otra parte tenemos que pensar en que los tokens son los elementos que sirven de comunicación entre el módulo de análisis léxico y el de análisis sintáctico, por lo que merece la pena repasar cuáles son los atributos que con más frecuencia suelen ser útiles con los tokens más comunes. También estudiaremos de qué forma se puede recoger información léxica precisa para después poder mostrar mensajes de error en los lugares correctos.

Atributos de los identificadores

Tenemos dos posibilidades:

- Devolver su lexema como atributo, o sea, la secuencia de caracteres reconocida.
- Devolver una entrada a la tabla de símbolos. Hay que hacer un procedimiento dentro del analizador léxico que busque, actualice y se posicione en la tabla de símbolos.

Entre ambas opciones, la primera es la más adecuada, aunque son muchos los textos de construcción de compiladores que recomiendan la segunda. La razón que desaconseja el uso de ésta última es que los analizadores léxicos están pensados para reconocer los tokens de los lenguajes de programación, por lo que por sí sólos no cuentan con información suficiente para determinar el contexto en el que aparecen esos tokens que está reconociendo.

Para solucionar este problema algunos autores recurren a utilizar variables de contexto, pero como veremos más adelante en el capítulo de análisis sintáctico, esto dificulta bastante la construcción del compilador y hace bastante difícil el tratamiento de los errores.

Por lo tanto, la mejor opción es que los identificadores cuenten con un atributo que recoge la hilera de caracteres que los componenen. En el capítulo dedicado al análisis sintáctico estudiaremos la forma de recuperar las entradas de la tabla de símbolos a partir de esas hileras.

Atributos de los números

De nuevo existen dos posibilidades fundamentales:

- Devolver directamente el lexema asociado.
- Pasar el valor asociado a esa hilera, para lo cual hay que implementar rutinas que efectúen la conversión de la misma a su valor numérico correspondiente.

La primera opción es bastante recomendable cuando dentro del compilador no tenemos que realizar operaciones aritméticas con los mismos, puesto que nos permite aislarnos

completamente de la precisión y rango de la máquina sobre la que se ejecutará el código objeto generado por el compilador. La segunda es más adecuada cuando el compilador debe realizar internamente operaciones con esos valores numéricos (por ejemplo, reducir todas aquellas expresiones en las que sólo intervienen literales numéricos a un único valor).

Atributos de los operadores

- No es necesario atributo. Se pasaría sólo el token.
- Podrían agruparse, por ejemplo, todos los operadores de un mismo tipo asociándoles el mismo token, y después asignar un atributo que los distinguiera.

  Por ejemplo, podríamos asociar todos los operadores binarios con el token `OPB` y colocarle un atributo que indique el tipo de operador concreto de que se trata.

La opción más habitual es la primera, y la segunda tan sólo es interesante cuando contamos con muchos operadores del mismo tipo con iguales precedencia y asociatividad. En el caso de que los operadores no tengan la misma asociatividad o precedencia el reconocimiento del lenguaje se complica bastante al usar la segunda técnica.

Por lo tanto, desde el punto de vista práctico, la opción más acertada suele ser la primera.

Atributos de las hileras de caracteres

- Pasar la hilera de caracteres completa.
- Crear un buffer para almacenar las hileras de caracteres, de manera que todas aquellas cadenas que son iguales compartan el mismo espacio de almacenamiento.

De entre las dos opciones la más acertada es la segunda puesto que permite reducir enormemente la cantidad de espacio de almacenamiento necesitada por el compilador. Es cierto que las hileras de caracteres exactamente iguales no suelen aparecer con frecuencia en el texto fuente de los programas, pero debemos pensar en que si implementamos este buffer, también podremos aprovecharlo para guardar las hileras de caracteres de los identificadores y estos si que se repiten multitud de veces.

Información léxica

Un problema que la mayor parte de los textos sobre compiladores suele dejar al margen es el de cómo se puede informar al usuario de la posición exacta del fuente en la que el compilador encuentra los errores.

Es evidente que uno de los factores que miden la calidad de un compilador es la precisión con la que informa de los errores que encuentra y ello lleva consigo informar de los mismos en los lugares exactos. Para conseguirlo lo mejor es asociar a cada uno de los tokens del lenguaje un atributo más que nos permite guardar la posición en la que aparece.

Generalmente, definiremos esta información como:

```
typedef struct {
    unsigned lin, col;  /* línea y columna */
    string nom_fich;    /* nombre del fichero */
} info_lex;
```

Cada token tendrá un atributo de tipo `info_lex` que el analizador léxico deberá rellenar con el número de línea, de columna y el fichero en el que aparece ese token.

Una solución a la que tradicionalmente se ha recurrido para encontrar la posición de los tokens es a dotar al analizador léxico de un conjunto de rutinas que nos informan sobre la posición en el fuente por la que se va analizando actualmente. Estas rutinas se han utilizado con frecuencia en el analizador sintáctico como se muestra en el siguiente ejemplo:

```
exp ::=  exp '+' exp
         { $$ = procesar_suma($1, $3,
                              yyline(), yycolumn(), yyfile(): }
```

La regla de sintaxis que hemos mostrado antes recoge la estructura de las sumas y la atribución lo que hace es procesarlas buscando errores (Por ejemplo, no se puede sumar una hilera de caracteres y un número entero). La rutina recibe además de las expresiones que estamos intentando sumar la información léxica para localizar el error. Pero supongamos que la expresion que se está tratando es la siguiente:

```
                    1         2         3
columna:  12345678901234567890123456789
texto:    'a' + (1 * 2 / 3 - (2 * 4))
```

Esta expresión es errónea, lo que ocurre es que cuando se detecta el problema ya se ha leido del fuente toda la subexpresión a la derecha del signo `+`. Por lo tanto el error se indicará en la columna 27, cuando realmente se encuentra en la columna número 5. Para evitar este problema y conseguir que el compilador informe de los errores en la posición exacta en la que se producen es preciso añadir a todos los tokens de interés un atributo de tipo `info_lex`.

3.3 Problemas con el diseño

3.3.1 Problemas con los identificadores

Tratamiento de las palabras clave

Lo primero a plantearse es ver si permitimos en nuestro lenguaje que las palabras clave puedan ser o no identificadores utilizables por el usuario. Lo más sencillo es hacer que las palabras clave sean además reservadas, y en este caso lo que debemos hacer es asociar a cada una de ellas un token diferente.

El único problema al hacer esto es que las expresiones regulares que describirán las palabras clave serán casos particulares de la expresión regular que describe los identificadores. Por ejemplo, dada la especificación:

```
Patrón                          Token
---------------------------------
"begin"                         BEGIN
"end"                           END
{letra}({letra}|{número})*      ID
```

el problema está en que cuando en el fuente aparece la palabra `begin`, esta encaja tanto en la expresión regular del token `BEGIN` como en la del token ID. Los analizadores léxicos generados por `lex`, por ejemplo, resuelven la ambiguedad considerando que en estos casos el token que se ha reconocido es el primero que se haya especificado, en este caso `BEGIN`.

De todas formas, aunque el problema queda resuelto, no es conveniente usar esta técnica porque los lenguajes de programación suelen contar con un número importante de palabras reservadas y los autómatas que genera `lex` para su reconocimiento suelen ser bastante grandes e ineficientes.

Lo más aconsejable es reconocer las palabras reservadas usando la expresión regular que describe de forma general los identificadores. La acción que asociaremos a esta expresión regular se encargará de determinar si el lexema que se acaba de reconocer es una palabra reservada o no usando algún algoritmo eficiente. Lo más habitual es guardar en una tabla ordenada los lexemas de las palabras reservadas y sus tokens asociados. De esta forma, para saber si un lexema recién reconocido es una palabra reservada o no bastará con realizar una búsqueda binaria del mismo en esta tabla. Si aparece se devuelve el token que se le haya asociado y en caso contrario se devuelve el token `ID`. Más adelante en este mismo tema volveremos a este problema y mostraremos con un ejemplo completo la forma de resolverlo.

Sensibilidad a mayúsculas y minúsculas

Cada lenguaje especifica si existe diferencia entre un identificador escrito en mayúsculas y en minúsculas. Si lo que queremos es que no distinga las mayúsculas y minúsculas una solución es:

Se ponen los lexemas de la tabla de palabras reservadas en mayúsculas. Antes de nigun procesamiento el lexema actual se pone en mayúsculas y luego se busca en la tabla correspondiente. Suponemmos que tenemos una funcion

```
void     mayusculas(char *t);
```

que trasforma t a mayúsculas.

Si queremos que sea sensible a mayúsculas y minúsculas no se usa la función anterior.

Longitud de los identificadores

En general, tendremos un buffer que nos permite lexemas todo lo grande que queramos. Si en el lenguaje se especifica que los identificadores no pueden se mas largos de MAXL entonces lo que haremos es cortar el lexema hasta long=MAXL para terminar buscando el lexema truncado en las diferentes tablas. Suponemos que tenemos disponible la función.

```
void     truncar(char *t,MAXL);
```

De todas formas, en la actualidad es poco habitual que los lenguajes modernos limiten la longitud máxima que pueden tomar los identificadores. Generalmente esta tan sólo queda limitada por la cantidad de memoria que quede disponible, por lo que es bastante recomendable guardar los lexemas de todos los identificadores que encontremos en el fuente en un buffer en el que los lexemas que son iguales comparten el mismo espacio de almacenamiento.

Problemas con los numeros

Se debe al hecho de que los lenguajes admiten diferentes representaciones de los números (entero, real,...). En vez de usar el token NUM utilizaremos un token para cada uno de los distintos tipos de números: enteros, reales, doble, etc. El problema es la notación que escojemos para escribir los enteros, reales, etc.

La solución más cómoda es adoptar las convenciones del C, es decir, adoptar expresiones tal como vendrían definidas en C. Como el C será el lenguaje en el que escribiremos nuestros compiladores, podremos utilizar las funciones atoi() y atof() de C para calcular el valor a partir del lexema.

Las expresiones regulares mas usuales para describir los números son:

```
{digito}+[uUlL]?              CONSTANTE_ENTERA;
{digito}+\.{digito}*{exp}     CONSTANTE_FLOAT;
\.{digito}+{exp}              CONSTANTE_FLOAT;
```

Tratamiento de comentarios

Con los comentarios los problemas a evitar son:

Detectar los fines de línea para mantener el número de línea actual. Evitar que en las implementaciones pueda aparecer un desbordamiento debido a que el lexema asociado al comentario sea demasiado largo. Estos problemas se pueden evitar con la expresiones regulares y acciones asociadas siguientes:

```
"/*"                          BEGIN(COMENTARIO);

<COMENTARIO>"*/"              BEGIN(0);
<COMENTARIO>[^*\n]+           ;
<COMENTARIO>\n                aumentar en uno el numero de lineas;
<COMENTARIO>"*"               ;
```

En este caso la solución ha sido usar dos analizadores léxicos: uno que funciona fuera de los comentarios y otro dentro de los mismos. Con `BEGIN(COMENTARIO)` cambiamos a este analizador léxico. Con `BEGIN(0)` vovlvemos al analizador léxico por defecto.

Tratamiento de las Hileras de Caracteres

El problema de la hileras de caracteres es encontrar una expresión regular que las describa adecuadamente. Una posibilidad es la expresión regular:

```
\"([^"\\\n]|{esc})*\"
```

Capítulo 4

La herramienta lex

`lex` es una herramienta informática que traduce la especificación de un analizador léxico a un programa escrito en C que lo implementa.

Para especificarlo usaremos, en esencia, expresiones regulares a las que se puede asociar acciones escritas en C. Cada vez que el analizador encuentra en la cadena de entrada un prefijo que encaja en una de las expresiones regulares especificadas, ejecutará la acción que le hallamos asociado.

El principal problema que se suele achacar a `lex` es el hecho de que genera programas poco eficientes, por lo que su aplicación en entornos de producción industrial se ve claramente menguada. En la actualidad, se ha desarrollado una versión que genera un código final más eficiente, pero mantiene una completa compatibilidad con la herramienta original. Se trata de `flex`, un producto GNU de dominio público del que existen versiones para UNIX y MS-DOS.

4.1 Esquema de un fuente en `lex`

Un fuente en `lex` se divide en las tres zonas que se muestran a continuación:

(zona de definiciones)
%%
(zona de reglas y acciones)
%%
(zona de rutinas de usuario)

El símbolo **%%** actúa como separador entre las tres secciones. Las dos primeras son obligatorias, aunque pueden aparecer vacías, y la tercera es opcional, de forma que en caso de omitirse no es necesario añadir el segundo separador **%%**.

El fuente `lex` más pequeño es el siguiente:

```
%%
```

No incluye ninguna regla y tan sólo reconoce el lenguaje formado por la palabra vacía.

4.1.1 Zona de definiciones

En esta sección se incluye todo el código C necesario para expresar las acciones asociadas a las expresiones regulares, se definen macros y también se definen entornos de reconocimiento. Veremos más adelante qué es lo que significa esto último.

Un ejemplo sencillo es el siguiente:

```
%{
    /* Un ejemplo en lex */

    int y = 0;          /* Una variable */
    char * f(int *);    /* Un prototipo */

    #include <stdio.h>
%}


digito    (0|1|2|3|4|5|6|7|8|9)
letra     (a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z)
ident     {letra}({letra}|{digito})*
```

La porción de texto entre los símbolos %{ y %} se considera que es código C, pero `lex` no lo tiene en cuenta. Lo copia íntegramente en el fichero de salida, por lo que si cometemos algún error en esta zona tan sólo se detectará al compilar el fichero resultado producido por `lex`.

Las macros se definen usando el siguiente formato:

(nombre) *(expresión regular)*

De esta forma, `digito` se considera como un nombre de macro equivalente a la expresión regular `(0|1|2|3|4|5|6|7|8|9)`. Para usarlas es preciso encerrar sus nombres entre llaves, como se muestra en el caso de la definición de ident. Suele ser un error frecuente entre los principiantes definir esta macro de la siguiente forma:

```
ident    letra(letra|digito)*
```

Pero si se hace así la macro equivaldría a una expresión regular que reconoce las palabras {`letra, letraletra, letradigito, letraletraletra,` ...}, pues si no utilizamos las llaves entonces `lex` no lleva a cabo sustitución alguna.

No hemos incluido de momento ningún ejemplo de entorno de reconocimiento puesto que los trataremos con mucho detalle más adelante.

4.1.2 Zona de reglas

Esta sección tiene el siguiente formato:

1a columna	2a columna en adelante
	(Código C)
(exp_reg$_1$)	*(acción$_1$)*
...	...
(exp_reg$_n$)	*(acción$_n$)*

Cada línea de la forma

(exp_reg) (acción)

recibe el nombre de regla, y es muy importante que escribamos el primer carácter de la expresión regular en la primera columna de texto, pues de lo contrario se consideraría que se trata de un trozo de código C.

De manera esquemática, el fichero que `lex` produce al compilar un fuente tiene la siguiente estructura:

(prototipos)
(código global)

```
int yylex(void) {
      (código local)
      (simulación del autómata)
}
```

(rutinas de usuario)
(tablas de transición del AFD)

`yylex()` es el nombre de la rutina que simula el analizador léxico especificado. Su prototipo estándar es el que se ha indicado y además no se puede cambiar.

La zona de código global es en la que `lex` coloca todo el código C que se ha escrito en la zona de definiciones del fuente y la de código local es en la que copia el código que se escribe al comienzo de la sección de reglas sin asociarlo a ninguna expresión regular. Por lo tanto, este código se ejecuta cada vez que se llama a la rutina `yylex()` para analizar un lexema de la entrada.

Veamos un ejemplo más completo. En él pretendemos especificar un analizador que reconozca cadenas de identificadores y números naturales separados por guiones. Se supondrá que la cadena termina con un signo `$` y se espera que al encontrarlo el analizador imprima el número de identificadores y de literales numéricos encontrados.

La especificación en lenguaje `lex` es la siguiente [1]:

```
%{
    #include <stdio.h>
```

[1] Los puntos suspensivos que aparecen en la expresión regular no forman parte del lenguaje `lex`. Indican que Vd. debe completar la secuencia manualmente, escribiendo todas las letras entre la `a` y la `z` y entre la `A` y la `Z`.

```
    int num_id = 0, num_nat = 0;
%}

letra    (a|b|c|...|z|A|B|C|...|Z)
digito   (0|1|2|...|9)

ident    {letra}({letra}|{digito})*
natural  {digito}{digito}*

%%

-           ;
{ident}     num_id++;
{natural}   num_nat++;
"$"         printf("\nid=%d, nat=%d\n", num_id, num_nat);
```

Con lo que hemos visto hasta ahora no le resultará difícil entender el significado de este programa. Tan sólo destacaremos que el carácter `$` tiene un significado especial en `lex` y para que sea interpretado literalmente hay que entrecomillarlo.

Si compilamos esta especificación y producimos un ejecutable, éste tomará la cadena a reconocer de la entrada estándar. Si contiene:

```
123-456-789-abc-def$
```

la salida que se producirá será:

```
id=2, nat=3
```

Las acciones que hemos escrito en este ejemplo son sencillas, puesto que tan sólo involucran una sentencia en C. Cuando la acción involucra varias sentencias es conveniente escribir cada una de ellas en una línea independiente, pero entonces hay que encerrarlas entre llaves, al estilo del lenguaje C. Por ejemplo:

exp_reg { *acción1; acción2;* ... *acciónN;* }

En otras ocasiones, varias expresiones regulares comparten la misma acción. En estos casos se pueden emplear reglas de la siguiente forma:

exp_reg1 |
exp_reg2 |
⋮ ⋮
eexp_regN *acción_común*

El signo `|`, al utilizarse como una acción para la primera expresión regular, indica que se debe tomar como acción para esta regla la misma que se especifique para la segunda expresión regular. Para esta se tomará la misma que para la tercera expresión, etcétera.

4.1.3 Zona de rutinas de usuario

En esta zona se puede escribir código C adicional necesario para compilar la especificación del analizador.

En la práctica, casi no haremos uso de esta sección, puesto que introducir rutinas o variables en ella dificulta enormemente la depuración y la modificación del analizador léxico.

4.2 Cómo se compila un fuente `lex`

`lex` es una herramienta estándar en todos los sistemas UNIX. El proceso que describiremos en esta sección muestra cómo se compila un fuente para `lex` en AIX y al final del capítulo mostraremos las diferencias más importantes con respecto a otros sistemas operativos.

Para tratar una especificación es preciso introducirla en un fichero con extensión `.l`. En el último ejemplo que vimos, este fichero podría ser `nat_id.l`. Una vez creado se utiliza el comando

```
$ lex nat_id.l
```

para compilarlo. Si el fuente contiene errores se mostrarán en la salida estándar de error. En caso de no haber ninguno, `lex` produce un fichero de nombre `lex.yy.c` con la estructura que vimos antes.

Este fichero se puede compilar usando un compilador ANSI C, y lo normal es introducir el código objeto resultado en un archivo de biblioteca para su uso posterior.

4.2.1 Opciones estándar

Las opciones más habituales de esta herramienta son las que se muestran a continuación:

- `-t`, para que `lex` produzca su resultado en la salida estándar en vez de en el fichero `lex.yy.c`. Es útil cuando esta herramienta se utiliza en conjunción con otras.
- `-v`, instruye a `lex` para que muestre en la salida estándar cierta información acerca del número de estados que tiene el autómata que ha generado, el número de transiciones, etcétera.
- `-f`, realiza un proceso de compilación rápida. Generalmente no optimiza el código resultado, ni minimiza el autómata que genera al usar esta opción.

Estas opciones están bastante estandarizadas y la mayoría de las implementaciones de la herramienta disponen de ellas, pero lo mejor es consultar el manual de aquella que estemos utilizando.

4.2.2 Depuración de un fuente en lex

lex incluye en el código que genera alguna ayuda para la depuración. Para conseguirla es necesario definir la macro LEXDEBUG en la zona de código C global. Al hacerlo, lex genera un código que al ejecutarse nos da información detallada sobre el estado en que se encuentra el autómata, la transición que se va a realizar, etcétera.

4.2.3 Prueba rápida de un fuente

Una forma de probar rápidamente el resultado de compilar un fuente lex, es utilizando la biblioteca estándar libl.a. Esta contiene, entre otras cosas, la función:

```
int main(void) {
    while (yylex())
        ;
    return 0;
}
```

De esta forma, el comando:

```
$ cc -ll lex.yy.c
```

compila el fichero lex.yy.c enlazándolo con la biblioteca libl.a, con lo que el resultado es un fichero ejecutable que invoca al analizador léxico en repetidas ocasiones hasta que se detecte el final de la cadena de entrada. En ese momento la función yylex() devuelve el valor 0 y termina la ejecución de la rutina main().

4.3 Expresiones regulares en lex

En lex todas las expresiones regulares que escribamos definen lenguajes sobre el juego completo de caracteres que use nuestra máquina (ASCII, EBCDIC, CDC, ...).

No existe forma de escribir la expresión regular λ ni tampoco $\emptyset$, pero ofrece como contrapartida un conjunto muy amplio de operadores.

4.3.1 Caracteres

Cualquier carácter a que no sea un operador se considera en lex una expresión regular cuyo lenguaje asociado es $\{a\}$. Los caracteres que tienen un significado especial y se consideran operadores son todos los siguientes:

```
"  [ ] ^ - ? . * + | ( ) $ / { } % < >
```

Si queremos usar alguno de estos operadores sin su significado especial y utilizarlos como cualquier otro carácter es necesario precederlos de una *barra mariquita* o escribirlo entre comillas dobles. Por ejemplo:

```
\$ \" "["
```

Las comillas también se pueden emplear para encerrar cadenas largas de caracteres. Por ejemplo:

```
"[ser|^ser]"
```

`lex` reconoce las siguientes secuencias de escape estándar:

- `\a` para hacer sonar el altavoz del terminal.
- `\b` para hacer retroceder el cursor un carácter hacia atrás en el terminal.
- `\f` para provocar un avance de página en la impresora y un borrado de pantalla en el terminal.
- `\n` para avanzar a la siguiente línea de texto.
- `\t` para avanzar hacia la siguiente posición de tabulación horizontal.
- `\v` para avanzar hacia la siguiente posición de tabulación vertical.

La última secuencia de escape es estándar, pero son muy pocas las implementaciones de la herramienta que la reconocen.

4.3.2 Clases de caracteres

Una clase elemental de caracteres es la expresión regular

`[`$ab\ldots c$`]`

que tiene como lenguaje asociado el conjunto $\{a, b, \ldots, c\}$.

Dentro de las clases es posible utilizar rangos de caracteres de la forma:

`[`a`-`b`]`

Esta expresión describe el lenguaje formado por todos los caracteres que se encuentran entre a y b. Por ejemplo,

```
[a-zA-Z]
```

Describe el lenguaje formado por cualquier letra mayúscula o minúscula. Pero es preciso matizar un poco más esta definición, puesto que `lex` es dependiente del juego de caracteres que utilice nuestra máquina y por tanto las clases de caracteres también lo son. Eso significa que si en nuestra máquina entre la `a` y la `z` tan sólo hay letras minúsculas, y entre `A` y la `Z` tan sólo hay letras mayúsculas, entonces la anterior clase sólo valida letras.

Pero si en nuestra máquina entre la `a` y la `z` está el carácter §, éste también será reconocido por la clase.

Las clases que hemos visto se suelen llamar positivas, en contraposición con las clases negativas que tienen la forma

`[^`$ab\ldots c$`]`

y describen el lenguaje formado por cualquier carácter que no sea ninguno de los indicados. Por ejemplo,

```
[^0-9]
```

valida cualquier carácter que no sea un dígito. Conviene hacer la misma puntualización de antes: estas clases también son dependientes del juego de caracteres de la máquina, de forma que si en la nuestra las letras se encuentran entre los dígitos, esta expresión también descartará todas las letras.

Dentro de las clases pierden su característica especial de operador los siguientes caracteres, que se podrán utilizar sin necesidad de precederlos de una barra invertida o escribirlos entre comillas:

```
? . * + | ( ) $ / { } % < >
```

4.3.3 Cómo reconocer cualquier carácter

En muchas ocasiones necesitamos reconocer cualquier carácter. Para ello podemos utilizar la expresión regular

```
.
```

que describe el lenguaje $\Sigma_M - \{\backslash n\}$, esto es, cualquier carácter del juego de nuestra máquina salvo el retorno de carro. Por ejemplo:

- `a.` reconoce las cadenas {`aa`, `ab`, `ac`, ... }.
- `a.*` reconoce cualquier cadena que empiece por la letra `a` y no contenga un retorno de carro.
- `a.b` reconoce cualquier cadena de tres caracteres que empiece por la letra `a` y termine por una `b`.
- `.|\n` reconoce cualquier carácter del alfabeto Σ_M.

4.3.4 Unión

El operador de unión se representa utilizando una barra vertical `|`. La única nota de interés es que el reconocedor sintáctico que genera `lex` intenta validar siempre el lexema más largo que pertenezca al lenguaje de las expresiones regulares que combinemos mediante este operador. De esta forma, una expresión como

```
ab|abc
```

ante una cadena de entrada `abcdef` podría validar tanto el prefijo `ab` como el prefijo `abc`, pero `lex` resuelve la ambigüedad validando siempre el más largo que sea posible. En este caso `abc`.

4.3.5 Clausuras

`lex` ofrece la posibilidad de utilizar dos formas de clausura: la de Kleene, que se denota usando el símbolo `*`, y la positiva que se denota usando el signo `+`. Por ejemplo:

- `[0-9]+` denota literales naturales [2].
- `[a-z][a-z0-9]*` denota identificadores que empiezan por una letra minúscula y continúan con cero, una o más letras minúsculas y dígitos.
- `[0-9]+"."[0-9]+` denota literales reales en notación de coma fija.

4.3.6 Repetición acotada

`lex` soporta la posibilidad de utilizar expresiones regulares que denotan la repetición de un cierto elemento un número exacto de veces. La nomenclatura que utiliza es la siguiente:

(exp_reg{n, m})

Esta expresión denota el lenguaje de aquellas palabras que encajan en la expresión exp_reg y están concatenadas unas con otras entre n y m veces. Por ejemplo:

- `a(b{3,3})` denota el lenguaje {`abbb`}.
- `a(b{1,3})` denota el lenguaje {`ab, abb, abbb`}.
- `[a-z]([a-z]{0,32})` denota el lenguaje formado por las palabras en minúsculas que tienen entre 1 y 33 caracteres [3].

[2] A partir de ahora supondremos que en el juego de caracteres de nuestra máquina no existe ningún caracter extraño entre las letras ni entre los números. De hecho esta asumpción es bastante sensata, pues tanto el juego de caracteres ASCII, como el EBCDIC y el CDC, que son los más utilizados, cumplen esta propiedad. Los juegos de caracteres que no la cumplen están en desuso.

[3] En algunas implementaciones de `lex` no se permite que el mínimo número de repeticiones sea nulo.

4.3.7 Opción

`lex` también proporciona el operador de opción, que se denota utilizando el signo `?`. Una expresión regular como

```
exp_reg?
```

describe el lenguaje formado por la unión de la palabra vacía y el lenguaje descrito por la expresión regular sobre la que actúa. Por lo tanto, podríamos ver este operador como uno que reconoce 0 ó 1 apariciones de las cadenas que encajan en la expresión `exp_reg`.

Por ejemplo, dadas las macros:

```
lit_nat   ([0-9]+)
exp       ([Ee])
sig       ([+])
```

la expresión regular

```
{sig}?{lit_nat}("."{lit_nat})?({exp}{sig}?{lit_nat})?
```

reconoce números reales y enteros con signo escritos en coma fija o en notación científica.

4.3.8 Paréntesis

En los ejemplos anteriores hemos visto que los paréntesis nos ayudan a cambiar la prioridad por defecto de los operadores y a agrupar expresiones para que resulten más fáciles de leer.

Su uso en las macros es fundamental, puesto que éstas se sustituyen allá donde aparecen literalmente. Esto significa que dada la macro

```
letra    a|b|c|...|z
```

la expresión regular

```
palabra    {letra}+"$"
```

no reconocerá cadenas de una, dos o más letras terminadas en el signo `$`, puesto que la sustitución que se realiza es literal. La macro `palabra` equivale a

```
a|b|c|...|z+"$"
```

por lo que el operador de clausura tan sólo actual sobre la última letra. El lenguaje descrito por esta expresión es {`a, b, ..., z, z$, zz$, zzz$, ...`}.

Para evitar problemas en la sustitución de las macros, lo mejor es colocar paréntesis alrededor de todas las definiciones.

4.4 Tratamiento de las ambigüedades

Una especificación escrita en `lex` puede ser ambigua por varias razones. Encontraremos una de ellas al analizar el siguiente ejemplo en el que el objetivo es especificar un analizador que distinga en el fuente entre identificadores y palabras reservadas de un lenguaje sencillo. En analizador deberá devolver en cada llamada a `yylex()` un valor entero que indique de qué tipo es la palabra reconocida.

Suponiendo que las únicas palabras reservadas son `if`, `then` y `else`, el fuente podría ser el siguiente:

```
%{
    #define PAL_RES 1
    #define IDENT   2
%}

ident    ([a-zA-Z][a-zA-Z0-9]*)

%%

if       |
then     |
else     return PAL_RES;
{ident}  return IDENT;
```

Supongamos que suministramos como entrada la palabra

```
then
```

¿Qué acción debe ejecutarse?. Es preciso darnos cuenta de que esta palabra encaja tanto en la expresión regular `then` como en `ident`, de forma que en principio ambas acciones podrían llevarse a cabo. `lex` soluciona la ambigüedad usando la siguiente regla:

> **Regla 1:** Si dos expresiones regulares reconocen en la entrada un prefijo de igual longitud, se ejecuta siempre la acción asociada a la primera expresión regular escrita.

Por lo tanto, al suministrar al reconocedor el fuente anterior, `yylex()` devolverá el valor `PAL_RES`.

Otra ambigüedad aparece al suministrar a este analizador la cadena de entrada

```
ifthenelse
```

Esta entrada se puede considerar compuesta por la concatenación de tres prefijos diferentes, correspondientes a tres palabras reservadas, o como una única palabra que encaja en la expresión regular `ident`. La regla que `lex` utiliza para resolver esta ambigüedad es la siguiente:

> **Regla 2:** Cuando se presentan varias posibilidades a la hora de reconocer una palabra, se opta siempre por aquella que reconozca en la entrada una palabra de mayor longitud.

En nuestro caso, una llamada a `yylex()` devolvería el valor `IDENT`.

4.5 Dependencias del contexto

`lex` ofrece varios mecanismos sencillos para resolver dependencias del contexto, por lo que más allá de un simple generador de analizadores de lenguajes regulares, se puede usar para reconocer otros lenguajes algo más complejos.

4.5.1 Dependencias simples

Estas dependencias simples están en relación a en qué lugar dentro de una línea aparece el texto que encaja en una expresión regular. De esta forma:

- `^exp_reg` valida sólo palabras descritas por `exp_reg` cuando aparecen al principio de una línea.
- `exp_reg$` valida las palabras descritas por `exp_reg` tan sólo cuando aparecen al final de una línea.
- `^exp_reg$` valida las palabras descritas por `exp_reg` tan sólo cuando aparecen aisladas dentro de una línea de texto.
- `^$` es una expresión regular especial que valida tan sólo aquellas líneas en las que no hay nada escrito.

De esta forma, la expresión

```
^#(include|define|undef|if|ifdef|ifndef|else|elsif)
```

validará algunas de las directivas del preprocesador del lenguaje C, pero sólo cuando aparecen al principio de una línea.

Es muy importante hacer hincapié en que las expresiones regulares `exp_reg$` y `exp_reg\n` no son equivalentes, puesto que la primera reconoce una palabra que encaje en la expresión regular, pero sin tener en cuenta el retorno de carro, mientras que la segunda también valida una palabra al final de una línea, pero incluyendo el retorno de carro.

Otra forma de contexto simple es la que proporciona el operador `/`. Una expresión de la forma

```
exp_reg1/exp_reg2
```

valida cadenas que encajan en la primera expresión regular, pero sólo cuando van seguidas en la entrada por una cadena que encaja en la segunda expresión regular. Esta segunda cadena no forma parte del prefijo reconocido y se mantiene en la entrada. De esta forma, `exp_reg$` se puede considerar casi equivalente a `exp_reg/\n`.

4.5.2 Dependencias complejas

Las dependencias complejas del contexto se resuelven utilizando entornos de reconocimiento. Para confinar una regla a un entorno se utiliza la siguiente sintaxis:

`<E1,E2,...,En>` *exp_reg*

Esto significa que dada una cadena de entrada, tan sólo se intentará determinar si encaja en la expresión regular indicada cuando nos encontremos en alguno de los entornos `E1`, `E2`, ... o `En`. Al comenzar el análisis nos encontramos siempre en el entorno predefinido de nombre `INITIAL`. Cualquier regla no confinada a un entorno se considera dentro de éste y se tiene en cuenta siempre, aunque explícitamente cambiemos de entorno. Volveremos a este tema un poco más adelante.

`E1, E2, ...` y `En` son nombres arbitrarios formados por letras y es preciso declararlos en la sección de definiciones del fuente utilizando la sintaxis siguiente:

```
%s E1 E2 ... En
```

Para cambiar de entorno explícitamente es preciso utilizar la acción especial `BEGIN(E)` [4]. Esta debe aparecer aislada o de lo contrario se ignorará a menos que toda la acción se encierre entre llaves. Por ejemplo:

```
<E>a        num_a++; BEGIN(F);
```

se considera exactamente equivalente a

```
<E>a        num_a++;
```

Para que `lex` no ignore esta sentencia especial es necesario encerrar toda la acción entre llaves, tal y como se muestra a continuación:

```
<E>a        { num_a++; BEGIN(F); }
```

Como ejemplo de uso de los entornos veremos la especificación de un analizador que preprocesa un fuente en C eliminando de él todos los comentarios.

```
%{
    #include <stdio.h>
%}

%s COM COD
```

[4] Muchas implementaciones de `lex` no permiten la acción `BEGIN(INITIAL)`. En estos casos hay que emplear la versión, menos intuitiva, pero más portable `BEGIN(0)`.

```
%%

                BEGIN(COD);

<COD>"/*"       BEGIN(COM);
<COD>.|\n       ECHO;

<COM>"*/"       BEGIN(COD);
<COM>.|\n       ;
```

En este fuente, `COD` representa el entorno en el que lo que se reconoce de la entrada es código, y `COM` el entorno en el que se reconocen los comentarios.

Inicialmente nos encontraremos en el entorno `COD` y cualquier carácter que encontremos en él, salvo la cadena `/*`, se mostrará en la salida estándar. `ECHO` es una macro predefinida que vuelca en la salida estándar el último prefijo reconocido. Para poder usarla es preciso añadir el fichero de cabecera `stdio.h`.

Cuando en el entorno de código encontremos la secuencia `/*` lo abandonaremos y pasaremos al entorno `COM` en el que ignoraremos todos los caracteres que encontremos hasta leer la cadena `*/`, que nos devolverá al entorno de código.

Es muy importante resaltar que todas las expresiones regulares del entorno `INITIAL` se pueden reconocer sea cual sea el entorno en el que nos encontremos.

Por ejemplo, al compilar la siguiente especificación se obtiene un analizador léxico que elimina todo lo que aparezca entre comentarios salvo aquellas cadenas que abarcan desde el símbolo `//` hasta el final de una línea.

```
%{
    #include <stdio.h>
%}

%s COM COD

%%

                BEGIN(COD);

"//".*\n        ECHO;

<COD>"/*"       BEGIN(COM);
<COD>.|\n       ECHO;

<COM>"*/"       BEGIN(COD);
<COM>.|\n       ;
```

Si enfrentamos a este analizador al texto

```
abc
```

```
/*
    cde
    // efg
*/
ghi
```

la salida que producirá será la siguiente:

```
abc
    // efg
ghi
```

pues la regla con la expresión regular `"//".*\n` tiene efecto dentro de cualquier entorno, no sólo dentro de `INITIAL`.

4.6 Variables predefinidas por `lex`

`lex` define varias variables estándar que le sirven de interface. Las estudiaremos en esta sección.

- `yytext`, es un puntero a una cadena de caracteres que contiene la última porción de texto validada por una expresión regular. Está declarada como `char *yytext`, de forma que podríamos modificar su contenido, pero no debemos hacerlo bajo ningún concepto. Apunta a una zona de memoria temporal cuyo contenido tan sólo es estable dentro de las reglas y entre llamadas consecutivas a la función `yylex()`.
- `yyleng`, contiene la longitud de la cadena `yytext`. Tampoco se debe modificar.
- `yyin`, es una variable declarada del tipo `FILE *`. Contiene un puntero al fichero con la cadena de entrada que pretendemos analizar en cada momento. Se puede cambiar libremente y de esta forma podemos conseguir que el analizador lea de varios ficheros distintos de entrada.
- `yyout`, es una variable declarada del tipo `FILE *`. Contiene un puntero al fichero estándar en el que escribe el analizador léxico al utilizar la acción `ECHO`. Se puede cambiar su valor libremente.

A continuación veremos un ejemplo muy sencillo. En él estudiaremos la especificación de un analizador que elimina todas las palabras que encuentra en su entrada y las sustituye por una cadena de la forma `(s, n)`, en donde `s` es la palabra leída y `n` su longitud. El nombre del fichero de entrada y de salida se pedirá al usuario por teclado.

```
%{
        #include <stdio.h>
%}

pal     ([a-zA-Z]+)
```

```
%%

{pal}    fprintf(yyout, "(%s, %d)", yytext, yyleng);
.|\n    ECHO;

%%

FILE *abrir_fichero(const char *nf, const char *m)
{
    FILE *p = fopen(nf, m);

    if (p == NULL)
        perror(nf);
    return p;
}

void main(void)
{
    char nfe[100], nfs[100];

    printf("Entrada: "); scanf("%s", nfe);
    printf("Salida: ");  scanf("%s", nfs);

    yyin = abrir_fichero(nfe, "r");
    yyout = abrir_fichero(nfs, "w");
    if (yyin != NULL && yyout != NULL)
        yylex();
}
```

4.7 Acciones predefinidas

`lex` predefine un cierto número de acciones y rutinas, algunas de las cuales pueden ser redefinidas por el usuario para adaptarlas a sus necesidades. De todas formas, modificar estas rutinas requiere de unos conocimientos profundos sobre cada implementación concreta de la herramienta, de forma que remitimos al lector interesado a los manuales correspondientes. En esta sección tan sólo trataremos su significado.

- `ECHO`, es una macro cuya definición estándar es

  ```
  #define ECHO fprintf(yyout, "%s", yytext)
  ```

  Cuando en la cadena de entrada se encuentra un carácter no reconocido por ninguna de las reglas especificadas, `lex` lleva siempre a cabo esta acción por defecto.

- `BEGIN(E)`, cambia del entorno actual de análisis al de nombre `E`.

- `output(c)`, escribe en el fichero yyout el carácter `c`.

- `unput(c)`, devuelve el carácter `c` al fichero `yyin`. Tan sólo se garantiza el poder devolver con éxito un carácter.
- `input()`, retira un carácter del fichero `yyin` y devuelve su valor como un número entero.

Todas estas acciones predefinidas son bastante elementales. En las secciones siguientes trataremos algunas más complejas.

4.7.1 yyless(n)

En ocasiones, una vez reconocido un lexema, resulta que no necesitamos todos los caracteres que lo forman. Al ejecutar `yyless(n)` nos quedamos con los `n` primeros caracteres del prefijo reconocido y se devuelven los restantes al fichero de entrada.

Un ejemplo histórico de utilización de esta acción fue al crear analizadores sintácticos para las primeras versiones del lenguaje C. En ellas, una expresión como `x=-a` resultaba ambigua puesto que se podía interpretar como `x=(-a)` o también como la expresión `x =- a`, en donde el signo `=-` era la versión antigua del actual operador de decremento y asignación. Lo que se hacía en aquellos tiempos era considerar que en estos casos, el programador realmente había querido escribir la segunda interpretación, pero se daba siempre un mensaje de aviso.

Para conseguir este comportamiento, podemos utilizar la siguiente especificación:

```
"=-"[a-zA-Z_]  {
                 fprintf(stderr, "=- es ambiguo");
                 yyless(2);
                 ...
               }
```

Los caracteres que se devuelven a la cadena de entrada se volverán a analizar.

4.7.2 yymore()

Generalmente, cuando se valida una expresión regular, se borra el contenido previo de `yytext`. Si queremos que esto no ocurra y que el siguiente lexema validado se añada a `yytext` es preciso hacer una llamada a `yymore()`.

Por ejemplo, supongamos que tenemos que reconocer cadenas de caracteres encerradas entre comillas. El carácter comilla puede aparecer dentro de la cadena precedido de una barra invertida. Una forma de solucionar este problema es utilizando el siguiente fuente:

```
%{
    #include <stdio.h>
%}

%%
```

```
\"[^\"]*\"  {
                if (yytext[yyleng - 2] == '\ ')
                    yyless(yyleng - 1), yymore();
                else
                    ECHO;
            }
.|\n        ;
```

4.7.3 REJECT

Como hemos visto hasta ahora, el analizador generado por `lex` no intenta encajar un prefijo de entrada en todas las posibles expresiones regulares que escribamos. Esto es un problema si, por ejemplo, deseamos contar cuántas veces aparecen las secuencias de letras `el` y `ella` en un texto. La especificación

```
"el"        num_el++;
"ella"      num_ella++;
.|\n        ;
```

no es correcta, pues al enfrentar el analizador a la cadena

```
ellaella
```

diría que `ella` aparece dos veces y el no aparece ninguna, cuando realmente las dos secuencias aparecen en dos ocasiones. La solución es que cada vez que se reconozca ella se intente encajar el prefijo reconocido en otra expresión, y para esa función podemos utilizar la acción `REJECT`.

```
"el"        num_el++;
"ella"      { num_ella++; REJECT; }
.|\n        ;
```

De esta forma cada vez que se encuentre la cadena `ella` en el fuente se incrementará la variable `num_ella` y se ejecutará la acción `REJECT` que devolverá todo el texto leído al fichero de entrada e intentará encajarlo en la expresión regular de otra regla distinta. En este caso la única que se activará será la que reconoce la palabra `el`.

Veamos otro ejemplo. En él intentaremos contar el número de veces que aparece la palabra `el`, la palabra `ella` y el número de pares consecutivos de letras en una cadena terminada con un signo `$`.

```
%{
    #include <stdio.h>
    int num_el = 0, num_ella = 0, num_par = 0;
%}
```

```
%%

\$                fprintf(yyout, "el=%d, ella=%d, par=%d", \
                          num_el, num_ella, num_par);

el                { num_el++; REJECT; }
ella              { num_ella++; REJECT; }
[a-zA-Z][a-zA-Z]  { num_par++; REJECT; }
.|\n              ;
```

Cuando `REJECT` forma parte de una secuencia de acciones, debe aparecer en último lugar y la secuencia completa entre llaves. Como nota final, indicaremos que `REJECT` y `BEGIN` son acciones incompatibles dentro de la misma regla.

4.7.4 yywrap()

Esta es una macro/función que debe devolver un valor que indique al analizador generado por `lex` qué hacer cuando alcanza el final del fichero de entrada. El algoritmo que lo ayuda a decidir es el siguiente:

```
if (yywrap())
    return 0; /* Final del análisis léxico */
else
    <Continuar con el análisis de yyin>
```

Puede parecer extraño que si `yywrap()` devuelve cero se intente continuar con el análisis del fichero de entrada. Lo que ocurre realmente es que esta función puede cambiar el fichero al apunta la variable `yyin`, con lo cual tras su evaluación el reconocedor puede continuar en un fichero completamente distinto.

Esta función se suministra en la biblioteca estándar `libl.a`, y por eso cuando no la enlazamos en nuestro ejecutable es preciso definirla de forma explícita. Generalmente basta con utilizar la siguiente macro:

```
#define yywrap() 1
```

Pero si estamos implementando un preprocesador que maneje ficheros de inclusión, su definición podría ser bastante más compleja, como se muestra en el siguiente esquema:

```
%{
    /* Esquema de un preprocesador */

    int yywrap(void);
    ...
%}
...
```

```
%%
...
{include}        <tratar la directiva include>
...
%%
...
int yywrap()
{
    int r = 0;

    if (<este fichero recién leído es de inclusión>)
        yyin = <fichero que lo incluía>;
    else
        yyin = NULL, r = 1;

    return r;
}
```

4.8 Algunas implementaciones comerciales de `lex`

En este capítulo hemos descrito básicamente la versión de esta herramienta que acompaña al sistema operativo AIX, que es bastante estándar. En esta sección estudiaremos algunas versiones más y sus principales diferencias con la que hemos descrito.

4.8.1 PC `lex`

Esta es la implementación de `lex` que proporciona Abraxas Software para correr en el sistema operativo MS-DOS.

Entre sus características más negativas destaca el hecho que tan sólo admite caracteres ASCII de 7 bits, y si se le suministra un fuente con algún carácter con el bit 8 alto, se cicla. Las versiones más modernas han solucionado este problema.

También ofrece entornos de evaluación exclusivos, que se declaran utilizando la opción `%x`. Dentro de los entornos exclusivos nunca se tienen en cuenta las reglas escritas dentro del entorno `INITIAL`. Los operadores `^` y `$` no se pueden utilizar en la definición de macros, tan sólo en la sección de reglas. Si se utilizan para definir una macro, se interpretan como si se tratase de caracteres normales.

4.8.2 Sun `lex`

Esta es la implementación de la herramienta ofrecida por Sun Microsystems. Su diferencia más importante es que la función `yywrap()` no se encuentra en la biblioteca estándar `libl.a`, por lo que debe ser definida siempre por el usuario. Tampoco admite la expresión regular `^$` para reconocer líneas de texto vacías.

4.8.3 Ejemplos

En esta sección mostraremos varios ejemplos de especificaciones léxicas.

4.8.4 Ejemplo 1

Obtener un programa en lenguaje `lex` que dado un fichero de entrada ofrezca como salida un fichero en el que se han eliminado de la entrada las líneas en blanco, los espacios en blanco al principio o al final de línea y que tan sólo deja un blanco entre cada dos palabras consecutivas.

```
%{
        #include <stdio.h>
%}

%%

^$          ;
^" "+\n     ;
^" "+       ;
" "+$       ;
" "+        putchar(' ');
```

4.8.5 Ejemplo 2

Obtener un programa en `lex` que pase a minúsculas un fichero de texto.

```
%{
    #include <ctype.h>
    #include <stdio.h>
%}

%%

[A-Z]    putchar(tolower(yytext[0]));
```

4.8.6 Ejemplo 3

Obtener un programa fuente `lex` que manipule un fichero de texto en el que cada línea está compuesta de varios enteros y justo antes del fin de línea aparece el carácter `S` o `N` (sin tener en cuenta si está en mayúscula o en minúscula).

El programa debe generar un fichero de salida que deje inalteradas las líneas que terminen en `N` y sume uno a todos los números que aparecen en las líneas terminadas en `S`.

```
%{
```

```
    #include <stdio.h>
    #include <string.h>
%}

S       (" "+[sS]" "*)
N       (" "+[nN]" "*)
nat     ([0-9]+)
lnat    (" "*nat(" "+nat)*)

%s NORMAL MODIF

%%

                        BEGIN(NORMAL);

<NORMAL>{lnat}/{S}      { BEGIN(MODIF); yyless(0); }

<MODIF>{nat}            fprintf(yyout, "%d", atoi(yytext) + 1);
<MODIF>{S}              { ECHO; BEGIN(NORMAL); }
```

4.8.7 Ejemplo 4

Escribir un fuente `lex` que dado un fichero de texto cuente:

1. El número de palabras escritas totalmente en mayúsculas.
2. El número de palabras escritas totalmente en minúsculas.
3. El número de palabras con minúsculas y mayúsculas.
4. El número de palabras de cinco letras.

En el fichero de salida se truncarán todos los números reales que se encuentren, incluidos aquellos que aparezcan en notación científica. El final del fichero se marcará con el signo especial `$`.

```
%{
         #include <stdio.h>
         #include <stdlib.h>

         int num_may = 0, num_min = 0, num_cin = 0, num_mix = 0;
%}

cinlet  (([a-zA-Z]){5,5})
may     ([A-Z]+)
min     ([a-z]+)
mix     ([a-zA-Z]+)
dig     ([0-9])
nat     ({dig}+)
```

```
exp     ([Ee][+]?{nat})
real    (({nat}"."{nat}{exp}?)|({nat}{exp}))

%%

{real}          fprintf(yyout, "%d", (int)atof(yytext));
{cinlet}        { num_cin++; REJECT; }
{may}           { num_may++; ECHO; }
{min}           { num_min++; ECHO; }
{mix}           { num_mix++; ECHO; }
\$             printf("cinco=%d, may=%d, min=%d," \
                    "mix=%d\n", num_cin, num_may, num_min, num_mix);
```

4.8.8 Ejemplo 5

Escribir un programa `lex` que encripte un texto en el que hay letras mayúsculas o minúsculas, comas, espacios, puntos y retornos de carro. Para ello, con cada palabra se hará lo siguiente:

- Si termina en una vocal, cada una de sus letras se cambiará por la posterior en el alfabeto. Después de la zeta se supone que viene la a.
- Si la palabra termina en una consonante, cada una de sus letras se cambiará por la anterior en el alfabeto. Antes de la a, se considera que va la zeta.
- El resto de los caracteres se quedan igual.

```
%{
    #include <stdio.h>

    #define SIG(c) ((c) == 'z' ? 'a' : \
                                ((c) == 'Z' ? 'A' : (c) + 1))
    #define ANT(c) ((c) == 'a' ? 'z' : \
                                ((c) == 'A' ? 'Z' : (c) - 1))
%}

p       [,\ .\n]
v       [aeiouAEIOU]
c       [bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ]
l       ({v}|{c})
pal     ({l}+)

%s INICIO CASO1 CASO2

%%

                        BEGIN(INICIO);

<INICIO>{v}             putchar(SIG(*yytext));
```

```
<INICIO>{c}              putchar(ANT(*yytext));
<INICIO>{pal}/{v}{p}     { yyless(0); BEGIN(CASO1); }
<INICIO>{pal}/{c}{p}     { yyless(0); BEGIN(CASO2); }

<CASO1>{l}               putchar(SIG(*yytext));

<CASO2>{l}               putchar(ANT(*yytext));

<INICIO,CASO1,CASO2>{p} { ECHO; BEGIN(INICIO); }

<INITIAL,INICIO>.        |
<CASO1,CASO2>.           fprintf(stderr, "Carácter" \
                                 "incorrecto %c\n", *yytext);
```

4.8.9 Ejemplo 6

Obtener la especificación en `lex` del reconocer léxico para un sencillo lenguaje de programación con las siguientes características:

- Palabras reservadas: `begin`, `if`, `then`, `else`, `end`. Pueden aparecer en mayúsculas, minúsculas o una mezcla de ambas.
- Símbolos especiales: `(`, `)`, `:=`.
- Identificadores, formados por letras, dígitos decimales y los caracteres `@`, `+` , `-`, `*` y `/`. Los identificadores pueden empezar por cualquiera de estos caracteres, pero no pueden estar compuestos íntegramente por dígitos decimales.
- Literales naturales y reales en notación de coma fija o punto fijo.
- En el lenguaje se ignoran todos los caracteres blancos, esto es: el espacio, el tabulador, el salto de línea y el salto de página.

Se pretende que la función `yylex()` obtenida devuelva en cada llamada un número distinto correspondiente al tipo de prefijo que ha reconocido. Para ello se ha preparado un fichero de nombre `y.tab.h` que contiene la información siguiente:

```
#ifndef _Y_TAB_H
#define _Y_TAB_H

typedef struct {
    int nat;
    char *ident;
    double real;

    unsigned lin, col;
} YYSTYPE;  /* OJO:  Este no es el yystype generado por Yacc */

extern YYSTYPE yylval;
```

```
#define PR_BEGIN 257    /* begin */
#define PR_IF 258       /* if */
#define PR_THEN 259     /* then */
#define PR_ELSE 260     /* else */
#define PR_END 261      /* end */
#define PAR_AB 262      /* ( */
#define PAR_CE 263      /* ) */
#define ASIG 264         /* := */
#define IDENT 265
#define NAT 266
#define REAL 267

#endif
```

Las directivas `#define` nos ofrecen nombres simbólicos para los números que la función `yylex()` debe devolver. La variable `yylval` servirá de interface con otros módulos del programa en el que se pretende utilizar este analizador. Cada vez que se reconozca un literal natural deberá actualizarse `yylval.nat` con su valor numérico, cada vez que se reconozca un literal real deberá actualizarse `yylval.real`, y cada vez que se reconozca un identificador deberemos copiar su lexema en `yylval.ident`. En cualquiera de los casos debemos apuntar la línea y la columna en la que aparece el lexema en las variables `yylval.lin` e `yylval.col`

También se desea que el analizador ofrezca dos rutinas, de nombre yyline() y yycolumn() con las que informará permanentemente a cerca de la posición (línea y columna) del último lexema reconocido en la cadena de entrada.

```
%{
        #include <stdio.h>
        #include <stdlib.h>
        #include <string.h>
        #include "y.tab.h"

        static int lin = 1, col = 1;

        #define RETURN(token) \
                    { yylval.lin = lin, yylval.col = col; \
                    return token; }
%}

dig     ([0-9])
let     ([a-zA-Z0-9@+*/])

nat     ({dig}+)
ident   ({let}+)
real    ({nat}("."{nat})?([Ee][+]?{nat})?)
blanco  ([\ \t\f])
ret     (\n)
```

```
%%

                        col += yyleng;

{blanco}+               col += yyleng;
{ret}                   col = 1; lin++;

[Bb][Ee][Gg][Ii][Nn]    RETURN(PR_BEGIN);
[Ii][Ff]                RETURN(PR_IF);
[Tt][Hh][Ee][Nn]        RETURN(PR_THEN);
[Ee][Ll][Ss][Ee]        RETURN(PR_ELSE);
[Ee][Nn][Dd]            RETURN(PR_END);

"("                     RETURN(PAR_AB);
")"                     RETURN(PAR_CE);
":="                    RETURN(ASIG);

{nat}                   { yylval.nat = atoi(yytext);
                          RETURN(NAT); }
{real}                  { yylval.real = atof(yytext);
                          RETURN(REAL); }

{ident}                 { yylval.ident = strdup(yytext);
                          RETURN(IDENT); }

.                       col++; fprintf(stderr, "Error léxico");

%%

int yyline(void)
{
  return lin;
}

int yycolum(void)
{
  return col;
}
```

Hemos presentado una de las posibles soluciones a este problema. El lenguaje que pretendemos analizar tiene diversas particularidades que hacen muy interesante el estudio del orden en el que se han escrito las reglas.

4.8.10 Ejemplo 7

Obtener la especificación en `lex` del reconocer léxico para un sencillo lenguaje que permite la manipulación de intervalos reales y numéricos. Un ejemplo de este lenguaje es el siguiente:

```
A = [1..2];
B = [3...4];
```

Se permiten registros de una letra, números enteros y números reales en punto fijo, con la característica que un número entero seguido de un punto se considera un número real con parte decimal nula.

Los tokens del lenguaje son: '=', DOSP ('..'), '[', ']', ';', LETRA, ENTERO y REAL. Al igual que en el problema anterior, se puede suponer la existencia de un fichero y.tab.h con características similares

```
%{
        #include <stdio.h>
        #include <stdlib.h>
        #include <string.h>
        #include "y.tab.h"

        static int lin = 1, col = 1;

        #define RETURN(token) \
                    { yylval.lin = lin, yylval.col = col; \
                      return token; }
%}

dig     ([0-9])
let     ([a-zA-Z])

nat     ({dig}+)
real    ({nat}"."{nat}?)
blanco  ([\ \t\f])
ret     (\n)

%%

                        col += yyleng;

{blanco}+               col += yyleng;
{ret}                   col = 1; lin++;

"["                     RETURN('[');
"]"                     RETURN(']');
"="                     RETURN('=');
".."                    RETURN(DOSP);

{nat}/".."              |
{nat}                   { yylval.nat = atoi(yytext);
                          RETURN(NAT); }
{nat}"."/".."           |
{real}                  { yylval.real = atof(yytext);
                          RETURN(REAL); }
```

```
{let}                 { yylval.let = *yytext;
                        RETURN(LETRA); }

.                     col++; fprintf(stderr, "Error léxico");

%%

int yyline(void)
{
  return lin;
}

int yycolum(void)
{
  return col;
}
```

4.8.11 Ejemplo 8

Obtener un programa `lex` que permita reconocer comentarios anidados al estilo del lenguaje Turbo C de la forma más eficiente posible, es decir reconociendo en la entrada los prefijos más largos posibles dentro de los comentarios y no caracter a caracter.

```
%{

    #include <stdio.h>


    static unsigned num_coment, lin_coment;
    static int lin = 1, col = 1;

%}

%x COMENT

%%

...

"/*"                  { lin_coment = lin;
                        num_coment = 1;
                        BEGIN(COMENT); }

<COMENT>"/*"          col += yyleng; num_coment++;
<COMENT>[^*bakslashn]*          col += yyleng;
<COMENT>[^**bakslashn]*\n        lin++; col = 1;
<COMENT>"*"+[^*/*bakslashn]*     col += yyleng;
<COMENT>"*"+[^*/*bakslashn]*\n   lin++; col =1;
```

```
<COMENT>"*"+"/"         { col += yyleng;
                          num_coment--;
                          if (num_coment == 0) BEGIN(0); }


int yywrap(void)
{
  if (num_coment > 0)
    fprintf(stderr, "Fin de fichero antes de cerrar "
                    "el comentario de la línea   return 1;
}

int yyline(void)
{
  return lin;
}

int yycolum(void)
{
  return col;
}
```

Capítulo 5

Análisis sintáctico

5.1 Introducción

Aquí se pretende desarrollar las habilidades necesarias para la construcción de analizadores sintñacticos. El primer objetivo es desarrollar especificaciones de los aspectos sintñacticos mñas importantes que aparecen en los lenguajes de programación: listas de declaraciones y sentencias, separadores, expresiones, prioridad de los operadores, etc. Primero resolveremos todos esos problemas mediante una especificación que combina una gramática posiblemente ambigua en notación BNFE con un conjunto de reglas para romper la ambigüedad. Luego introduciremos un conjunto de tranformaciones de esa especificación.

Luego presentaremos YACC como un lenguaje adecuado para escribir las especificaciones de la sintaxis de un lenguaje de programación y usaremos las facilidades que incorpora para procesar gramñaticas ambiguas.

Posteriormente presentaremos la menera de implementar un reconocedor manualmente a partir de la especificación.

5.2 Especificación del Analizador Sintáctico

El analizador sintñactico es la parte del compilador que aceptando secuencias de token decide si esta secuencia es correcta o no en un lenguaje dado. La secuencia de tokens proviene del analizador léxico que los ha ido formando a partir de la secuencia de caracteres de entrada.

```
Tokens   ---  A.S.   ---    {si, no}
```

El primer problema que nos queremos plantear es escoger el lenguaje de especificación adecuado para describir el analizador sintñactico. Tal como se ha visto anteriormente las expresiones regulares tienen un poder expresivo equivalente a las Gramñaticas Lineales. Pero con ellas no son suficientes para describir los aspectos sintácticos de los lenguajes usuales. La siguiente posibilidad es escoger las gramáticas independientes del contexto como lenguaje de especificación. Aun estas son insuficientes para describir determinados aspectos. Asi el lenguaje $L = \{a^n b^n c^n, n \geq 0\}$ no puede ser descrito con una gramñatica

independiente del contexto. Sin embargo una gramñatica dependiente del contexto puede describirlo. Así la gramñatica G({a,b,c},{A,B,C}, P, A) con las producciones

A : a A B C
A : a B C
b C : b c
c C : c c
C B: B C
a B : a b
b B : b b

describe el lenguaje. Hay otros aspectos de los lenguajes de programación que no pueden ser descritos mediante gramñaticas independientes del contexto, pero si se describirian mediante gramáticas dependientes del contexto. Entre esos aspectos tenemos:

- No se pueden repetir dos identificadores en la lista de parñametros formales de una función.
- El n
  'umero de parñametros formales y reales deben ser el mismo.
- Los tipos de la parte izquierda y derecha de una asignación deben ser compatibles.
- Etc.

Muchos de estos aspectos se podrían describir mediante gramñaticas G1, pero esto tiene el inconveniente de que la especificación sería muy difícil de leer y lo que es mas importante que sería practicamente imposible obtener implementaciones eficientes a partir de la especificación. Todo ello nos lleva a escoger las gramáticas independientes del contexto como lenguaje de especificación puesto que:

- Son fñacilmente legibles.
- Es relativamente fñacil obtener implementaciones eficientes a partir de la especificación.

La notación que usaremos para escribir las gramñaticas serña BNF o EBNF segun en cada caso sea mas interesante por claridad y concisión de la especificación. Partimos de de que la notación EBNF puede ser transformada a BNF automñaticamente tal como se ha visto en capítulos anteriores. Las gramñaticas independientes del contexto pueden ser ambiguas. Esto quiere decir que podrían existir diferentes árboles de reconocimiento sintñactico para una secuencia dada. Si esto ocurre nuestra especificación es incompleta. Para que nuestra especificación sea completa deberiamos restringirnos a gramñaticas que sean no ambiguas. Pero esto no es deseable porque en la mayoría de los casos la especificación queda mas clara usando gramñaticas ambiguas. Una posibilidad es ampliar el lenguaje de especificación con un conjunto de reglas que indiquen de forma inequívoca la manera de decidir el árbol escogido en caso de ambigüedad. Es decir a
nadir a la gramñatica reglas de desambiguación. Esta es la alternativa que escogemos. Nuestro lenguaje de especificación de analiadores sintñacticos se compondrña de:

- Una gramñatica independiente del contexto, posiblemente ambigua, escrita en notación BNF o EBNF.
- Un conjunto de reglas de desambiguación.

Al igual que en el caso de la especificación del analizador léxico el lenguaje de especificación tendrña la siguiente estructura:

```
%{
------
%}
%start S
Simbolos de la gramatica
Reglas de desambiguacion
%%
Producciones  en notacion BNFE
%%
----
```

Las reglas de desambiguación especifican la forma en como se rompe la ambigüedad. Mediante esas reglas proporcionamos información sobre la prioridad realtiva de cada token y producción gramatical. Esta información se escribe mediante un conjunto de líneas. Cada línea comienza con las palabras clave: *%token, %left, %right, %nonassoc* y sigue con un conjunto de tokens. Los tokens en una misma línea tienen la misma prioridad. Los tokens escritos en una línea anterior tinen menos prioridad que los escritos en una línea posterior. Todos los tokens en una misma línea tienen la misma asociatividad. Esta viene especificada por la palabra de comienzo de la línea. *%left* especifica asociatividad por la izquieeda, *%right* asociatividad por la derecha, *%nonassoc* no asociatividad y *%token* no da información sobre la asociatividad. Dentro del conjunto de producciones, cada una de ellas hereda la prioridad y asociatividad del token mas a derecha de la producción. Este mecanismo de herencia se puede romper usando, a la derecha de la producción, la palabra clave *%prec tk* para especificar que la prioridad de la regla es la del token *tk*. A partir de la información anterior cada token y cada producción puede tener asociada una prioridad y una asociatividad. Tambien habrña tokens y producciones que no tengan asociada prioridad y asociatividad. Esto ocurre cuando no damos esta información en el conjunto de reglas de desambiguación. Partiendo de lo anterior ahora debemos dar un mecanismo para romper la ambigüedad. Este mecanismo especifica la forma de escoger la aplicación de una producción de entre dos posibles. El mecanismo se resume en las siguientes reglas:

1. Reglas por defecto: Para dar prioridad a una regla de la que no hemos especificado nada.
   - Es mas prioritaria la producción mas larga
   - Si son iguales de largas la que estña escrita primero
2. Reglas de ruptura de la ambigüedad.
   - Escoger la producción mas prioritaria
   - Si son igualmente prioritarias se decide seg
     'un su asociatividad

- Si son asociativas por la izquierda, escoger como mas prioritaria la que tiene el token que aparece primero en la cadena a reconocer.
- Si son asociativas por la derecha, escoger como mas prioritaria la que tiene el token que aparece mas tarde en la cadena a reconocer.
- Si son no asociativas hay un error

Veamos un ejemplo para aclarar estas ideas.

```
%{
Codigo C
%}

%start list
%token IF ELSE
%token ID NUM
%nonassoc EQU < >
%left '+' '-'
%left '*' '/'
%left UMINUS
%right '^'

%%
list : stat*
     ;
stat : expr  '\n'
     | ID '=' expr  '\n'
     ;
expr : '(' expr ')'
     | expr EQU expr
     | expr '>' expr
     | expr '<' expr
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | '-' expr      % prec UMINUS
     | '+' expr      % prec UMINUS
     | expr '^' expr
     | IF '(' expr ')' expr
     | IF '(' expr ')' expr ELSE expr
     | ID
     | NUM
     ;

%%

    < codigo C >
```

En las dos reglas

```
                | expr '+' expr
                | expr '/' expr
```

la primera hereda las propiedades del +, la segunda las del /. La regla

```
                  | '-' exp    % prec UMINUS
```

hereda la prioridad del UMINUS, no del '-'. Y además es el de mayor prioridad. Entre las reglas

```
          | IF '(' expr ')' expr
          | IF '(' expr ')' expr ELSE expr
```

la prioritaria es la mas larga. Con la información especificada en el ejemplo las cadenas siguientes tendrán los árboles de reconociemiento sintáctico asociados:

```
(1)     a+b*c                   expr
                  +-------------+-------------+
                  expr          +             expr
                  -+-                   +------+-------+
                   a                    expr   *       expr
                                        -+-            -+-
                                         b              c

(2)     a+b-c                   expr
                  +-------------+-------------+
                  expr          -             expr
              +------+-------+                 -+-
              expr   *       expr              c
             -+-             -+-
              a               b

(3)     a^b^c                   expr
                  +-------------+-------------+
                  expr          ^             expr
                  -+-                   +------+-------+
                   a                    expr   ^       expr
                                        -+-            -+-
                                         b              c

(4)     a EQU b EQU c          Error
```

La (1) por ser *expr '*' expr* mas prioritaria que *expr '*' expr*. La (2) por ser *expr '+' expr*, *expr '-' expr* igualmente prioritarias y asociativas por la izquierda. La (3) por ser *expr expr* asocitiva por la derecha y la (4) por ser *expr EQU expr* no asociativa.

5.2.1 Especificación de Secuencias

Algunos comentarios sobre la forma de escribir secuencias de elementos con y sin separadores entre ambos. Sea el siguiente ejemplo:

```
list : stat*
     ;
stat : expr  '\n'
     | ID '=' expr  '\n'
     ;
```

Mediante las dos producciones anteriores, se ha especificado una lista de enunciandos, posiblemente vacía. Cada enunciado lleva un terminador que es el fin de línea. Otra posibilibad es que el fin de línea no fuera un terminador de enunciado sino un separador de enunciados. Entonces las producciones adecuadas serian:

```
list : stat ('\n' stat)*
     ;
stat : expr
     | ID '=' expr
     ;
```

Asi en general una secuencia de objetos de tipo "a" con un ";" como separador y al menos un elemento, se especificará:

```
s : a ( ';' a)*
  ;
```

Si la secuencia pudiera estar vacia:

```
s :
  |  a ( ';' a)*
  ;
```

Si la secuencia es posiblemente vacia, pero sus elementos no tienen separador, pero el terminador ";", entonces:

```
s : b*
  ;
b : a ';'
  ;
```

y si la secuencia tiene al menos un elemento:

```
s : b+
  ;
b : a ';'
  ;
```

Ejemplo de secuencias son separador es *ID,ID,ID* y con terminador la secuencia de sentencias en C con el termiandor el ;".

5.3 Transformación de Especificaciones

5.3.1 Incorporación de las reglas de ruptura de la ambiguedad a la Gramática

El método consiste en ir recorriendo cada línea de las que especifican las prioridades y asociatividades. Cada línea da lugar a una secuencia de objetos operados por operadores de esa línea. A su vez cada uno de esos objetos contarña con operadores de mayor prioridad. Dependiendo de la asociatividad la lista de objetos se escribirña como

```
(1)             expr1 : expr2 ( op1 expr2 )*
                      ;
(2)             expr1 : expr2 [ op1 expr1 ]
                      ;
(3)             expr1 : expr2 [ op1 expr2 ]
                      ;
(4)             expr1 :  op1 expr1
                      |  expr2
                      ;
```

Donde *op1* serñan todos los operadores de la correspondientes línea. La forma (1) corresponde a la asociatividad por la izquierda, la (2) a la asociatividad por la derecha, la (3) a la no asociatividad y la (4) a los operadores unarios. La opción (4) tambien admite ser escrita en la forma

```
(4.1)           expr1 :  [op1] expr2
                      ;
```

En la variante (4.1) sólo permitimos la posibilidad de un operador unario. Con la (4) se permite una secuencia de operadores unarios. El ejemplo anterior se transforma en:

```
%{
 Codigo C
%}
%start list
%token IF ELSE
%token NUM ID EQU
%%
list : stat *
     ;
stat : expr '\n'
     | ID '=' expr '\n'
     ;
expr : expr1 [ opr expr1]
     ;
expr1: term (ops term)*
     ;
term : fact1 (opm fact1)*
```

```
      ;
fact1: opu fact1
      | fact
      ;
fact : base ['^' fact]
      ;
base :  ID
      |  NUM
      | '(' expr ')'
      | IF '(' expr ')' expr
      | IF '(' expr ')' expr ELSE expr
      ;
 ops : '+'
      | '-'
      ;
 opm : '*'
      | '/'
      ;
 opr : EQU
      | '<'
      | '>'
      ;
 opu : '+'
      | '-'
      ;
 %%
    Codigo C
```

5.3.2 Transformación de BNFE a BNF

Esto se puede hacer con la siguientes equivalencias:

```
s: a*       ---     s:
 ;                   | s a
                     ;
s: a*       ---     s:
 ;                   | a s
                     ;
s: [a]      ---     s:
 ;                   | a
                     ;
s: c(a|b)   ---     s: c d
 ;                   ;
                    d: a
                     | b
                     ;
s: a+       ---     s: a
 ;                   | s a
                     ;
```

```
s: a+       ---      s: a
 ;                     | a s
                       ;
```

Capítulo 6

La herramienta yacc

YACC es una herramienta informática que permite obtener a partir de un texto fuente que especifica la gramática de un lenguaje de contexto libre, la implementación de un autómata de pila que controlado por una tabla *parse* LALR(1) permite reconocer frases de ese lenguaje. Además permite especificar la ejecución de una cierta porción de código C cada vez que se lleva a cabo la reducción de una regla de la gramática.

Los programas generados por YACC se basan en una rutina de análisis léxico de nombre `yylex()` que suele ser producida por la herramienta de generación de analizadores léxicos LEX.

6.1 Introducción

El núcleo de la especificación de entrada es un conjunto de reglas gramaticales en formato BNF. Cada regla describe una porción lógica que se pretende reconocer en el texto de entrada. Un ejemplo de una regla gramatical podría ser:

```
fecha
  : dia '-' nombre_mes '-' anyo
  ;
```

`fecha`, `dia`, `nombre_mes`, y `año` representan estructuras de interés en fichero de entrada y se espera que se encuentren separadas entre sí por guiones. En algún otro lugar del programa YACC será preciso definir la estructura interna de cada uno de estos elementos. Por ejemplo,

```
dia
  : LIT_NAT
  ;

nombre_mes
  : ENERO
  | FEBRERO
  | ...
```

```
  | DICIEMBRE
  ;

anyo
  : LIT_NAT
  ;
```

En este caso hemos definido `dia` como una porción del texto de entrada que contiene un literal natural. Suponemos que el analizador léxico `yylex()` devuelve el *token* `ID_NAT` cada vez que encuentra en el fuente una secuencia de dígitos númericos que representa un número natural. De igual forma `nombre_mes` reconoce cualquier porción del texto de entrada en la que el analizador sintáctico encuentre una subcadena que encaje en el patrón del *token* `ENERO`, `FEBRERO`, ... o `DICIEMBRE`. Por ejemplo, la entrada

```
12-DIC-197o
```

podria ser un texto reconocido por la regla anterior.

Como vemos, una parte muy importante del trabajo de análisis sintáctico recae sobre el analizador léxico `yylex()` que es el que se encarga de agrupar los caracteres que encuentra en el fichero de entrada y devolver los *tokens* asociados a cada uno de los patrones reconocidos. Generalmente llamaremos símbolos terminales a los *tokens* reconocidos por el analizador léxico y símbolos no terminales a los símbolos que encabezan las reglas gramáticales. Es costumbre escribir los primeros en mayúsculas y los segundos en minúsculas, justo al contrario en la teoría general de lenguajes.

Es importante destacar que los analizadores generados por YACC reconocen tan sólo un subconjunto de los lenguajes de contexto libre, entre ellos los regulares. Cualquier lenguaje regular se puede reconocer usando un generador producido por YACC, por lo que en un principio puede parecer innecesario el uso combinado de la herramienta LEX. Esto es, si definimos la rutina `yylex()` como

```
int yylex(void)
{
  return getchar();
}
```

y nuestro anterior fuente lo reescribimos como

```
fecha
  : dia '-' nombre_mes '-' anyo
  ;

dia
  : lit_nat
  ;

nombre_mes
```

```
  : 'E' 'N' 'E'
  | 'F' 'E' 'B'
  | ...
  | 'D' 'I' 'C'
  ;

anyo
  : lit_nat

lit_nat
  : lit_nat digito
  | digito
  ;

digito
  : '0' | '1' | ... | '9'
  ;
```

es evidente que el analizador obtenido por YACC también será capaz de reconocer en la entrada el mismo lenguaje. La razón de por qué los componentes léxicos sean tarea del analizador generado por LEX es que los reconocedores generados por YACC son particularmente ineficientes en el caso de lenguajes regulares, y los generados por LEX están especializados en esta última clase de lenguajes.

Otra razón importante para usar un reconocedor léxico específico es que en la mayoría de los lenguajes de ordenador suelen ignorar por completo ciertas secuencias de caracteres que aparecen en la entrada. Por ejemplo, en la mayoría de los lenguajes los espacios en blanco y los comentarios son completamente irrelevantes, pero introducirlos en la gramática de YACC puede resultar bastante complejo y engorroso. Por eso es mejor dejar todas estas tareas a LEX. En este capítulo estudiaremos con detalle la forma de enlazar los reconocedores producidos de forma independiente por cada una de las herramientas.

Un problema aparte es el hecho de que la entrada de un programa en rara ocasión es correcta desde el punto de vista gramatical, sobre todo cuando el fichero ha sido creado de forma manual. En estos casos YACC nos proporciona un conjunto de herramientas bastante simples para poder tratar con ficheros de entrada incorrectos. La técnica no está demasiado depurada en el sentido de que es difícil de utilizar y no permite analizar los errores de una forma precisa, por lo que le dedicaremos un capítulo especial a estos asuntos.

6.2 Especificaciones Básicas

Un fuente en YACC se divide en las tres zonas que se muestran a continuación:

(zona de definiciones)
%%
(zona de reglas)
%%
(zona de rutinas de usuario)

El símbolo `%%` actúa como separador entre las tres secciones, de las que tan sólo son obligatorias las dos primeras. La zona de definiciones puede estar vacía y se utiliza para incluir código C que sea necesario en la atribución de las reglas gramaticales. También se incluye la declaración de los *tokens* del lenguaje y los atributos de todos los símbolos, terminales o no.

Los blancos, tabuladores y líneas en blanco son completamente ignorados y además es posible insertar comentarios al estilo del lenguaje C (`/* ... */`) en cualquier punto. Los identificadores usados para represetar los símbolos pueden tener una longitud arbitraria y estar formados por letras, dígitos, puntos o subrayados y como es habitual deben comenzar necesariamente por una letra. Mayúsculas y minúsculas se consideran diferentes y generalmente se reservan las primeras para los nombres de los *tokens* y las segundas para los nombres de los símbolos no terminales.

Algunos *tokens* se pueden representar como caraacteres literales encerrados entre comillas simples y al igual que en C se puede usar la barra invertida para representar ciertos caracteres especiales. Los más importantes se muestran en la tabla 6.1. Cualquier otro caracter precedido por una barra se considera equivalente a él mismo. Tan sólo hay que tener especial cuidado con el caracter nulo que se representa como `'\ 0'` y no puede usarse dentro de las reglas gramaticales, aunque YACC no advertirá de ello en caso de que aparezca en un fuente provocando la aparición de errores difíciles de encontrar.

'\n'	Salto de línea
'\r'	Retorno de carro
'\"	Apóstrofe
'\\'	Barra invertida
'\t'	Tabulador horizontal
'\v'	Tabulador vertical
'\b'	Caracter de retroceso
'\f'	Salto de página
'\ooo'	El caracter cuyo código octal es `ooo`

Tabla 6.1: Literales estándar.

Un tipo especial de regla gramatical es el de las λ–producciones, que en teoría de lenguajes se suelen representar como $A ::= \lambda$ y sirven para describir una porción del texto de entrada en la que no aparece ningún caracter. En YACC no existe ningún símbolo especial para representar las λ–producciones, y estas se escriben de la siguiente forma:

```
A : ;
```

Generalmente, para hacer más legible la gramática es conveniente añadir algún comentario que indique que se trata de una λ–producción, pero YACC lo ignora por completo.

```
A : /* lambda */
  ;
```

Los identificadores que representan símbolos terminales se deben declarar de forma explícita en la zona de definiciones utilizando la palabra reservada `%token`.

```
%token TOKEN_1 TOKEN_2 ...
```

De entre todos los símbolos no terminales de la gramática es necesario destacar su axioma. Por defecto se toma como axioma de la gramática el símbolo no terminal que da nombre a la primera regla que aparezca en el fuente, pero es posible cambiarlo usando en la zona de definiciones la palabra reservada `%start`. Por ejemplo,

```
%start S
```

hace que YACC tome como símbolo principal de la gramática `S`, ignorando cuál es el que encabeza la primera regla de producción escrita.

6.3 Acciones

Con cada regla gramatical, el usuario puede asociar acciones para que se realicen cada vez que en el fichero de entrada se encuentra una secuencia de *tokens* que encaja en ella. Estas acciones se codifican en lenguaje C y pueden devolver valores y utilizar valores preparados por acciones que se han llevado a cabo previamente al reducir alguna otra regla o por el analizador léxico. Las acciones se especifican entre llaves.

En el siguiente trozo de código hemos ampliado nuestra gramática para reconocer fechas incluyendo como acciones la impresión de un mensaje que nnos indica a qué categoría sintáctica pertenece cada una de las porciones de texto que son reconocidas.

```
fecha
  : dia '-' nombre_mes '-' anyo
    { printf("Reconocido fecha\n"); }
  ;

dia
  : LIT_NAT   { printf("Reconocido dia\n"); }
  ;

nombre_mes
  : ENERO      { printf("Reconocido enero\n"); }
  | FEBRERO    { printf("Reconocido febrero\n"); }
  | ...
  | DICIEMBRE  { printf("Reconocido diciembre\n"); }
  ;

anyo
  : LIT_NAT { printf("Reconocido anyo\n"); }
  ;
```

Pero además de mostrar mensajes también es posible realizar cálculos y asociar valores a los símbolos que aparecen en la gramática. Para ello se utilizan los pseudosímbolos `$$`, `$1`, `$2`, ..., `$n`. `$$` referencia siempre el valor que se calcula para el símbolo a la cabeza de

una regla de producción y `$i` el valor calculado previamente para el símbolo `i`–ésimo de la regla de producción. En este punto es importante recordar que YACC utiliza la técnica de reconocimiento LALR(1), que es una técnica de tipo ascendente. Esto significa que en principio YACC tan sólo es capaz de tratar con gramáticas S–atribuidas en las que todos los atributos involucrados son de tipo sintetizado. Esto es:

```
A : A_1 A_2 ... A_n
    { $$ = f($1, $2, ..., $n);
}
```

Más adelante veremos que es posible utilizar ciertos tipos de atributos heredados, pero utilizando unos mecanismos bastante inseguros.

A continuación mostramos una ligera modificación del ejemplo anterior en la que cada vez que se reconoce una fecha se muestra en pantalla en formato numérico `dd/mm/aa`.

```
fecha
  : dia '-' nombre_mes '-' anyo
    { printf("Fecha = %d/%d/%d\n", $1, $3, $4); }
  ;

dia
  : LIT_NAT   { $$ = $1 }
  ;

nombre_mes
  : ENERO      { $$ = 1; }
  | FEBRERO    { $$ = 2; }
  | ...
  | DICIEMBRE  { $$ = 12; }
  ;

anyo
  : LIT_NAT { $$ = $1; }
  ;
```

Cada vez que se reconoce un mes, se devuelve su código numérico correspondiente, y cada vez que se reconoce un literal natural correspondiente a un día o a un año, su valor numérico se toma del token `LIT_NAT`. Este valor deberá haber sido sintetizado por el analizador léxico cuando reconoció el patrón, y más adelante veremos de qué forma se hace esto.

En muchas ocasiones es necesario llevar a cabo una acción, no cuando se ha terminado de reconocer una porción completa de la entrada, sino cuando tan sólo hemos leído un trozo de la misma. A continuación se muestra la gramática simplificada de un típico lenguaje de programación con estructura de bloques anidados:

```
bloque
  : BLOCK ID
```

```
    VARS
      dec_vars
    BEGIN
      sentencia
    END
  ;

sentencia
  : /* lambda */
  | sentencia ';' sentencia
  | bloque
  | ID '=' exp
  | ...
  ;
```

Un programa escrito en este lenguaje podría ser:

```
BLOCK X
VARS
  a: Integer;
BEGIN
  a := 1;      (* 1 *)
  BLOCK Y
  VARS
    a : Real;
  BEGIN
    a := 2.3   (* 2 *)
  END
  a := 4;      (* 3 *)
END
```

Como es de esperar, cada bloque introduce en el lenguaje un nuevo ámbito de nombres, de forma que el identificador `a` que aparece en las líneas 1 y 3 se refiere al identificador declarado como `Integer` en el bloque `X`, y el identificador `a` que aparece en la línea 2 se refiere al declarado como `Real` en el bloque `Y`.

Para conseguir diferenciar entre unos y otros identificadores, los compiladores deben crear de forma explícita un ámbito de declaración cada vez que entran dentro de un bloque y lo destruyen cada vez que salen de él. Para conseguir esto sería necesario poder ejecutar una acción de creación de ambito cada vez que se reconoce la palabra `BLOCK` en el fuente. En YACC es posible insertando el código entre llaves en el lugar oportuno. Por ejemplo:

```
bloque
  : BLOCK ID { crear nuevo ámbito }
    VARS
      dec_vars
    BEGIN
      sentencia
    END      { destruir ámbito }
  ;
```

En cualquier caso debemos entender esto como una abreviatura que YACC transforma de manera automática en la siguiente gramática:

```
bloque
  : BLOCK ID $ACC
    VARS
      dec_vars
    BEGIN
      sentencia
    END      { destruir ámbito }
  ;

$ACC
  : /* lambda */ { crear nuevo ámbito }
  ;
```

en donde `$ACC` es un nuevo símbolo no terminal introducido de forma automática por YACC.

6.4 Relación con el analizador léxico

El usuario debe suministrar un analizador léxico que lea del fichero fuente caracteres y los agrupe para dar lugar a *tokens*, cada uno con los atributos que sean necesarios. YACC espera que este analizador esté implementado en una fucniòn con el siguiente *interface*:

```
int yylex(void);
```

de tal forma que cada vez que sea llamada devuelva un valor entero correspondiente al siguiente *token* encontrado en el fichero de entrada. Los valores para los atributos se del *token* devuelto se deben guardar en la variable de nombre `yylval`, que estudiaremos con mucho más detalle en la sección ??.

Los números que representan los *tokens* pueden ser elegidos libremente por el usuario (a excepción del 0, que debe representar sirmpre el fin de fichero), o bien se puede dejar a YACC la labor de generarlos de forma automática.

A continuación mostramos un pequeño ejemplo en el que hemos realizado un analizador léxico que lee caracteres de la entrada y los clasifica en letras, dígitos y otros. En cualquier caso, la variable yylval toma una copia del caracter leído.

```
#include <ctype.h>

#define LETRA  1
#define DIGITO 2
#defien OTRO   3

int yylex()
```

```
extern char yylval;
int c;
int result = 0;

yylval = c = getchar();

if (isalpha(c))
  result = LETRA;
else if(isdigit(c))
  result = DIGITO;
else if (c == EOF)
  result = 0;
else
  result = OTRO;

return result;
```

En esta ocasión, hemos elegido los números para los *tokens* manualmente, pero tal y como hemos indicado, lo más adecuado es dejar que YACC los genere de forma automática.

Una herramienta muy utilizada para construir analizadores léxicos es el programa "Lex" desarrollado por Mike Lesk descrito en el capítulo ??. Estos analizadores léxicos son diseñados para trabajar conjuntamente con los analizadores sintácticos generados por YACC.

6.5 Ambigüedados y conflictos

Un conjunto de reglas gramaticales es ambigüo si hay alguna cadena de entrada que pueda ser derivada a partir del axioma principal de la gramática utilizando dos o más conjuntos de derivaciones. Por ejemplo, la regla gramatical:

```
exp : exp '+' exp
    | LIT_ENT
```

es una forma sencilla de indicar que una expresión puede formarse tomando como base otras dos expresiones combinadas con el operador suma. Por desgracia, esta regla es ambigüa. Por ejemplo, si la entrada es:

```
1 + 2 + 3
```

entonces son posibles dos derivaciones:

```
exp → exp '+' exp → (exp '+' exp) '+' exp
exp → exp '+' exp → exp '+' (exp '+' exp)
```

En el primer caso se ha derivado primero el no terminal más a la izquierda y en el segundo el no terminal más a la derecha. Es decir, en el primer caso se ha realizado la derivación suponiendo que el operador '+' es asociativo por la izquierda, mientras que en el segundo caso se ha considerado asociativo po la derecha.

Este tipo de ambigüedades se traducen enc conflictos *shift/reduce* cuando se construye el autómata de reconocimiento LALR(1), tal y como se estudió en la sección ??. YACC utiliza dos reglas sencillas que aplica cada vez que en la entrada aparece un *token* conflictivo:

1. Ante un conflicto *shift/reduce*, se lleva a cabo siempre el desplazamiento
2. En un conflicto *reduce/reduce*, por defecto se reduce por la regla gramatical que hay en primer lugar (en la secuencia de entrada).

La Regla 1 implica que esas reducciones son aplazadas siempre que hay una alternativa, en favor de los shifts. La Regla 2 le da al usuario la responsabilidad sobre el control del comportamiento del analizador sintáctico en ésta situación, pero los conflictos reduce/reduce deberían ser evitados siempre que fuese posible.

Los conflictos pueden presentarse debido a equivocaciones en la entrada o en la lógica, o porque las reglas gramaticales, aunque consistentes, requieren un analizador sintáctico más complejo que el que YACC pueda construir. La utilización de las acciones dentro de las reglas pueden también provocar conflictos, si la acción debe estar hecha antes de que el analizador sintáctico pueda estar seguro de que regla está siendo reconocida. En estos casos, la aplicación de reglas sin ambigüedades es inapropiado, y lleva a un analizador sintáctico incorrecto. Por ésta razón, YACC siempre devuelve el número de conflictos shift/reduce y reduce/reduce resueltos por la Regla 1 y la Regla 2.

Como ejemplo del poder de las reglas sin ambigüedades, consideremos un fragmento de un lenguaje de programación resolviendo una construcción llif-then-else":

```
stat : IF '(' expr ')' stat
     | IF '(' expr ')' stat ELSE stat
     | otra
```

En éstas reglas, `IF` y `ELSE` son *tokens*, "cond" es un símbolo no terminal que describe expresiones, y "otra" es un símbolo no terminal que describe otras sentencias. La primera regla será llamada "regla de simple if" y la segunda como "regla if-else".

Estas dos reglas forman una construcción ambigüa, dada una entrada de la forma:

```
IF (C1) IF (C2) S1 ELSE S2
```

puede ser estructurado de acuerdo con éstas reglas de dos formas:

```
IF (C1)
{
  IF (C2)
```

```
    S1
}
ELSE
  S2
```

o bien como:

```
IF (C1)
{
  IF (C2)
    S1
   ELSE
    S2
}
```

La segunda interpretación es la que más se toma en los lenguajes de programación que tienen ésta construcción. Cada ELSE es asociado con el último IF no precedido de un ELSE. Con éste ejemplo, consideremos la situación donde el analizador sintáctico ha visto:

```
IF ( C1 ) IF ( C2 ) S1
```

y en ese momento está mirando el ELSE. Puede inmediatamente reducir por la "regla de simple if"":

```
IF (C1) stat
```

y entonces lee la entrada que queda:

```
ELSE S2
```

y reduce

```
IF (C1) stat ELSE S2
```

por la "regla if-else". Esto lleva a la primera de las agrupaciones anteriores de la entrada.

Por otra parte, al ELSE puede hacérsele una operación de shift, se lee S2, y entonces la parte derecha de:

```
IF ( C1 )  IF ( C2 ) S1 ELSE S2
```

puede ser reducida por la "regla if-else":

```
IF ( Cl )  stat
```

la cual puede ser reducida por la "regla de simple if". Esto lleva a la segunda de las agrupaciones anteriores de la entrada, lo que normalmente es deseado.

De nuevo el analizador sintáctico puede hacer dos cosas válidas (hay un conflicto shift/reduce). La aplicación de la regla sin ambigüedades 1 dice que el analizador sintáctico haga la operación shift en éste caso, lo cual lleva a la agrupación deseada.

Este conflicto shift/reduce se presenta únicamente cuando hay un símbolo en la entrada actual particular, ELSE, y entradas particulares que ya han sido vistas, tales como:

```
IF ( Cl ) IF ( C2 ) Sl
```

En general pueden darse muchos conflictos, y cada uno será asociado con un símbolo de entrada y un conjunto de entradas leidas previamente. Las entradas leidas previamente son caracterizadas por el estado del analizador sintáctico. Los mensajes de conflictos de YACC son comprendidos mejor examinado el fichero de salida con la opcion de prefijo (-v). Por ejemplo, la salida correspondiente al estado de conflicto anterior podría ser:

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

  stat : IF cond stat_ (18)
  stat : IF cond stat_ ELSE stat

  ELSE shift 45
  reduce 18
```

La primera línea describe el conflicto, dando el estado y el símbolo de entrada. La descripción del estado normal continúa dando las reglas gramaticales activas en ese estado, y las acciones del analizador sintáctico. El símbolo de subrayado "_" marca la parte de las reglas gramaticales que ya ha sido vista. Así que en el ejemplo, en el estado 23, el analizador sintáctico ha visto la entrada correspondiente a:

```
IF ( cond ) stat
```

y las dos reglas gramaticales son activas al mismo tiempo. El analizador sintáctico tiene dos posibilidades para continuar. Si el símbolo de entrada es ELSE, es posible hacer un shift al estado 45. El estado 45 tendrá, como parte de su descripción, la línea:

```
stat : IF '(' cond ')' stat
```

De nuevo, hay que hacer notar que los números que siguen a los comandos "shift" se refieren a otros estados, mientras que los números que siguen a los comandos "reduce" se refieren a números de reglas gramaticales. En el fichero "y.output", los números de

las reglas son imprimidos después de que esas reglas puedan ser reducidas. Es posible hacer la accion de reducir, y éste será el comando por defecto. El usuario que encuentre inesperados conflictos shif t/reduce probablemente querrá mirar la salida para decidir si las acciones por def ecto son apropiadas o no. De hecho, en los casos difíciles, el usuario podría necesitar conocer más sobre el comportamiento y la construcción del analizador sintáctico que fuese conveniente en éste caso. En éste caso, una de las ref erencias teóricas (2, 3, 4) podrían ser consultadas.

6.6 Precedencia

Hay una situcion bastante común en las que las reglas dadas anteriormente para resolver conflictos no son suficientes; y está en el análisis de expresiones aritméticas. Además de las construcciones utilizadas comúnmente para las expresiones aritméticas pueden ser descritas normalmente por una notación de niveles de "precedencia" para los operadores, junto con información acerca de la asociatividad por la izquierda o por la derecha. Eliminar esas gramáticas ambigüas con apropiadas reglas sin ambigüedades pueden ser utilizadas para crear analizadores sintácticos que sean rápidos y fáciles de escribir, en vez de para analizadores sintácticos construidos desde gramáticas inequívocas. La notación básica es escribir reglas gramaticales con la forma:

```
expr : expr OP expr
```

y

```
expr : UNARY expr
```

para todos los operadores binarios o unarios deseados. Esto produce una gramática muy ambigüa, con muchos conflictos de análisis. Como en las reglas sin ambigüedades, el usuario especifica la precedencia, u obliga con fuerza, de todos los operadores, y la asociatividad de los operadores bínarios. Esta información es suficiente para permitir a YACC que resuelva los conflictos de análisis en concordancia con éstas reglas, y construir un analizador sintáctico que realice las precedencias y asociatividades deseadas.

Las precedencias y asociatividades aparecen junto a los tokens en la seccion de declaraciones. Esto aparece como una serie de líneas encabezadas por una palabra clave de YACC: ""de la misma linea se les asume que tienen el mismo nivel de precedencia y asociatividad; las lineas son listados en orden de crecimiento de precedencia. Así:

```
%left '+' '-'
%left '*' '/'
```

describe la precedencia y asociatividad de los cuatro operadores aritméticos. Los operadores 'más' y 'menos' son asociativos por la izquierda, y tienen menos precedencia que los operadores de 'multiplicación' y 'división', los cuales son también asociativos por la izquierda. La palabra clave lllos operadores asociativos por la derecha, y la palabra clave lloperador ".LT." en Fortran, que nos se puede asociar; así que,

```
A .LT. B .LT. C
```

es ilegal en Fortran, y por lo tanto es un operador que debería ser descrito por la palabra clave "de estas declaraciones, la descripción:

```
%right '=='
%left '+' '-'
%left '*' '/'

expr: expr '==' expr
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | NAME
```

podría utilizarse para estructurar la entrada:

```
a ==  b == c * d - e - f * g
```

como sigue:

```
a == ( b == (((c*d)-e) - (f*g)))
```

Cuando se utilizan éstos mecanismos, a los operadores unarios deben, en general, dárseles una precedencia. A veces un operador binario y un operador unario tienen la misma representación simbólica, pero precedencias distintas. Un ejemplo es el '-' unario y binario; al menos unario se le debe dar la misma precedencia que a la multiplicación, o incluso mayor, mientras que el menos binario tiene una precedencia menor que la multiplicación. La palabra clave "%prec", cambia el nivel de precedencia asociado con una regla gramatical particular. "%prec" aparece inmediatamente despues del cuerpo de la regla gramatical, antes de la acción o del punto y como que cierra la regla, y está seguido por un nombre de token o literal. Provoca que la precedencia de la regla gramatical convenga con la del siguiente nombre de token o literal. Por ejemplo, hacer que el menos unario tenga la misma precedencia que la multiplicación, podría parecerse a la siguientes reglas:

```
%left '+' '-'
%left '*' '/'

%%

expr: expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | '-' expr %prec '*'
    | NAME
```

Un token declarado por "%left", "%right", y "%nonassoc" no necesitan, pero podrían, ser declarados también por "%token". YACC usa las precedencias y asociatividades para resolver los conf lictos de análisis; éstos dan lugar a las reglas sin ambigüedades. Formalmente, las reglas funcionan así:

1. Las precedencias y las asociatividades son grabadas para estos tokens y literales que los tienen.

2. Una precedencia y una asociatividad está asociada con cada regla gramatical; es la precedencia y la asociatividad del último token o literal en el cuerpo de la regla. Si la construcción "%left", invalida éste defecto. Algunas reglas gramaticales pueden no tener precedencia y asociatividades asociadas a ellas.

3. Cuando hay un conflicto reduce/reduce, o un conflicto shift/reduce y cada uno de los símbolos de entrada o las reglas Iramaticales no tienen precedencia ni asociatividad, entonces se usan las dos reglas sin ambigüedades dadas al principio de la seccion, y se informa de los conflictos.

4. Si hay un conflicto shift/reduce y tanto la regla gramatical como el caracter de entrada tienen precedencia y asociatividades asociadas con ellos, entonces el conflicto se resuelve en favor de la acción (shift o reduce) asociada con la precedencia mayor. Si las precedencias son las mismas entonces se utiliza la asociatividad; la asociatividad por la izquierda implica reduce, la asociatividad por la derecha implica shift, y la no asociatividad implica error.

Los conflictos resueltos por la precedencia no se cuentan en el número de conflictos shift/reduce y reduce/reduce presentados por YACC. Esto significa que los errores en la especificación de las precedencias pueden ocultar errores en la gramática de la entrada; es una buena idea prescindir de las precedencias, y utilizarlas básicamente como un "libro de cocina",, hasta que se haya conseguido alguna experiencia. El fichero lly.output" es muy útil al decidir si el analizador sintáctico está de verdad haciendo lo que se propone.

Capítulo 7

Tratamiento de errores sintácticos en yacc

Algún autor ha indicado que, atendiendo a su uso habitual, un compilador se debería definir como una herramienta informática cuyo objetivo es encontrar errores. Y es cierto, pues la mayor parte de las veces los compiladores deben enfrentarse a textos de entrada que son incorrectos. Por lo tanto, será preciso tratar muy bien las situaciones de error si queremos que nuestro compilador sea apreciado por los usuarios.

En este capítulo trataremos el problema de los errores sintácticos y su solución. Aprenderemos a construir *parsers* LR o recursivos descendentes robustos que tratan adecuadamente los textos con errores, discutiremos las técnicas más habituales para conseguir ésto y además daremos algunas indicaciones sobre las causas más comunes de errores y la forma en que se deben mostrar a los usuarios.

7.1 Conceptos generales

Como hemos indicado en la introducción, el tratamiento de los errores sintácticos debe estar muy cuidado en cualquier compilador, puesto que la mayoría de las veces estas herramientas se usan precisamente para eso: para encontrar errores en vez de para generar un código eficiente.

En esta sección trataremos varios conceptos generales relacionados con los errores. Comenzaremos por estudiar qué son los errores sintácticos, veremos de qué forma deben comportarse ante ellos los buenos compiladores, cómo se deben diseñar los mensajes de error y de qué forma se deben mostrar al usuario.

7.1.1 Los mensajes de error

Resulta bastante evidente que un *parser* magnífico capaz de encontrar el 95% de los errores en un fuente, sirve de bastante poco si no nos informa adecuadamente de su posición, sus causas y las posibles soluciones que podemos tomar. Pero ojo: tan malo resulta ser parcos como excesivamente verbosos.

La estructura elemental de un mensaje de error sintáctico debe ser la siguiente:

```
Error sintáctico en "fichero" (línea, columna):
  Se esperaba { token_1, token_2, ..., token_n }.
```

Esto es, la información mínima que un mensaje de error debe de proporcionarnos es el nombre del fichero, la línea y la columna en la que se ha producido el error y cuáles son los *tokens* válidos en esa posición. Por lo tanto, el analizador léxico del compilador deberá actualizar continuamente esta información que será accesible al exterior mediante las rutinas siguientes:

1. `int yynchars()`, para obtener el número de caracteres leídos.
2. `int yylineno()`, para obtener el número de línea.
3. `int yycolumn()`, para obtener el número de columna.
4. `const char *yyfilenm()`, para obtener el nombre del fichero.

Estas rutinas no son estándar en las herramientas de construcción de compiladores que nosotros vamos a estudiar en este trabajo, pero últimamente están apareciendo algunas nuevas [1] que incorporan varias de ellas.

También es importante que el *parser* indique si ignora *tokens* de la entrada o si inserta alguno, pues de esta forma el usuario podrá comprender mucho mejor su funcionamiento posterior y quizá algunos mensajes de error espurios provocados por estas acciones. Por ejemplo, ante la entrada `1 + + 3` en la calculadora, su *parser* podría producir los siguientes mensajes:

```
Error sintáctico en "(stdin)" (1, 5):
  Se esperaba { NUM }.
  Se ha insertado un 0.
```

7.1.2 Herramientas para la corrección de errores

Durante muchos años, los informes de error de los compiladores eran largos listados que el usuario podía imprimir en papel para a continuación dedicarse a la dificultosa tarea de eliminarlos manualmente.

En la actualidad, las máquinas tienen muchas capacidades que nos pueden ayudar a mejorar la forma en que se muestran los errores y se corrigen. Una de las posibilidades más simples pasa por tener un editor de textos capaz de interpretar los errores que da el compilador y situar el cursor automáticamente en la posición del fuente en que se encuentran.

Existen editores capaces de entender casi cualquier mensaje de error, puesto que permiten programar su estructura utilizando expresiones regulares o gramáticas de contexto

[1] `bison` y `flex`, por ejemplo.

libre con algunas restricciones. Otros son más modestos y tan sólo son capaces de interpretar mensajes con una estructura predeterminada. En cualquier caso, es muy deseable que el compilador tenga en cuenta la existencia de estos editores y produzca mensajes de error con una estructura sencilla, pero suficientemente informativa.

7.2 Detección, recuperación y reparación de errores en analizadores LR

La incorporación de mecanismos adecuados para tratar las situaciones de error ha sido uno de los problemas más difíciles de solucionar en la familia de *parsers* LR. Nos concentraremos en la familia LALR(1) y en concreto en la herramienta `yacc`.

7.2.1 La rutina estándar de errores

En la sección siguiente estudiaremos de qué forma se comportan los *parsers* generados por `yacc` con los errores, pero antes es preciso conocer cómo informa de ellos.

Para tal misión, `yacc` exige al usuario que suministre una rutina con el siguiente interface:

```
void yyerror(const char *msg)
```

El *parser* llamará a esta rutina cada vez que encuentre un error en el fuente, pasándole una cadena de caracteres que describe su naturaleza. Esta cadena suele ser el mensaje `syntax error` o alguno parecido.

La definición habitual de la rutina es la siguiente:

```
void yyerror(const char *msg) {
  fprintf(stderr, "Error:  %s\n", msg);
}
```

Por lo tanto, parece que uno de los componentes del *parser* que necesitará de una mayor atención es precisamente esta rutina. Más adelante estudiaremos de qué forma se puede mejorar de tal manera que además de ese mensaje estándar nos proporcione la línea y la columna en la que se ha encontrado el error, y la lista de *tokens* admisibles en esa posición.

7.2.2 Comportamiento predefinido ante los errores

Para ver cómo se comporta el *parser* le proponemos que observe qué ocurre al ofrecer la cadena `1++2+3+4++5` al *parser* que `yacc` genera para la siguiente gramática:

```
%token NUM /* Un número decimal */
```

```
%%

exp
   : exp '+' exp
   | NUM

%%

... yyerror(), yylex() y main() ...
```

Si completa esta gramática [2] y la compila verá como el programa generado le responde con un sencillo `syntax error`, que resulta poco informativo. Además tan sólo ha detectado un error cuando en la cadena de entrada realmente existen dos.

Este es el comportamiento por defecto de un *parser* LALR(1): siempre que se puedan llevar a cabo transiciones en el autómata continua analizando la entrada, y en aquellos casos en que es imposible, se muestra un mensaje de error y se detiene el proceso.

7.2.3 Un mecanismo elemental de tratamiento de errores

El comportamiento por defecto del *parser* es claramente insatisfactorio, pues no nos permite detectar más que un error en cada compilación.

Una técnica muy básica para detectar más de un error consiste en añadir a la gramática algunas producciones con los errores más comunes. Por ejemplo, si suponemos que en el lenguaje de la calculadora es frecuente equivocarse y escribir dos operadores juntos, podríamos añadir una regla como la siguiente:

```
exp
   : exp '+' exp
   | exp '+' '+' exp            { yyerror("Dos operadores juntos"); }
   | NUM
```

Esta técnica es muy sencilla, pero resulta difícil de generalizar su uso puesto que a priori es imposible determinar en qué situaciones puede cometer errores el usuario.

7.2.4 Mecanismo elaborado de tratamiento de errores

Una solución mucho mejor pasa por tratar los errores de la entrada como un tipo especial de símbolo terminal. Así es como lo hace `yacc`, puesto que en su lenguaje acepta la palabra reservada `error` como un símbolo terminal predefinido que nos ayuda a solucionar las situaciones de conflicto. `error` se puede utilizar en las reglas de producción como cualquier otro terminal, con la única diferencia de que es un *token* ficticio no devuelto por el analizador léxico, sino generado por el propio *parser* cuando en el estado actual del autómata de reconocimiento no es posible realizar ninguna transición con el actual símbolo

[2] Esta gramática tiene un conflicto shift/reduce, pero se resuelve por defecto de la forma que nos interesa.

de entrada. Una vez que el *token* `error` se ha generado internamente de esta forma, el *parser* llama a la rutina `yyerror()` y lo acepta como si se tratase de cualquier otro.

Consideremos por ejemplo la siguiente modificación de la gramática que nos sirve de ejemplo:

```
exp
   : exp '+' exp
   | NUM
   | error
```

En este caso el *parser* generado por `yacc` aceptará entradas erróneas sin detenerse en el primer error, pero para comprender bien por qué lo hace es preciso estudiar el autómata que `yacc` produce en el fichero `y.output` al darle la opción `-v`. Es el siguiente:

```
state 0
        $accept : _exp $end

        error  shift 3
        NUM  shift 2
        .  error

        exp  goto 1

state 1
        $accept :  exp_$end
        exp :  exp_+ exp

        $end  accept
        +  shift 4
        .  error

state 2
        exp :  NUM_    (2)

        .  reduce 2

state 3
        exp :  error_    (3)

        .  reduce 3

state 4
        exp :  exp +_exp

        error  shift 3
        NUM  shift 2
        .  error
```

```
        exp  goto 5

5: shift/reduce conflict (shift 4, red'n 1) on +
state 5
        exp :  exp_+ exp  (1)
        exp :  exp + exp_

        +  shift 4
        .  reduce 1

        exp  goto 5
```

Fíjese en que la palabra `error` aparece como un *token* en los estados 0 y 4, pero también como una operación en los estados 0, 1 y 4. A continuación estudiaremos con detalle el funcionamiento de este *token* especial en todas las situaciones posibles.

`error` como un *token* ficticio

Veamos de qué forma reacciona este *parser* ante la entrada errónea `1++2`. En la siguiente tabla recogeremos la evolución de la entrada, la pila y la acción que lleva a cabo el *parser* en cada momento.

Entrada	Pila	Acción
1++2		push [0]
++2	[0]	shift NUM, push NUM [2]
++2	[0] NUM [2]	reduce exp : NUM, push exp [1]
++2	[0] exp [1]	shift +, push + [4]
+2	[0] exp [1] + [4]	error

Como vemos en la traza, tras llegar al estado 4 [3] no existe ninguna transición válida con el carácter `+`, por lo que el *parser* invocaría la rutina `yyerror()` para mostrar el oportuno mensaje de error y produciría internamente el *token* de entrada `error`. Si consultamos la tabla del autómata resulta que ante este *token* se debe producir la secuencia de acciones `shift error`, `push error` [3], con lo que la traza podría continuar tal y como se indica a continuación.

[3] Para distinguir en la pila los estados, los hemos escrito dentro de un recuadro. Por ejemplo [0], [1], etcétera.

Entrada	Pila	Acción
...	...	...
+2	0 exp 1 + 4	error
+2	0 exp 1 + 4	shift error, push error 3
+2	0 exp 1 + 4 error 3	reduce exp : error, push exp 5
+2	0 exp 1 + 4 exp 5	shift +, push + 4
2	0 exp 1 + 4 exp 5 + 4	shift NUM, push NUM 2
	0 exp 1 + 4 exp 5 + 4 NUM 2	reduce exp : NUM, push exp 5
	0 exp 1 + 4 exp 5 + 4 exp 5	reduce exp : exp + exp, push exp 5
	0 exp 1 + 4 exp 5	reduce exp : exp + exp, push exp 1
	0 exp 1	accept

Por lo tanto, si resumimos la secuencia de reducciones que nos ha llevado a llevado a aceptar esta cadena resulta que es la siguiente:

$$\begin{aligned}&\text{exp} \rightarrow \text{exp} + \text{exp} \rightarrow \text{exp} + \text{exp} + \text{exp} \rightarrow \\ &\rightarrow \text{exp} + \text{exp} + \text{NUM} \rightarrow \text{exp} + \text{error} + \text{NUM} \rightarrow \\ &\rightarrow \text{NUM} + \text{error} + \text{NUM}\end{aligned}$$

Es decir, que en este caso el *token* `error` se ha comportado como un *token* ficticio que se ha insertado entre los dos operadores de suma, de forma que la expresión que realmente se ha reconocido ha sido `1 + error + 2`, en donde `error` representa a cualquier valor válido para el *token* `NUM`, el único posible en esa situación.

`error` para ignorar parte de la entrada

Como vemos, el mecanismo de corrección de errores se ha comportado bastante bien, pero aún se le puede obtener más juego ya que en ocasiones también es capaz de ignorar *tokens* de la entrada. En estos casos se suele decir que el *parser* "tira" parte de la entrada antes de continuar el reconocimiento. Podremos ver este comportamiento cuando le suministramos la cadena `1-+2`. En esta cadena se supone que el signo `-` es un *token* que existe en el contexto de nuestra gramática pero que no debe aparecer aquí. Si utilizamos como rutina de análisis léxico la que se muestra a continuación, este *token* erróneo se produce de una forma muy simple:

```
int yylex(void) {
  int c = getchar();

  return isdigit(c) ?  NUM : c
}
```

Para ver la forma en que el *parser* ignora el *token* incorrecto haremos, como en el caso anterior, una traza de su funcionamiento.

Entrada	Pila	Acción
1-+2		push 0
-+2	0	shift NUM, push NUM 2
-+2	0 NUM 2	reduce exp : NUM, push exp 1
-+2	0 exp 1	error

De nuevo nos encontramos en una situación en la que no existe ninguna transición para el *token* que hay en la entrada. Al igual que antes, el *parser* elabora internamente un *token* `error` y empieza a comportarse como si el analizador léxico se lo hubiese enviado. El problema es que en el estado 1 del autómata no hay prevista ninguna transición con este *token* . En estas situaciones el *parser* comienza a desapilar símbolos de la pila, y lo hace hasta que se produzca un *underflow* o hasta que en el pila aparezca un número de estado para el cual sí están previstas transiciones etiquetadas con el *token* `error`. En el primer caso, la rutina de análisis sintáctico falla produciendo como resultado un valor distinto de cero, y en el segundo continua el reconocimiento a partir del estado que quede en la cima de la pila.

En nuestro ejemplo hemos tenido suerte, pues basta con desapilar dos elementos de la pila para encontrar el estado 0 en el que sí se permiten transiciones de error.

Entrada	Pila	Acción
...	...	...
-+2	0 exp 1	error
-+2	0 exp 1	pop 1 exp, push error 3
-+2	0 error 3	reduce exp : error, push exp 1
-+2	0 exp 1	...

Según se ve en la traza, después de desapilar hasta dejar en la pila tan sólo el estado 0, se ha producido la reducción `exp : error`, en la cima volvemos a tener el estado 1 y el *token* - sigue en la entrada. Es evidente que se repitiésemos el mismo proceso entraríamos en un bucle sin fin, por lo que el *parser* evita la posibilidad de que esto ocurra eliminándo el *token* - de la entrada, ignorándolo.

Una vez hecho puede continuar el proceso de reconocimiento de la forma normal.

Entrada	Pila	Acción
...	...	...
-+2	0 exp 1	shift -
+2	0 exp 1	shift +, push + 4
2	0 exp 1 + 4	shift NUM, push NUM 2
	0 exp 1 + 4 NUM 2	reduce exp : NUM, push exp 5
	0 exp 1 + 4 exp 5	reduce exp : exp + exp, push exp 1
	0 exp 1	accept

Modo normal, de sincronización y de recuperación

Ya sabemos que el token `error` se puede utilizar en unas ocasiones para suplir símbolos que faltan en la entrada y en otras para descartar basura. También sabemos que cuando

se incluyen producciones de error, el proceso de reconocimiento continua a menos que se produzca un *underflow* de la pila. ¿Pero qué ocurre exactamente cuando dos errores aparecen uno casi al lado del otro?. ¿Qué ocurre, por ejemplo, si a nuestro *parser* le suministramos para analizar la cadena `1++2-+3`?.

Si realizamos una traza a mano del funcionamiento del *parser* podremos convencernos de que los dos errores son efectivamente detectados, pero si probamos con el *parser* generado por `yacc`, resulta que tan sólo se obtiene un mensaje `syntax error`.

La razón de que ocurra esto es que el *parser* generado por `yacc` utiliza una heurística sencilla que le permite evitar las cascadas de errores que se producen con frecuencia al encontrar uno en un fuente. Esta heurística dice que el *parser* no generará ningún mensaje de error hasta que haya logrado leer satisfactoriamente al menos tres *tokens* de la entrada después de haber detectado un error. Por eso, en el ejemplo anterior, el segundo error se detecta pero no se llama a la rutina `yyerror()` de nuevo hasta que se logre leer al menos tres *tokens* . Este número es en cierto modo mágico, pues fue elegido atendiendo a estudios experimentales y hoy en día todos los *parsers* derivados de `yacc` siguen esta norma.

Aunque existen situaciones en que esta heurística funciona muy bien, hay otras como la que nos ocupa, en las que este funcionamiento no es del todo adecuado. Por eso `yacc` proporciona una acción predefinida, `yyerrok`, que cuando se ejecuta instruye al *parser* para que vuelva a su modo de funcionamiento normal. El *parser* se puede encontrar en tres modos diferentes:

1. Normal. Es el modo habitual de funcionamiento cuando no se están produciendo errores. Este es el único modo de operación en el que se invoca la rutina `yyerror()`.
2. Sincronización. Se dice que el *parser* se encuentra en este modo cuando tras encontrar un error empieza a desapilar elementos hasta encontrar en la cima un estado que admita transiciones con el *token* `error`.
3. Recuperación. En el caso de encontrar en la cima de la pila algún estado que admita transiciones de error, se lleva a cabo dicha transición y se entra en el modo de recuperación. Se permanecerá en él hasta que se logre leer en la entrada tres *tokens* correctos. En el modo de recuperación, todos los *tokens* que no admitan transiciones se ignoran inmediatamente.

Al ejecutar la acción predefinida `yyerrok`, el *parser* vuelve inmediatamente al modo de operación normal, por lo que es necesario utilizarla con mucho cuidado. Por ejemplo, si nuestra gramática tiene una regla de la forma:

```
exp
   : ...
   | error                          { yyerrok; }
```

el *parser* se ciclará irremediablemente ante una entrada de la forma `1-+2`. La razón es que si en el mismo momento en que se detecta el error se pasa al modo normal, el *parser* nunca podrá descartar el *token* `-` de la entrada.

En nuestro ejemplo, parece bastante sensato asociar la acción `yyerrok` a la regla `exp: NUM`, puesto que parece evidente que al leer un número correcto podremos reducir al menos una parte de la expresión completa y detectar más errores.

Dónde colocar los *tokens* `error`

Es evidente que al utilizar `yacc` hay que ser un poco habilidosos, de forma que las producciones de error y las acciones `yyerrok` estén colocadas en los sitios adecuados para que no se descarten demasiados *tokens* de la entrada y tampoco se produzcan ciclos.

En esta sección tan sólo daremos algunas sugerencias basadas en la experiencia. En general la posición de los *tokens* de error debe venir guiada por las siguientes reglas:

1. Tan cerca como sea posible del símbolo principal de la gramática, de forma que nunca se pueda producir un *underflow* de la pila cuando nos encontramos en el modo de sincronización.

2. Tan cerca como sea posible de símbolos terminales importantes de la gramática. Son símbolos terminales importantes aquellos que delimitan secciones de la entrada, por ejemplo el `';'` que indica el final de cada sentencia en la mayoría de los lenguajes de programación, el `')'` que cierra las listas de parámetros o las palabras `'end'` o `'}'` que cierran el final de las sentencias compuestas.

   Al hacer esto se persigue que en cada error se descarte la menor cantidad posible de la entrada. Si combinamos bien estas reglas con acciones `yyerrok` podremos obtener *parsers* que detectan muchos errores en cada pasada.

3. En cada construcción recursiva, de forma que la presencia de un error en alguno de sus componentes no nos haga descartar toda la lista de elementos.

4. Preferiblemente, `error` no debería aparecer al final de las reglas de producción, puesto que si se ejecuta demasiado pronto una acción `yyerrok` el *parser* podría ciclarse.

5. Lo más cerca posible de los lugares en los que habitualmente se comenten errores. Por ejemplo, en las comas que sirven para separar elementos de una lista, en los paréntesis de cierre de las expresiones, en medio de las expresiones binarias, etcétera.

6. Sin introducir conflictos shift/reduce. Este objetivo puede ser bastante difícil de conseguir, puesto que la presencia de este *token* suele ser bastante conflictiva. En muchos casos hay que cambiar la gramática que pensamos inicialmente y rehacerla sacando factor común o aplicando otras transformaciones que nos permitan eliminar los conflictos introducidos.

Es evidente que muchos de estos objetivos son contradictorios entre sí, pues por ejemplo, si deseamos hacer recuperación de errores en una construcción recursiva, tendremos que colocar alguna vez el *token* `error` al final de una producción. En la práctica el problema de colocar adecuadamente las producciones de error no tiene solución única y estas heurísticas tan sólo son una guía.

Como ejemplo, podemos ver los caso más frecuentes de construcciones repetitivas. Son los siguientes:

1. Lista opcional sin separadores

```
lista
   : λ
   | lista item                { yyerrok; }
   | lista error
```

2. Lista obligatoria sin separadores

```
lista
   : lista item                { yyerrok; }
   | item
   | lista error
   | error
```

3. Lista obligatoria con separadores

```
lista
   : lista SEP item            { yyerrok; }
   | item
   | lista error
   | lista error item          { yyerrok; }
   | lista SEP error
   | error
```

Los tres ejemplos que se han mostrado son bastante exhaustivos en el tratamiento de los errores y podríamos incluso pensar que al seguirlos obtendremos *parsers* robustos [4]. En la práctica todo depende que la versión de `yacc` que estemos utilizando: si nuestra implementación es un derivado de la serie `BSD 4.2`, entonces tendremos algunos problemas, pues en todas estas versiones existe un error: si en un estado la acción por defecto es reduce y no existe ninguna transición etiquetada con el siguiente *token* de la entrada, el *parser* opta por la reducción, cuando realmente debería optar por una transición con el *token* `error`. Esto significa que el error se detectará, pero demasiado tarde, cuando ya hemos eliminado de la pila un estado en el que probablemente se admitían transiciones para el *token* `error`. El efecto de esto puede ser un *parser* que vacía inesperadamente la pila ante una entrada errónea [5].

Estas producciones de error se pueden mejorar considerablemente cuando se utilizan en una gramática más compleja que las incluye. Veamos, por ejemplo, el caso de las listas de parámetros en la cabecera de un procedimiento:

```
cabecera
   : PROCEDURE ID '(' lista_par ')'
   | PROCEDURE ID '(' error ')'

lista_par
   : lista_par ';' param
   | param
```

[4] Por favor, no tome esta heurística como un dogma de fé universalmente válido. Cada problema requerirá un tratamiento especial de error, y lo que se ha mostrado es tan sólo una sugerencia

[5] Las versiones de `yacc` de Abraxas y AIX, contienen este error.

```
   | lista_par error ID

param
   : ID
   | error
```

En este caso hemos eliminado la regla `lista_par : error` y la hemos embebido en la segunda alternativa de `cabecera`. De esta forma el *token* error se encuentra delante del *token* `')'` que le servirá de recuperación en caso de conflicto. Las restantes reglas eliminadas de `lista_par` las hemos introducido en la regla `param`, esto es, siempre lo más cerca posible de aquellos puntos en los que se producen errores.

Dónde colocar las acciones `yyerrok`

La acción `yyerrok` se debe colocar, a parte de donde hemos indicado, detrás de aquellos símbolos terminales de la gramática que sirvan de delimitadores de secciones. Por ejemplo, suelen ser de gran interés el paréntesis de cierre, el punto y coma delimitador del final de una sentencia, y la palabra que pone fin a sentencias compuestas, procedimientos, clases, tipos, etcétera.

Si lo hacemos así, podremos comprobar que esta acción se convierte en una de las más repetidas, por lo que lo mejor suele ser crear nuevos símbolos no terminales con el aspecto siguiente:

```
par_ce
   : ')'                        {  yyerrok;  }

punto_coma
   : ';'                        {  yyerrok;  }

end
   : END                        {  yyerrok;  }
```

De esta forma, cada vez que necesitemos un paréntesis cerrado, en vez de usar el terminal `')'` usaremos el no terminal `par_ce` que lo reconoce y además ejecuta la acción `yyerrok`.

7.2.5 Ejemplo: Una calculadora avanzada

Como ejemplo de todo lo que hemos estado viendo, daremos la especificación completa en `yacc` de una calculadora avanzada. Cuenta con 26 registros identificados con letras entre la `a` y la `z`, admite las operaciones aritméticas tradicionales y la expresión `if ... then ... else`.

Algunos ejemplos de las expresiones posibles son los siguientes:

1. `? 1 + 2`, mostrar el resultado de 1+2.

2. `a := 2`, cambiar el valor del registro `a` por 2.

3. `b := if a > 2 | a < 0 then 1 else 2 end`, asignar al registro `b` el valor `1` si `a` $>$ `2` ó si `a` $<$ `0`. En otro caso asignarle un `2`.

Hemos elegido este ejemplo porque es bastante simple, pero de una gran utilidad, puesto que en los fuentes de los programas son precisamente las expresiones las construcciones que aparecen con mayor frecuencia. Por lo tanto estudiar una gramática con corrección de errores para las expresiones será de un gran interés.

La gramática de la calculadora

A continuación se muestra una gramática lista para ser suministrada a `yacc`. La definición del analizador léxico no se incluye, puesto que es muy sencilla.

```
%{
    #include <stdio.h>
    #define ERROR(e) fprintf(stderr, "REGLA: %s\n", e);
%}

%token NUM REG ASIG MAI MEI IF THEN ELSE END

%%

prog
    : prog linea
    | linea

linea
    : '?' exp ret
    | REG ASIG exp ret
    | ret
    | error exp ret        { ERROR("linea: error exp ret"); }
    | REG error exp ret    { ERROR("linea: REG error exp ret"); }
    | error ret            { ERROR("linea: error ret"); }


exp
    : exp '|' exp_and
    | exp_and
    | exp error exp_and    { yyerrok; ERROR("exp: exp error exp_and"); }
    | error exp            { yyerrok; ERROR("exp: error exp"); }

exp_and
    : exp_and '&' exp_rel
    | exp_rel

exp_rel
    : exp_adit '<' exp_adit
```

```
    | exp_adit '>' exp_adit
    | exp_adit MAI exp_adit
    | exp_adit MEI exp_adit
    | exp_adit '=' exp_adit
    | exp_adit

exp_adit
    : exp_adit '+' exp_mult
    | exp_adit '-' exp_mult
    | exp_mult

exp_mult
    : exp_mult '*' exp_base
    | exp_mult '/' exp_base
    | exp_base

exp_base
    : IF exp THEN exp ELSE exp end
    | NUM
    | REG
    | '(' exp par_ce
    | '!' exp_base
    | '(' error par_ce    { ERROR("exp_base; ( error par_ce"); }
    | '(' exp error       { yyerrok; ERROR("exp_base: ( exp error"); }

ret
    : '\n'    { yyerrok; }

par_ce
    : ')'    { yyerrok; }

end
    : END    { yyerrok; }
```

Conflictos

Al compilar esta gramática se obtienen dos conflictos shift/reduce. El fichero `y.output` nos indica que se localizan en el estado 31 del autómata.

```
...
31: shift/reduce conflict (shift 28, red'n 13) on |
31: shift/reduce conflict (shift 29, red'n 13) on error
state 31
        exp :  exp_| exp_and
        exp :  exp_error exp_and
        exp :  error exp_    (13)

        error  shift 29
        |  shift 28
```

```
        .  reduce 13
...
```

Ambos indican que después de sintetizar en la pila la cadena `error exp`, si en el fuente aparece un signo '|' o bien un `error` el parser puede optar por apilar estos símbolos o por reducir la regla `exp: error exp`. De acuerdo con el método habitual de resolver conflictos, sabemos que el parser se decantará por apilar el terminal en vez de por realizar una operación reduce.

Por lo tanto, si queremos que nuestro parser encuentre dentro de las expresiones tantos errores como sea preciso, entonces este comportamiento es bastante adecuado.

Los *tokens* importantes de la gramática

Tal y como indicamos en la sección ??, dentro de nuestra gramática existen varios *tokens* importantes que nos permiten asegurar que cualquier posible error ya ha sido corregido cada vez que los encontramos y realizamos alguna transición con ellos. Son el retorno de carro, que pone fin a las líneas de entrada, el paréntesis de cierre y la palabra `end` que aparece al final de las expresiones condicionales.

Por lo tanto, lo que hemos hecho es incluir en la gramática símbolos no terminales especiales que cada vez que se reducen ejecutan una acción `yyerrok`.

```
ret
    : '\n'    { yyerrok; }

par_ce
    : ')'    { yyerrok; }

end
    : END    { yyerrok; }
```

Tratamiento de errores en la producción `linea`

Dentro de las alternativas para `linea` se han incluido las siguientes reglas de error.

```
linea
    : error exp ret     { ERROR("linea: error exp ret"); }
    | REG error exp ret { ERROR("linea: REG error exp ret"); }
    | error ret         { ERROR("linea: error ret"); }
```

Generalmente, la primera vez que se intenta hacer una gramática con recuperación de errores, se tiende a colocar producciones de error por todas partes, intentando adivinar hasta el último sitio en que el usuario puede cometer un error. Esto, a parte de dificultar enormemente la depuración de la gramática, puesto que suelen aparecer muchísimos conflictos, no sirve de mucho en la práctica. Lo más adecuado es determinar en qué sitios aparecen errrores con más frecuencia y concentrarnos en resolver tan sólo esos casos.

En nuestra calculadora, hemos supuesto que dos errores bastante frecuentes pueden ser escribir una expresión directamente, sin precederla por un signo `'?'`, y equivocarse al colocar el signo de asignación. Un fuente como el siguiente, puede ser bastante habitual:

```
1 + 2
a = 3
```

Por eso entre las producciones para `linea`, hemos añadido dos reglas para reparar estos errores, pero en absoluto hemos intentado determinar todas las posibles situaciones de error. En cualquier momento, el usuario puede proporcionar a la calculadora un fuente con un error inimaginable, y en ese caso se activaría la tercera regla, que juega el papel de *último recurso* en el que ante un error imprevisto, se descarta toda la línea hasta encontrar un retorno de carro.

Al suministrar el fuente anterior, el parser produce el siguiente resultado:

```
1 + 2
syntax error
REGLA: linea: error exp ret
a = 3
syntax error
REGLA: linea: REG error exp ret
```

Esta salida demuestra que en la primera línea ha reparado el error insertando una `'?'` y en la segunda descartando el `'='` e insertando un `':='`.

El operador de menor precedencia

En todos los lenguajes, los operadores se jerarquizan atendiendo a unas reglas de precedencia que nos permiten deshacer ambigüedades. Todos sabemos que el resultado de una expresión como `1 + 2 * 3` es 7, puesto que el operador `*` tiene más precedencia que `+` y por lo tanto las operaciones en las que interviene se realizan siempre antes.

La forma de tratar los errores en las expresiones suele ser muy parecida en casi todos los lenguajes. El método más sencillo suele consistir en seleccionar el operador con menor precedencia y añadir a sus producciones reglas de error. En las reglas del resto de los operadores no se añade ninguna.

```
exp
    : exp error exp_and    { yyerrok; ERROR("exp: exp error exp_and"); }
    | error exp            { yyerrok; ERROR("exp: error exp"); }
```

Estas dos reglas de error son bastante generales. La primera nos permite reparar de forma específica aquellas situaciones en las que el usuario olvida escribir un operador entre dos expresiones. La atribución de esta regla supondrá que el operador que falta es justamente el operador menos prioritario del lenguaje. La segunda regla actua como un último recurso, para tratar otros casos de error no previstos.

En ambas situaciones se ejecuta la acción `yyerrok` para devolver el parser a su modo de funcionamiento normal. En otro caso podría ocurrir que descartasemos demasiados *token* de la entrada y no informásemos de algunos errores adyacentes.

Para ver como funcionan estas reglas podemos ofrecer al parser la entrada siguiente:

```
? 1 + 2 3
? 1 2
? 1 + end + 2 2
```

El resultado es el siguiente:

```
? 1 + 2 3
syntax error
REGLA: exp error exp_and
? 1 2
syntax error
REGLA: exp error exp_and
? 1 + end + 2 3
syntax error
REGLA: exp: error exp
syntax error
REGLA: exp error exp_and
```

En el primer y segundo caso, el parser ha detectado perfectamente la ausencia de un operador y lo ha insertado. El tercer caso es más complejo, pues en él primero se ha enfrentado a un error no previsto, la aparición de la palabra `end`, y en el segundo a la ausencia de un operador. En los dos casos ha reaccionado bien, pues primero ha reducido la regla de último recurso, y después la regla que inserta un operador ausente.

Las expresiones entre paréntesis

Suele ser bastante frecuente utilizar paréntesis para alterar la prioridad por defecto de los operadores al escribir expresiones complejas. En estos casos un error muy frecuente es olvidar cerrar alguno de los paréntesis de la expresión, por lo que en la regla para `exp_base` hemos añadido dos alternativas de error bastante útiles.

```
exp_base
    : '(' error par_ce    { ERROR("exp_base; ( error par_ce"); }
    | '(' exp error       { yyerrok; ERROR("exp_base: ( exp error"); }
```

La primera de las alternativas nos sirve de nuevo como una regla de último recurso cuando entre paréntesis encontramos un error que no hemos previsto. La segunda nos permite insertar paréntesis de cierre que el usuario ha olvidado.

Por ejemplo, ante la entrada:

```
? (end)
? (1+2
? ((1+2
```

el parser produce la siguiente salida:

```
? (end)
syntax error
REGLA: exp_base; ( error )
? (1+2
syntax error
REGLA: exp_base: ( exp error
? ((1+2
syntax error
REGLA: exp_base: ( exp error
syntax error
REGLA: exp_base: ( exp error
```

Es muy importante colocar la acción `yyerrok` en la última alternativa de la forma que hemos hecho, pues en caso contrario, el parser no sería capaz de detectar más que la falta de un paréntesis.

7.2.6 Otra forma de conseguir la misma calculadora

En esta sección mostraremos otra forma de resolver el mismo problema de la calculadora, pero en esta ocaasión, en vez de utilizar una gramática en la que la precedencia y asociatividad de los operadores queda implícita, usaremos una gramática ambigüa en la que estas características de los operadores se han indicado de forma explícita.

Mostraremos la gramática sin más comentarios, pues una vez estudiada la sección anterior, al lector no le costará el más mínimo trabajo entender por qué las producciones de error se han colocado en donde se encuentran.

```
%{
    #include <stdio.h>
    #define ERROR(e) fprintf(stderr, "REGLA: %s\n", e);
%}

%token NUM REG ASIG MAI MEI IF THEN ELSE END
%left     '|' '&'
%nonassoc '=' '<' '>' MAI MEI
%left     '+' '-'
%left     '*' '/'
%right    '!'

%%

prog
```

```
    : prog linea
    | linea

linea
    : '?' exp ret
    | REG ASIG exp ret
    | ret
    | error ret             { ERROR("linea: error ret"); }
    | error exp ret         { ERROR("linea: error exp ret"); }
    | REG error exp ret     { ERROR("linea: REG error exp ret"); }


exp
    : bin
    | exp '|' bin
    | exp error bin         { yyerrok;  ERROR("exp: exp error bin"); }
    | error exp             { yyerrok;  ERROR("exp: error exp"); }

bin
    : bin '&' bin
    | bin '<' bin
    | bin '>' bin
    | bin MAI bin
    | bin MEI bin
    | bin '=' bin
    | bin '+' bin
    | bin '-' bin
    | bin '*' bin
    | bin '/' bin
    | base

base
    : IF bin THEN bin ELSE bin end
    | NUM
    | REG
    | '(' bin par_ce
    | '!' bin
    | '(' bin error         { yyerrok; ERROR("base: ( bin error"); }
    | '(' error par_ce      { ERROR("base: ( error par_ce"); }

ret
    : '\n'   { yyerrok; }

par_ce
    : ')'    { yyerrok; }

end
    : END    { yyerrok; }
```

Capítulo 8

Árboles con atributos

8.1 Introducción

Lo que llevamos estudiado hasta ahora acerca de los lenguajes, nos permite especificar sus aspectos léxicos (expresiones regulares), sus aspectos sintácticos (gramáticas independientes del contexto), su estructura profunda (gramáticas abstractas) y su semántica estatica (gramáticas con atributos). A las dos primeras especificaciones le hemos encontrado una utilidad práctica, al construir herramientas que tomando como base dichas especificaciones producen reconocedores léxicos y sintácticos respectivamente.

Por su parte, la sintaxis abstracta y las gramáticas con atributos, además de proporcionarnos el conocimiento de la estructura del lenguaje, aún no nos ha aportado nada especialmente útil desde el punto de vista de la implementación. El objetivo de este capítulo es precisamente ese, mostrar como el conocimiento del lenguaje que nos proporciona la sintaxis abstracta nos va a servir para definir e implementar una estructura de datos que utilizaremos como representación intermedia de dicho lenguaje. La estructura de datos utilizada será un tipo particular de árbol, que se ajustará en cada caso a la estructura profunda del lenguaje que estemos procesando.

La razón por la que utilizaremos árboles como método de representación intermedio, es evidente si tenemos en cuenta que los lenguajes que estamos tratando se describen mediante gramáticas independientes del contexto, y que este tipo de gramáticas está estrechamente relacionada con los árboles de derivación. Precisamente esta relación, extrapolada a las gramáticas abstractas, hace que nos planteemos la representación de la estructura profunda de un lenguaje (determinada por su sintaxis abstracta) mediante lo que denominaremos *árboles de sintaxis abstracta*, o más genéricamente *árboles con atributos*.

Al existir una relación directa entre la estructura del lenguaje y la estructura del árbol que servirá para representar programas escritos en dicho lenguaje, nos tendremos que plantear la definición e implementación de distintos tipos de árboles para lenguajes que tengan estructuras diferentes. Con la idea de liberarnos totalmente de la tarea de implementar los árboles, propondremos una notación que nos permita especificar las características de un tipo de árbol, y que sirva como entrada a una herramienta que produzca automáticamente la implementación correspondiente. Esta notación, como se puede prever, va a resultar muy cercana a la notación utilizada para describir la sintaxis abstracta

del lenguaje.

8.1.1 Notación orientada a la manipulación de árboles

La especificación de los árboles con atributos es una variante de la notación usada para la sintaxis abstracta. Esta notación esta pensada para obtener una implementación de los mismos de manera automática. Para conseguirlo introducimos símbolos no terminales nuevos por cada parte derecha de no terminal con mas de una parte derecha. Igualmente eliminamos símbolos superfluos que solo aparecian en la sintaxis abstracta para conseguir legibilidad. La declaración de instacias de de las categorías sintácticas la pasamos a la parte de reglas y solo usamos instancias en los casos necesarios.

Asi el ejemplo de sintaxis abstracta:
Categorías sintácticas

b ∈ Bloque
a ∈ Asig
n ∈ Num
v ∈ Var
ob ∈ Opb
ou ∈ Opu
e, e1, e2 ∈ Exp

Atributos

< v:int > Exp, Num
< e:Ent > Var

Definiciones

```
b  : a*
   ;
a  : v = e
   ;
e  : v
   | n
   | ou e1
   | e1 e2 ob
   ;
ou : -
   ;
ob : + | - | * | /
   ;
```

quedaría:

Categorías sintácticas

Bloque
Asig
Num
Var
Unaria
Binaria
Opb
Opu
Exp

Atributos

< v:int Exp, Num
< e:EntVar

Definiciones

```
Bloque : Asig*
       ;
Asig   : v:Var ;e:Expr
       ;
Exp    : Var | Num | Unaria | Binaria

       ;
       | n
Unaria | ou:Opu; e:Exp
       ;
Binaria| e1,e2:Exp; ob:Opb
       ;
Opu    : Menos
       ;
Opb    : Suma | Resta | Mult | Div
       ;
```

8.1.2 Formato de entrada

La información anterior para hacerla mas facilmente procesable la vamos a escribir en una estructura muy similar a una especificación para la herramienta YACC, ya que realmente lo que se especifica es algo muy parecido a una gramática independiente del contexto, aunque se interpreta como una definición de un árbol. La especificación se escribirá en un fichero con extensión ".esa" y con el siguiente formato:

```
%{
  CODIGO DE DECLARACION
%}
%union{
   tipo1 campo1;
```

```
   tipo2 campo2;
     ...
   tipoN campoN;}
%type <campo1> LISTA DE CATEGORIAS
   ...
%type <campoN> LISTA DE CATEGORIAS
%%
REGLAS
```

La zona de código de declaración está destinada a la inclusión de ficheros de cabecera para poder utilizar símbolos (generalmente nombres de tipos) propios de otros módulos.

En la sección *union* se enumeran los distintos atributos que podrán tener los constructores, definiendolo a través de un nombre de tipo (que será un tipo básico de C o un tipo declarado en alguno de los ficheros incluidos anteriormente) y un nombre de campo que servirá para diferenciar a cada uno de los atributos.

En cada línea *type* se denota que los constructores que aparecen en la lista correspondiente tienen un atributo del tipo que se indica entre < y > a través del nombre de un campo de *union*. Un constructor puede aparecer en más de una línea *type*, con lo que se permite que un constructor tenga más de un atributo. Los constructores que aparezcan en la parte derecha de una producción de elección, heredarán automáticamente los atributos que tenga el constructor de la parte izquierda. Además pueden tener otros atributos.

Por último en la zona de reglas se describirá la estructura de las categorias.

El siguiente ejemplo presenta la especificación árboles con atributos capaz de almacenar programas con la sintaxis abstracta descrita arriba.

```
%{
#include "tipos.h"
%}
%union{
    int v;
    Entrada e;}
%type <v> Exp
%type <e> Var
%%
Bloque :: Asig *
Asig :: v: Var; e: Exp
Exp:: Var | Num | Unaria | Binaria
Binaria :: e1, e2: Exp; ob: Opb
Unaria :: e:Exp; ou: Opu
Opb :: Suma | Resta | Mult | Div
Opu :: Menos
```

Como se puede ver, la notación utilizada en la zona de declaraciones es muy similar a la de la herramienta YACC para especificar los atributos de los símbolos, a excepción de las reglas de desambiguación que no tienen sentido a la hora de especificar árboles.

8.2 Taxonomía de las reglas: Constructores y Tipos

Si analizamos la estructura de las reglas de las especificaciones nos damos cuenta de que sólo nos encontramos tres tipos de esquemas sintácticos, la *agregación*, la *secuencia* y la *elección*. La agregación aparece en reglas cuya parte derecha se compone de un número fijo de elementos heterogéneos. La repetición o secuencia permite la posibilidad de construir listas de un número variable de elementos homogéneos. Y por último, la elección proporciona diferentes alternativas para la estructura de un determinado objeto. Las categorías sintácticas que no aprarecen en la parte izquierda de una regla seran terminales.

Una característica de esta notación es que no se permite que se mezclen los distintos tipos de producciones, esto es así con la idea de obtener una descripción clara de la estructura del árbol. De esta forma simplemente sabiendo el tipo de un determinado constructor (agregado, secuencia o elección), sabremos que operaciones tendremos que utilizar para manipularlo.

Para conseguir una implementación a partir de la especificación debemos definir el conjunto de funciones que nos sirvan para construir los árboles. Estas funciones las llamaremos constructores. Igualmente necesitamos un conjunto de tipos asociados a los árboles que construimos.

Para ello usaremos el siguiente procedimiento:

- Una categoría sintáctica terminal define un constructor con el nombre de la categoria. Este constructor no tiene parámetros.
- Una regla de tipo agregado define un constructor con el mismo nombre de la categoría en la parte izquierda. Los parámetros formales de este constructor son los indicados en la parte derecha. Los tipos de los parámetros formales son indicados en la parte derecha igualmente.
- Una producción de tipo secuencia define varios constructores con el nombre de la parte izquierda y algunos sufijos. Los constructores son:
  - Uno que no toma nigun parámetro y devuelve una secuencia vacía (su nobre se forma concatenando *_v* al nombre de categoría sintáctica,
  - Un segundo constructor toma dos parámetros: el primero del tipo indicado en la parte derecha y el segundo de tipo secuencia de esos elementos. Su nombre se forma concatenado *_dl* al nombre de la categoría sintáctica.
  - Un tercer constructor toma dos parámetros: el primero de tipo secuencia de los elementos de la parte derecha y el segundo del mismo tipo que los elementos de la parte derecha. Su nombre se forma concatenado *_dt* al nombre de la categoría sintáctica.
  - El primer constructor forma una secuencia vacia, el segundo va formando secuencias añadiendo elementos por delante y el tercero por atras.
  - Además de los constructores anteriores podemos disponer de otros que construyan una secuencia de uno, dos, etc, elementos. Los nombres de estos constructores se formaran con el sufijo _1, _2, _3, etc.
- La producción elección define un tipo asociado a cada símbolo de la regla. Define además que los tipos asociados a la parte derecha son subtipos del tipo asociado a la parte izquierda.

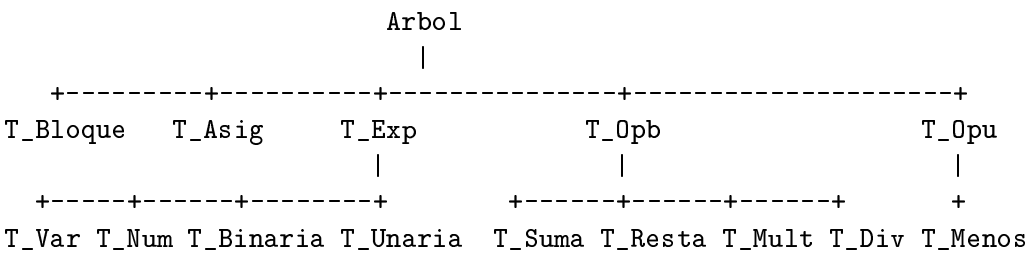
Por convención los identificadores de los constructores serán de los de la categoría sintáctica asociada cambiando la primera letra a mayúscula. Los identificadores de los tipos los formaremos concatenando $T_$ al nombre de la categoría sintáctica asociada.

Deberemos destacar uno de los constructores de la especificación como constructor *Raiz*, este constructor representará al elemento de mayor nivel del lenguaje (Programa, Paquete y Modulo son ejemplos de constructores raiz en los lenguajes PASCAL, Ada y Modula-2, respectivamente) y constituirá la raiz del árbol de un programa escrito en dicho lenguaje.

Los tipos que se derivan de la especificación son:

```
Arbol
T_Bloque
T_Asig
T_Exp
T_Binaria
T_Unaria
T_Var
T_Num
T_Opb
T_Suma
T_Resta
T_Mult
T_Div
T_Opu
T_Menos
```

Entre ellos existe una relación de subtipado que se deduce de la especificación.




```
T_Bloque = Seq(Asig)
```

Siempre aparecerá un tipo árbol que un supertipo de todos los definidos. Asumimos que hay un valor *Vacio* que pertenece al tipo Arbol.

El conjunto de constructores que se definen a partir de la especificación es

```
T_Bloque  Bloque_v();
T_Bloque  Bloque_dl(T_Asig e, T_Bloque b);
T_Bloque  Bloque_dt(T_Bloque b, T_Asig e);
T_Bloque  Bloque_1(T_Asig e);
T_Bloque  Bloque_2(T_Asig e1,e2);
```

```
T_Asig Asig(T_Var v, T_Exp e);
T_Binaria Binaria(T_Exp e1,e2, T_Opb ob);
T_Unaria  Unaria(T_Exp e,T_Opu ou);
T_Var Var();
T_Num Num();
T_Suma Suma();
T_Resta Resta();
T_Mult  Mult();
T_Div Div();
T_Menos Menos();
```

A partir de los constructores definidos podemos construir árboles. Un árbol se constuirá con una llamada a un constructor terminal o a un constructor no terminal tomando como parámetros árboles de los tipos adecuados.

Un árbol posible es

```
Binaria(
        Binaria(Var(),Num(),Suma()),
        Unaria(Num(),Menos()),
        Mult()
        );
```

Que representaría el árbol asociado a la expresión del tipo (x+3)*(-2).

Tambien es posible construir árboles que tienen subárboles no construidos. Como por ejemplo:

```
Binaria(
        Binaria(Var(),Num(),Suma()),
        a,
        Mult()
        );
```

donde a es una variables tipo T_Exp. Esta variable puede ser instanciada posteriormente por un árbol de ese tipo. Los árboles con variables pueden ser adecuados para repressentar a un conjunto de árboles con una estructura dada. En el ejemplo anterior se representan todas las expresiones Binarias cuyo operador de mas alto nivel es *, cuyo operando izquierdo es la suma de una varible y un numero y cuyo operando derecho es cualqueir expresión.

Llamaremos árboles constantes y árboles con variables a los dos tipos de árboles.

8.3 Atributos de los árboles

Hasta ahora, nos hemos preocupado simplemente de los aspectos estructurales de los árboles. Sin embargo, en algunas ocasiones puede darse el caso de que simplemente con los aspectos estructurales de un constructor, no se describan totalmente las características de un especímen de ese constructor. Supongamos la siguiente definición del constructor *Asig*:

```
Asig :: v: Var; e: Exp
```

Si asumimos que *Var* es un constructor terminal, tenemos que con la estructura anterior, no hay suficiente información como para diferenciar los siguientes especímenes del constructor *Asig*:

```
A := 1 + 2;          B := 1 + 2;
```

Esto se debe a que sólo somos capaces de determinar que el destino de una asigandción es una variable, pero no somos capaces de decir qué variable está involucrada en la operación de asignación. Este mismo problema lo encontraríamos a la hora de determinar que números forman la expresión de la parte derecha de la asignación.

El el caso del problema concreto de la variable, el atributo asociado podría ser una cadena de caracteres para representar el nombre o un entrada a la tabla de símbolos para tener incluso referencia directa a sus atributos.

Desde el punto de vista de manipulación del TAD *Arbol*, por cada constructor que tenga atributos (y para cada atributo, si es que dicho constructor tiene más de uno) necesitaremos una operación de recuperación y otra de actualización.

La operación de recuperación, devolverá el valor de un atributo, conociendo, el nombre del atributo y el árbol del que queremos extraer la información.

La operación de actualización, modificará el valor de un atributo, conociendo el nombre del atributo, el nuevo valor y el árbol que queremos actualizar.

Las identificadores de las operaciones de acceso se constuirán con el prefijo del constructor correspondiente, y con el sufijo del atributo al que queremos acceder, tomando como argumento el padre. Algunas funciones que se derivan de la anterior especificación son:

```
Ent Var_e(T_Var a);
int Exp_v(T_Exp a);
```

Estas operaciones deben estar implementadas de tal manera que puedan ser usadas la parte izquierda de una asignación para actualizar el correspondiente atributo. Asi es posible usar

```
Exp_v(a)=3;
```

para actualizar el atributo v del nodo a que tipo T_Exp.

La especificación anterior se podría escribir tambien en la forma:

```
%{
#include "tipos.h"
%}
%union{
    int v;
    Opb ob;
```

```
    Opu ou;
    Entrada e;}
%type <v> Exp
%type <ob> Binaria
%type <ou> Unaria
%type <e> Var
%%
Bloque :: Asig *
Asig :: v: Var; e: Exp
Exp:: Var | Num | Unaria | Binaria
Binaria :: e1, e2: Exp;
Unaria :: e:Exp;
```

Ahora se han pasado a forma de atributos lo que antes se hacía mediante árboles. La especificación define menos constructores que antes y es mas compacta. En este caso los tipos Opb, Opu y sus operaciones deben estar definidos en tipos.h. La forma de acceder al operador de una expresión Binaria se hará como al resto de los atributos.

8.4 Otras operaciones de manipulación de árboles

Junto con los constructores del árbol y las funciones para manipular atributos debemos tener disponibles otro conjunto de funciones que busquen un determinado hijo dado el padre.

8.4.1 Manipulación de agregados

Ademas del constructor, ya definido necesitamos operaciones de acceso. Para manipular los árboles de un constructor agregado, necesitaremos una operación de *construcción* y tantas operaciones de acceso como hijos tenga dicho constructor. Tomemos como ejemplo la regla:

```
Asig :: v: Var; e: Exp
```

Los identificadores de las operaciones de acceso se construirán con un prefijo que es el nombre del constructor correspondiente y un sufijo que es el nombre del hijo al que queremos acceder. Las funciones que se derivan de la regla anterior son:

```
T_Asig Asig(T_Var v; T_Exp e);
T_Var Asig_v(T_Asig a);
T_Exp  Asig_e(T_Asig a);
```

Entre estas funciones hay una serie de relaciones del tipo

```
Asig_v(Asig(v,e))==v
Asig_e(Asig(v,e))==e
```

8.4.2 Manipulación de secuencias

Para manipular los árboles de un constructor secuencia, necesitaremos unas operaciones de *construcción* (ya definidas) y operaciones de acceso que nos permitan recorrer todos los elementos de la secuencia. Tomemos como ejemplo la regla *Bloque*:

```
Bloque :: Asig *
```

Las funciones de acceso serán un conjunto de funciones genéricas para manipular secuencias de objetos. Tendremos dos formas de acceder a todos los elementos de una secuencia, de manera directa o de manera secuencial. Las funciones disponibles son:

```
Nat Longitud(Seq(T_g) a);
T_g Elemento(Seq(T_g) a; Nat n);
T_g Primero(Seq(T_g) a);
T_g Ultimo(Seq(T_g) a);
void En_Primero(Seq(T_g) a);
void Siguiente(Seq(T_g) a);
Bool Es_Ultimo(Seq(T_g) a);
void En_Ultimo(Seq(T_g) a);
void Anterior(Seq(T_g) a);
Bool Es_Primero(Seq(T_g) a);
T_g Elem_act(Seq(T_g) a);
void Insert_dl(Seq(T_g) s, T_g e);
void Insert_dt(Seq(T_g) s, T_g e);
Seq(T_g) Concat(Seq(T_g) s1,s2);
```

En lo anterior *T_g* representa un tipo genérico que sea un subtipo del tipo *Arbol*. El significado de las anteriores operaciones es:

- *Nat Longitud(Seq(T_g) s);* da longitud de una secuencia.
- *T_g Elemento(Seq(T_g) s; Nat n);* da el elemento n-esimo de la secuencia.
- *T_g Primero(Seq(T_g) s);* da el primer elemento de la secuencia.
- *T_g Ultimo(Seq(T_g) s);* da el ultimo elemento de la secuencia.
- *void En_Primero(Seq(T_g) s);* coloca el elemento actual de s en el primero.
- *void Siguiente(Seq(T_g) s);* hace elemento actual el siguiente.
- *Bool Es_Ultimo(Seq(T_g) s);* nos dice si es el último.
- *void En_Ultimo(Seq(T_g) s);* coloca el elemento actual en el último.
- *void Anterior(Seq(T_g) s);* coloca el elemento actual en el anterior.
- *Bool Es_Primero(Seq(T_g) s);* nos indica si es el primero.
- *T_g Elem_act(Seq(T_g) s);* nos da el elemento actual.
- *void Insert_dl(Seq(T_g) s; T_g e);* inserta e delante del elemento actual.

- *void Insert_dt(Seq(T_g) s; T_g e);* inserta e detras del elemento actual.
- *Seq(T_g) Concat(Seq(T_g) s1,s2);* concatena las dos secuencias.

La forma directa de acceder a los elementos de una secuencia será calculando su longitud y accediendo de forma selectiva al elemento i-ésimo, mediante la función *Elemento*.

La forma secuencial consiste en acceder al primer elemento, disponer de una operación que calcule el siguiente de un elemento y de un predicado que determine si un determinado elemento es el último de una secuencia. Igualmente las secuencias podrian ser recorridas hacia atras. Las secuencias deben de ser implementadas de forma que permitan las operaciones anteriores.

8.4.3 Manipulación de elecciones

Para consultar dicha clasificación dispondremos de la función *Es*. Esta función toma dos árboles uno completamente construido y otro con variables. La función responde si ambos árboles tienen los constructores de mas alto nivel iguales. Asi en el ejemplo anterior tendriamos:

```
Es(Asig(a,b),Asig(c,d))==T;
Es(Asig(a,b),Binaria(c,d,e))==F;
etc..
```

En el ejemplo la primera llamada resulta verdadero y la segunda falso.

8.4.4 Consideraciones Generales sobre las diferentes operaciones

- Para cualquier tipo de constructor se pueden definir uno o varios atributos, para los que se tendrán operaciones de recuperación y actualización.
- Los constructores terminales tienen operaciones de construcción pero no de acceso.
- Los constructores agregado y secuencia tienen operaciones de construcción y de acceso, sirviendo esta últimas para *navegar* a través de un árbol.
- Los constructores elección tienen operaciones de consulta, que sirven para determinar la estructura de un árbol (de entre varias alternativas). No tienen ni operaciones de construcción ni de acceso.

Por último se dispone de un conjunto de funciones genéricas de manipulación de listas y árboles, algunas de las más importantes son:

```
void Borra(Arbol a);
Arbol Copia(Arbol a);
void Modifica(Arbol antiguo, Arbol nuevo);
Bool Igual(Arbol a1, Arbol a2);
void EscribeA(FILE *f,Arbol a);
void LeeA(FILE *f,Arbol a);
```

La función *Borra* toma un árbol como argumento y lo borra recursivamente. La función *Copia* devuelve una copia del árbol que toma como argumento, copia también los atributos. La función *Modifica* toma los árboles *antiguo* y *nuevo* y sustituye el antiguo por nuevo. La función *Modifica*, usada en un contexto determinado, sustituye el árbol antiguo por el nuevo y hace que el padre y hermanos del nuevo sean los del del antiguo. La función *Igual* comprueba si dos árboles son estructuralmente iguales, no comprueba la igualdad de atributos.

8.5 Funciones sobre árboles

En este punto ya somos capaces de especificar qué tipo de árbol queremos manipular, con la ventaja añadida de que dicha especificación se puede utilizar para generar de forma automática una implementación para dicho TAD. El siguiente paso es, evidentemente, utilizar dicho TAD *Arbol*. Pero la pregunta a responder es ¿para qué se pueden utilizar estos árboles en la construcción de compiladores?.

En esta sección nos encargaremos de responder a esa pregunta a medias, contestando que sobre esos árboles podemos implementar evaluadores de atributos y transformadores de árboles. La utilidad de estos dos procesos en el ámbito de los compiladores se desvelará en posteriores capítulos. Hay dos razones por las que hemos preferido dar esta *media respuesta*, la primera es intentar no mezclar la problemática de la manipulación de los árboles con los problemas particulares que nos plantearán las siguientes etapas del procesamiento del lenguaje. La segunda y más importante es completar de forma independiente el tratamiento de estos árboles, por que los consideramos con suficiente entidad como para ser estudiados por separado, y aplicables en en el desarrollo de otras aplicaciones que no tengan nada que ver con los compiladores.

8.5.1 Concordancia de patrones sobre árboles

Vamos a introducir una nueva estructura de control, basada en la concordancia de patrones sobre árboles, que nos permita obtener especificaciones de funciones sobre árboles más compactas y más claras. Esta nueva estructura se llamará *segun* y nos dará la posibilidad de elegir entre distintas alternativas, dependiendo de la estructura o patrón del árbol que recorremos. Los patrones se especificarán en base a los constructores definidos en la correspondiente especificación del árbol, pudiéndose incluso definirse de manera anidada. Este anidamiento permitirá además de comprobar la estructura de un árbol, comprobar la estructura de sus descendientes. La estructura de control *segun* tendrá el siguiente formato:

```
segun(<lista de argumentos>)
 {
  <lista de reglas>
 }
```

En la *lista de argumentos*, se especificarán los nombres de variables sobre los que se va a efectuar la concordancia de patrones, en pricipio estos argumentos sólo serán del tipo *Arbol*, aunque más adelante veremos como también se permitirá la aplicación de la concordancia

de patrones sobre otros tipos de datos, pudiéndose de esta forma generalizar la estructura *segun* para estos tipos de datos.

En la *lista de reglas*, se indicarán las distintas posibilidades de actuación dependiendo de los valores concretos de las variables de la *lista de argumentos*. Cada *regla* estará compuesta de una guarda y de un efecto, el formato de una regla será:

```
<lista de terminos> '[' '=>' <condicion>']' ':' <accion> [<modificador>]
```

La parte anterior al carácter ':' representará la guarda (o condición de aplicación) de la regla y se compondrá de una *lista de términos* (uno por cada argumento del *segun*) y de una *condición*. La lista de términos se evaluará como *cierto* si cada uno de valores de los argumentos concuerda con el patrón especificado por el término correspondiente. Por otro lado mediante la *condición* (que se pondrá de manera opcional) se podrán especificar otras restricciones que resulten más fácilmente expresables mediante una expresión lógica. Dicha condición se escribirá en lenguaje C.

La parte posterior al carácter ':' representará el efecto de la regla y se compondrá de una *acción* descrita por un fragmento de programa en lenguaje C (entre los caracteres '{' y '}') y de un modificador (esto último de manera opcional) que nos permitirá introducir aspectos iterativos en la estructura *segun*. Estos modificadores pueden ser NEXT, REJECT, LOOP.

Veamos ahora que notación vamos a utilizar para describir los distintos patrones sobre los cuales comprobaremos la concordancia ante valores concretos. En primer lugar veremos el tipo de patrones que se podrán describir sobre árboles, estos patrones representarán o bien al árbol *Vacio* o a árboles cuyo constructor sea alguno de los definidos en la gramática abstracta.

Para el caso de árboles no vacíos, el patrón se forma con el nombre del constructor, opcionalmente seguido por una lista de términos que servirán para analizar además los valores de los atributos del árbol (encerrados entre los símbolos '[' y ']') y por último una serie de términos para representar la estructura de los hijos del árbol.

En el caso de los constructores terminales, al no tener ningún hijo, el patrón se formará simplemente con el nombre del constructor y opcionalmente los términos para analizar los atributos:

```
<constructor terminal> '[' <atributos> ']'
```

Para los constructores agregados tendremos tantos términos como hijos tenga dicho constructor, de esta forma con los términos asociados a un constructor agregado, analizaremos la estructura de los hijos directos de un árbol.

```
<constructor agregado> '[' <atributos> ']' '('<lista de terminos>')'
```

Para los constructores secuencia podremos especificar un caso base, a través de una lista de cero términos para representar sus hijos. También podremos especificar un caso general, mediante un término para la cabeza de la lista y otro para el resto.

```
<constructor secuencia> '['<atributos> ']' ()
<constructor secuencia> '['<atributos> ']' '('<termino>,<termino>')'
```

En definitiva un patron es un árbol con variables, donde entre el nombre del constructor principal y los hijos hemos colocados los patrones que describen los atributos entre corchetes. Mas generalmente en los patrones se permite en cada sitio donde puede aparecer una variable de tipo Arbol las tres posibilidades siguientes:

```
Patron
ID
ID = Patron
_
```

Donde ID es el identificador de una variable de tipo Arbol.

Si es de la forma ID, donde ID será el nombre de un identificador no declarado hasta ahora [1], se tendrá el efecto de una declaración de una variable de ámbito local de nombre ID (del mismo tipo que el valor correspondiente). Dicha variable sólo será conocida en la condición y en la acción de la regla correspondiente.

Si un término es de la forma $ID = Patron$, al igual que en el caso anterior se declara de la variable ID con ambito en la regla.

Si un término es de la forma _, se considerará que cualquier valor concuerda con el término, sin guardar el valor en ninguna variable.

El último formato que se permite para formar un patron sirve para especificar patrones sobre tipos que no sean el tipo *Arbol*, estos patrones se utilizan generalmente para analizar la estructura de los atributos de un árbol, aunque su uso no se renstringe a eso, y pueden ser utilizados incluso para variables de la lista de argumentos de una estructura *segun*. En este caso se permite un literal del tipo de datos dado.

```
<literal de un TAD>
```

Con lo visto anteriormente ya podemos describir con detalle la semántica de la estructura *segun*. La estructura descrita es de la forma:

```
segun(a1,...,an)
 {
  t11,...,t1n => c1 : r1 [m1]
  ......
  tj1,...,tjn => cn : rn [mn]
 }
```

a1, ...,an representarán variables de tipo Arbol u otro tipo predefinido. Se supone que a1, ...,an son árboles constantes o valores de otro tipo de dato. En primer lugar se intenta Unificar a1, ...,an con tk1,...., tkn para valores sucesivos de k. Si para un valor dado

[1] Quedan reservados todos los nombres de constructores definidos en la correspondiente especificación del árbol, además del nombre $Vacio$

k unifican entonces se evalua ck y si es verdadero se ejecuta rk. En el proceso de unificación se instancian las variables locales de tk1,...,tkn. El valor de estas variables una vez instanciadas es usado para evaluar ck y rk.

El proceso de unificación puede definirse de la siguiente manera:

- Dos patrones unifican si tienen el mismo constructor principal y unifican cada uno de sus hijos. A este nivel los atributos se tratan como unos hijos mas.
- Un patron constante siempre unifica con una variable y esta queda instanciada al árbol correspondiente.
- Un patron p1 constante unifica con ID =p2 si p1 unifica con p2 e ID queda instanciado a p1.
- Un patron p siempre unifica con _.

Si no se utiliza ningún modificador, se aplicará la primera regla que unifique y la condición sea cierta y terminará la ejecución de la estructura *segun*. Si ninguna regla se puede aplicar se acabará la estructura *segun*. Si la regla que aplicamos tiene modificador entonces:

- El modificador *LOOP* hace que se vuelva de nuevo al comienzo de la estructura *segun*.
- El modificador *REJECT* hace que se vuelva de nuevo al comienzo de la estructura *segun*, descartando la aplicación de la regla actual.
- El modificador *NEXT* hace que se inspeccionen las reglas situadas más abajo de la regla actual.

En cada caso la unificación se hará con los valores, posiblemente cambiados de a1,....,an.

8.5.2 Descripción metódica de recorridos

Definiremos una función para recorrer cada constructor que aparezca en la especificación del árbol. El nombre de cada función se formará añadiendo un prefijo identificativo del recorrido [2] al nombre del constructor. Cada función tomará de su primer parámetro el árbol a recorrer, utilizándose el resto de los parámetros para otros cometidos.

```
void Chequea_Exp(Arbol a, Tipo *t);
```

En este caso la función *Chequea_Exp*, efectuará el chequeo de tipos de una expresión dada en el árbol a, devolviendo el tipo de dicha expresión en el atributo sintetizado t. Si sólo tenemos un valor que devolver, una forma alternativa de devolverlo sería como resultado de la función.

[2] Así evitaremos posibles conflictos de nombre si describimos distintos recorridos para chequear la semántica estática, generar código, simplificar expresiones, etc.

Dependiendo del tipo de constructor, las funciones seguirán un patrón distinto, así para los constructores tipo *agregado* se recorrerán secuencialmente cada uno de los hijos del árbol, para los constructores tipo *secuencia* se recorrerán iterativamente todos los elementos de una lista y para los constructores tipo *elección* se determinará de que forma ha de recorrerse el árbol.

Por ejemplo para recorrer una sentencia de asignación cuyo constructor *agregado* se describirá por la siguiente regla:

```
Asig:: v: Var ; e: Exp
```

la función correspondiente podría tener la siguiente estructura:

```
void recorre_Asig(Arbol a)
 {
  segun(a){
  Asig(v,e): {recorre_Var(v);recorre_Exp(e);}
  }
 }
```

A la hora de describir el recorrido, siempre tendremos la posibilidad de recorrer los hijos en cualquier orden (alterando el orden de las llamadas) y tambien podremos recorrer varias veces un mismo hijo (incluso no recorrer un hijo, si no es relevante para el proceso que estemos describiendo).

Para recorrer un bloque de asignaciones, descrito por la siguiente regla tipo *Bloque* de sintaxis abstracta:

```
Bloque :: Asig *
```

podríamos tener una función con la siguiente estructura:

```
void recorre_Bloque(Arbol a)
 {
  int i=0,l;
  l = Longitud(a);
  while(i <= l)
   {
    recorre_Asig(Elemento(a,i));
    i++;
   }
 }
```

otra posibilidad es

```
void recorre_Bloque(Arbol a)
 {
  If(longitud(a)>0)
  {
```

```
    En_Primero(a);
    while(!Es_Ultimo(a))
     {
      e=Elem_act(a);
      recorre_Asig(e);
      Siguiente(a);
     }
    recorre_Asig(e);
   }
 }
```

y una tercera

```
void recorre_Bloque(Arbol a)
 {
  segun(a){
   Bloque_v():      {}
   Bloque_dl(e,s):   {recorre_Asig(e);
                      recorre_Bloque(s);
                     }
  }
 }
```

donde *Longitud* y *Elemento* son funciones de manipulación de secuencias generadas por la herramienta HESA, que nos dan la longitud y el elemento i-ésimo de un bloque de sentencias. Al igual que para los agregados, podríamos plantearnos otros órdenes y número de visitas distintos a los descritos en esta función.

Las reglas de tipo *eleccion*, tienen asociado un esquema como el siguiente. Sea una expresión, como la descrita en los tendríamos pues el esquema:

```
void recorre_Exp(Arbol a)
 {
  segun(a)
   {
    Binaria(e1,e2,ob): {recorre_Exp(e1);
                        recorre_Exp(e2);
                        recorre_Opb(ob);
                        }
    Unaria(e,ou):       {recorre_Exp(e);
                         recorre_Opu(ou);
                        }
    Var[e]():           {..}
    Num[v]():           {..}
   }
 }
```

8.5.3 Ejemplo

Veamos con un ejemplo como definir un recorrido sobre un árbol con esta notación. Sea la siguiente gramática abstracta, que nos sirve para describir la estructura de los árboles de expresiones aritméticas:

```
%{
#include "tabsim.h"
%}
%union{
   Entrada entrada;
   int valor;
   }
%type<entrada> Variable
%type<valor> Constante
%%
Expresion :: Binaria | Variable | Constante
Binaria :: termi,termd : Expresion; op: OpBin
OpBin :: Suma | Resta | Mult | Div
```

Una función *evalua* que recorre el árbol asociado a una expresión y calcula su valor sería de la forma:

```
int evalua(Arbol a)
 {
  segun(a)
  {
   Constante[v]:
      {return(v);}
   Variable[e]:
      {return(valor(e));}
   Binaria(ti,td,Suma):
      {return(evalua(ti) + evalua(td));}
   Binaria(ti,td,Resta):
      {return(evalua(ti) - evalua(td));}
   Binaria(ti,td,Mult):
      {return(evalua(ti) * evalua(td));}
   Binaria(ti,td,Div):
      {return(evalua(ti) / evalua(td));}
  }
 }
```

Como se ve, la posibilidad de anidar términos (como ocurre en el caso del término $Binaria(ti, td, Suma)$ del ejemplo), hace que mediante una descripción muy compacta se puedan expresar muchas condiciones, además de recuperar en variables de ámbito local ciertos valores (que en el término anterior serían ti y td) sin llamar explícitamente a las correspondientes funciones de acceso generadas por la herramienta HESA.

Esta notación es pues muy adecuada para la especificación de recorridos complejos. Para obtener una implemetación (por ejemplo en lenguaje C) a partir de dicha especi-

ficación simplemente habrá que disponer de una herramienta que aplane la estructura *segun*, expresándola en términos de otras estructuras disponibles en el lenguaje de implementación elegido.

8.6 Automatización de la implementación

En esta sección vamos a presentar una herramienta que tomando como entrada una especificación de árbol con atributos en una notación similar indicada arriba genera una implementación (en lenguaje C) del TAD *Arbol* correspondiente. El formato del fichero de entrada es el indicado arriba.

8.6.1 Formato de salida

A partir de la especificación de entrada se generarán dos ficheros de salida, ambos con el mismo nombre que el fichero de entrada y con la extensión ".h" y ".c" donde se concretarán los detalles de interface e implementación respectivamente del TAD *Arbol*.

La herramienta está diseñada de tal forma que no hace falta describir todas las funciones para cada constructor, sino que se definen un conjunto de macros para cada uno que configuran adecuadamente las llamadas de unas funciones genéricas que manipulan directamente el árbol. De esta forma la interface se ofrece a través de dicho conjunto de macros que ocultan los detalles de implementación del árbol.

La estructura elegida para soportar el tipo de datos árbol es un registro con un puntero a una lista de hijos y un puntero a una lista de hermanos, de esta forma podremos representar árboles multicamino. También disponemos de una referencia al padre que nos será de utilidad en las operaciones de modificación[3]:

```
typedef struct s_arbol{
   int constructor;
   union s_atributo{
      ...
   } atributo;
   int nh,np;
   struct s_arbol *hijos, *hermano, *padres;
  } *Arbol;
```

En la estructura anterior, el campo *constructor* sirve para determinar el tipo de árbol y el campo *atributo* nos sirve para almacenar los atributos asociados a un árbol[4].

Los constructores se codifican mediante números enteros, sin embargo para su manipulación simbólica disponemos de una serie de macros que asocian a cada nombre de constructor (en mayúsculas, para no tener conflictos con los nombres de las funciones constructoras) un número entero:

[3] El hecho de tener una sóla referencia al padre nos limita el número de padres a uno, con lo que tendremos que ser muy cuidadosos al intentar representar cualquier tipo de DAG (Directed Acyclic Graph) sobre esta implementación

[4] La definición de *union s_atributo* es más compleja que en la primera versión de HESA, para poder permitir múltiples atributos y herencia en los constructores elección.

```
#define PROGRAMA   1
#define SENTENCIA  2
  ...
#define BINARIA    8
```

Para cossultar el constructor de un nodo disponemos de una función que lo devuelve

```
#define Constructor(a)  a.constructor
```

Para manipular la información asociada a los constructores tipo elección, dispondremos de una tabla de filiación que nos permitirá determinar en cada momento a que constructores pertenece un árbol dado, por ejemplo para el siguiente conjunto de reglas:

```
OpBin :: OpArit | OpRel | OpLog
OpArit :: OpSuma | OpResta
OpRel :: OpMayor | OpMenor
OpLog :: OpO | OpY
```

se define un árbol de filiación donde se establecen las relaciones de inclusión entre los distintos tipos asociados, esta información se reflejará en la tabla de filiación de la siguiente forma:

```
(HIJO     ,  PADRE)
-------------------
(OPBIN    ,    -  )
(OPARIT   ,  OPBIN)
(OPREL    ,  OPBIN)
(OPLOG    ,  OPBIN)
(OPSUMA   , OPARIT)
(OPRESTA  , OPARIT)
(OPMAYOR  ,  OPREL)
(OPMENOR  ,  OPREL)
(OPO      ,  OPLOG)
(OPY      ,  OPLOG)
```

Igualmente hay que mantener la información asociada a los tipos que son secuencias de otros. Esto puede hacerse en una tabla. Igualmente la tabla de filiación habría que estructurarla de tal manera que para cada tipo poder saber los constructores de sus subtipos.

Para la manipulación de árboles tipo agregado, dispondremos de unas funciones genéricas para construir el árbol a partir de los hijos, y acceder a los hijos de un determinado árbol.

```
Arbol _agregado(int costructor, int num_hijos, ...);
Arbol _hijo_iesimo(Arbol a, int orden);
Arbol _padre_iesimo(Arbol a, int orden);
int   _num_hijos(Arbol a);
int   _num_padres(Arbol a);
```

Para la siguiente regla:

```
Asignacion :: destino: Variable; fuente: Expresion
```

obtendríamos el siguiente conjunto de macros:

```
#define Asignacion(a1,a2) _agregado(ASIGNACION,2,a1,a2)
#define Asignacion_destino(a) _hijo_iesimo(a,1)
#define Asignacion_fuente(a) _hijo_iesimo(a,2)
```

Para la manipulación de árboles tipo secuencia, dispondremos de tres funciones constructoras. La que construye la secuencia vacía, la que añade un elemento por delante y la que añade uno por detras. Otras funciones genéricas de interés son:

```
Nat Longitud(Arbol a);
Arbol Elemento(Arbol a, Nat n);
Arbol Primero(Arbol a);
Arbol Ultimo(Arbol a);
void En_Primero(Arbol a);
void Siguiente(Arbol a);
Bool Es_Ultimo(Arbol a);
void En_Ultimo(Arbol a);
void Anterior(Arbol a);
Bool Es_Primero(Arbol a);
Arbol Elem_act(Arbol a);
void Insert_dl(Arbol s, Arbol e);
void Insert_dt(Arbol s, Arbol e);
Arbol Elem_a_Seq(Arbol e, Arbol s);
Arbol Concat(Arbol s1, Arbol s2);
Bool Es(Arbol a1,a2);
void Borra(Arbol a);
Arbol Copia(Arbol a);
void Modifica(Arbol antiguo, Arbol nuevo);
Bool Igual(Arbol a1, Arbol a2);
void EscribeA(FILE *f,Arbol a);
void LeeA(FILE *f,Arbol a);
```

Para los constructores terminales disponemos de una función que construye el árbol asociado, por ejemplo para el constructor terminal *Variable* se define la macro:

```
#define Variable() _agregado(VARIABLE,0)
```

Para la gestión de atributos, disponemos de una macro que accede a un atributo concreto de un árbol dado, así para acceder al atributo *valor* de una *Constante* tendríamos la siguiente macro:

```
#define Constante_valor(a) a->atributo.Constante.valor
```

En este caso vemos como la macro al expandirse dará lugar a una expresión con L-VALUE, lo que nos posibilita utilizar esa expresión tanto para recuperación como para

la actualización [5]de atributos. En la primera de las siguientes sentencias recuperamos el valor del atributo del árbol a para almacenarlo en la variable v, mientras que en la segunda actualizamos el atributo del árbol a con el valor 33:

```
v = Constante_valor(a);
Constante_valor(a) = 33;
```

8.6.2 Implementación de la estructura segun

```
int evalua(Arbol a)
 {
  segun(a)
  {
   Constante[v]:
      {return(v);}
   Variable[e]:
      {return(valor(e));}
   Binaria(ti,td,Suma):
      {return(evalua(ti) + evalua(td));}
   Binaria(ti,td,Resta):
      {return(evalua(ti) - evalua(td));}
   Binaria(ti,td,Mult):
      {return(evalua(ti) * evalua(td));}
   Binaria(ti,td,Div):
      {return(evalua(ti) / evalua(td));}
  }
 }
```

puede hacerse de esta manera:

```
int evalua(Arbol a)
  {
   int c,c1;
   Arbol ti,td,op;
   while(True){
     c=Constructor(a);
     switch(c){
     {
      CONSTANTE:{
          v=Constante_valor(a);
          return v;
      }
      VARIABLE:{
          e=Variable_entrada(a);
          return (valor(e));
```

[5]En el caso de la actualización queda una expresión que a primera vista puede parecer un poco extraña. Es importante reseñar que sólo tiene sentido si se implementa el acceso a los atributos a través de expansiones de macros

```
      }
      BINARIA:{
          ti=Binaria_termi(a);
          td=Binaria_termd(a);
          op=Binaria_op(a);
          c1=Constructor(op);
          switch(c1){
            SUMA: return(evalua(ti) + evalua(td));
            RESTA: return(evalua(ti) - evalua(td));
            MULT: return(evalua(ti) * evalua(td));
            DIV: return(evalua(ti) / evalua(td));
          }
      }
      break;
    }
  }
```

La implementación de los modificadores se hara de siguiente manera

- CONTINUE se implementará poniendo la sentencia *continue;* al final de la correspondiente regla.
- NEXT se implementará dejando que se ejecute secuencialmente el codigo de la siguiente regla.
- REJECT se inplementará poniendo un a variable a False para que la siguiente esta regla no pueda ejecutarse en la siguiente vez y posteriormente *continue*.

Capítulo 9

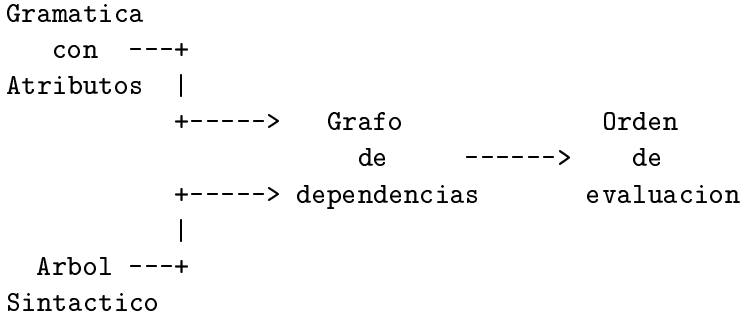
Implementación de Gramáticas con atributos

9.1 Tipos de gramáticas con Atributos

Una gramática con atributos tal y como hemos visto nos sirve para determinar que valores tomarán los atributos de cada símbolo en función de los valores de otros símbolos. Se establece pues, una relación funcional entre los valores de los atributos, pero sin embargo, no se determinan explícitamente los pasos a seguir a la hora de calcularlos. Dicho en otras palabras, mediante una gramática con atributos obtenemos una especificación *no ordenada*, en la que se describen las relaciones funcionales entre los valores de los atributos de los símbolos sin describirse explícitamente el orden de evaluación de dichos atributos.

Una forma de extraer un orden de evaluación a partir de una gramática con atributos pasa por la construcción de un grafo de dependencias para dicha gramática. El grafo se construirá para una palabra dada, y sobre el árbol sintáctico de dicha palabra. Por su parte, el árbol debe tener capacidad para guardar los distintos atributos de los símbolos.

El grafo de dependencias, será un grafo dirigido y se obtendrá a partir de las relaciones funcionales definidas para los atributos de los símbolos. De esta forma, ante una relación funcional del tipo $a = f(a_1, ..., a_n)$, añadiremos al grafo de dependencias arcos de la forma (a_i, a) indicando que el valor del atributo a no podrá ser calculado hasta que se calculen todos los valores de los atributos a_i. A partir de este grafo de dependencias podremos obtener un orden de evaluación para los atributos de los símbolos a través de una ordenación topológica del dicho grafo dirigido.



Otra consecuencia directa de la construcción del grafo de dependencias, es la detección de gramáticas con atributos no evaluables, en el sentido de que serán gramáticas para las que no se podrá encontrar un orden de evaluación correcto. Estas gramáticas serán aquellas en las que podamos encontrar algún ciclo en el grafo de dependencias asociado, de tal forma que para los atributos que se encuentren en dicho ciclo no habrá manera de determinar por cual de ellos hay que empezar a evaluar ni por cual habrá que terminar.

De esta forma establecemos una primera clasificación para las gramáticas con atributos, dividiéndolas en circulares y no circulares, y considerando a las gramáticas con atributos circulares sin sentido y no evaluables. En cualquier caso podremos diseñar un test de circularidad para determinar si una gramática con atributos es circular o no a partir de su grafo de dependencias.

```
   Grafo                Test
     de        ---->     de       --->  {SI,NO}
dependencias          circularidad
```

Para evaluar los atributos de los símbolos que aparecen en un árbol sintáctico estableceremos una serie de recorridos sobre dicho árbol. La forma en la que se llevarán a cabo estos recorridos quedará determinada al tenerse que respetar la ordenación topológica derivada del grafo de dependencias asociado a la gramática con atributos. Dado el grafo de dependencias de una gramática con atributos, es posible que no baste con un sólo recorrido ya que puede que un atributo heredado de un símbolo dependa a su vez de atributos sintetizados del mismo símbolo, con lo que habría que efectuar un primer recorrido para calcular los atributos sintetizados, y un segundo recorrido conociendo ya el valor del atributo heredado.

En principio, la forma de obtener una descripción operacional a partir de una gramática con atributos es a través de una ordenación de recorridos sobre el árbol. Sin embargo desde un punto de vista práctico (y con el fin de asegurar implementaciones eficientes) es interesante disponer de técnicas que permitan incorporar información acerca del orden de evaluación de los atributos.

En este sentido plantearemos dos métodos, en el primero se aprovechará el orden sintáctico para hacer que los atributos se evalúen al mismo tiempo que se efectúa el reconocimiento sintáctico, utilizando gramáticas concretas. En el segundo método se especificarán directamente los recorridos sobre el árbol (que habrá sido construido previamente en el proceso de reconocimiento sintáctico) a través de un conjunto de funciones que se llamarán de manera recursiva, en este caso el conocimeinto de los detalles de sintaxis concreta ya no es importante, por lo que nos podremos quedar con la información estructural que nos proporcionan las gramáticas abstractas.

No hay que perder de vista que estos dos métodos son sólo formas implementar gramáticas con atributos. En ningún caso debemos confundirlos con las gramáticas con atributos, que son más declarativas y por lo tanto más útiles a la hora de especificar.

9.2 Evaluación sobre reconocedores sintácticos

En esta sección estudiaremos como aprovechar el proceso de reconocimento sintáctico para evaluar atributos. Una característica de estos métodos va a ser que no necesitarán el árbol

(en forma de estructura de datos) para evaluar los atributos, ya que la pila asociada al reconocimiento sintáctico contendrá en cada momento una representación de un camino del árbol sintáctico, que será suficiente para la evaluación de los atributos. Las especificaciones que obtendremos aquí serán del tipo de las utilizadas por herramientas como YACC o similares, y servirán para la obtención directa de analizadores sintácticos. Por un abuso de la nomenclatura a este tipo de notaciones se les denomina también gramáticas con atributos. Sin embargo atendiendo a la definición hecha en 10.5, no se pueden considerar estrictamente como tales, ya que implícitamnete incorporan aspectos operacionales dados por el algoritmo de análisis utilizado para la obtención del reconocedor.

A partir del conocimiento de una ordenación implícita, en estas notaciones tiene sentido incluso intercalar acciones semánticas entre los símbolos no terminales de una parte derecha de una producción, cosa que no estaba contemplada en la definición original de las gramáticas con atributos.

Otra consecuencia del aprovechamiento del orden sintáctico vendrá dada por las limitaciones a la hora de calcular atributos, impuestas por los métodos de análisis. Estas limitaciones son debidas al hecho de no disponer de una representación completa del árbol sintáctico. Atendiendo al tipo de atributos que podremos calcular, tendremos la siguiente clasificación:

- gramáticas l-atribuidas
- gramáticas s-atribuidas
- gramáticas lc-atribuidas

En los dos siguientes apartados veremos como evaluar atributos junto a reconocedores ascendentes y descendentes, y las implicaciones que cada uno de estos métodos conllevan con respecto a las notaciones permitidas para especificar los reconocedores sintácticos.

9.2.1 Reconocedores ascendentes

Mediante un reconocedor ascendente se construye el árbol sintáctico desde las hojas hacia la raiz, el algoritmo correspondiente se denomina de *desplazamiento y reducción*. En este tipo de algoritmos se colocan los símbolos en la cima de la pila (desplazamiento) a medida que van siendo analizados y se comprueba si los símbolos situados más arriba en la pila coinciden con alguna parte derecha α de una producción $A : \alpha$, sustituyéndolos (reducción) por el símbolo A de la parte izquierda de la producción.

Por ejemplo, para una gramática que contenga la producción $S : aAb$, tras apilar el símbolo terminal a, analizar el símbolo no terminal A y apilar el símbolo terminal b la pila quedaría en una configuración en la que se podría reducir la secuencia aAb y sustituirla por S:

```
b
A        ---->
a    (reduccion)    S
```

Según esta implementación, el símbolo de la parte izquierda de la regla no existe en la pila hasta que desaparecen sus *hijos*, por lo que no es posible conseguir que un símbolo

herede atributos de un antecesor. Los atributos sintetizados si son soportados en esta implementación, la síntesis de atributos se lleva a cabo en el momento de una reducción, al disponer de los atributos de los símbolos de la parte derecha de la producción para poder calcular el atributo del símbolo de la parte izquierda.

Existe, sin embargo, la posibilidad de soportar un tipo de herencia de atributos en el momento del desplazamiento. Esta herencia sólo podrá ser de los símbolos de una parte derecha α que en ese instante existan en la pila, que son precisamente los hermanos izquierdos (en el ámbito de la producción $A : \alpha$) del símbolo que apilamos.

Las especificaciones gramaticales válidas para obtener a partir de ellas reconocedores ascendentes de este tipo se pueden clasificar en dos:

- **Gramáticas s-atribuidas**, que sólo contendrán atributos sintetizados.
- **Gramáticas lc-atribuidas**, que permitirán atributos sintetizados y atributos heredados de hermanos izquierdos.

La herramienta YACC, aparece como ejemplo más paradigmático de generación de este tipo de reconocedores ascendentes.

9.2.2 Evaluación en YACC de gramáticas s-atribuidas: Construcción de árboles de SA en el reconocimiento sintáctico

Ejemplo:

Un ejemplo de evaluación de gramáticas s-atrbuidas es la construcción de árboles a partir del reconocimiento sintáctico. Esto puede ser expresado como una gramática s-atribuida. El atributo que se se va porpagando desde los hijos hacia los padres el árbol construido. Este procedimiento será muy importante a la hora de construir compiladores. Nos permite construir árboles que representaran el procesameinto intermedio del compilador. En general nos servira para asociar de una forma univoca la sintaxis concreta y la absttracta. Sea un lenguaje cuya sintaxis abstracta puede ser descrita mediante los árboles descritos en:

```
%{
#include "tipos.h"
%}
%union{
    int v;
    Hilera n;}
%type <v> Exp
%type <n> Var
%%
Bloque :: Asig *
Asig :: v: Var; e: Exp
Exp:: Var | Num | Unaria | Binaria
Binaria :: e1, e2: Exp; ob: Opb
Unaria :: e:Exp; ou: Opu
Opb :: Suma | Resta | Mult | Div | Pot
Opu :: Menos
```

Sea el lenguaje con la sintaxis concreta del ejemplo siguiente. En el se especifica tambien el proceso de evaluación de la gramática con atributos. Este proceso va construyendo los árboles. La notación usada por YACC para nobrar los atributos es $$ para los atributos del símbolo en la parte izquierda, $1 para los del primer símbolo de la derecha, etc. Si un símbolo tiene varios atributos entonces uno de ellos en concreto se nombra posniendo $n.atr donde atr es el nombre del atributo.

```
%{
  #include "arboles.h"
%}

% start list

% union {
     Arbol nodo;
     int val;
     char * name;
}

% token <val> NUM
% token <name> VAR
% type <nodo> list expr

% left '+' '-'
% left '*' '/'
% right '^'
% left UMINUS
%%

list  :                  { $$ = Bloque_v(); }
      | list stat        { $$ = Bloque_dt($1,$2); }
      | error
      ;
stat  : VAR '=' expr '\n' {Arbol t;t= Var(); copia(Var_n(t),$1);
                           $$ = Asig(t,$3);}
      | error '\n'       {yyerrok;}
      ;
expr  : '(' expr ')'     { $$ = $2; }
      | expr '+' expr    { $$ = Binaria($1,$3,Suma() ); }
      | expr '-' expr    { $$ = Binaria($1,$3,Resta() ); }
      | expr '*' expr    { $$ = Binaria($1,$3,Mult() ); }
      | expr '/' expr    { $$ = Binaria($1,$3,Div() ); }
      | expr '^' expr    { $$ = Binaria($1,$3,Pot() ); }
      | '-' expr         { $$ = Unaria($2,Menos()); }
                          % prec UMINUS
      | NUMBER           { $$ = Num(); Num_v($$)=$1;}
      | VAR              { $$ = Var(); copia(Var_n($$),$1);}
      | expr error expr  {;}
      | '(' expr error   {;}
      | error            {;}
```

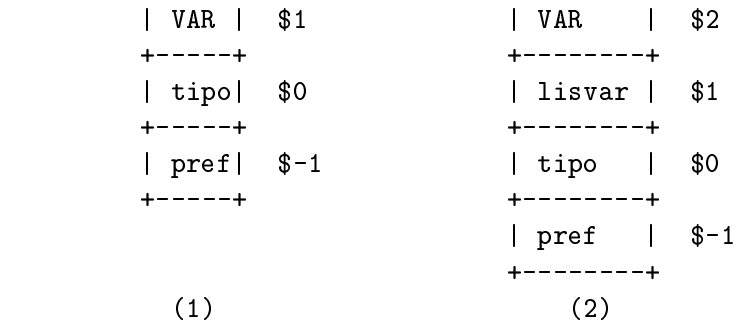
```
      ;
%%

< C\'odigo C >
```

9.2.3 Evaluación en YACC de gramáticas lc-atribuidas

En YACC es posible evaluar gramáticas lc-atribuidas. Hay un mecanismo para acceder a atributos de símbolos que estan en la pila en un momento dado. Asi

```
pref :          {$$=0;}
     |  PUNT    {$$=1;}
     ;
tipo : ENT      {$$=T_Ent;}
     | REAL     {$$=T_Real;}
     ;
decl : pref tipo lisvar
     ;
lisvar : VAR          {Pt3($1,$<t>-1,$<t>0);}
       | lisvar VAR   {Pt3($2,$<t>-1,$<t>0);}
       ;
```

Donde Pt3 es una función que los parámetros recibidos hace algunas actualizaciones. En un momento dado la pila puede estar como en la posición (1) o (2):



La notación $1 indica el primer símbolo de la parte derecha de una regla. $0 el que esta debajo de el, etc. La notación $¡t¿0 se usa para indicar cual de las componentes de la unión es el adecuado.

9.3 Evaluación sobre árboles con atributos

Hay muchos casos en los que los atributos no se podrán calcular al mismo tiempo que se efectúa el análisis sintáctico, o bien porque se necesite más de una pasada para el cálculo de los atributos o bien por que se necesite heredar atributos de hermanos colocados más a la derecha en la especificación gramatical. En estos casos necesitaremos una representación

del árbol sintáctico para poder recorrerlo tantas veces como queramos y en el orden que necesitemos.

Para representar los árboles sintácticos, utilizaremos los árboles con atributos presentados en el capítulo anterior, trasladando los métodos allí propuestos para la evaluación de atributos sobre árboles a la implementación de las gramáticas con atributos.

Presentaremos el paso de una gramática abstracta al método de evaluación a través de un ejemplo, ya que el método en sí ya se estudió anteriormente.

El ejemplo elegido es el de las restricciones de ubicación de las sentencias *break* y *continue* visto en el capitulo de gramáticas con atributos.

9.3.1 Especificación

Categorías sintácticas

s,s1,s2 ∈ Sentencia
exp ∈ Expresion
c,c1 ∈ Casos
prog ∈ Programa

Atributos

< ew,es: Bool > Prog, Sentencia, Casos

Definiciones

prog:	s	$\{s.ew := F$
	;	$s.es := F\}$
s	: s1 ; s2	$\{s1.ew := s.ew$
		$s2.ew := s.ew$
		$s1.es := s.es$
		$s2.es := s.es\}$
	| if (exp) s1 else s2	$\{s1.ew := s.ew$
		$s2.ew := s.ew$
		$s1.es := s.es$
		$s2.es := s.es\}$
	| while (exp) s1	$\{s1.ew := T$
		$s1.es := s.es\}$
	| switch (exp) c	$\{c.es := s.es\}$
	| break	$< (s.ew = T) \vee (s.es = T) >$
	| continue	$< s.ew = T >$
	;	
c	: exp → s	$\{s.es := c.es$
		$s.ew := c.ew\}$
	| c1 [] exp → s	$\{s.es := c.es$
		$s.ew := c.ew\}$

9.3.2 Implementación

Una vez que hemos especificado claramente el problema a resolver, el siguiente paso es obtener una descripción equivalente en una notación *ejecutable*. En este caso efectuaremos la evaluación de los distintos atributos que aparecen en la gramática anterior, apoyándonos en un árbol con atributos.

Lo primero que tendremos que hacer es determinar que árbol necesitamos, y la forma de hacerlo será especificándolo tal y como vimos en el capítulo 8. Como ya sabemos, el conocimiento de la sintaxis abstracta del lenguaje nos servirá de mucha ayuda para escribir esta especificación:

```
  %{
  #include "tipos.h"
  %}
  %union{
   Bool t;
  }
  %type <t> Sentencia Casos Opcion
%%
  Programa :: Sentencia *
  Sentencia :: If | While | Switch | Compuesta | Break | Continue
  If :: cond: Expresion; rama_si, rama_no : Sentencia
  While :: cond: Expresion; cuerpo: Sentencia
  Switch :: disc: Expresion; cuerpo: Casos
  Casos :: Opcion *
  Opcion :: exp: Expresion; cuerpo: Sentencia
  Compuesta :: Sentencia *
```

Estructuraremos el recorrido del árbol que evaluará los atributos *ew* y *es* en tres funciones.

```
void Chequea_Programa(Arbol p)
{
 segun(p)
 {
  Programa() : {}
  Programa(s1,resto):
       {Chequea_Sentencia(s1,FALSO,FALSO);
        Chequea_Programa(resto);}
 }
}
```

La siguiente función realiza la parte fundamental de la evaluación de los atributos, haciendo que los grupos de sentencias que pertenecen a las estructuras *if* y *while* hereden los atributos correspondientes. Se ha separado la gestión de los casos de la estructura *switch*, por pertenecer éstos a una categoría sintáctica diferente. Por último, destacar que las condiciones que aparecen en la gramática con atributos se sustituyen por tratamientos de errores, ya que resulta más interesante obtener información de los errores detectados, que impedir la evaluación de más atributos.

```
void Chequea_Sentencia(Arbol s, Bool ew, Bool es)
{
 segun(s)
 {
  If(_,rama_si,rama_no):
      {Chequea_Sentencia(rama_si,ew,es);
       Chequea_Sentencia(rama_no,ew,es);}
  While(_,cuerpo):
      {Chequea_Sentencia(cuerpo,CIERTO,es);}
  Switch(_,cuerpo):
      {Chequea_Sentencia(cuerpo,ew,es);}
  Compuesta(): {}
  Compuesta(s1,resto):
      {Chequea_Sentencia(s1,ew,es);
       Chequea_Sentencia(resto,ew,es);}
  Casos(): {}
  Casos(Opcion(_,s),resto):
      {Chequea_Sentencia(s,CIERTO,es);
       Chequea_Sentencia(resto,ew,es);}

  Break:
      {if((ew!=CIERTO)||(es!=CIERTO))
        error("break fuera de ambito");}
  Continue:
      {if(es!=CIERTO)error("continue fuera de ambito");}
 }
}
```

Como se puede ver, la obtención de un evaluador[1] a partir una gramática con atributos es bastante directa, y podríamos decir que casi metódica. He aquí unos pasos o directrices que pueden servir de ayuda en dicho proceso:

1. Especificar la estructura del árbol con atributos que utilizaremos en la evaluación. Esta especificación coincidirá prácticamente con la sintaxis abstracta del lenguaje.

2. Utilizar la concordancia de patrones proporcionada por la estructura *segun* para consultar la estructura del árbol. Con esto representamos la parte derecha de las producciones de la gramática abstracta, que queda ligada a la correspondiente parte izquierda al conocerse *dentro* de qué función nos encontramos.

3. En dichas funciones, los atributos heredados de cada símbolo se modelarán con parámetros de entrada y los atributos sintetizados con parámetros de salida.

4. Por último, tendremos que determinar el lugar donde han de colocarse las acciones semánticas dentro de las funciones que efectúan el recorrido. Este paso es quizá el menos metódico, ya que supone la elección de un orden que no está especificado en la gramática abstracta.

[1] Con el término *evaluador* nos referimos a un recorrido de un árbol con atributos, que efectúa el cálculo de nuevos atributos

Parte III

Semántica Estática

Capítulo 10

Sintaxis Abstracta y gramáticas con atributos

10.1 Introducción

El principal objetivo de este capítulo es presentar la sintaxis abstracta como formalismo adecuado para representar los aspectos relevantes de un lenguaje. En esta representación, nos olvidaremos de la problemática asociada al reconocimiento sintáctico, para quedarnos sólo con los detalles que puedan ser de utilidad en las restantes etapas del proceso de compilación. Dicho en otras palabras, nos quedaremos sólo con los elementos del lenguaje que tengan *significado*, desechando aquellos otros elementos que sólo sirven para expresar de una forma más clara y compacta dicho *significado* (ejemplos de este tipo de elementos son los separadores y delimitadores).

Hasta ahora, en la tarea de especificación del lenguaje a compilar, sólo nos hemos preocupado de los detalles de representación o apariencia externa. En este empeño, hemos utilizado las expresiones regulares para la especificación de los aspectos léxicos y las gramáticas independientes del contexto para la especificación de los aspectos sintácticos. La utilidad de este tipo de representaciones en el proceso de reconocimiento del lenguaje está fuera de toda duda, ya que incluso disponemos de métodos para obtener de forma automática implementaciones de los reconocedores léxicos y sintácticos.

Sin embargo a la hora de tratar el significado de los programas, hay detalles que dejan de tener importancia, por lo que las representaciones que reflejan dichos detalles dejan de ser interesantes para convertirse incluso en inadecuadas. Entre estos detalles destacaremos pricipalmente dos, lo que denominaremos *azúcar sintáctico* y los mecanismos de *resolución de ambigüedad*. Todos estos detalles deben quedar reflejados en lo que hasta ahora hemos denominado simplemente *sintaxis*, y que a partir de ahora denominaremos *sintaxis concreta* (para diferenciarla de la sintaxis abstracta).

10.1.1 Detalles de sintaxis concreta

Consideraremos *detalles de sintaxis concreta* a todas aquellas características del lenguaje que están diseñadas para hacerlo más claro y legible. Entre estas características se encuentran las palabras clave, los delimitadores, los separadores, los operadores, etc. Estos

elementos son irrelevantes desde el punto de vista estructural, e incluso se puede decir que, una vez cumplida su función, pueden llegar a estorbar en el proceso de análisis semántico, ya que hacen perder la visión global de la estructura del lenguaje.

La siguiente producción describe la forma que debe tener una sentencia condicional en algún lenguaje de programación:

```
condicional : SI expresion_logica ENTONCES
              COMIENZO sentencia FIN
              SINO sentencia FIN
```

Sin embargo, desde un punto de vista semántico lo único importante que tenemos que saber es que una sentencia condicional está compuesta de dos grupos de sentencias y de una expresión lógica que discrimina cual de ellos ha de ejecutarse.

Algunos de estos elementos cumplen además una segunda función en la fase de análisis sintáctico, haciendo que no tengamos que construir reconocedores excesivamente potentes (se puede asegurar la condición LL(1) colocando palabras clave en puntos determinados del programa, por ejemplo TYPE, VAR y CONST en PASCAL), o proporcionando puntos de seguridad que ayuden en la recuperación de errores sintácticos (el más típico es el carácter ';').

10.1.2 Ambigüedad y árboles

La ambigüedad se presenta cuando ante una gramática dada, para una palabra (perteneciente al lenguaje generado por dicha gramática) se pueden encontrar dos o más árboles de derivación. El problema es pués decidir con cual de ellos quedarnos, y la solución es precisamente proporcionar mecanismos que permitan tomar esa decisión de manera automática.

Al estudiar la etapa de análisis sintáctico, hemos visto como resolver este problema durante el proceso de reconocimiento, planteando incluso notaciones que permiten especificar reglas de desambiguación. Por esta razón no hace falta que desde el punto de vista semántico tengamos que replantearnos un problema que ya ha sido resuelto.

Por lo tanto, la resolución de ambigüedades no es algo que deba expresarse mediante sintaxis abstracta, ya que es un problema que aparece al relacionar cadenas con árboles (reconocimiento sintáctico), y que carece de sentido en etapas posteriores, donde se toma un árbol concreto como punto de partida. Es más, desde el punto de vista de la sintaxis abstracta ni siquiera nos preocupan los árboles de derivación, sino unos árboles mucho más *descargados*, en los que sólo aparecen elementos que son semánticamente relevantes.

10.2 Gramáticas abstractas

Denominaremos gramática abstracta a una especificación de sintaxis abstracta para un lenguaje, es decir una especificación de su estructura profunda. Podemos plantear distintas notaciones para representar gramáticas abstractas, todas ellas servirán para representar las características estructurales de un lenguaje, sin embargo algunas resultarán adecuadas

para aprovechar dichas características en la construcción de traductores y otras serán más adecuadas como base para describir formalmente los aspectos semánticos del lenguaje. Aquí presentaremos una primera notación

En esta notación se describen una serie de *categorías sintácticas* en la que se agrupan objetos que tienen una estuctura común. La especificación se compondrá de dos partes, la sección de *categorías sintácticas* y la sección de *definiciones*. En la primera sección, se enumeran las distintas categorías sintácticas, y se intruducen distintos identificadores que representarán instancias de una categoría sintáctica determinada. Por ejemplo, en el lenguaje de las expresiones aritméticas, tendríamos las siguientes categorías:

```
b          ∈ Bloque
a          ∈ Asig
n          ∈ Num
v          ∈ Var
ob         ∈ Opb
ou         ∈ Opu
e, e1, e2 ∈ Exp
```

En la segunda sección se define la estructura de aquellas categorías que lo requieran, mediante producciones similares a las utilizadas en la notación BNFE. En el caso de que dos o más objetos de una misma categoría aparezcan en una regla, se diferenciarán usando instancias con nombres diferentes. Aquellas categorías que no tengan una estructura relevante, las denominaremos terminales y no tendrán ninguna producción asociada. En las partes derechas de la reglas pueden aparecer símbolos que son irrelevantes para la sintaxis abstracta, pero la hacen mas legible. Las definiciones de las categorías del ejemplo anterior serían:

```
b  : a*
   ;
a  : v = e
   ;
e  : v
   | n
   | ou e1
   | e1 ob e2
   ;
ou : -
   ;
ob : + | - | ⋆ | /
   ;
```

Como se puede ver con esta notación se pueden obtener especificaciones muy compactas. Además, a través de las categorías sintácticas, nos proporciona un mecanismo para clasificar los distintos elementos estructurales de un lenguaje, que es precisamente lo que queremos destacar con una gramática abstracta.

La especificación completa queda en la forma:

Categorías sintácticas

```
b        ∈ Bloque
a        ∈ Asig
n        ∈ Num
v        ∈ Var
ob       ∈ Opb
ou       ∈ Opu
e, e1, e2∈ Exp
```

Definiciones

```
b  : a*
   ;
a  : v = e
   ;
e  : v
   | n
   | ou e1
   | e1 ob e2
   ;
ou : -
   ;
ob : + | - | ⋆ | /
   ;
```

10.3 Aplicaciones de la sintaxis abstracta

Como hemos visto la sintaxis abstracta es un formalismo que nos permite describir la estructura profunda de un lenguaje. De esta forma, al quedarnos sólo con los aspectos esenciales, somos capaces de reflejar más claramente su estructura, y por lo tanto se simplifica el posterior tratamiento semántico.

Cuando se aborda la descripción formal de un lenguaje existente o la definición de un nuevo lenguaje, suele ser un típico error concentrarse inicialmente en la definición de los detalles de sintaxis concreta en lugar de comenzar determinando claramente cuales son las características esenciales del lenguaje. Esta premura suele dar lugar a descripciones poco claras en el caso de lenguajes existentes o a lenguajes innecesariamente cargados en el caso de la definición de nuevos lenguajes.

La sintaxis abstracta permite definir los aspectos estructurales, de tal forma que a partir de esa estructura tenemos una base para describir formalmente la semántica estática y dinámica del lenguaje. Podemos, de esta forma, completar descripción de los aspectos esenciales del lenguaje sin tener que mezclarlos con los aspectos externos o representación.

10.4 Gramáticas con atributos

Existen muchas características de los lenguajes de programación que no pueden ser descritas mediante una gramática independiente del contexto, o que si lo son, es a costa de

complicar sobremanera la especificación gramatical. A la hora de describir estas características necesitaremos una notación que nos permita especificarlas formalmente al igual que hemos hecho con los aspectos puramente léxicos y sintácticos (a través de las expresiones regulares y los lenguajes independientes del contexto respectivamente). Las gramáticas con atributos se presentan como una buena notación para ampliar el poder descriptivo de las gramáticas independientes del contexto.

El objetivo perseguido con la ampliación del poder descriptivo de las gramáticas independientes del contexto, no es otro que el poder expresar características que se salen del ámbito de lo sintáctico. De esta forma utilizaremos las gramáticas con atributos como método de especificación de ciertos aspectos semánticos de los lenguajes, en concreto los aspectos que se conocen con el nombre de *restricciones contextuales* o también *semántica estática* de los lenguajes.

10.4.1 Un ejemplo simple

Veamos uno de esos casos en los que no es del todo apropiado utilizar simplemente gramáticas independientes del contexto como método de descripción:

```
main()
 {
   ...
  while(...)
   {
     ...
    break;
     ...
   }
   ...
 }
```

En el ejemplo anterior se muestra una utilización de la sentencia *break* del lenguaje de programación *C*, dicha sentencia nos permite abandonar ciertas estructuras de control, así en el caso del ejemplo la sentencia *break* haría que el flujo del programa continuase en la sentencia inmediatamente posterior a la estructura *while* (que es la estructura que engloba directamente a la sentencia *break*). Una posible gramática que generase el lenguaje asociado a las sentencias de *C* sería:

```
sentencia : WHILE expresion sentencia
          | '{' sentencias '}'
            ...
          | BREAK ';'
```

Esta gramática es válida en el sentido de que el lenguaje generado por ella contiene todas las posibles sentencias que involucren estructuras *while* y sentencias *break*. Sin embargo, no todas las sentencias generadas por esta gramática son válidas ya que se definen ciertas restricciones en la ubicación de las sentencias *break*:

```
main()
 {
   ...
  break;
   ...
 }
```

Este sería un fragmento de un programa erróneo, ya que las sentencias *break* sólo pueden encontrarse dentro de una sentencia iterativa (*while*, *for* o *do*) o dentro de una sentencia alternativa múltiple (*switch*), pero sin embargo, es un programa sintácticamente correcto si atendemos a la gramática expuesta anteriormente.

Está claro que la gramática propuesta no recoge todas las características del lenguaje. Para describirlas tenemos dos posibilidades, o bien definir otra gramática independiente del contexto que generase estrictamente el lenguaje en cuestión, o bien aprovechar la gramática propuesta y utilizar otra notación que fuese capaz de describir las restricciones de ubicación de las sentencias *break*. El inconveniente de la primera opción es que puede dar lugar a especificaciones gramaticales bastante complejas y sobre todo poco legibles, además de encontrarnos con la limitación de descripción de los lenguajes independientes del contexto (que se demuestra con el lema del bombeo para los lenguajes independientes del contexto). Las gramáticas con atributos son un ejemplo de la segunda opción permitiendo ampliar una gramática independiente del contexto mediante atributos asociados a los símbolos y reglas de cálculo de dichos atributos.

En el ejemplo de la sentencia *break*, una solución puede ser que cada símbolo no terminal *sentencia* reciba el valor de un atributo que determine en que ámbito se encuentra dicha sentencia (pudiéndose así decidir si se permite o no la aparición de una sentencia *break*). Más adelante se definirá este tipo de atributos con el nombre de atributos heredados, y nos servirán fundamentalmente para que cada símbolo pueda recibir información contextual del conjunto de símbolos que le rodean.

10.4.2 ¿Gramáticas concretas o gramáticas abstractas?

Las gramáticas con atributos, como hemos comentado anteriormente, se definen como una extensión a las gramáticas independientes del contexto. Hasta ahora hemos utilizado las gramáticas independientes del contexto para definir dos aspectos del lenguaje que queremos describir, por un lado su apariencia externa (gramáticas concretas) y por otro su estructura profunda (gramáticas abstractas). ¿Con cual de estas dos *versiones* nos quedaremos para definir gramáticas con atributos?.

Como veremos a lo largo de este capítulo, la respuesta a la pregunta anterior es que los dos usos que hemos dado a las gramáticas independientes del contexto van a ser interesantes para su extensión con atributos. Para el caso de las gramáticas concretas, existen herramientas que permiten especificar reconocedores sintácticos, que al mismo tiempo que analizan la entrada son capaces de evaluar valores de atributos asociados a determinados símbolos. Por otra parte la especificación e la semántica estática de un lenguaje es mas conveniente hacerla con gramáticas abstractas con atributos.

10.5 Definición y especificación de gramáticas con atributos

Una gramática con atributos es una gramática independiente del contexto, donde:

- Se asocia un conjunto de atributos a cada símbolo,
- se definen reglas para calcular dichos atributos y
- se asocia a cada producción un predicado que deben cumplir atributos de los símbolos de cada producción.

10.5.1 Notación

Para representar estas gramáticas con atributos, vamos a utilizar una notación que es una extensión de la notación para la sintaxis abstracta vista anteriormente. En la especificación introduciremos una sección para definir los atributos de la categorías sintácticas. Las producciones las ampliaremos en el siguiente sentido:

P <C> {A}

En este caso P es una producción de una gramática independiente del contexto, y nos sirve de base para la descripción de la gramática con atributos. C es un predicado calculado en términos de los atributos de los símbolos, que condiciona la aplicación de la producción gramatical. Por último A es una acción semática que especifica qué valores toman ciertos atributos en función de los valores de otros atributos.

La notación que utilizaremos para referenciar a los atributos de los símbolos dentro de la condición C y de la acción A será la siguiente:

símbolo.atributo

Dado que en una misma producción pueden aparecer más de una instancia de un mismo símbolo, usaremos diferentes identificadores para diferentes instancias de una misma categoría sintáctica en la misma regla.

Veamos un ejemplo de un lenguaje que no es independiente del contexto y que se podrá especificar fácilmente mediante una gramática con atributos. El lenguaje en cuestión es $L_{dc} = \{a^n b^n c^n \mid n \geq 1\}$ que es un típico ejemplo de lenguaje dependiente del contexto. Una gramática con atributos que generase dicho lenguaje sería:

Categorías sintácticas

S ∈ Leng
A,A1∈ La
B,B1∈ Lb
C,C1∈ Lc

Atributos

< n:int > La, Lb, Lc

Definiciones

S :	A B C	$< A.n = B.n \wedge A.n = C.n >$
	;	
A :	a	$\{A.n := 1\}$
	| A1 a	$\{A.n := A1.n + 1\}$
	;	
B :	b	$\{B.n := 1\}$
	| B1 b	$\{B.n := B1.n + 1\}$
	;	
C :	c	$\{C.n := 1\}$
	| C1 c	$\{C.n := C1.n + 1\}$

En este caso el operador :=, refleja una asignación, indicando la forma en la que se evaluará el atributo que aparece a su izquierda, en función de la expresión que aparece a su derecha. Si analizamos la gramática anterior, vemos que realmente definimos el lenguaje L_{Leng} como el conjunto de las palabras del lenguaje $L_{Leng} = \{a^i b^j c^k \mid i, j, k \geq 1\}$ (que es el generado por la gramática sin atributos y que se demuestra que es regular) que cumplen la condición $A.n = B.n \wedge A.n = B.n$, donde $A.n$, $B.n$ y $C.n$ son el número de a's, b's y c's respectivamente. Veamos otra forma de especificar el mismo lenguaje:

Categorías sintácticas

S	∈ Leng
A,A1	∈ La
B,B1	∈ Lb
C,C1	∈ Lc

Atributos

< n:int >	La, Lb, Lc
< l:Bool >	Leng

Definiciones

S :	A B C	$\{S.l := A.n = B.n \wedge A.n = B.n\}$
	;	
A :	a	$\{A.n := 1\}$
	| A1 a	$\{A.n := A1.n + 1\}$
	;	
B :	b	$\{B.n := 1\}$
	| B1 b	$\{B.n := B1.n + 1\}$

```
  ;
C: c          {C.n := 1}
 | C1 c       {C.n := C1.n + 1}
```

En esta gramática definimos el mismo lenguaje L_{Leng}, como el conjunto de las palabras de L_{Leng} que hacen que el atributo l (que ha de ser de tipo lógico) del símbolo S tenga el valor *cierto*. En este caso hemos conseguido modelar la condición C a través de un atributo de tipo lógico asociado al símbolo S.

Podemos incluso plantearnos eliminar las acciones semánticas, expresándolas por medio de condiciones. Siguiendo con el lenguaje L_{dc}, tendríamos la siguiente especificación:

Categorías sintácticas

```
S     ∈ Leng
A,A1∈ La
B,B1∈ Lb
C,C1∈ Lc
```

Atributos

```
< n:int >∈La, Lb, Lc
< l:Bool >∈Leng
```

Definiciones

```
S  : A B C    < A.n = B.n ∧ A.n = B.n >
   ;
A  : a        < A.n = 1 >
   | A1 a     < A.n = A1.n + 1 >
   ;
B  : b        < B.n = 1 >
   | B1 b     < B.n = B1.n + 1 >
   ;
C  : c        < C.n = 1 >
   | C1 c     < C.n = C1.n + 1 >
```

En este caso obtenemos una especificación mucho más declarativa, al perder el aspecto procedural que introduce el operador :=, con su sentido de evaluación de derecha a izquierda.

Como se puede ver, la notación elegida es bastante potente y flexible, permitiéndonos por un lado especificaciones más procedurales (utilizando acciones semánticas), y por otro lado especificaciones mucho más abstractas (utilizando condiciones). Sin embargo, a la hora de abordar la implementación de las gramáticas con atributos, partiremos de producciones que contengan sólo acciones semánticas, precisamente por su mayor cercanía a la implementación.

10.5.2 Atributos heredados y sintetizados

Dada una gramática concreta y una palabra del lenguaje generado por dicha gramática, existe un *árbol sintáctico* también denominado *árbol de derivación*. Las gramáticas abstractas describen el lenguaje como un conjunto de árboles. Ya sean gramáticas concretas o abstractas los árboles referidos verifican que si su raiz es A y sus hijos $A_1...A_n$, entonces hay una regla en la gramática del tipo $A : A_1...A_n$. En este sentido, ante una gramática con atributos, podemos establecer una clasificación entre los atributos de sus símbolos en función de que se calculen en base a descendientes en el árbol o antecesores en el mismo.

En términos gramaticales diremos que para una producción $A : A_1...A_n$, con $a, a_1, ..., a_n$ los atributos de los diferentes símbolos:

- a será un atributo sintetizado del símbolo A, si existe una relación funcional del tipo $a = f(a_1, ..., a_n)$ asociada a dicha producción.
- a_i será un atributo heredado de uno de los símbolos A_i si exite una relación funcional del tipo $a_i = f(a, a_1, ., a_{i-1}, a_{i+1}, .., a_n)$ asociada a dicha producción.

Como se puede observar, esta clasificación sólo tiene sentido para las relaciones funcionales definidas en una acción semántica A, y no es aplicable a gramáticas en las que las relaciones entre los distintos atributos se definen en base a condiciones, ya que con ellas no se introduce ningún tipo de sentido de evaluación.

10.5.3 Especificación

Con lo que hemos visto estamos en condiciones de especificar el problema que nos habiamos planteado mas arriba. El ejemplo elegido es el de las restricciones de ubicación de las sentencias *break* y *continue* en un subconjunto del lenguaje C, que nos sirvió de motivación al principio del capítulo para introducir las gramáticas con atributos.

A la hora de plantearnos la especificación del problema, lo primero que tenemos que tener claro es la estructura de los elementos del lenguaje que vamos a procesar. La mejor forma de hacerlo es describiendo su sintaxis abstracta:
Categorías sintácticas

```
s,s1,s2 ∈ Sentencia
exp     ∈ Expresion
c,c1    ∈ Casos
prog    ∈ Programa
```

Definiciones

```
prog : s
       ;
s    : s1 ; s2
     | if (exp) s1 else s2
     | while (exp) s1
```

| switch (exp) c
| break
| continue
c : exp → s
| c1 [] exp → s

Con la gramática abstracta disponemos de la base para definir la gramática con atributos. sin embargo, aún no tenemos todos los datos del problema, nos falta por conocer precisamente las restricciones de ubicación de las sentencias *break* y *continue*. Dichas restricciones se podrían resumir (de manera *informal*) de la siguiente forma:

- Una sentencia *break* deberá aparecer en el ámbito de una sentencia *while* o de una sentencia *switch*.
- Una sentencia *continue* deberá aparecer en el ámbito de una sentencia *while*.

Para especificar las restricciones de ubicación vamos a definir dos atributos de tipo lógico *ew* y *es*, que reflejarán si nos encontramos o no dentro de un *while* o dentro de un *switch*.

- El atributo *ew* de una sentencia, será T si dicha sentencia se encuentra dentro de un *while* y F en otro caso.
- El atributo *es* de una sentencia (o también de un caso), será T si dicha sentencia o caso se encuentran dentro de un *switch* y falso en otro caso.

Los atributos *ew* y *es* serán atributos heredados, que en la raíz del árbol (símbolo *prog*) serán falsos, y que se harán ciertos en los ámbitos de las estructuras de control que correspondan. La gramática con atributos que describe la forma en la que se calcularán los atributos *ew* y *es* es la siguiente:

Categorías sintácticas

s,s1,s2 ∈ Sentencia
exp ∈ Expresion
c,c1 ∈ Casos
prog ∈ Programa

Atributos

< ew,es: Bool > Prog, Sentencia, Casos

Definiciones

prog : s $\{s.ew := F$
; $s.es := F\}$

s	: s1 ; s2	$\{s1.ew := s.ew$
		$s2.ew := s.ew$
		$s1.es := s.es$
		$s2.es := s.es\}$
	| if (exp) s1 else s2	$\{s1.ew := s.ew$
		$s2.ew := s.ew$
		$s1.es := s.es$
		$s2.es := s.es\}$
	| while (exp) s1	$\{s1.ew := T$
		$s1.es := s.es\}$
	| switch (exp) c	$\{c.es := s.es\}$
	| break	$< (s.ew = T) \vee (s.es = T) >$
	| continue	$< s.ew = T >$
	;	
c	: exp $\rightarrow$ s	$\{s.es := c.es$
		$s.ew := c.ew\}$
	| c1 [] exp $\rightarrow$ s	$\{s.es := c.es$
		$s.ew := c.ew\}$

De esta forma, una vez que se han calculado todos los atributos *es* y *ew*, expresamos las restricciones de ubicación a través de las condiciones asociadas a las sentencias *break* y *continue*.

Capítulo 11

Tablas de Símbolos

11.1 Las declaraciones en los lenguajes de programación

Las restricciones en la parte de declaraciones en los lenguajes de programación son cuestiones que no se pueden especificar mediante gramáticas independientes del contexto. En este capítulo vamos a ver algunos de ejemplos de especificación y las técnicas usadas para verificar esas restricciones en tiempo de compilación.

En lo que sigue vamos a usar un tipo de datos llamado *mapa(D,R)*. Este tipo de datos es el tipo de las aplicaciones finitas de un tipo de dato D en otro R. Las operaciones de que viene dotado y sus definicione informales son:

$\{\}$	Construye un mapa vacio
$\{a1 \mapsto b1\}$	Construye un mapa con el par definido
$\{a1 \mapsto b1, ..., an \mapsto bn\}$	Construye un mapa con los pares definidos
$dom(m)$	Da el dominio de m
$rng(m)$	Da el rango de m
$\{m[a \mapsto b]\}$	Construye un nuevo mapa haciendo que la imagen de a sea b
$\{m1 \oplus m2\}$	Construye un mapa con los pares definidos sustituyendo las imagenes redefinidas en m2
$\{m[a]\}$	Da la imagen de a en la aplicacion m

Con el tipo de dato anterior podemos especificar las restricciones contextuales de las declaraciones de la variables y los ambitos de las misma. En la especificación siguiente el tipo *mapaIT* es un instanciacón del tipo anterior tomando como dominio el tipo *Cadena* y como rango el tipo $Tipo$ que se verá mas adelante.

Dependiendo del lenguaje estas restricciones serán de un tipo u otro.

Categorías sintácticas

p ∈ Prog
b ∈ Bloque
d,d1∈ Declaraciones
a ∈ Declsim
s,s1 ∈ Sentencias
c ∈ Sentbas
e ∈ Expresion
i ∈ Identificador
f ∈ Informacion

Atributos

< m : mapaIT >	Bloque, Sentencia, Expresion, Declaraciones
< l: Cadena >	Identificador
< t: Tipo >	Informacion

Definiciones

p	: b	{b.m:=mini;}
	;	
b	: d s	{ s.m:=d.m; b.m:=d.m;}
	;	
d	: a	$\{d.m := \{a.l \mapsto a.t\}$
	| d1 a	$< a.l \notin dom(m) > \{d.m := d1.m \oplus \{a.l \mapsto a.t\}$
	;	
a	: i f	$\{a.l := i.l; a.t := f.t;\}$
	;	
s	: c	$\{c.m := s.m;\}$
	| s1 c	$\{c.m := s.m; s1.m := s.m;\}$
c	: i = e	$< i.dl \in dom(c.m) > \{e.m := c.m\}$
	| b	$\{b.m := c.m;\}$
	| while (e) s	$\{e.m := c.m; s.m := c.m\}$
	| if (e) then s1 else s2	$<> \{e.m := c.m; s1.m := c.m; s2.m := c.m\}$
	;	
e	: i	$< i.l \in dom(e.m); > \{\}$
	| n	$\{\}$
	| e1 ob e2	$\{e1.m := e.m; e2.m := e.m;\}$
	| e1 ou	$\{e1.m := e.m\}$
	;	

En esta especificación hemos dicho que no se pueden repetir identificadores con el mismo lexema en una declaración. Ademas cada variable debe estar declarada antes de ser usada y se especifican las reglas de ambito dentro de los diferentes bloques. Igualemente se especifica que el lenguaje tiene un conjunto de símbolos predefinidos, pero esos símbolos pueden ser cambiados de significado si dentro de un bloque declaracmos una variable local con el mismo lexema.

En los compiladores la manera usual de implementar la verificación de estas restricciones es mediante la tabla de símbolos. La tabla de símbolos es de hecho una implementación

del tipo $mapa$ comentado anterioremente. Al ser necesario acceder a la tabla se símbolos desde sitios muy diversos para conseguir mayor eficiencia se hace que esta sea un objeto global. A este objeto accederán los distintos módulos del compilador. Pero no olvidemos que lo que estamos implementando está especificado meddiante una gramática con atributos. Como en el ejemplo de antes.

11.2 La tabla de símbolos

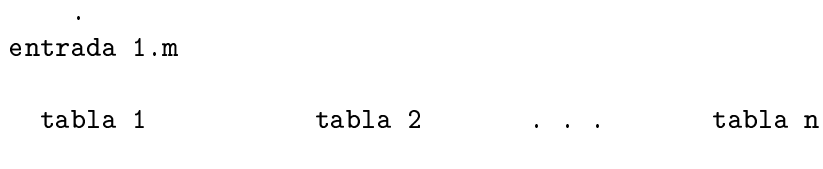
La tabla de símbolos proporciona un mecanismo para asociar valores (atributos) con nombres. Estos atributos son una representación del significado (semántica) de los nombres a los que están asociados, en este sentido, la tabla de símbolos desempeña una función similar a la de un diccionario. La tabla de símbolos aparece como una estructura de datos especializada que permite almacenar las declaraciones de los símbolos, que sólo aparecen una vez en el programa, y acceder a ellas cada vez que se referencie un símbolo a lo largo del programa, en este sentido la tabla de símbolos es un componente fundamental en la etapa del análisis semántico ya que será aquí donde necesitaremos extraer el significado de cada uno de los símbolos para llevar a cabo las comprobaciones necesarias.

Plantearemos el estudio de las tablas de símbolos intentando separar claramente los aspectos de especificación y los aspectos de implementación, de esta forma a través de la definición de una interface conseguiremos una visión abstracta de la tabla de símbolos en la que sólo nos preocuparemos de definir claramente las operaciones que la manipularán sin decir como se van a efectuar. Desde el punto de vista de la implementación, comentaremos diversas técnicas de distinta complejidad y eficiencia, lo que nos permitirá tomar decisiones con respecto a la implementación a elegir en función de las características del problema concreto que abordemos.

Definiremos la tabla de símbolos como un tipo abstracto de datos, es decir un tipo que contiene una serie de géneros y una serie de operaciones sobre esos géneros. A la hora de definir la interface debemos especificar los géneros utilizados, que van a ser conjuntos de valores (que en C se implementarán mediante tipos básicos o tipos definidos por el programador) y por otro lado debemos especificar el conjunto de operaciones que se aplicarán sobre dichos géneros, asociando una signatura o molde a cada nombre de operación que nos dirá que valores toma y que valores devuelve.

En muchos casos será necesario disponer de varias tablas de símbolos con la idea de almacenar de forma separada grupos de símbolos que tienen alguna propiedad en común (por ejemplo todos los campos de un registro o todas las variables de un procedimiento), de esta forma nuestra tabla de símbolos será una estructura que será capaz de albergar varias tablas, que estarán compuestas a su vez por una serie de entradas (una por cada símbolo que contenga la tabla) donde cada entrada se caracterizará por el lexema que representa al símbolo y por los atributos que representan el significado del símbolo.

```
entrada 1.1
entrada 1.2
    .
    .
```



Con este esquema de tabla de símbolos, los géneros necesarios serán *Tabla* para representar una tabla de símbolos, *Entrada* para representar una entrada dada de una tabla de símbolos, *Cadena* que será una cadena de caracteres que representará el lexema asociado a un símbolo y *Atributos* que será un registro donde cada uno de sus campos representará uno de los atributos del símbolo. En el caso de los atributos de un símbolo también se podría plantear la manipulación de cada uno de los atributos individualmente sin tener por que referenciarlos a todos, como ya veremos, esto influirá en la definición de las operaciones de recuperación y actualización de atributos.

La manera de representar los géneros en C será a través de la definición de tipos, en el siguiente fragmento de declaración en C representaremos la definición de estos géneros dejando en puntos suspensivos los detalles correspondientes a la implementación que no deben concretarse al hablar de interface.

```
typedef ... Cadena;
typedef ... Tabla;
typedef ... Entrada;
typedef ... Atributos;

#define TABLA\_NULA ...
#define ENTRADA\_NULA ...
```

Las constantes *TABLA_NULA* y *ENTRADA_NULA*, serán dos valores necesarios para detectar irregularidades acontecidas al aplicar ciertas operaciones sobre tablas de símbolos.

Veamos ahora cuales son las operaciones necesarias para manipular adecuadamente la estructura de datos que nos hemos planteado, en primer lugar comentaremos las funciones de creación y destrucción de tablas de símbolos. La función *crea_tabla* se encarga de crear la estructura de una nueva tabla de símbolos con todas sus entradas vacías devolviendo *TABLA_NULA* si se ha producido algún error en la creación de la nueva tabla. La función *destruye_tabla* se encarga de eliminar la estructura de una tabla de símbolos existentes despues de borrar todas sus entradas, los prototipos de estas funciones son los siguientes:

```
Tabla crea\_tabla(void);
void destruye\_tabla(Tabla);
```

El siguiente grupo de funciones se encarga de la gestión de entradas dentro de una tabla de símbolos. La función *busca_simbolo* busca un determinado lexema dentro de una tabla de símbolos devolviendo la entrada coorespondiente o *ENTRADA_NULA* si dicho símbolo no existe. La función *instala_simbolo* crea una entrada para un nuevo símbolo en una tabla dada devolviendo la referencia a la entrada o *ENTRADA_NULA* si se produce algún error en la creación de la nueva entrada. La función *borra_simbolo* se encarga de eliminar la entrada de un símbolo, los prototipos de estas funciones son:

```
Entrada busca\_simbolo(Cadena, Tabla);
Entrada instala\_simbolo(Cadena, Tabla):
void borra\_simbolo(Entrada);
```

Algunas de las operaciones vistas anteriormente son implementaciones de las operaciones definidas para el tipo mapa. Pero esta implementación tiene en cuenta el tratameinto de errores y los correspondientes mensajes de error.

Las operaciones de manipulación de atributos se van a encargar de actualizar y recuperar los valores de los atributos de un determinado símbolo. Tenemos dos poibilidades, la primera es la de utilizar el género *Atributos* y disponer de dos funciones *actualiza_atributos* y *recupera_atributos* que tratan en bloque a todo el registro de atributos, debiéndose definir operaciones para construir y disgregar elementos del género *Atributos*:

```
void actualiza\_atributos(Entrada, Atributos);
Atributos recupera\_atributos(Entrada);
```

La segunda posibilidad de manipular los atributos no hace referencia al género *Atributos* definiéndose varias parejas de funciones que traten individualmente cada uno de los atributos, indicando el nombre y el tipo del atributo concreto:

```
void actualiza\_<nombre>(Entrada, <tipo>);
<tipo> recupera\_<nombre>(Entrada);
```

Con el conjunto de funciones planteado, disponemos de una interface para manipular varias tablas de símbolos, capaces de albergar varias símbolos y asociarles a cada uno de ellos un conjunto de atributos. Esta interface constituye un buen punto de partida y contiene las funciones necesarias para resolver los problemas planteados a la hora de procesar un lenguaje simple, sin embargo puede que en algunos casos necesitemos gestionar otras características adicionales propias del lenguaje a procesar. Podemos pues considerar la interface hasta ahora propuesta como el núcleo de cualquier tabla de símbolos que se podrá ampliar en el sentido impuesto por las características específicas de cada lenguaje.

11.3 Tablas de símbolos con estructura de bloques

En esta sección vamos a presentar una extensión a la interface de tabla de símbolos anterior que nos ayudará a resolver el problema de los ámbitos de visibilidad. Este problema se plantea en algunos lenguajes de programación, donde se divide el programa en bloques o módulos en cada uno de los cuales son visibles un conjunto determinado de símbolos. Ejemplos de estos ámbitos lo constituyen las variables locales dentro de un procedimiento o la los campos dentro de un registro. En algunos casos estos ámbitos no son disjuntos y se pueden solapar, complicándose el problema cuando en ámbitos solapados se definen símbolos con el mismo nombre, como pasa por ejemplo en el anidamiento estático de procedimientos de PASCAL. Veamos un fragmento de un programa donde se planteen conflictos de este tipo:

```
Modulo primero
   Variables
```

```
      A,B,C : Entero;
   Comienzo
   Modulo segundo
      Variables
         X,Y : Real;
      Comienzo
         ...
      Fin segundo;
   Modulo tercero
      Variables
         C,D : Caracter;
      Comienzo
         ...       <----- (*)
      Fin tercero;
Fin primero;
```

Este ejemplo plantea un problema similar al del anidamiento de procedimientos en PASCAL, en este caso a cada módulo se le asocia un ámbito que contiene las variables que se declaran en su sección de *Variables* además de las que hereda del módulo que lo contiene, cuando un módulo superior contiene una variable con el mismo nombre que una del módulo actual, la herencia no se lleva a cabo y prevalece la declaración actual. De esta forma en el punto marcado con (*) del ejemplo las variables C y D son de tipo *Caracter*, A y B son de tipo *Entero* y X e Y no son visibles. Para gestionar estos ámbitos de variables, describiremos una ampliación a la interface de tabla de símbolos que permita organizar el conjunto de tablas existentes en cada momento para llevar a cabo correctamente la búsqueda de un nombre y detectar a que tabla pertenece. En el caso concreto del anidamiento de procedimientos, la organización adecuada es la estructura de datos pila, de tal forma que en la cima de la pila se coloque la tabla correspondiente al módulo actual y en las posiciones más inferiores las tablas correspondientes a los módulos predecesores:

```
tercero <---- CIMA
primero
```

Las operaciones a añadir a la interface de la tabla de símbolos se encargarán fundamentalmente de gestionar la pila de tablas de símbolos, en este sentido las operaciones *apila_tabla*, *desapila_tabla* y *tabla_actual* se encargarán de manipular la pila de tablas, las dos primeras insertarán y eliminarán una tabla en la pila y la tercera simplemente consultará la cima de la pila sin modificarla:

```
void apila\_tabla(Tabla);
Tabla desapila\_tabla(void);
Tabla tabla\_actual(void);
```

Además se proporciona la función *busca_globalmente* que se encargará de buscar un nombre de identificador dentro de todas las tablas existentes, es en esta función donde se especifican claramente las reglas de búsqueda de un nombre:

```
Entrada busca\_globalmente(Cadena);
```

Estas operaciones son una forma de implementar la operación $\oplus$ definida al pricipio sobre los mapas.

A la hora de implementar esta extensión de la interface nos encontramos con que tenemos varias tablas de muy diversos tamaños, con lo que se hace muy difícil escoger un método de implementación que se ajuste a todas (ya que su eficiencia y complejidad depende del número estimado de símbolos) o el dimensionamiento adecuado para algunos métodos concretos, como es el caso de las tablas hash. Una posible solución es mezclar todas las tablas en una misma estructura, con lo que sólo tendríamos que estimar la cantidad de símbolos que se albergarán globalmente y que cada símbolo dentro de esta estructura tenga asociada la información referente a la tabla a la que pertenece, en este sentido una posible representación para el tipo *Tabla* sería a través de números enteros:

```
typedef int Tabla;
```

Las tablas hash y los árboles binarios ordenados son métodos adecuados para soportar esta estructura general. Un inconveniente de esta opción es que hay ciertas funciones se complican, como es el caso de *destruye_tabla*, que debería encargarse de recorrer toda la estructura para eliminar selectivamente los símbolos pertenecientes a la tabla a destruir. A la hora de decidir si esta implementación constituye una buena solución, deberemos valorar el nivel de utilización de las funciones que resultan más ineficientes.

11.4 Soluciones a problemas particulares

En esta sección veremos como ampliar la interface propuesta hasta ahora para resolver problemas específicos que plantean algunos lenguajes. No se ha introducido las tablas con estructura de bloques por considerarse el problema de los ámbitos con una entidad propia al ser común en muchos lenguajes de programación. Estudiaremos aquí la probremática asociada a la definición de campos y registros, la importación y exportación de símbolos entre tablas y la sobrecarga de definiciones para un símbolo.

11.4.1 Campos y registros

En la mayoría de los lenguajes de programación los nombres de campos dentro de los registros son locales a la decalaración del registro, esto hace que distintos campos puedan tener el mismo nombre. En estos casos utilizaremos una solución desde el punto de vista de la tabla de símbolos, similar a la estructuración por bloques, asignando una tabla de símbolos a cada declaración de registro y haciendo de esta forma que los nombres de los campos sean locales a dicha declaración:

```
A : Registro
       A : Entero;
       B : Registro
              A: Real;
              C: Caracter;
           Fin Registro;
    Fin Registro;
```

En la declaración anterior, el lexema *A*, aparece en tres lugares distintos como variable tipo registro, campo de tipo entero y campo de tipo real, debiendo aparecer pues en tres tablas como símbolo distinto. A diferencia del anidamiento estático de procedimientos, cuando se abre un nuevo ámbito, las nueva tabla no amplía el conjunto de símbolos que pueden aparecer sino que lo sustituye ya que cuando se abre un nuevo ámbito de registro a través del prefijo correspondiente (prefijos válidos en el ejemplo serían *A.* o *A.B.*) sólo se permite la ocurrencia de los campos de dicho registro.

11.4.2 Reglas de importación y exportación

Cuando se definen módulos puede ser de interés que ciertos símbolos pertenecientes a un módulo sean visibles desde otro, por ejemplo para definir variables globales compartidas entre varios módulos o para diseñar librerías que exporten funciones hacia otros módulos. Veamos con un ejemplo como utilizando las reglas de importación y exportación somos capaces llevar a cabo esta comunicación entre módulos.

```
Modulo Pila
  Exporta Apila, Desapila;
  Variables
     pila : Tabla [100] de Entero;
     cima : Entero;
  Funcion Apila(Entero)
     ...
  Fin Apila;
  Funcion Desapila : Entero
     ...
  Fin Desapila;
Comiezo
   pila = 1;
Fin Pila;

Modulo Evaluador
   Exporta Evalua;
   Importa Apila, Desapila de Pila;
      ...
Fin Evaluador;
```

En el fragmento de programa anterior se describen dos módulos, el primero se encarga de implementar una pila de enteros y deja visibles sólo las funciones *Apila* y *Desapila*, ocultando la propia implementación de la pila, el segundo módulo será un evaluador de expresiones aritméticas, dejará visible la función *Evalua* y tomará las funciones *Apila* y *Desapila* del módulo *Pila*. Mediante las reglas de importación y exportación somos capaces de determinar selectivamente que símbolos han de ser conocidos entre diversos módulos. Para que nuestra tabla de símbolos soporte este tipo de reglas debemos ampliar la interface con dos funciones. La función *exporta_simbolos* tomará como argumento dos tablas y moverá de una tabla a otra todos los símbolos de la primera calificados como exportables (esta calificación puede ser un atributo del símbolo). La función *importa_simbolo* tomará como argumentos dos tablas y un nombre de símbolo y copiará la definición del símbolo de la primera a la segunda tabla, devolviendo la entrada creada en la tabla receptora.

```
void exporta\_simbolos(Tabla, Tabla);
Entrada importa\_simbolo(Tabla, Tabla, Cadena);
```

Cuando los módulos se describen en distintas unidades de compilación (usualmente un fichero), nos encontramos ante lo que se denomina compilación separada, con lo que nos tendremos que encargar de almacenar y recuperar tablas de un fichero objeto. La función *almacena_tabla* tomará una tabla de símbolos y la guardará en un fichero con algún formato. La función *recupera_tabla* leerá del fichero especificado creando la correspondiente tabla en memoria.

```
void almacena\_tabla(Tabla, Fichero);
Tabla recupera\_tabla(Fichero);
```

En el caso de la compilación separada, el procedimiento será el siguiente: para creación del fichero objeto, se creará una tabla que contenga todos los símbolos exportables, mediante la función *exporta_simbolos* y se guardará en el fichero con la función *guarda_tabla*, para la lectura de declaraciones de otros módulos se creará una tabla leyéndola del fichero con *recupera_tabla* y luego se irán tomando selectivamente los símbolos con la función *importa_simbolo*.

11.4.3 Sobrecarga

En muchos lenguajes de programación se permite sobrecargar la definición de símbolos, es decir dar más de un significado a un símbolo y decidir cual de ellos es dependiendo del contexto en el que se encuentre. En estos casos la tabla de símbolos deberá encargarse de agrupar todos los significados de un símbolo y de proporcionar un mecanismo para ofrecer todos los significados de un símbolo. Una posible solución para recuperar todos los significados de un símbolo consiste en agrupar varias entradas en una sola, mediante la función *numero_significados* somos capaces de conocer cuantos significados tiene una entrada, y mediante la función *significado_iesimo* recuperamos un significado concreto:

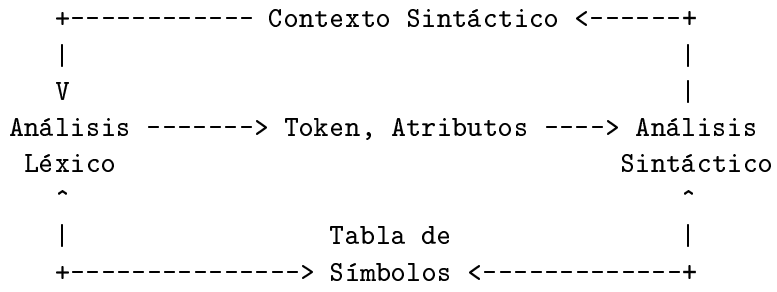
```
int numero\_significados(Entrada);
Entrada significado\_iesimo(Entrada, int);
```

De esta forma seremos capaces de acceder a todos los significados de un símbolo, preguntando cuantos tiene y accediendo luego a cada uno de ellos.

11.5 Relación entre el analizador léxico, el sintáctico y la tabla de símbolos

En principio, la labor del analizador léxico de un compilador es agrupar los caracteres que aparecen en la entrada formando con ellos *tokens* que suministra al analizador sintáctico. Además debe calcular un cierto número de atributos para cada *token* que permitirán al analizar sintáctico conocer la posición exacta del fuente en la que aparecen o saber cuál es el valor numérico de una hilera que representa un número, entre otras cosas.

De todas formas algunos autores insisten en que a pesar de que la tabla de símbolos es un componente claramente semántico, se puede plantear el diseño del compilador de forma que ya en el analizador léxico se le pueda sacar partido. Según estos autores, el analizador léxico, además de reconocer *tokens* en la entrada, debe buscar los identificadores en la tabla de símbolos y devolver su entrada correspondiente o incluso insertarlos en ella.



Según se ve en este gráfico, la tabla de símbolos es leída y actualizada por ambos analizadores. El contexto sintáctico suele ser una variable que indica qué construcción de la gramática se está reconociendo en este momento y ayuda al analizador léxico a determinar si debe insertar los identificadores que encuentra en la tabla o buscarlos. También puede indicar si la búsqueda debe realizarse en la tabla local actualmente activa o de forma global.

En esta sección estudiaremos los problemas que se derivan del uso de la tabla de símbolos dentro del analizador léxico y veremos cuál es la forma más adecuada de utilizarla.

11.5.1 ¿Debe el analizador léxico insertar símbolos en la tabla de símbolos?

Antes de contestar a la pregunta analizaremos un ejemplo bastante clásico en el que el analizador léxico se encarga de insertar todos los identificadores que encuentra en una línea de declaración de variables. En este caso, la atribución de la gramática sería algo similar a lo siguiente:

```
dec_var
  : ENTERO {contexto(DECLARAR);} lista_id {contexto(USAR);} ';'
  ;
```

Básicamente hemos supuesto la existencia de dos contextos que se fijan con la rutina `contexto()`. Cuando nos encontramos en el contexto `DECLARAR` el analizador léxico inserta todos los identificadores que encuentre en la tabla de símbolos, y cuando nos encontramos en el contexto `USAR` los busca, dando un error en caso de no encontrarlos.

La solución parece buena, pero su correcto funcionamiento es tremendamente dependiente de la técnica con la que hayamos desarrollado en analizador y de la forma en que ésta se haya implementado internamente. Para darnos cuenta de lo críticos que pueden ser estos dos aspectos analizaremos el siguiente trozo de programa:

```
Entero a, b;
```

Supongamos que intentamos reconocer este texto usando un *parser* que lleve un carácter de lectura adelantada o *lookahead*. A continuación mostramos la traza del mismo y las acciones que lleva a cabo (Supondremos que el contexto inicial es `USAR`):

```
Entrada   Lookahead   Acción
=========================================================
ENTERO    ID(a)       contexto(DECLARAR) (el  parser)
ID(a)     ','
','       ID(b)       insertar('b'); (el  scanner)
ID(b)     ';'
...
```

Si analizamos la traza cuidadosamente nos daremos cuenta de que después de leer el *token* `ENTERO`, el *parser* lee inmediatamente el siguiente *token* de la entrada, en este caso el identificador `a`. Pero fíjese en que dado que el *parser* lee inicialmente dos *tokens* de la entrada para poder inicializar el *lookahead*, el cambio de contexto no se produce en el momento oportuno y el analizador léxico devuelve el *token* `ID` correspondiente al lexema `a` cuando aún se encuentra en el contexto `USAR` inicial. Por lo tanto, intentará buscar `a` en la tabla de símbolos actual y si no lo encuentra dará un mensaje de error o en caso de existir devolverá una entrada errónea y no detectará el error de reduplicación. En cualquier caso, la variable `a` no se insertará en la tabla de símbolos.

Si nuestro *parser* implementa el *lookahead* de otra manera, entonces el problema no aparecerá tal y como lo hemos mostrado pero probablemente se manifestará de otra forma. El problema realmente importante aparece cuando usamos un *parser* generado por `Yacc`, cuya característica es que no siempre lleva un *token* de lectura adelantada, sino sólo a veces. Podemos encontrarnos con un compilador que a veces inserta correctamente los símbolos y a veces no.

Otro problema derivado del uso de los contextos es que su correcta sincronización cuando el *parser* es capaz de detectar múltiples errores dentro del mismo programa fuente resulta bastante difícil. Continuando con nuestro ejemplo anterior, ¿qué ocurre si en mitad de la lista de identificadores se produce un error y no llega a ejecutarse la acción que cambia el contexto a `USAR`? Es evidente que sincronizar los contextos en estos casos puede resultar bastante complicado.

Por lo tanto, de lo que hemos estudiado debemos sacar en conclusión que esta forma de trabajo es más un truco que una técnica que podamos generalizar con facilidad y sobre todo aplicar de una manera que resulte portable y segura.

11.5.2 ¿Debe el analizador léxico consultar la tabla de símbolos

Algunos autores se han dado cuenta del problema anterior y han propuesto limitar la interacción del analizador léxico con la tabla de símbolos a la consulta. En estos casos el analizador léxico es capaz de devolver "*tokens* refinados" que dan una información bastante precisa sobre las características del lexema que se acaba de reconocer. Por ejemplo, ya no hablaremos del *token* genérico `ID`, sino de los *tokens* refinados `VARIABLE`, `PROCEDIMIENTO` o `FUNCIÓN`. Desgraciadamente, para que el analizador léxico pueda distinguir aquellos casos en los que debe consultar la tabla de símbolos de aquellos otros en los que no debe hacerlo (por ejemplo, durante la declaración de un conjunto de variables) es necesario

utilizar entornos y ya estudiamos en la sección anterior todos los problemas que esto lleva implícito.

A título de ejemplo, examinaremos una gramática que recoge la estructura de las sentencias de asignación y de las llamadas a rutinas en un cierto lenguaje de programación:

```
sentencia : ID '=' expresión ';'
          | ID '(' lista_exp ')' ';'
          | ID ';' /* Llamada a procedimiento sin parámetros */

expresión : NUM
          | ID '.' ID
          | ID '(' lista_exp ')'
          | ID   /* Variable o función sin parámetros */
```

Es evidente que para reconocer el lenguaje descrito por esta gramática no se puede utilizar un reconocedor LL(1) ni tampoco un LR(0). `Yacc` admite esta gramática, pero en cuanto la complicamos un poco más con nuevos tipos de expresiones, informa de conflictos *shift/reduce*. Si el analizador léxico pudiese consultar la tabla de símbolos, en vez de devolver el *token* genérico `ID` podría ofrecernos un *token* más refinado que nos ayudaría a simplificar la gramática de la siguiente manera:

```
sentencia : VARIABLE '=' expresión ';'
          | PROC_CON_PAR '(' lista_exp ')' ';'
          | PROC_SIN_PAR ';'
          ;

expresión : NUM
          | VARIABLE
          | VARIABLE '.' CAMPO
          | FUNC_CON_PAR '(' lista_exp ')'
          | FUNC_SIN_PAR
          ;
```

La idea es en principio bastante atractiva puesto que además de simplificar las gramáticas las hace substancialmente más legibles. En cualquier caso no podemos aconsejarlo en la práctica por diversos motivos que expondremos en los párrafos siguientes.

El primero y más importante es que la técnica exige de un conocimiento bastante preciso de la implementación del *parser*, y en caso de que éste lleve un *token* permanente de lectura adelantada no funciona. Para convencernos de ello podemos examinar el siguiente programa:

```
Entero a;
a := 2;
```

descrito por la gramática atribuida:

```
prog
  : decl asig
  ;

decl
  : ENTERO { contexto(DECLARAR); } ID ';'
    { contexto(USAR); insertar($2); }
  ;

asig
  : VARIABLE '=' expresión
  ;
```

De nuevo detectaremos el error realizando la traza del *parser*.

```
Entrada   Lookahead  Acción
==================================
ENTERO    ID(a)
ID(a)     ';'
';'       ID(a)      insertar('a')
ID(a)     '='        error
```

Fíjese en que dado que el *parser* que estamos usando lleva un *token* de lectura adelantada, la llamada a insertar en la tabla de símbolos el identificador `a` se produce después de que el *parser* haya leído su *token* de *lookahead*. Cuando el analizador léxico lo leyó, su información aún no se había insertado en la tabla de símbolos y tampoco se había cambiado al contexto `USAR`. Por eso devuelve el *token* general `ID` en vez del *token* más específico `VARIABLE`.

Evidentemente, si somos cuidadosos eligiendo el lugar en el que colocamos la atribución de las reglas, podremos solucionar con relativa facilidad algunos de estos problemas, pero no deja de ser un truco que funciona con nuestro *parser* y probablemente fallará al usar otro. Aún en el caso en el que estemos dispuestos a admitir la perdida de portabilidad, la solución no es adecuada puesto que:

- Considerar casos particulares durante la etapa de análisis sintáctico no nos permite plantear una solución general al problema de la comprobación de tipos del lenguaje. Tendremos que tener en cuenta todos los controles que ya ha realizador el *parser*, con lo que el compilador que obtengamos no tendrá una arquitectura limpia. Las distintas funciones que debe desarrollar el compilador no estarán contenidas dentro módulos separados sino que estarán repartidas a través de diversos módulos cuya interrelación se hace más compleja y difícil de modificar, ampliar, etcétera.
- Muchos errores semánticos se muestran como errores de tipo sintáctico. Por ejemplo, si en la asignación `x := 3`, resulta que `x` es el nombre de un procedimiento, el *parser* indicará un error sintáctico y entrará en el modo de recuperación de errores, que como ya sabemos es bastante difícil de manejar. Si ya es difícil corregir los verdaderos errores sintácticos como la falta de un punto y coma, que el *parser* indique errores semánticos hace la tarea de recuperación de errores mucho más compleja y difícil de implementar satisfactoriamente.

11.5.3 Entonces, ¿dónde se usa entonces la tabla de símbolos?

En el analizador léxico no, esto es evidente después de estudiar las secciones anteriores. Debemos tener en cuenta que la tabla de símbolos es una forma de representar la información que el compilador puede extraer a partir del fuente de un programa, y que está muy relacionada con la estructura de bloques del lenguaje de que estemos compilando. Por lo tanto, dado que esa información sobre el contexto en el que van apareciendo los identificadores está implícita en las reglas de producción de la gramática del lenguaje, es en la atribución que realicemos de la misma en donde debemos usar la tabla de símbolos, tanto para insertar como para consultar identificadores.

Un problema que algunos autores achacan a esta técnica es que las gramáticas resultantes son crípticas y mucho más complejas de leer que las que se obtienen con la técnica que mostramos en la sección anterior. También les achacan que es necesario repetir y repetir la búsqueda de los identificadores una y otra vez en la atribución de la gramática. Pero esto no es cierto, y en breve veremos la justificación. Consideremos para ello la siguiente gramática:

```
...



prog
  : decl sent

decl
  : tipo id_ndec ';'

tipo
  : ENTERO
  | REGISTRO
       ENTERO id_ndec ';'
    FIN_REG

sent
  ; variable '=' expr ';'

expr
  : variable
  | NUM

variable
  : id_dec
  | id_dec '.' id

/* Reglas para el tratamiento de los identificadores */

id_dec
  : ID   { Entrada e = buscar($1);
```

```
            if (e != ENTRADA_NULA)
              $$ = e;
            else
              $$ = crear_entrada($1), error();
          }

id_ndec
  : ID    { Entrada e = buscar($1);

            if (e == ENTRADA_NULA)
              $$ = $1
            else
              $$ = crear_lexema(), error();
          }

id : ID   { $$ = $1; }
```

Fíjese en que desde un punto de vista descriptivo, la gramática resulta bastante legible y compacta. En ella destacaremos el tratamiento que se ha dado a los identificadores. Para ellos hemos creado tres reglas de producción que describen respectivamente los identificadores que deben haber sido declarados previamente, aquellos que no deben estar previamente declarados y aquellos a los que de momento no exigiremos ninguna propiedad especial.

Cada vez que en un contexto sintáctico necesitemos un identificador que haya sido declarado previamente, usaremos en vez del *token* `ID` el símbolo no terminal `id_dec` que reconocerá en la entrada un identificador y lo buscará en la tabla de símbolos. Si aparece devolverá su entrada y si no aparece lo declarará de forma automática como una variable de tipo `TError`, por ejemplo, en la rutina `crear_entrada()`, [1]. Un ejemplo de esto se muestra en la regla de producción

```
variable: id_dec
```

si un identificador aparece en el contexto en el que se espera una variable, entonces ese identificador ha debido ser declarado previamente. Otro ejemplo es la regla de producción

```
variable: id_dec '.' id
```

con la que representamos los accesos a campos de las variables de tipo registro. En una expresión como `x.y` se exige que la raíz `x` sea un identificador declarado con anterioridad, pero no podemos exigir nada al campo puesto que estos son locales a su raíz y realizar la comprobación de que ha sido declarado exigiría que el analizador sintáctico comprobase el tipo de `x` y sintetizase su tabla de símbolos, lo que es claramente una tarea del comprobador de tipos. En un lenguaje como el que hemos mostrado hacer esto no es

[1] Fíjese en que esta idea se puede mejorar bastante. Por ejemplo, haciendo que si el identificador aparece justo antes de un índice se considere que es una variable de tipo tabla, o si aparece justo antes de una lista de parámetros entre paréntesis se considere que es el nombre de un procedimiento. En cualquier caso, la idea que se ha expuesto es sencilla, efectiva y se puede adaptar a las características de cada lenguaje particular con facilidad.

complicado, pero en lenguajes más realistas en los que las raíces de las referencias a campo pueden ser expresiones arbitrariamente complejas exigiría ir realizando todo el trabajo del comprobador de tipos durante la fase de análisis sintáctico.

Esta es también la razón de que tan sólo hablemos de identificadores declarados o no declarados, y no tengamos producciones de la forma:

```
id_var
  : ID { Entrada e = buscar($1);

         if (e == ENTRADA_NULA || e.objeto != VARIABLE)
             $$ = crear_entrada_var($1), error();
           else
             $$ = $1;
       }

id_proc
  : ID { Entrada e = buscar($1);

         if (e == ENTRADA_NULA || e.objeto != PROCEDIMENTO)
             $$ = crear_entrada_proc($1), error();
           else
             $$ = $1;
       }

⋮
```

Hacer esto implica complicar excesivamente la fase de análisis sintáctico, cargándola con comprobaciones que corresponden al comprobador de tipos del lenguaje y que en definitiva tan sólo acaban complicando la interacción entre los diversos módulos del compilador.

`id_ndec` se utilizará en aquellos contextos sintácticos en los que deba aparecer un identificador no declarado. En estos casos del identificador tan sólo nos interesa conocer su lexema. La regla tan sólo reconoce un prefijo que encaje en el *token* `ID` y comprueba que no haya sido declarado con anterioridad. En caso de haberlo sido se informa de ello con un mensaje de error y se devuelve un lexema inventado para corregirlo.

En la práctica, la experiencia nos demuestra que la tabla de símbolos puede ser manejada de una forma eficiente, limpia y no problemática por el analizador sintáctico de la forma que hemos mostrado en los párrafos anteriores. Cualquier otra interacción con las tablas de símbolos en otros módulos del compilador acaba siendo contraproducente y dificultando la construcción modular del mismo.

Capítulo 12

La comprobación de tipos

La semántica de un lenguaje se puede considerar dividida en dos apartados: la semántica estática, que está en relación con el conjunto de propiedades que todos los programas sintácticamente correctos deben cumplir antes de poder ser ejecutados, y la dinámica, que concierne al comportamiento en tiempo de ejecución de aquellos programas que son correctos sintáctica y estáticamente.

La semántica estática tiene multitud de facetas e incluye desde la comprobación de que todos los operadores utilizados son aplicados sobre valores de los tipos adecuados, hasta la comprobación de que en una construcción de la forma `procedure` $\texttt{ID}_1$`()` ... `end` $\texttt{ID}_2$, ambos identificadores deben ser idénticos. También se incluyen dentro de la semántica estática ciertos controles sobre el uso de algunos elementos del lenguaje que son dependientes del contexto y por tanto difíciles de tratar desde el punto de vista sintáctico (por ejemplo, que la sentencia `break` del lenguaje C debe aparecer dentro de una sentencia `switch`, `while` o `do`, que la sentencia `while` o `until` tan sólo puede aparecer cero o una vez dentro de los bucles `loop` en Casale I, o que dentro del mismo ámbito un identificador tan sólo puede declararse en una ocasión).

En este capítulo nos centraremos en la comprobación de tipos (*type checking*), cuya tarea esencial es comprobar que los tipos de todos los elementos que intervienen en una construcción sintáctica son los adecuados dentro de ese contexto. Por ejemplo, comprobar que en la llamada a una rutina se emplean todos los parámetros necesarios y que estos son de los tipos adecuados, que las constantes simbólicas equivalen a expresiones en las que no intervienen elementos variables, que tan sólo se indexan expresiones de tipo tabla, etcétera.

El comprobador de tipos es un módulo del compilador que toma el árbol con atributos creado por el analizador sintáctico y realiza sobre el mismo todas las operaciones necesarias para garantizar su validez antes de que se lleve a cabo la etapa de generación de código intermedio. Supondremos que las comprobaciones sobre unicidad en la declaración de identificadores han sido realizadas previamente con la ayuda de la tabla de símbolos y que todas las situaciones de error han sido tratadas adecuadamente para que el árbol que recibe este módulo del compilador esté completamente libre de casos especiales provocados por la aparición de errores durante la fase de análisis sintáctico. Por ejemplo, si durante ésta se detecta el uso de una variable no declarada, entonces se introducirá automáticamente en la tabla de símbolos y le colocaremos por defecto el tipo `TError`; si el usuario utiliza un nombre de tipo no declarado, lo insertaremos en la tabla de símbolos suponiendo

que es un alias para `TError`; si llamamos a una rutina no declarada, la declararemos automáticamente infiriendo su interface a partir de los parámetros de la llamada; etcétera. En general deberemos tomar decisiones similares en cada caso de error, de forma que la etapa de comprobación de tipos pueda llevarse a cabo sin tener en cuenta estos casos especiales.

Conforme vayamos estudiando este capítulo, nos iremos dando cuenta de que la construcción de un comprobador de tipos en absoluto es una tarea trivial. Por ello intentaremos siempre que las técnicas desarrolladas sean lo más generales posibles en el sentido de que puedan ser adoptadas con facilidad a multitud de lenguajes.

12.1 Los sistemas de tipos

El tipo de un objeto se puede entender como un adjetivo que nos permite conocer cuál es el conjunto de valores que ese objeto puede tomar en tiempo de ejecución. Por ejemplo, decir que `c` es una variable de tipo `Carácter` indica que en tiempo de ejecución esta variable podrá tomar valores sobre el conjunto de caracteres de nuestra máquina; decir que `pe` es de tipo `Puntero a Entero` indica que esa variable puede tomar como valores el conjunto de todas las direcciones disponibles para almacenar números enteros en nuestra máquina [1].

El sistema de tipos de un lenguaje está constituido por el conjunto de reglas que indica si una construcción correcta desde el punto de vista sintáctico es válida en nuestro lenguaje desde el punto de vista semántico, es decir, si lo que hemos escrito tiene significado. También se incluyen en el sistema de tipos las reglas que nos indican cómo calcular el tipo de cada una de las construcciones que permite el lenguaje.

A continuación se muestran algunas de las reglas del sistema de tipos del lenguaje C:

1. Los literales de la forma {`dígito`}`+` son del tipo `Entero`.
2. Los operandos de los operadores aritméticos deben ser de tipo `Entero` o `Real`.
3. Si los dos operandos de un operador aritmético son de tipo `Entero`, entonces la expresión completa es de ese mismo tipo.
4. La expresión que controla el bucle `while` debe ser de tipo `Entero`.
5. En cualquier contexto en el que sea válido un valor de tipo `Entero` también se admite un valor de tipo `Carácter`
6. Etcétera.

En algunos lenguajes estas reglas se pueden comprobar estáticamente durante la compilación de los programas y se dice que su sistema de tipos es estático (Pascal, Ada, Eiffel, C++), pero en otros lenguajes todas las comprobaciones se llevan a cabo en tiempo de ejecución y se dice que su sistema de tipos es dinámico (SmallTalk, BASIC). En general, la tendencia es hacia lenguajes con sistema de tipos estáticos, puesto que en ellos el compilador puede detectar muchos errores y entonces los programas construidos serán más

[1] No se sorprenda de esto, pues las máquinas que reservan ciertas posiciones de memoria para almacenar valores de determinados tipos son bastante frecuentes.

robustos y fáciles de depurar. Los lenguajes con sistema de tipos dinámico son más lentos ejecutándose puesto que todas las reglas del sistema de tipos deben ser comprobadas una y otra vez durante la ejecución de cada sentencia.

12.2 Taxonomía de los tipos de datos

En esta sección estudiaremos de forma general cuáles son los tipos de datos que con más frecuencia se encuentran en los lenguajes de programación habituales. Generalmente se establecen dos grandes grupos: los básicos, todos aquellos que desde el punto de vista del programador no se pueden descomponer en otros elementos más sencillos, y los complejos o estructurados, que se construyen a partir de tipos básicos y complejos aplicando ciertos constructores. Los más frecuentes son el constructor de tabla, el de registro, el de unión, el de producto cartesiano, el de puntero y el de rutina.

```
Tipo :: TBásico | TComplejo
```

En las secciones siguientes los estudiaremos con detalle, indicando cuál es la forma más adecuada de representarlos usando árboles con atributos.

12.2.1 Tipos Básicos

Los tipos básicos más comunes en los lenguajes de programación podemos clasificarlos en:

- Enumerables: booleano, entero, carácter, enumerado y subrango.
- No enumerables: real.
- Otros: tipo error y tipo vacío.

Diremos que un tipo es enumerable cuando es posible obtener todos sus valores a partir de uno concreto al que llamaremos cero del tipo [2]. Si analizamos con cierto detalle esta definición, nos daremos cuenta de que las capacidades de memoria de los ordenadores son limitadas y por lo tanto los conjuntos de valores de un cierto tipo deben ser necesariamente finitos. No obstante, el tipo `Real` se considera no enumerable por razones históricas puesto que los lenguajes tradicionales nunca han proporcionado una forma de enumerar todos sus valores a partir del cero de este tipo.

El tipo error no representa ningún valor concreto pero es absolutamente necesario para compilar cualquier lenguaje, puesto que es el tipo de todas aquellas construcciones que no cumplen con las reglas de nuestro sistema de tipos. Por ejemplo, si en nuestro lenguaje no se puede sumar números enteros y caracteres, entonces el tipo de una expresión como `1 + 'a'` será erróneo. El tipo error se suele considerar compatible con cualquier otro, puesto que de esta manera se pueden evitar con gran facilidad cascadas de errores. Por ejemplo, en la expresión `(((1 + 'a') - 2) * 3)`, existe un único error en el subtérmino `(1 +`

[2] No confunda el cero del tipo `Entero` con el valor entero 0. Un valor bastante habitual para el cero del tipo `Entero` suele ser -32.768.

'a') por lo que toda la expresión tendrá tipo error. Para evitar que este error provoque una cascada cuando el comprobador de tipos analice los otros dos operadores basta con hacer que este tipo sea admisible para cualquier operador del lenguaje.

El tipo vacío es un tipo especial del que tampoco existen valores concretos, pero no es indicativo de error. Es útil para representar el tipo de los procedimientos y el de las sentencias.

Otra clasificación habitual de los tipos básicos es en predefinidos y definidos por el usuario, atendiendo a si el lenguaje los proporciona automáticamente o no. Los tipos predefinidos más habituales son el booleano, el entero, el real y el carácter. Entre los definidos por el usuario destacaremos:

- Enumerado, en el que el usuario indica de forma explícita cuál es el conjunto de valores que son de ese tipo. Por ejemplo, en Pascal

  ```
  TYPE
    Color = (Blanco, Amarillo, Naranja, Rojo, Azul, Verde)
  ```

  introduce un nuevo tipo enumerado de nombre `Color` que cuenta con los valores `Blanco`, `Amarillo`, `Naranja`, `Rojo`, `Azul` y `Verde`. Generalmente los lenguajes suelen definir de forma automática algunos operadores que actúan sobre los valores de estos tipo. Por ejemplo, Pascal ofrece `pred` y `succ` para calcular el predecesor y el sucesor de un cierto valor enumerado.

- Subrango, cuyos valores son siempre un subconjunto de los valores que puede tomar algún tipo enumerado al que llamaremos padre del subrango. Por ejemplo, en Pascal

  ```
  TYPE
    Color_Claro = Blanco .. Naranja;
    Adolescente = 13 .. 19;
  ```

introduce dos tipos subrango. El primero es el subconjunto de valores del tipo enumerado `Color` que se encuentran entre el `Blanco` y el `Naranja`. El segundo es el subconjunto de valores del tipo predefinido `INTEGER` que se encuentran entre el 13 y el 19.

Especificación en HESA

El siguiente programa HESA muestra la definición de un árbol con atributos que nos permite representar todos estos tipos básicos:

```
    Entrada ent;
    Intervalo_Entero ie;
}



TBásico :: TEnumerable | TNo_Enumerable
TEnumerable :: TEPredefinido | TEDefinido_Usuario
```

```
TEPredefinido :: Booleano | Carácter | Entero | Real
TEDefinido_Usuario :: Enumerado | Subrango
Enumerado :: Valor*
Subrango :: padre: TEnumerable
TNo_enumerable :: Real
```

El tipo `Entrada` representa referencias a entradas de la tabla de símbolos e `Intervalo_Entero` representa intervalos de números enteros caracterizados por su límite inferior y superior. Este tipo viene dotado de las operaciones.

```
int li(Intervalo a);  /* Límite inferior */
int ls(Intervalo a);  /* Límite superior */
```

Tan sólo comentaremos un poco la forma en que representamos el tipo enumerado y el subrango. El primero, como hemos indicado, permite al usuario determinar cuáles son los valores concretos que pueden tomar las variables de ese tipo. Como estos se representan de forma simbólica, cada uno de ellos tendrá su propia entrada en la tabla de símbolos. De hay que este tipo se represente como una lista de nodos con referencias a las entradas que describen sus valores.

Los subrangos son siempre subconjuntos de valores de otro tipo enumerable. Por esta razón, todos esos valores se pueden caracterizar con ayuda de un número entero, con lo que el tipo subrango lo representaremos con un nodo que tiene el tipo enumerable padre y un atributo con el intervalo de valores que se permite dentro de él. Por ejemplo, si asociamos a los valores del tipo `Color` los enteros entre 0 y 5, entonces el subrango `Colores_Claros` estará formado por los valores del tipo `Color` cuyos índices se encuentran en el intervalo entero [0, 2].

12.2.2 Constructores de tipos

Los constructores de tipo son herramientas que proporcionan los lenguajes para definir nuevos tipos de datos basándonos en tipos básicos u otros tipos complejos definidos con anterioridad.

En esta sección se estudian los constructores de tipos más habituales en la práctica.

```
TComplejo
  :: Tabla | Registro | Unión | Producto | Puntero | Rutina
```

Tabla

El constructor tabla forma un nuevo tipo a partir de un tipo base y una lista que determina el conjunto de valores válidos para los índices en cada una de las dimensiones de la tabla [3]. Cada lenguaje sigue un criterio diferente a la hora de indicar cuáles son estos conjuntos.

[3] Consulte la sección 12.3.1 para un estudio más detallado de estos constructores en el que se muestra la forma idónea de modelarlos teniendo en cuenta las características de igualdad de tipos con las que cuenta el lenguaje.

Por ejemplo, en Modula se especifica usando tipos enumerables, de forma que cualquier valor concreto de los mismos constituye un valor correcto para los índices. Por ejemplo, la declaración

```
TYPE
  T = ARRAY Color [-20..20] CHAR (Normal, Extra) OF INTEGER
```

introduce un tipo tabla de cuatro dimensiones. Los índices de la primera deben ser valores del tipo enumerado `Color`, los de la segunda valores dentro del subrango `[-20..20]`, los de la tercera caracteres y los de la cuarta valores en el conjunto {`Normal, Extra`}. En otros lenguajes tan sólo se permiten índices de tipo entero y en muchos el límite inferior es predefinido.

Las tablas que hemos visto hasta ahora tienen como característica que su número total de componentes es fijo, esto es, se puede determinar en tiempo de compilación. Pero existen lenguajes como Ada, en los que los rangos de valores para los índices de cada dimensión son dinámicos y dependen de los valores que ciertas variables toman en tiempo de ejecución. Por ejemplo:

```
ARRAY[a..b] OF INTEGER
```

es un tipo tabla de una dimensión y el rango de valores permitido como índice de la misma es desconocido a priori puesto que depende de las variables `a` y `b`.

Teniendo en cuenta todas estas consideraciones, un buen diseño de árbol con atributos para representar los tipos tabla es el siguiente:

```
Tabla :: rangos: Lista_Rangos; tipo_base: Tipo
Lista_Rangos :: Rango*
Rango :: tipo_base: TEnumerable; lim_inf, lim_sup: Expresión
```

En donde `Expresión` representa cualquiera de las expresiones que se pueden construir en nuestro lenguaje. Si éste tan sólo permite tablas estáticas, el comprobador de tipos deberá asegurarse de que las expresiones que aparecen como límite inferior y superior sean en efecto valores constantes.

Registro y unión

Los constructores registro y unión forman un tipo a partir de una lista de campos. Cada uno de ellos es una tupla formada por un identificador y un tipo, por lo que esta información se puede guardar fácilmente en una tabla de símbolos.

Para representar estos tipos podemos usar la siguiente definición en HESA:

```
    Entrada ent;
}
```

```
Registro :: Campo *
Unión    :: Campo *
```

Desde el punto de vista estructural, registros y uniones tienen las mismas características. La diferencia es respecto a la forma en que los campos se organizan en la memoria del ordenador, puesto que en los registros cada campo cuenta con una o varias casillas reservadas para almacenar valores, mientras que en las uniones todos los campos comparten el mismo espacio de almacenamiento.

Producto cartesiano

Muchos de los lenguajes que permiten la definición de registros también incluyen la posibilidad de que el usuario escriba expresiones constructoras de tuplas compatibles respecto a la asignación o al paso de parámetros con estos tipos. Por ejemplo, dada la siguiente declaración en ADA

```
type Persona is record
       Edad: Integer;
       Sexo: (Hombre, Mujer);
     end Persona;
```

la expresión `(12, Hombre)` construye un valor cuyo tipo es compatible con `Persona`. Evidentemente, el tipo de esta expresión no es idéntico a `Persona` puesto que en ella no aparecen por ningún lado los nombres de los campos y tampoco hay indicación alguna de si éstos deben compartir o no el mismo espacio de almacenamiento. Estas expresiones se dice que tiene tipo `Producto`, que no es más que una lista de otros tipos.

```
Producto :: Tipo *
```

Puntero

Este constructor toma un tipo base cualquiera y produce uno nuevo que toma como valores el conjunto de direcciones de la máquina que pueden alojar valores de ese tipo base.

```
Puntero  :: tipo_base : Tipo
```

Rutinas

Una rutina, ya devuelva valores o no, puede verse como una "función" que asocia a cada elemento de su dominio un cierto valor dentro de su rango. Por eso, en principio, su tipo podemos representarlo así

```
Rutina :: dominio: Producto; rango: Tipo
```

El problema de esta especificación es que la mayoría de los lenguajes de programación permiten la posibilidad de que los parámetros que se pasan a las rutinas sufran efectos laterales. Cada parámetro se caracteriza además de por su tipo, por un modo de paso que determina el uso que dentro de la rutina se puede hacer de él. Los modos de paso más habituales son `Lectura` (el valor del parámetro tan sólo se puede consultar) y `Lectura/Escritura` (su valor se puede leer y también modificar). Teniendo esto en cuenta, podemos realizar la siguiente especificación en HESA:

```
Rutinas :: dominio: Producto; modos: Lista_Modos; rango: Tipo

Lista_Modos :: Modo*

Modo :: Lectura | Lectura_Escritura
```

De todas formas, en adelante seguiremos usando la primera definición que hicimos del tipo, puesto que como veremos en la sección 12.7.1 es mucho más general y fácil de adaptar a lenguajes diferentes que esta última.

Otros constructores en desuso

Muchos lenguajes tradicionales han proporcionado constructores de tipos de datos complejos que no hemos mencionado con anterioridad ya que en los lenguajes modernos tienden a estar en desuso.

Por ejemplo, Pascal proporciona el tipo `Conjunto de X`, LISP el tipo `Lista`, Modula el tipo `Fichero de X` y UCSD Pascal el tipo `Cadena de longitud n`. En la actualidad, los lenguajes modernos no incluyen este tipo de constructores puesto que suelen ofrecer herramientas suficientes para modelarlos como tipos abstractos de datos. Esto es muy conveniente porque de esta forma los compiladores resultan mucho más fáciles de construir y además ampliar el conjunto de tipos ofrecido por el lenguaje resulta sencillo sin necesidad de ampliar su definición y escribir nuevos compiladores.

La razón de por qué los lenguajes antiguos incluían estos constructores de tipos es que todos ellos son genéricos, esto es, no existe el tipo `Conjunto` como tal, sino los tipos concretos `Conjunto de Entero`, `Conjunto de 1..30`, etcétera. Los lenguajes antiguos no proporcionaban herramientas para modelar los tipos de datos genéricos y por lo tanto era necesario que el lenguaje los ofreciese de forma predefinida. Nosotros estudiaremos la forma de tratarlos en la sección 12.6.

12.2.3 Nombres de tipo

Muchos lenguajes ofrecen al usuario la posibilidad de dar nombres a los tipos que define. En estos casos puede ser útil incluir en la especificación del sistema de tipos esta posibilidad.

```
    Entrada ent;
}
```

12.3 Igualdad de tipos

Un aspecto importante con respecto a la especificación de cualquier sistema de tipos es determinar con claridad cuándo dos tipos se consideran iguales y cuándo se consideran compatibles.

En cuanto a la igualdad de tipos existen dos posibilidades fundamentales: que en nuestro lenguaje la igualdad sea de tipo estructural o por nombre. En el primer caso, dos tipos son iguales si tienen la misma estructura, en el segundo tan sólo cuando se les ha dado el mismo nombre.

Sean por ejemplo las siguientes declaraciones:

```
T0 = Entero;
T1 = Registro
       a, b, c: Entero;
     Fin registro;
T2 = Registro
       x: Entero;
       y, z: T0;
     Fin registro;
```

En un lenguaje con igualdad estructural de tipos, `T1` y `T2` se consideran equivalentes, puesto que exceptuando los detalles sintácticos, ambos tipos son dos registros con tres campos de tipo entero. En cambio, en un lenguaje con igualdad por nombre, dos tipos son iguales si y sólo si se les ha dado el mismo nombre, por lo que `T0` no sería equivalente a `Entero` y por lo tanto `T1` no sería equivalente a `T2`.

12.3.1 Igualdad de tipos y tablas

Los lenguajes con igualdad de tipos por nombre no permiten construcciones a las cuales no se les pueda asignar un tipo con nombre. Por ejemplo, dada la siguiente declaración:

```
Tipos
    T = Tabla[1..10][2..20] de Entero;
Variables
    x: T;
```

La expresión de indexación `x[1]` sería incorrecta puesto que el tipo de la misma sería `Tabla[2..20] de Entero`, pero el usuario no ha definido ningún nombre para este tipo.

En la sección 12.2.2 mostramos la definición de un árbol con atributos capaz de representar los tipos tabla que es muy adecuada para lenguajes con igualdad de tipos por

nombre. En cambio, en aquellos lenguajes en los que la igualdad es estructural suele ser más conveniente representar el tipo tabla de una forma alternativa, considerando que tan sólo es posible definir tablas de una dimensión. De esta forma una tabla de dos dimensiones como la anterior se representaría en estos lenguajes de forma equivalente como:

```
T = Tabla[1..10] de Tabla[2..20] de Entero;
```

Esta simplificación [4] está en consonancia con la idea de igualdad estructural y además simplifica considerablemente la comprobación de tipos puesto que todas las tablas tienen el mismo número de dimensiones y se evita la posibilidad de que el usuario pueda escribir expresiones parcialmente indexadas cuyos tipos pueden resultar difíciles de calcular. Esta simplificación, como veremos más adelante, también permite generar código para el acceso a las tablas de una forma bastante sencilla y en absoluto hace imposible aplicar algunas optimizaciones bastante comunes en este campo.

12.3.2 Algunos comentarios respecto a la implementación de la igualdad de tipos

Para implementar la igualdad de tipos podemos una función con el interface siguiente:

```
Boolean Igual_Tipo(Tipo t1, Tipo t2);
```

Si la igualdad de tipos es por nombre, esta función se limitará a determinar si `t1` y `t2` tienen o no el mismo nombre. En el caso de igualdad estructural, deberá comprobar la estructura de los árboles, pero teniendo en cuenta que los registros y las tablas necesitarán de un tratamiento especial.

Al comparar los registros hay que tener en cuenta que los nodos que representan sus campos no contienen más que una referencia a la entrada de la tabla de símbolos que los describe. Por lo tanto, para comparar los campos deberán compararse los tipos guardados para los mismos en sus entradas de la tabla de símbolos.

En el caso de las tablas también hay que tener en cuenta que la mayoría de los lenguajes no exige que los rangos para los índices de sus diversas dimensiones sean exactamente los mismos. Lo habitual es que para que dos tablas sean iguales tan sólo se exija que tengan tipos base equivalentes, el mismo número de dimensiones y el mismo número de componentes en cada una, pero con independencia de los valores concretos que puedan tomar los índices. Por ejemplo, las declaraciones

```
T1 = Tabla[1..10] de Entero;
T2 = Tabla[10..20] de Entero;
```

suelen considerarse estructuralmente equivalentes puesto que los tipos bases son equivalentes y ambas tienen una dimensión con 10 componentes.

[4] Sintácticamente, el programador podría seguir escribiendo `Tabla[1..10][2..20] de Entero`. Lo único que ocurre es que internamente el compilador representará este tipo de la forma simplificada que hemos indicado.

12.4 Compatibilidad de tipos

La compatibilidad de tipos está en relación a que ciertas construcciones de un lenguaje exigen que algunas de las subconstrucciones que aparezcan en ella sean de un tipo específico. Un conjunto de tipos se dice que es compatible en el contexto de una cierta construcción si es válido desde el punto de vista del sistema de tipos del lenguaje en ese contexto.

Por ejemplo, los tipos `Entero` y `Real` suelen ser compatibles con respecto al operador `+` en la mayoría de lenguajes. En otros, en cambio, estos dos tipos no son compatibles con respecto a ese operador.

12.4.1 Implementación de la compatibilidad de tipos

Para implementar estos aspectos del sistema de tipos se pueden las siguientes funciones:

```
Tipo Result_bin(Tipo t1, Tipo t2, Operador_Binario ob);
Tipo Result_un(Tipo t, Operador_Unario ou);
Bool Compatible_bin(Tipo t1, Tipo t2, Operador_Binario ob);
Bool Compatible_un(Tipo t, Operador_Unario ou);
```

Las dos primeras determinan si el operador que se les pasa puede actuar o no sobre los tipos de datos que se le indica. En caso de poder hacerlo devuelven el tipo resultado de aplicar el operador. En caso contrario devuelven `TError`. Las dos funciones siguientes se basan en las anteriores y devuelven un valor booleano que nos dice si los operandos son compatibles con el operador indicado. Generalmente las definiremos como:

```
Bool Compatible_bin(Tipo t1, Tipo t2, Operador_Binario ob)
{
  return !Igual(Result_bin(t1, t2, ob), TError())
}

Bool Compatible_un(Tipo t, Operador_Unario ou)
{
  return !Igual(Result_un(t, ou), TError())
}
```

12.4.2 Un ejemplo sencillo

En esta sección mostraremos un ejemplo que ilustrará la forma en que se pueden implementar estas funciones para un lenguaje tipo que incluye operadores aritméticos, lógicos y la instrucción de asignación (que desde este punto de vista se puede considerar como un operador especial).

Consideraremos que los valores de tipo `Booleano` sólo pueden operarse con operadores lógicos y que los de tipo `Entero` o `Real` se pueden operar con operadores de tipo aritmético y relacionales. Con respecto a la asignación, supondremos que los valores de tipo `Entero` pueden asignarse sobre valores de tipo `Real` pero no al revés.

```
Tipo Result_bin(Tipo t1, Tipo t2, Opb ob)
{
  Tipo Result;

  según(t1, t2, ob)
  {
    Booleano, Booleano, op => Es_op_lógico(op): { Result = t1; }

    Entero, Entero, op => Es_op_arit_o_rel(op): { Result = t1; }
    Entero, Real, op => Es_op_arit_o_rel(op):   { Result = t2; }
    Real, Entero, op => Es_op_arit_o_rel(op):   { Result = t1; }
    Real, Real, op => Es_op_arit_o_rel(op):     { Result = t1; }

    Real, Entero, Asig:                { Result = TVacío(); }
    _, _, Asig => Igual_tipo(t1, t2): { Result = TVacío(); }

    _, _, _: { Result = TError(); }
  }

  return Result;
}
```

Fíjese en que en algunos casos el tipo resultado es `TVacío`, indicando que la construcción es correcta, pero no devuelve ningún valor. No debe confundirse con el caso en el que el resultado es de tipo `TError` que indica que en la construcción existe algún error de tipos.

Esta función es bastante compacta y fácil de entender, pero tiene como inconveniente que es demasiado específica, puesto que hay que adaptarla a cada lenguaje concreto y difícil de ampliar en el caso de añadir nuevos operadores o alterar las reglas del sistema de tipos de nuestro lenguaje. En la próxima sección presentaremos una solución bastante más general al problema que se basa en el concepto de sobrecarga de operadores.

12.5 Sobrecarga de operadores y rutinas

Un símbolo sobrecargado es aquél que tiene diferentes interpretaciones dependiendo del contexto en el que se está haciendo uso de él. Por ejemplo, en las matemáticas el símbolo – está sobrecargado puesto que se utiliza tanto para negar, como para restar números enteros, complejos o matrices. Los paréntesis matemáticos también son símbolos sobrecargados puesto que se utilizan para agrupar los parámetros sobre los que se aplica una función o para construir matrices y vectores.

Es bastante frecuente que todos los lenguajes de programación ofrezcan operadores sobrecargados (fundamentalmente los aritméticos y los paréntesis), aunque tan sólo los más modernos permiten al usuario definir libremente sus propios operadores y rutinas sobrecargados.

Realmente, se trata de operadores y rutinas diferentes, cada una con su propio código (no es lo mismo, por ejemplo, sumar dos enteros que sumar dos reales, puesto que las

representaciones como hileras de bits son muy diferentes) y es tarea del comprobador de tipos del lenguaje determinar en cada caso cuál es la rutina u operador real que el usuario desea invocar cada vez que lo escribe en una expresión. Por ejemplo, si el operador `-` admite las sobrecargas

```
-:  Entero  ×  Entero  →  Entero
-:             Entero  →  Entero
```

entonces, dada una expresión como `-(i - j)`, debe ser capaz de determinar cuál de las dos sobrecargas es la que estamos utilizando en cada una de la apariciones del símbolo `-`. En este caso es fácil puesto que la aridad de las dos sobrecargas es diferente, pero en otros es bastante más complicado. Dedicaremos el resto de la sección a estudiar estos problemas. Para simplificar la exposición consideraremos que todos los operadores y rutinas sobrecargados toman parámetros con modo de paso `Lectura`. En la sección 12.7.1 mostraremos una forma bastante sencilla de salvar esta restricción.

12.5.1 Posibles tipos de una subexpresión

Cuando tratamos con operadores y rutinas sobrecargadas, es frecuente que la misma subexpresión pueda tener diversos tipos en función a cómo interpretemos los símbolos que aparecen en ella. Por ejemplo, consideremos un lenguaje de programación en el que el operador `+` cuenta con las sobrecargas:

```
+:  Entero  ×  Entero  →  Entero
+:  Entero  ×  Entero  →  Real
+:   Real   ×   Real   →  Real
```

A cada una de estas sobrecargas nos referiremos como $+_1$, $+_2$ y $+_3$ respectivamente. Supongamos que los literales `1` y `2` son de tipo `Entero` y que el literal `3.4` es de tipo `Real`. Si en el fuente de un programa nos encontramos con la subexpresión `(1 + 2)`, el comprobador de tipos deberá determinar que puede ser tanto de tipo `Entero` como `Real`, puesto que el signo `+` que aparece se puede interpretar como $+_1$ o como $+_2$. Si esta expresión apareciese aislada (en una sentencia de escritura, por ejemplo) el comprobador de tipos debería informarnos con un error de que la expresión puede interpretarse de varias formas y por lo tanto resulta ambigua. Hay que tener en cuenta que el comprobador de tipos debe pasar al generador de código un árbol en el que cada construcción del lenguaje tiene un único tipo, de forma que éste sepa qué código generar en cada caso.

Cuando la subexpresión anterior aparece dentro de un contexto en el que su tipo se puede determinar de forma unívoca, el comprobador de tipos depurará el conjunto de tipos calculado inicialmente. Por ejemplo, si aparece en el contexto de la expresión `(1 + 2) + 3.4`, resulta evidente que en la subexpresión `(1 + 2)` el signo `+` debe interpretarse como la sobrecarga $+_2$ y el otro signo como la sobrecarga $+_3$

En las secciones anteriores hemos considerado a los operadores proporcionados por el lenguaje como unos elementos diferentes a las rutinas definidas por el usuario, pero como se desprende de lo que hemos estado estudiando anteriormente, la única diferencia es que los primeros son predefinidos, es decir, su interface es conocido por el compilador, y los segundos deben ser definidos completamente por el usuario. Por lo tanto, podemos

considerar que los operadores no son más que un caso particular de rutina. En el resto de esta sección trataremos a los operadores predefinidos como si fuesen rutinas normales que se representan en formato prefijo y que por lo tanto se describen usando entradas de la tabla de símbolos que el compilador define de forma automática durante su proceso de inicialización.

La siguiente gramática con atributos describe la forma en que se calcula de forma ascendente el conjunto de tipos posible para una expresión:

Categorías Sintácticas

e	$\in$	Expresión
v	$\in$	Variable
le	$\in$	Literal_Entero
r	$\in$	Rutina
p	$\in$	Parámetros

Atributos

<Conj_Tipos tipos>	Expresión
<Entrada ent>	Variable Rutina

Definiciones

$$e' : e \quad \{e'.tipos = e.tipos\}$$

$$\begin{array}{lll} e : v & \{e.tipos = \{v.ent.tipo\}\} \\ \mid le & \{e.tipos = \{Entero()\}\} \\ \mid r(p) & \{e.tipos = \{t \mid \exists s \in p.tipos \cdot Rutina(s,t) \in r.ent.sobs\}\} \\ \mid r() & \{e.tipos = \{Rutina(Producto_v(), t) \mid t \in r.ent.sobs\}\} \end{array}$$

$$p_1 : e\ p_2 \quad \left\{ p_1.tipos = \bigcup_{\substack{s \in e.tipos \\ t \in p_2.tipos}} Producto(s,t) \right\}$$

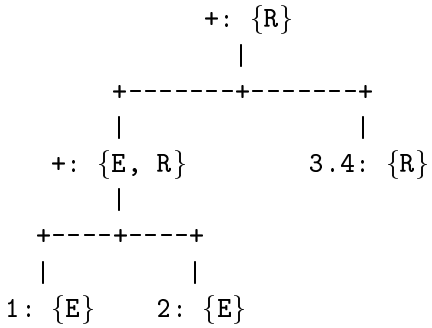
$$\mid e \quad \{p_1.tipos = \{Producto1(s) \mid s \in e.tipos\}\}$$

Como hemos dicho, los operadores se tratarán como rutinas normales desde el punto de vista de la comprobación de tipos, y sus diferentes sobrecargas se guardarán como diversos tipos permitidos en las entradas de la tabla de símbolos que los describen (atributo `sobs` de cada entrada).

La primera regla indica que el conjunto de tipos de una variable tiene cardinalidad uno y está formado por el tipo que se guarda en su correspondiente entrada de la tabla de símbolos. La segunda regla indica que el único tipo de un literal entero es `Entero` y la tercera merece que nos detengamos con un poco más de detalle. Esta regla formaliza la aplicación de un símbolo de rutina sobre una lista de parámetros, e informalmente significa que si s es uno de los tipos de p y una de las sobrecargas de r es $Rutina(s,t)$, entonces t es uno de los posibles tipos de la expresión $r(p)$. En el caso de que no exista

ninguna sobrecarga aplicable, el conjunto de tipos para esta expresión será vacío [5], lo que nos servirá temporalmente para indicar un error de tipos.

En la siguiente figura se ilustra este proceso ascendente de cálculo de tipos posibles aplicado a la expresión `(1 + 2) + 3.4`. Al lado de cada nodo del árbol se indica el conjunto de tipos posibles. Para abreviar, supondremos que `E` representa el tipo `Entero` y `R` el tipo `Real`.



Si suponemos las sobrecargas que vimos anteriormente, en la expresión más interna el operador + se aplica sobre el dominio `Entero` × `Entero`, por lo que son posibles las sobrecargas $+_1$ y $+_2$. En el primer caso, el resultado de la función sería `Entero` y en el segundo `Real`. Cuando procesamos el nodo raíz, se observa que el operador + está actuado sobre valores en los dominios `Entero` × `Real` y `Real` × `Real`. Por lo tanto, la única sobrecarga válida en estas condiciones es $+_3$ con lo que el tipo de la expresión completa será `Real`.

12.5.2 Reducción del conjunto de tipos posibles

Con lo que hemos estudiado en la sección anterior nos es posible determinar el conjunto de tipos posibles para una expresión. En el caso de que éste sea único, deberemos refinar el tipo de cada una de las subexpresiones que intervienen para que éste también sea único y el generador de código sepa para qué sobrecarga generar código en cada caso.

La reducción del conjunto de tipos posibles plantea nuevos problemas. Por ejemplo, supongamos que en la expresión `r(x)`, `r` es una rutina que admite las sobrecargas `a` → `c` y `b` → `c` y que `x` es una expresión de tipo `a` o `b`. Resulta evidente que la expresión completa es de tipo `c`, pero ahora es preciso interpretar todos los símbolos que aparezcan en `x` de forma que esta subexpresión tan sólo tenga un único tipo y sea posible elegir la sobrecarga adecuada. Si `x` es un símbolo de rutina sin parámetros sobrecargado entonces no será posible resolver la sobrecarga y habrá que informar de ello con un mensaje de error.

La reducción de tipos se lleva a cabo mediante el recorrido que se muestra en la siguiente gramática con atributos.

[5] Asegúrese de que no lo confunde con el tipo TVacío.

Categorías Sintácticas

e	$\in$	Expresión
v	$\in$	Variable
le	$\in$	Literal_Entero
r	$\in$	Rutina
p	$\in$	Parámetros

Atributos

<Conj_Tipos tipos>	Expresión
<Entrada ent>	Variable Rutina
<Tipo tipo>	Expresión

Definiciones

$$e' : e \quad \{e.tipo = if\ e.tipos = \{t\}\ then\ t\ else\ TError()\}$$

$$\begin{array}{lcll} e & : & v & \\ & | & le & \\ & | & r(p) & \left\{ \begin{array}{l} t = e.tipo \\ S = \{s \mid s \in p.tipos \wedge Rutina(s,t) \in r.ent.sobs\} \\ p.tipo = if\ S = \{r\}\ then\ r\ else\ TError() \end{array} \right\} \\ & | & r() & \left\{ \begin{array}{l} t = e.tipo \\ if\ \exists Rutina(Producto_v(), t) \in r.ent.sobs\ then \\ \quad e.tipo = t \\ else \\ \quad e.tipo = TError() \end{array} \right\} \end{array}$$

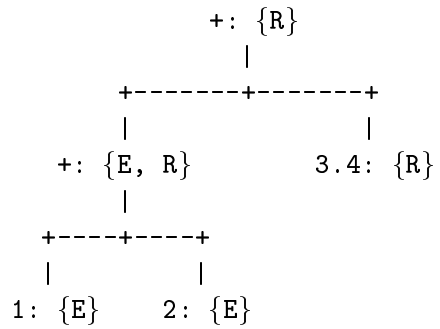
$$\begin{array}{lcll} p_1 & : & e\ p_2 & \left\{ \begin{array}{l} if\ Producto(t,r) = p_1.tipo\ then \\ \quad e.tipo = t \\ \quad p_2.tipo = r \\ else \\ \quad e.tipo = p_2.tipo = TError() \end{array} \right\} \\ & | & e & \left\{ \begin{array}{l} if\ Producto1(t) = p_1.tipo\ then \\ \quad e.tipo = t \\ else \\ \quad e.tipo = TError() \end{array} \right\} \end{array}$$

El atributo `tipo` determina el tipo único de cada expresión o `TError()` en caso de que sea incorrecta. Es heredado y por lo tanto debe calcularse usando un recorrido de la raíz a las hojas del árbol con atributos, una vez que se haya sintetizado el conjunto de tipos posibles para cada una de las expresiones.

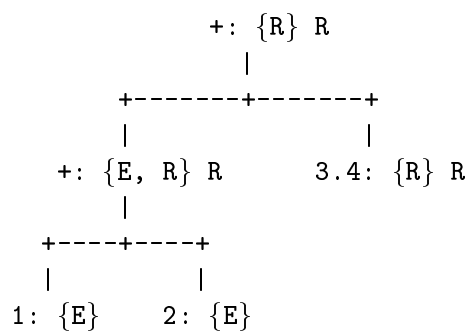
Todas las expresiones son generadas en última instancia por `e'`, por lo que si `e'.tipos` no es un conjunto de cardinalidad uno, entonces no habrá sido posible interpretar la expresión de forma única y se producirá un error de tipos. Dentro de `e`, las únicas reglas que necesitan de un tratamiento son la tercera y la cuarta. En ellas se indica que si el resultado de llamar a una rutina `r(p)` es de tipo `t` entonces debe haber una única sobrecarga de la forma $Rutina(s,t)$ para `r`. En otro caso se producirá un error de tipos.

Una vez determinada la sobrecarga correspondiente, el tipo de cada uno de los parámetros sobre los que se aplica la función se puede determinar recursivamente.

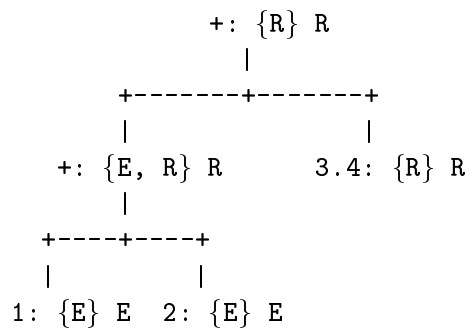
Como ejemplo de aplicación del algoritmo, podemos tomar la misma expresión de antes, cuyo árbol atribuido después de calcular el conjunto de tipos posibles para cada expresión es el siguiente:



El tipo del nodo raíz es `Real`, por lo que a priori son posibles dos sobrecargas: `+: Real` $\times$ `Real` $\rightarrow$ `Real` y `+: Entero` $\times$ `Entero` $\rightarrow$ `Real`. Esto significa que la tupla de parámetros sobre la que se aplica este operador debe ser de tipo `Real` $\times$ `Real` o bien `Entero` $\times$ `Entero`. Teniendo en cuenta los conjuntos de tipos posibles para los argumentos que calculamos previamente, descubrimos que la única sobrecarga posible es `+: Real` $\times$ `Real` $\rightarrow$ `Real`. Después de realizar este cálculo, el árbol queda así:



Una vez hecho esto, tenemos que refinar los tipos en la subexpresión `1 + 2`. Dado que el tipo de esta subexpresión es `Real`, se presentan de nuevo las dos mismas alternativas de antes. En este caso, dado los tipos de los parámetros, la única posible es `+: Entero` $\times$ `Entero` $\rightarrow$ `Real`, por lo que el árbol final queda así:



12.5.3 Unos ejemplos más complicados

En esta sección ilustraremos los algoritmos para refinar el tipo de una expresión en presencia de operadores sobrecargados con ejemplos más complicados.

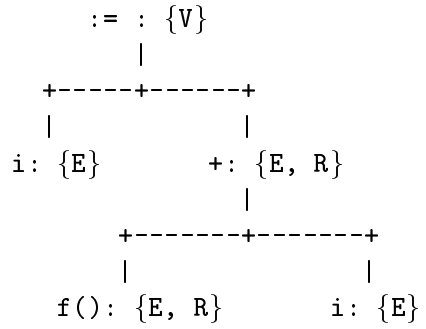
Supongamos que tenemos los operadores y sobrecargas que se muestran a continuación.

```
:=:  Entero  ×  Entero  →  TVacio
:=:   Real   ×  Entero  →  TVacio
:=:   Real   ×   Real   →  TVacio
 +:  Entero  ×  Entero  →  Entero
 +:  Entero  ×   Real   →  Real
 +:   Real   ×  Entero  →  Real
 +:   Real   ×   Real   →  Real
 f:                     →  Entero
 f:                     →  Real
```

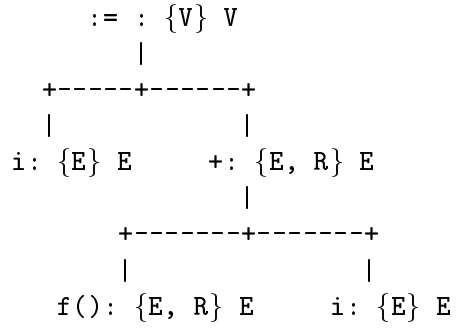
Además supondremos que la variable `i` es de tipo `Entero` y que la variable `x` es de tipo `Real`.

Ejemplo: `i := f() + i`

En el calculo de los conjuntos de tipos posibles se obtiene el siguiente árbol:

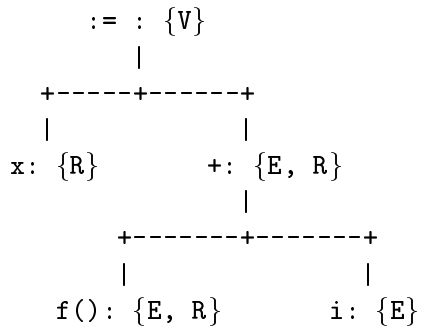


Una vez calculados estos conjuntos, el refinamiento de tipos nos permite determinar unívocamente el tipo de cada subexpresión:



Ejemplo: `x := f() + i`

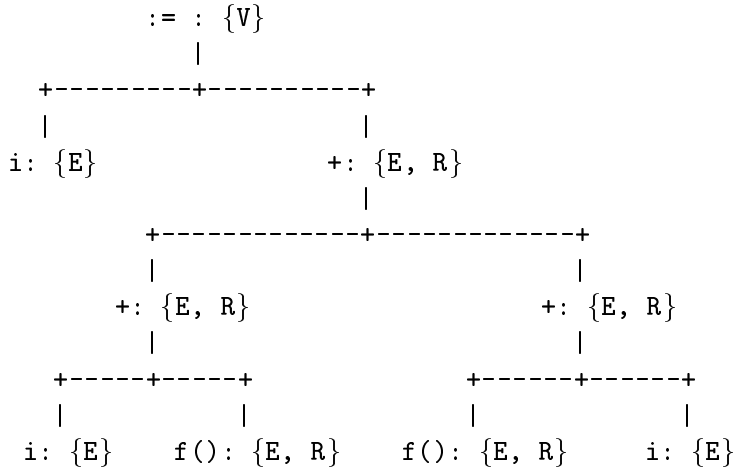
En el calculo de los conjuntos de tipos posibles se obtiene el siguiente árbol:



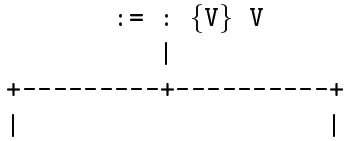
Fíjese en que el nodo raíz tan sólo tiene un tipo, lo que era una de las condiciones necesarias para que toda la expresión fuese correcta, aunque no suficiente. Como veremos a continuación, el proceso de refinamiento de tipos no puede inferir la sobrecarga del operador `+` a la que nos estamos refiriendo, puesto que con el tipo `TVacío` para la raíz del árbol, son posibles dos sobrecargas del operador `:=`: `Real` $\times$ `Entero` $\rightarrow$ `TVacío` y `Real` $\times$ `Real` $\rightarrow$ `TVacío`. Por lo tanto, esta expresión debería ser rechazada por el comprobador de tipos.

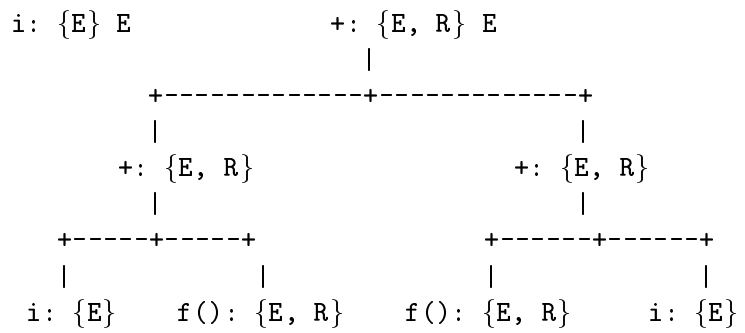
Ejemplo: `i := (i + f()) + (f() + i)`

En este caso, el primer recorrido del árbol nos ofrece el siguiente resultado:

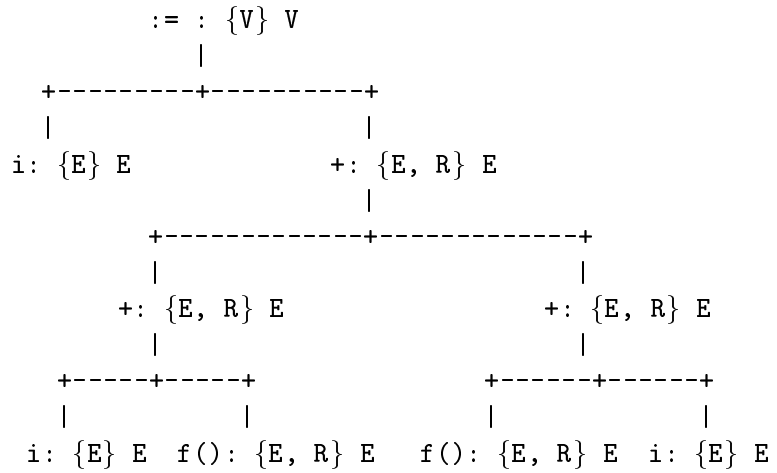


La única sobrecarga posible para el operador del nodo raíz es `Entero` $\times$ `Entero` $\rightarrow$ `TVacío`, por lo que el operando derecho debe ser necesariamente de tipo `Entero`.





De forma similar, dado que la expresión fuente de la asignación debe ser de tipo `Entero`, la única sobrecarga aplicable para el operador de suma es `Entero` $\times$ `Entero` $\rightarrow$ `Entero`, con lo que el árbol final queda como se muestra a continuación:



12.6 Polimorfismo

En muchas ocasiones el concepto de polimorfismo se suele confundir con el de sobrecarga de operadores y rutinas, pero no debemos equivocarnos, entre otras cosas porque el polimorfismo es un calificativo que se aplica a los tipos de datos [6], mientras que el de sobrecarga tan sólo es aplicable a operadores y rutinas. Se dice que un tipo de datos `T(`$\mathtt{X}_1$`, ..., `$\mathtt{X}_n$`)` es polimorfo o genérico porque representa un conjunto potencialmente infinito de tipos de datos concretos o polimorfos que se obtienen al instanciar algunas de las variables $\mathtt{X}_i$ que aparecen en él. Por ejemplo, `T = Tabla(Rango(1, 10), B)` es un tipo de dato polimorfo que representa el conjunto de todas las tablas de una dimensión con diez componentes de tipo `B`. En este caso, `B` es una variable de tipo y al instanciarla se obtienen valores concretos del tipo `T`. Por ejemplo, si sustituimos `B` por `Entero`, obtenemos una tabla de enteros, y si la sustituimos por `Puntero(S)`, obtenemos una tabla de punteros al tipo de datos `S`, que queda sin especificar.

La confusión en los conceptos de polimorfismo y sobrecarga suele darse porque es muy frecuente que los lenguajes ofrezcan algunos operadores predefinidos cuya signatura está

[6] Algunos autores llaman *genericidad* al polimorfismo, y en vez de hablar de tipos de datos polimorfos lo hacen de tipos de datos genéricos. Otros autores incluso hablan de tipos con variables para referirse al mismo concepto.

definida sobre tipos de datos genéricos. Por ejemplo, en el lenguaje C, el operador `_[_]` tiene el perfil `Puntero(T)` $\times$ `Entero` $\rightarrow$ `T`, y se dice de él que es un operador polimorfo al actuar sobre tipos de datos polimorfos. La diferencia fundamental está en que en el caso de la sobrecarga, el mismo símbolo de función puede aplicarse sobre valores de distintos tipos, pero el código que se ejecuta sobre esos valores es diferente en cada caso. En cambio, las rutinas polimorfas siempre ejecutan el mismo código (que está basado en las características comunes de los tipos de datos sobre los que actua la rutina). Por ejemplo, en el caso del operador `_[_]`, todos los datos sobre los que se puede aplicar son punteros.

Por supuesto, de lo que hemos dicho en el párrafo anterior no debe en absoluto desprenderse que los conceptos de sobrecarga y polimorfismo de datos sean incompatibles entre sí. De hecho veremos al final de este capítulo que un comprobador de tipos que tenga en cuenta sobrecarga y polimorfismo será adaptable a multitud de lenguajes diferentes sin modificación alguna, lo que supone unas enormes ventajas desde el punto de vista de la reutilización y robustez del *software* construido, facilidad de ampliación, etcétera.

En esta sección estudiaremos algunos de los problemas más comunes a la hora de compilar un lenguaje que permita tipos de datos polimorfos.

12.6.1 Igualdad de tipos polimorfos

Decidir si dos tipos sin variables son iguales o no es sencillo, puesto que para ello basta con comprobar si los árboles que los representan son estructuralmente iguales o no. La representación de los registros y las tablas son una excepción, pero no plantea problemas demasiado difíciles de resolver. En cambio, decidir si dos tipos con variables son iguales es más complejo, puesto que cada variable representa un conjunto quizá infinito de tipos y no basta con comprobar si los árboles son estructuralmente iguales, sino que en caso de no serlo es necesario determinar si existe algún valor concreto de las variables de tipo que los hace estructuralmente iguales.

Por ejemplo, los árboles de tipo `Puntero(T)` y `Puntero(Puntero(R))` no tienen la misma estructura. Cuando esto ocurre es necesario encontrar si existe algún subconjunto de los tipos que representan las variables que aparecen en los dos árboles, de forma que para esos subconjuntos los dos árboles sí que sean iguales estructuralmente. En este caso, si restringimos `T` al conjunto de tipos de la forma `Puntero(R)` y no restringimos `R`, resulta que los dos árboles son iguales.

Llamaremos sustitución a un conjunto de asignaciones a variables de tipo que denotaremos como $\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$. Las sustituciones las denotaremos como $\sigma, \delta, \rho, \ldots$ y se pueden aplicar sobre expresiones de tipo usando la nomenclatura $\sigma(t)$. El resultado de aplicar una sustitución σ a una expresión de tipo t es una nueva expresión de tipo en la que cada una de las variables que aparece se ha sustituido por el tipos correspondiente en la sustitución. Por ejemplo,

$$\{T \mapsto Puntero(R)\}(Puntero(T)) = Puntero(Puntero(R))$$

El problema de determinar si dos tipos t y s son iguales es equivalente al problema de encontrar una sustitución σ tal que $\sigma(t)$ sea estructuralmente igual a $\sigma(s)$. Para resolver este problema se utiliza el llamado algoritmo de unificación, una de cuyas versiones más sencillas es la que se muestra a continuación:

```
Unifica(Tipo t1, Tipo t2) dev (unificable: bool, σ: sust)
{
  k: int
  τ: sust

  si t1 o t2 es una variable de tipo, entonces
    sea x la variable y sea t la otra expresión de tipo
    si x = t, entonces
      (unificable, σ) := (true, ∅)
    si no, si x aparece en t, entonces
      (unificable, σ) := (false, ∅)
    si no
      (unificable, σ) := (true, {x ↦ t})
    fin si
  si no
    sea t1 = f(X_1, ..., X_n) y
        t2 = g(Y_1, ..., Y_m) (n=0, 1, ...)
    si f != g ó m != n, entonces
      (unificable, σ) := (false, ∅)
    si no
      (k, unificable, σ) := (0, true, ∅)
      mientras k < n y unificable
        k := k + 1
        (unificable, τ) := Unifica(σ(X_k), σ(Y_k))
        si unificable, entonces
          σ := τ ∘ σ
        fin si
      fin mientras
    fin si
  fin si
}
```

La función `Unifica` toma dos expresiones de tipo `t1` y `t2` como argumentos y devuelve una tupla con la componente `Unificable`, que es cierta si y sólo si las dos expresiones de tipo son unificables, y σ, la sustitución que hace las dos expresiones iguales.

Como ejemplo podemos ver de forma intuitiva el funcionamiento del algoritmo sobre las expresiones de tipo `t1 = Puntero(Tabla(R, Puntero(X)))` y `t2 = Puntero(Tabla(Rango(1..10)` `Puntero(Puntero(W))))`. Dado que `t1` y `t2` son dos constructores del mismo tipo y con la misma aridad, entonces el algoritmo intenta determinar si los parámetros del constructor son o no unificables. Esto es, tenemos que examinar `t11 = Tabla(R, Puntero(X))` y `t2 = Tabla(Rango(1..10), Puntero(Puntero(W)))`. Puesto que vuelven a ser dos tipos basados en el mismo constructor, el problema se reduce en este caso a unificar `R` y `Rango(1..0)` por una parte y `Puntero(X)` y `Puntero(Puntero(W))` por otra. El primer caso es trivial, puesto que se trata de unificar una variable con un constructor en el que no aparece esa variable, por lo que el unificador es $\sigma = \{R \mapsto \mathit{Rango}(1..10)\}$. En el segundo caso, ambos tipos son punteros, por lo que tenemos que unificar sus parámetros. En este caso `X` y `Puntero(W)`, que unifican trivialmente con el unificador $\rho = \{X \mapsto \mathit{Puntero}(W)\}$.

Por lo tanto, el unificador de los tipos `t1` y `t2` es

$$\sigma \circ \rho = R \mapsto Rango(1..10), X \mapsto Puntero(W)$$

Los dos tipos son iguales si sustituimos las variables de esta sustitución por los valores indicados. En ellos quedan aún variables de tipo que podremos sustituir por cualquiera de los tipos del lenguaje.

12.6.2 Inferencia de Tipos

La inferencia de tipos es el problema de determinar el tipo de una construcción de un lenguaje a partir del modo en que se usa. Este problema aparece en el momento en que un lenguaje permite utilizar identificadores que no han sido declarados previamente. Un ejemplo de ellos es C. Lo que se muestra a continuación es un programa válido en ANSI-C:

```
void main(void)
{
    int h;

    scanf("%d", &h);
    if (h == 1)
      printf("Es la una");
    else
      printf("Son las %d horas", h);
}
```

Dado que las rutinas `scanf` y `printf` son usadas, pero no han sido declaradas previamente, el compilador de C debe deducir su tipo a partir del uso que se ha hecho de ellas. A partir del primer uso de `scanf` se deduce que

`scanf: char * × int * → int`

Realmente, este no es el *interface* correcto de esta rutina, pero el compilador de C es el único que puede deducir a partir de su uso.

De la primera llamada a `printf` se deduce que

`printf: char * → int`

Por lo tanto, cuando se analiza la segunda llamada a `printf` el compilador mostrará un mensaje de error indicando que el segundo parámetro sobra si nos atenemos al primer uso que se ha realizado de la rutina.

Pero la inferencia de tipos también es de gran utilidad en aquellos lenguajes que exigen que todos los identificadores sean previamente declarados. En este caso el compilador, cada vez que se encuentra con un identificador no declarado, conjetura su tipo a partir del contexto en el que se usa por primera vez y en muchas ocasiones se pueden evitar cascadas de errores provocadas por el uso habitual de un identificador no declarado.

También es de utilidad en la herramienta PM que utilizamos en la asignatura para especificar recorridos de árboles con atributos. Por ejemplo, en el trozo de código:

```
int evalúa(Árbol a)
{
  int Result;

  según(a)
  {
    Constante[v]:           { Result = v); }
    Variable[e]:            { Result = valor(e); }
    Binaria(ti, td, Suma):  { Result = evalúa(ti) + evalúa(td); }
    Binaria(ti, td, Resta): { Result = evalúa(ti) - evalúa(td); }
    Binaria(ti, td, Mult):  { Result = evalúa(ti) * evalúa(td); }
    Binaria(ti, td, Div):   { Result = evalúa(ti) / evalúa(td); }
  }

  return Result;
}
```

PM debe deducir el tipo `v`, `e` `ti` y `td` automáticamente. Este problema se puede resolver adecuadamente usando el algoritmo de unificación de tipos polimorfos que hemos estudiado anteriormente.

12.7 Valores–l y valores–r

En todos los lenguajes de tipo imperativo (aquellos que incluyen instrucciones de asignación con efectos colaterales) hay una diferencia explícita entre el uso de un identificador en el lado izquierdo de una asignación y en el lado derecho.

Por ejemplo, en cada una de las asignaciones

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

aparece el identificador `i`, pero cuando aparece al lado derecho del signo `:=`, lo que nos interesa de este identificador es el valor que almacena, mientras que cuando aparece en el lado izquierdo lo que nos interesa es la dirección de memoria que tiene asignada para poder almacenar en ella el valor de la parte derecha del signo `:=`.

De una forma similar, si `p` y `q` son variables de tipo `Puntero(T)`, entonces en la instrucción en lenguaje C

```
*p = *q
```

de `*p` lo que nos interesa es la dirección de memoria que representa, mientras que de `*q` lo que nos interesa es el valor que contiene esa dirección.

En la literatura técnica la dirección de un valor se suele llamar *l–value* o valor–l, y el contenido de esa posición de memoria *r–value* o valor–r. Generalmente, todos los lenguajes ofrecen diversos operadores que generan tanto valores l como r. Por ejemplo, en C los operadores aritméticos, relacionales y lógicos siempre generan un valor–r. En cambio, el operador de contenido `*` genera ambos tipos de valores, dependiendo del contexto en el que se use.

Para poder recoger con facilidad todas las restricciones sobre los valores–l y valores–r podemos definir las siguientes funciones:

```
Bool Cmp_lv_bin(Bool, Bool, Operador_binario);
Bool Rst_lv_bin(Bool, Bool, Operador_binario);
Bool Cmp_lv_un(Bool, Operador_unario);
Bool Rst_lv_un(Bool, Operador_binario);
```

Las funciones que comienzan con `Cmp` toman como parámetro un operador y valores lógicos que indican si el operando sobre el que actúa ese operador es o no un valor–l. El resultado depende de si ese operador puede o no actuar sobre expresiones con esas características. Las funciones que comienzan con `Rst` determinan si el resultado de aplicar el operador sobre esos operandos es un valor–l o no.

En el caso del lenguaje C los únicos operadores binarios que construyen una expresión que es un valor–l son `_[_]`, `_._` y `_->_`. El único operador unario que construye una expresión que es un valor–l es el `*` (contenido). Los demás operadores construyen expresiones que son valores–r. Por supuesto cada operador necesita ser aplicado a operandos con un tipo y una característica de valor–l adecuados. Así por ejemplo el operador & tiene que ser aplicado a una expresión que sea un valor–l de tipo `T` y devuelve un valor–r con tipo `Puntero(T)`. El operador $*$ (contenido) tiene que ser aplicado a una expresión cuyo tipo sea $\widehat{\text{P}}$untero(t) y devuelve un valor–l con tipo t. Igualmente en el lenguaje C son valores–r: las constantes, los nombres de función y los nombres de array. Toda esa información debe ser especificada en las funciones comentadas anteriormente.

Con las funciones anteriores definidas en cada caso particular, las restricciones sobre los valores–l pueden expresarse como se especifica en la siguiente gramática con atributos:

Categorías Sintácticas

s	$\in$	Sentencia
e	$\in$	Expresión
v	$\in$	Variable
le	$\in$	Literal_Entero

Atributos

<Tipo tipo>	Expresión Variable Numero
<Bool lvalue>	Expresión Variable Numero

Definiciones

$$s \;:\; e_1 := e_2 \quad \left\langle \begin{array}{l} e_1.lvalue \wedge \\ Compatible_bin(e_1.tipo, e_2.tipo, Asig) \end{array} \right\rangle \; \{\}$$

$$e \;:\; e_1 \; ob \; e_2 \quad \left\langle \begin{array}{l} Compatible_bin(e_1.tipo, e_2.tipo, ob) \wedge \\ Cmp_lv_bin(e_1.lvalue, e_2.lvalue, ob) \end{array} \right\rangle \; \left\{ \begin{array}{l} e.tipo = Result_bin(e_1.tipo, e_2.tipo, ob); \\ e.lvalue = Rst_lv_bin(e_1.lvalue, e_2.lvalue, ob) \end{array} \right\}$$

$$\mid \; ou \; e_1 \quad \left\langle \begin{array}{l} Compatible_un(e_1.tipo, ou) \wedge \\ Cmp_lv_un(e_1.lvalue, ou) \end{array} \right\rangle \; \left\{ \begin{array}{l} e.tipo = Result_un(e_1.tipo, ou); \\ e.lvalue = Rst_lv_un(e_1.lvalue, ou) \end{array} \right\}$$

$$\mid \; v \quad \{e.tipo = v.tipo \wedge \; e.lvalue = True\}$$

$$\mid \; le \quad \{e.tipo = Entero(); e.lvalue = False;\}$$

12.7.1 Comprobación de l–values y r–values en presencia de un comprobador de tipos para operadores y rutinas sobrecargadas

La gramática que hemos mostrado anteriormente y la familia de funciones que hemos propuesto para la comprobación de los valores l y r es bastante sencilla de codificar, pero si examinamos este problema desde un punto de vista mucho más general, veremos que la comprobación del correcto uso de valores l y r no es más que un caso particular el problema de la comprobación de tipos en presencia de operadores y funciones sobrecargados que estudiamos en la sección 12.5.

Por ejemplo, el operador `+` suele admitir la sobrecarga `Entero` $\times$ `Entero` $\rightarrow$ `Entero` y el operador de asignación `:=` la sobrecarga `Entero` $\times$ `Entero` $\rightarrow$ `TVacío`. La única diferencia entre estas sobrecargas, aparte de la del tipo de retorno, es que la del operador `+` puede trabajar tanto con expresiones que generan l–values como r–values, mientras que el primer operando del operador de asignación debe ser forzosamente un l–value. Este problema se puede resolver con facilidad si a partir de ahora consideramos la característica de valor r o l como un elemento más de los tipos. Esto es:

```
LRTipo :: LValue | RValue

LValue :: t: Tipo

LValue :: t: Tipo

Tipo :: TBásico | TComplejo

TBásico :: ...
```

De esta forma, un literal entero como el `2` tendría el tipo `RValue(Entero)`, una variable `x` declarada como `Tabla[1..10] de Real` tendría el tipo `LValue(Tabla(Rango(1, 10), Real)` y la expresión de indexación `x[1]` sería de tipo `LValue(Real)`. De esta manera, para resolver el problema de la comprobación de *l–values* y *r–values*, lo único que tenemos que hacer es añadir en la entrada de la tabla de símbolos que describe los operadores las sobrecargas siguientes:

```
:=:     LValue(T)      ×     RValue(T)      →  RValue(Vacío)
:=:     LValue(T)      ×     LValue(T)      →  RValue(Vacío)
:=:    LValue(Real)    ×  RValue(Entero)    →  RValue(Vacío)
:=:    LValue(Real)    ×  LValue(Entero)    →  RValue(Vacío)

 +:  LValue(Entero)    ×  LValue(Entero)    →  RValue(Entero)
 +:  LValue(Entero)    ×  RValue(Entero)    →  RValue(Entero)
 +:  RValue(Entero)    ×  LValue(Entero)    →  RValue(Entero)
 +:  RValue(Entero)    ×  RValue(Entero)    →  RValue(Entero)
 +:   LValue(Real)     ×   LValue(Real)     →  RValue(Real)
 +:   LValue(Real)     ×   RValue(Real)     →  RValue(Real)
           ...
```

Resulta evidente que al usar esta técnica, la comprobación de tipos del lenguaje queda reducida a implementar adecuadamente el algoritmo que estudiamos en la sección 12.5. Aumentar el número de operadores del lenguaje o modificar sus características se reduce ahora a introducir las sobrecargas adecuadas en las entradas de la tabla de símbolos que describen esas rutinas, cosa que incluso se podría llevar a cabo de forma automática a partir de una especificación textual como la anterior.

La única objeción que se podría achacar a la técnica es el hecho de que ahora la comprobación de una expresión como `2.3 + 4` necesita recorrer la lista de sobrecargas, que puede ser bastante grande. De todas formas, no debemos olvidar que esa "lista" de sobrecargas se puede implementar muy eficientemente utilizando tablas *hash*, como demuestra la herramienta de construcción de compiladores Kimwitu. Por lo tanto, la técnica que hemos expuesto en este capítulo para la comprobación de tipos en presencia de operadores sobrecargados resulta de una gran utilidad práctica y se puede reaprovechar en la construcción de compiladores para lenguajes muy diferentes.

12.8 Comprobación de tipos para las sentencias

Una vez que hemos resuelto el problema de la comprobación de tipos para las expresiones (de una forma completamente general e independiente del lenguaje concreto que estemos compilando en el caso de usar un comprobador de tipos que tenga en cuenta la sobrecarga de funciones y los tipos polimorfos), especificar la comprobación de tipos para las sentencias es una tarea bastante sencilla.

A continuación se muestra una gramática atribuida para la comprobación de tipos de las sentencias más comunes en los lenguajes de programación.

Categorías Sintácticas

s $\in$ Sentencia
e $\in$ Expresión

Atributos

Definiciones

s_1	: $e_1 := e_2$	$\left\langle \begin{array}{l} e_1.l-value \wedge \\ Compatible_bin(e_1.tipo, e_2.tipo, Asig) \end{array} \right\rangle$	$\{\}$
	$\mid$ *if e then* s_2 *else* s_3	$< e.tipo = Booleano >$	$\{\}$
	$\mid$ *while e do* s_2	$< e.tipo = Booleano >$	$\{\}$
	$\mid$ *for* e_1:=e_2 *to* e_3 *do* s_2	$\left\langle \begin{array}{l} e_1.l-value \wedge \\ Compatible_bin(e_1.tipo, e_2.tipo, Asig) \\ Compatible_bin(e_1.tipo, e_3.tipo, Asig) \\ e_1.tipo = Entero \end{array} \right\rangle$	$\{\}$

12.9 Problemas

Problema 1: Especifique un comprobador de tipos para rutinas sobrecargadas que tenga en cuenta la posibilidad de rutinas con número variable de parámetros.

Problema 2: ¿Cómo podrían aprovecharse los resultados del problema anterior para generalizar la comprobación de tipos que hemos propuesto en la sección 12.8 para las sentencias.

Ayuda: Considere que la sentencia `while`, por ejemplo, es una función con el interface `Booleano` $\times$ `Sentencia` $\rightarrow$ `Sentencia`.

Problema 3: Algunos lenguajes, como Casale I, permiten tres modos de paso para los parámetros:

- Entrada, el valor del parámetro sólo se puede leer.

- Salida, tan sólo se puede asignar un valor al parámetro, aunque no se puede leer el valor que éste contiene.
- Entrada/Salida, el parámetro se comporta dentro de la rutina en la que aparece como una variable local más. Se puede consultar su valor y también cambiarse libremente.

De qué forma se podría implementar este lenguaje usando el comprobador de tipos que hemos estudiado en este capítulo sin introducir modificación alguna en el mismo.

Parte IV

Semántica Dinámica

Capítulo 13

Código intermedio. Generación

Al diseñar la parte de generación de código de un compilador debemos decidir si generar código directamente para una máquina real o indirectamente a través de la generación de un código intermedio. Este código intermedio posteriormente será interpretado o nuevamente procesado para obtener código de la máquina real. La primera de las alternativas choca con muchos inconvenientes aunque tiene algunas ventajas. El principal incoveniente en un curso de compiladores es que se entremezclan los problemas de la generación de código con los detalles particulares de la arquitectura de la máquina destino. Es mucho más conveniente didácticamente separar la generación de código intermedio de la generación de código para la máquina destino. En este capítulo presentamos las formas generales de representar el código intermedio. Así como los detalles para la generación de código intermedio de los elementos más usuales en los lenguajes de programación.

La parte general del capítulo puede prepararse a partir de la parte primera del capítulo ocho de [ASU86].

13.1 Introducción

En este capítulo vamos a ver las formas de convertir el grafo obtenido de la sintaxis abstracta en una estructura lineal. Hay diferentes formas de conseguir este objetivo. Todas ellas son denominadas códigos intermedios o lenguajes intermedios. Suponemos en general que tenemos almacenada la tabla de símbolos obtenida. Ello supone que los identificadores podemos representarlos como entradas en la tabla de símbolos y por lo tanto sus atributos pueden ser recuperados de allí en un momento determinado. En lo que sigue represntaremos los identificadores por x, y,... para hacer mas legible el texto, pero no olvidemos que con ello estamos denotando entradas a la tabla de símbolos.

13.2 Notaciones prefija y postfija

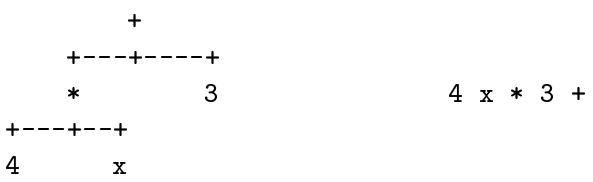
La notación postfija es un lenguaje intermedio que puede describirse con la siguiente gramática:

```
exp : VAR
```

```
    | NUM
    | exp ou
    | exp exp ob
    | exp exp exp ot
    ;
```

Obsérvese que lo que estamos describiendo es la estructura de una secuencia de símbolos. Por ello la gramática anterior es una gramática concreta. Es decir una expresión en esta notación se compone de una variable, un número, una expresión seguida de un operador unario, dos expresiones seguidas de un operador binario o tres expresiones seguidas de un operador ternario. Esta gramática nos indica la manera de reconocer un expresión escrita en notación postfija.

La manera de obtenerla a partir de un árbol es haciendo un recorrido en postorden del mismo. La expresión obtenida a partir del árbol del ejemplo es:

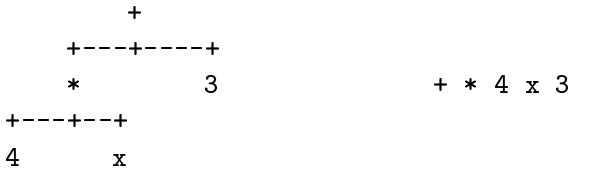


Esta notación tiene ventajas con respecto a notación infija usual en que no necesita parentesis y la evaluación de este tipo de expresiones es bastante sencilla. Aunque es posible extenderla para operadores n-arios, como inconveniente principal tiene que muy dificil expresar en ella las estructuras de control del lenguaje y no es adecuada para la fase de optimización de código. Todo ello hace que esta notación no sea muy usada como lenguaje intermedio en el caso de un compilador, aunque sería adecuada por su simplicidad en algunos casos concretos.

La *notación prefija* es similar a la anterior pero ahora es descrita por la gramática:

```
exp : VAR
    | NUM
    | ou exp
    | ob exp exp
    | ot exp exp exp
    ;
```

La forma de obtener a partir de un árbol es por un recorrido en preorden del mismo. La expresión asociada al árbol del ejemplo es ahora:



Las ventajas e inconvenientes de esta notación son similares al caso de la notación postfija.

13.2.1 Notacion cuartetos

Los cuartetos son la forma de código intermedio mas usada. Este lenguaje consiste de una secuencia de cuartetos, posiblemente precedidos de una etiqueta, donde cada cuarteto puede tener una de las siguientes formas:

```
dcl x tp;
dcp x tp;
x := y opb z;
x := opu y;
x := y;
x := y[z];
x := y;
x := *y;
*x:= y;
x[y]:=z;
goto etiq;
if v goto etiq;
ifnot v goto etiq;
param x;
call p,n;
return;
return x;
```

Donde el cuarteto *dcl x tp;* denota la declaración de la variable x del tipo tp, *dcp x tp;* denota la declaración de un parámetro de tipo tp, *x := y opb z;* es la asignación a x del resultado de operar y con z mediante el operador binario opb, etc. Como en el lenguaje C los operadores unarios *, denotan contenido y dirección.

Este lenguaje se denomina de cuartetos porque puede ser representado como un array de regristos de cuatro campos. Asi el cuarteto *x := y opb z;* rellenaría los cuatro campos con x, y ,z, opb colocados en un orden preestablecido. Las etiquetas pueden ser representadas por el numero de orden del registro correspondiente dentro del array.

Las variables pueden ser representadas por sus lexemas o por entradas en una tabla de simbolos.

Un diseño concreto del lenguaje de cuartetos implica escojer adecuadamente el conjunto de tipos básicos permitidos y los operadores binarios y unarios permitidos para esos tipos.

Como se verá mas adelante es un lenguaje muy adecuado para las representaciones intermedias del proceso de compilación y en concreto de la fase de optimización.

Otros posibles cuartetos pueden ser

```
x:=memp n;
x:=memm n;
x:=sizeof tb;
x:=free p;
```

Estos cuartetos son adecuados para la manipulación de memoria dinámica y de pila. Asi *x:=memp n;* pide un bloque de memoria de pila de tamaño n y deja su dirección en la tem-

poral x. *memm* pide la memoria del motón. Asi se podiran pensar en otras posibilidades. *sizeof* devuelve el tamaño de un tipo y *free* libera la memoria señalada por p.

13.2.2 Notacion Tercetos

```
/     b     c
=     a     (1)
*     c     d
=     d     (3)
```

Es una notación similar a la de los cuartetos, pero mas compacta. Aqui cada terceto puede tener asociado un resultado temporal. Este resultado temporal puede ser referenciado por un terceto posterior.

A (1) se el asocia el resultado de b/c

Esta notación es mas compacta que los cuartetos. Pero resulta problemática en el momento en el que sea necesario reordenar los tercetos.

13.2.3 Notacion tercetos indirectos

Es similar a la notación de tercetos, pero se añade un array que indica el orden en el que se ejecutarán los tercetos:

```
1          /     b     c
3          =     a     (1)
2          *     c     d
4          =     d     (3)
```

Aqui (1), (3) indican los resultados temporales de los tercetos que ocupan ese lugar, pero el array de la izquierda indica el orden de ejecución.

13.3 Generación de código intermedio

13.3.1 Traducción de expresiones y referencias a estructuras de datos.

El objetivo es aprender a procesar los aspectos más importantes que aparecen en los lenguajes de programación ligados con las expresiones y a generar el correspondiente código intermedio. Como elementos de las expresiones aparecen las referencias a estructuras de datos. Una parte importante es la manipulación de temporales.

El tema puede prepararse a partir de capítulo once de [FiL88] y el capítulo ocho de [ASU86].

Expresiones

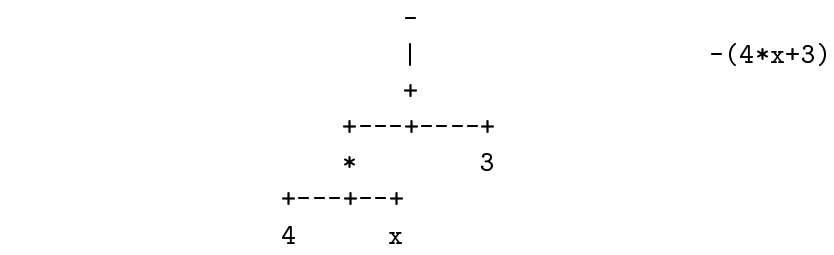
La transformación de una expresión en un código intermedio de cuartetos puede hacerse recorriendo el árbol de la expresión y por cada nodo intermedio crear una variable temporal nueva, del tipo calculado para esa subexpresión, y crear un cuarteto que asigne a esa temporal el resultado de operar, con el operador que tenga la expresión, las temporales asociadas a las subexpresiones.

Así tendremos las siguientes transformaciones

```
e1 op e2    =>      t := t1 op t2;
e1 op       =>      t := t1 op;
```

donde t es una variable temporal nueva del tipo asociado a e1 op e2, t1 es una variable temporal donde se dejará el resultado de e1 e igualmente para t2, e2.

Como ejemplo tengamos una expresión cuyo árbol es:



El código generado es:

```
t1 := 4 * x;
t2 := t1 + 3;
t3 := -t2;
```

Observese que se genera un cuarteto por cada nodo interior del árbol. El recorrido se va haciendo de la hojas hacia los padres. Los literales de las constantes y variables se generan sin transformación. La optimización del número de varibles temporales se verá en una capitulo próximo.

Operadores no estrictos

Hay operadores booleanos que tienen un código particular. Estos son operadores en los cuales el resultado de la operación puede ser determinado, en algunos casos, despues de evaluar el primer operando. Asi el operandor *or* da como resultado *True* si el primer operando es *True*. Eso implica que asumimos una semántica determinada para el operador *or*. Segun esta semántica, aunque el segundo operando diera como resultado algún tipo de error, el resultado de la operación sería *True* si el primer operando es *True*. Este tipo de operadores se denominan no estrictos. El código intermedio generado para el operador *or* sería:

```
      t1 := e1 ;
      if t1 goto et1;
      t2 := e2 ;
      t1 := t1 or t2;
et1 : .....
```

El resultado de la operación se deja en la temporal *t1*. De manera equivalente se hará para otros operadores del mismo tipo.

Tipo enumerado

Cada tipo enumerado puede sustituirse por el tipo entero. Cada valor del tipo enumerado se sustituirá por un entero de 1 a n, donde n es el número de valores del tipo enumerado.

Arrays Multidimensionales

Su declaración puede reducirse en forma general a:

```
T0= array[a1..b1,a2..b2,...,an..bn]  de  Tb
```

Esto puede hacerse si cada tipo enumerado se ha representado por un entero como se ha visto arriba. El objetivo que nos proponemos es transformar el tipo T0 definido arriba en otro T1 de la forma

```
T1 = array [0..D-1] de Tb
```

Si definimos

```
Di=bi-ai+1
```

entonces D puede calcularse como

```
D=D1*D2*...*Dn
```

Asi vemos que una variable v declarada de tipo T0 puede cambiarse a tipo T1 definido arriba. Pero el cambiar el tipo de la variable nos obliga a cambiar dentro de una expresión cada aparición de la forma

```
v[e1,e2,...,en]
```

por

```
v[x]
```

donde x es ahora de tipo entero. La transformación se puede hacer si consideramos el array multidimensional como un array unidimensional donde se han dispuesto las filas del primero consecutivamente. Asi se cumple que:

```
x:=((e1-a1) * D2 *... * Dn
   +(e2-a2) * D3 *... * Dn
   +................
   +(en-an))
```

Esta expresión compleja se puede descomponer de la siguiente forma:

```
D1=b1-a1+1;
D2=b2-a2+1;
....
Dn=bn-an+1;

c:=(a1 * D2 *... * Dn
   +a2 * D3 *... * Dn
   +................
   +an);

t:=(e1* D2 *... * Dn
   +e2 * D3 *... * Dn
   +................
   +en);

x:=t-c;
```

estas expresiones pueden a su vez transformarse en:

```
c:=a1;
c:=c*D2+a2;
c:=c*D3+a3;
....
c:=c*Dn+an;

t1:=e1;
...
tn:=en;

r:=t1;
r:=r*D2+t2;
...
r:=r*Dn+tn;

x  :=   t-c;
```

donde en todo lo anterior t, r, c son variables temporales nuevas. En lo anterior hay un conjunto de expresiones constantes que pueden ser calculadas en tiempo de compilación. Estas expresiones son: Di, D, c.

Otra transformación interesante el la que reduce una array unidimensional a una secuencia operaciones con punteros, sumas y contenidos: Asi la expresión

```
b[i]
```

puede transformarse a

```
*(pb+i)
```

Donde pb es de tipo *Puntero(Tb)* donde *Tb* es el tipo base del array b. Así

```
pb:=&b[0];
```

Estamos asumiendo que el código intermedio soporta el tipo *Puntero(t)*, donde t son los tipos básicos soportados. Tambien asumimos que la suma de un *Puntero(t)* mas un entero da *Puntero(t)*, con un significado igual al que tiene en el lenguaje C.

En algunos lenguajes como C el identificador de un array es un puntero a la base del mismo. En este caso la transformación anterior será:

```
*(b+i)
```

Registros y Uniones

Suponemos que tenemos ahora unos tipos T0, T1 declarados de la siguiente forma:

```
T0 = registro
        a1: T1;
        a2: T2;
        ....
        an: Tn;
     fin registro;
```

y una variable v declarada de ese tipo y usada dentro de una expresión de la forma:

```
v: T0;
......
.. v.a1 ...
....
..  v ....
```

El objetivo de nuevo es transformar la anterior declaración y uso de la variable en otra tal que sólo aparezcan los tipos elementales T1, T2, ... Este objetivo se puede conseguir sustituyendo la declaración de v por n declaraciones de variables con nombres nuevos. El uso de la variable de forma equivalente se sustiutirá por el uso de las n variables declaradas. El uso de un campo en particular se sustituirá por el uso de la variable correspondiente. Lo anterior se transformaría en:

```
v_a1: T1;
v_a2: T2;
```

```
 ....
 v_an: Tn;
 ......
 .. v_a1 ...
 .....
 ..  v_a1 ....
 ..  v_a2 ....
 ....
 ..  v_an ....
```

La transformación del tipo union se hace de forma similar, pero ahora las n varibles declaradas podrán ubicarse en la misma posición de memoria. Esta información se debe guardar para la hora de generar código en una máquina virtual o real.

Arrays dinámicos y Registros dinámicos

Los arrays dinámicos son aquellos en los cuales algunos o varios de los límites ai, bi no son constantes sino expresiones que dependen los valores de los parámetros del procedimiento o función. Supongamos la siguiente declaración, donde algún ai, bi puede ser una expresión:

```
T0= array[f1..h1,f2..h2,...,fn..hn]  de  Tb;
T0 v;
```

La transformación de arriba no nos vale. Ahora la transformación adecuada será a un tipo T1 de la forma:

```
T1 = registro
       int c,D;
       int a1,b1,D1;
       int a2,b2,D2;
       ...
       int an,bn,Dn
    fin registro;
T1 rT0;
Tv=Puntero(Tb);
Tv pv;
```

Ahora en la parte de inicialización de variables se generará código para rellenar los campos del registro *rT0*. A cada tipo dinámico le asociaremos un registro de este tipo. Este registro contendrá la información del tipo en tiempo de ejecución. Esto puede hacerse de la siguiente manera:

```
rT0.a1:=f1;
rT0.b1:=h1;
rT0.D1:=rT0.b1-rT0.a1+1;
...
c:=rT0.a1;
c:=c*rT0.D2+rT0.a2;
```

```
...
c:=c*rT0.Dn+rT0.an;
rT0.c:=c;

d:=D1 ;
d:=d*D2;
...
d:=d*Dn;

t1:=sizeof Tb;
rT0.D:=d*t1;
pv:=memp D;
```

Ahora la variable indexada:

```
v[e1,e2,...,en]
```

se convertirá en:

```
*(pv+x)
```

con x calculado por:

```
t1:=e1;
...
tn:=en;

t:=t1;
t:=t*rT0.D2+t2;
...
t:=t*rT0.Dn+tn;

x:=t-rT0.c;
```

En resumen se ha escogido una estructura de datos el resgistro T1 que contiene la información necesaria para el tipo y los datos se ubicarán de forma independiente en otra zona de memoria. Esta estructura de datos asociada al tipo se la suele denominar *vector dope*.

Los registros dinámicos serán los que tengan alguna componente dinámica, por lo que su tratamiento será similar al de los arrays.

Asignaciones Complejas

En determinados lenguajes de programación aparecen varias formas de asignaciones complejas que pueden ser reducidas a asignaciones sencillas. Aqui presentamos dos y su correspondiente transformación: La primera es de la forma:

```
e1:=e2:=...:=en;
```

que se transforma en

```
e(n-1):=en;
....
e2:=e3;
e1:=e2;
```

Donde se supone que las diferentes expresiones, menos la última, deben ser valores-l. La segunda es una asignación paralela:

```
<e1,e2,...,en> := <f1,f2,...,fn>;
```

donde ei, fi son expresiones. Esta se transforma en:

```
t1:=f1;
t2:=f2;
...
tn:=fn;
e1:=t1;
e2:=f2;
...
en:=tn;
```

Esta transformación puede simplificarse cuando las expresiones ei sean variables.

13.3.2 Traducción de las estructuras de control

El objetivo de esta sección es procesar las estructuras de control más representativas de los lenguajes de programación. Estas pueden ser clasificadas en tres categorías: estructuras condicionales, repetitivas y de transferencia directa del control. Entre las condicionales incluimos: un if-then-else generalizado y la estructura case. Las estructuras repetitivas incluyen: los bucles while, repeat, for, loop, etc, con enunciados de ruptura del bucle del tipo break, continue. Por último entre las estructuras de transferencia directa de control incluimos: break, continue, goto y excepciones. El tema puede prepararse a partir del capítulo doce de [FiL88] y el capítulo ocho de [ASU86].

Cada una de las estructuras de control adminte una transformación al código intermedio comentado anteriormente.

Sentencia *If*

```
if (e) then s1 else s2    ====      t:=e;
                                    ifnot t goto et2;
                                    codigo de s1;
                                    goto ets;
                               et2: codigo de s2;
                               ets: ....
```

La semántica adoptada para la sentencia *if (e) then s1 else s2* es la usual. Si la expresión e produce un resultado verdadero entonces se ejecuta s1 sino s2.

Sentencia *While*

```
while (e) s          ====       ete: t:=e;
                                     ifnot t goto ets;
                                     codigo de s1;
                                     goto ete;
                                ets: ....
```

```
while (e) {          ====       ete:t:=e;
    s1;                             ifnot t goto ets;
    continue;                       codigo de s1
    s2;                             goto ete;
    break;                          codigo de s2
    s3;                             goto ets;
 }                                  codigo de s3;
                                    goto ete;
                               ets: ....
```

La semántica adoptada para la sentencia *while (e) s* es la usual. Si la expresión e produce un resultado verdadero entonces se ejecuta s y posteriormente se vuelve a evaluar e. La semántica del *break* y *continue* se ha escogido como en el lenguaje C.

Sentencia *For*

```
for(s1,e,s2) s       ====            codigo de s1;
                               ete:  t:=e;
                                     ifnot t goto ets;
                                     codigo de s;
                                     codigo de s2;
                                     goto ete;
                               ets: ....
```

La semántica adoptada para la sentencia *for (s1,e,s2) s* es la del lenguaje C. Se ejecuta s1. Se evalua e y si da verdadero se ejecuta s y luego s2. Posteriormente se vuleve a evaluar e. Si da falso se sale de la estructura de control.

Sentencia *Switch*

En el caso del *switch* hay diversas posibilidades de transformación al código intermedio. En cualquier caso hay que tener en cuenta la semántica dada a esta estructura de control en cada caso. Aqui se propone una transformación que supone salir de la estructura una vez ejecutado el caso correspondiente. La transformación sería:

```
switch (e) {       ====            t:=e;
    E1 :  s1;                      goto etp;
    E2 :  s2;                  et1: codigo de s1;
    ....                           goto ets;
    En :  sn;                  et2: codigo de s2;
    default: s0;                   goto ets;
 }                             et3: codigo de s3;
                                    goto ets;
                                    ....
                               etn: codigo de sn;
                                    goto ets;
                               et0: codigo de s0;
                                    goto ets;
                               etp: t1:= E1 = t;
                                    if t1 goto et1;
                                    t1:= E2 = t;
                                    if t1 goto et2;
                                    ....
                                    t1:= En = t;
                                    if t1 goto etn;
                                    goto et0;
                                ets:...
```

El lenguaje C tiene otra semántica diferente. Solo se sale de la estructura de control si se explicita con una setencia *break* o estamos en el caso último. La transformación ahora sería:

```
switch (e) {       ====            t:=e;
    E1 :  s1;                      goto etp;
    E2 :  s2;                  et1: codigo de s1;
    ....                       et2: codigo de s2;
    En :  sn;                  et3: codigo de s3;
    default: s0;                    .....
 }                             etn: codigo de sn;
                               et0: codigo de s0;
                                    goto ets;
                               etp: t1:= E1 = t;
                                    if t1 goto et1;
                                    t1:= E2 = t;
                                    if t1 goto et2;
                                    ....
                                    t1:= En = t;
                                    if t1 goto etn;
                                    goto et0;
                                ets:...
```

En este caso la aparición de la sentencia *break* se traduciría por *goto ets;*. El trozo de código que va desde la etiqueta *etp* hasta *ets* admite varias formas de optimización cuando el número de etiquetas es muy alto. Una de ellas es usar una tabla hash para calcular la dirección de salto a partir del valor de la temporal t1. Otras formas son posibles.

La generación de código intermedio para la sentencias en el lenguaje fuente *goto, label* se hace con el uso de etiquetas.

13.3.3 Traducción de procedimientos y funciones

Hay dos partes esenciales para traducir procedimientos y funciones: el procesamiento de las declaraciones y el procesamiento de las llamadas. La parte de declaraciones se incluyó anteriormente. El objetivo de esta sección es el procesamiento de las llamadas. Un aspecto importante de ellas son los diferentes métodos de implementar el paso de parámetros. Otro es la traducción de enunciados del tipo return o exit. Otros problemas de la invocación de procedimientos, tales como la grabación y restauración de registros se posponen para el tema de la generación de código. El tema puede ser preparado a partir del capítulo trece de [FiL88]. La traducción de corrutinas, semáforos, etc. puede encontrarse en [Ben82] y [Ben90].

Paso de parámetros

Los parámetros en general pueden ser de entrada, de salida o entrada/salida. En algunos lenguajes como C esto no es posible. Sólo hay parámetros de entrada. Pero en general puede haber de los tres tipos anteriores.

Hay distintas formas de paso de parámetros:

Paso por valor: Se pasa el valor de la expresión que aparece como parámetro real. Este parámetro se comporta dentro del cuerpo del procedimeinto o función como una variable local más.

Paso por referencia: Se pasa la dirección de la expresión que aparece como parámetro real. Este parámetro se trata dentro del cuerpo de la función como un puntero al objeto dado.

Paso por nombre: Se realiza una expansión de macros con respecto al parámetro dado. En cada llamada se sustituye textualmente, en el cuerpo del procedimiento, el parámetro formal por la expresión que se recibe como parámetro real.

Las técnicas anteriores pueden ser usadas para implementar las distintas clases de parámetros. Una primera posibilidad es:

Paso por copia/restauración: Todos los parámetros se pasan por valor. Antes de hacerse la llamada al procedimiento, se calculan los valores de las expresiones que aparecen como parámetros reales. Luego se hace la llamada al procedimiento y despues de salir del procedimiento, se restaura el valor de los parámetros de salida o entrada/salida en la dirección adecuada. Supongamos un lenguaje determinado donde una declaración de función pudiera ser de la siguiente forma:

```
Funcion f(ent x1:T1; sal x2:T2; ent/sal x3:T3) dev y:T4;
```

y sea la llamada a la función:

```
r:=f(e1,e2,e3);
```

El esquema de generación de código intermedio sería:

```
evalua e1 en t1;
evalua dir de e2 en t2;
evalua dir de e3 en t3;
p1:=t1;
p3:=*t3;
param p1;
param p2;
param p3;
param pr;
call f,4;
*t2:=p2;
*t3:=p3;
r:=pr;
```

El orden de generación de los parámetros puede variar. En el capítulo siguiente se verán algunas posibilidades. Observese como se ha añadido un parámetro más que contendrá el resultado. Tanto t1, t2, t3 como p1, p2, p3, pr son variables temporales nuevas. En el cuerpo de la función p1, p2, p3, pr serán consideradas como unas variables locales mas. Para denotar que son parámetros los declararemos con *dcp p1 T1;* etc.

Paso de la mayor parte de parámetros por referencia: Los parámetros de entrada se pasan por valor, los de salida o entrada/salida se pasan por dirección. Incluso puede ocurrir que algun parámetro de entrada, que ocupe gran cantidad de memoria se pase por dirección para evitar el tiempo de copiado.

```
evalua e1 en t1;
evalua dir de e2 en t2;
evalua dir de e3 en t3;
p1:=t1;
p2:=t2;
p3:=t3;
param p1;
param p2;
param p3;
param pr;
call f,3;
r:=pr;
```

Ahora se pasa un puntero como parámetro en el caso de salida y entrada/salida. Dentro del cuerpo del procedimiento o función los parámetros de entrada salida han de tratarse como punteros a los objetos que señalan, por lo tanto cuando haya que usar o actualizar el valor del objeto usaremos el operador de contenido.

Cuerpo de función: El código intermedio para el cuerpo de una función comenzará con una etiqueta, que denotará el nombre de la función, después se declaran los parámetros por orden, las variables locales y las temporales, se generará el código de inicialización de variables locales y por último se generará el código para sentencias y expresiones.

```
f: dcp p1 T1;
   dcp p2 T2;
   dcp p3 T3;
   dcp pr T4;
   declaracion de locales;
   declaracion de temporales;
   inicializacion de variables;
   codigo de sentencias;
```

Dentro del cuerpo puede aparecer return e; que se traducirá igualmente por

```
evaluar e en t;
return t;
```

Si no aparece return explicitamente se incluirá un return; como última sentencia del cuerpo.

Paso de todos los parámetros por nombre: Este caso se puede implementar sustituyendo el procedimiento por una macro que expandirá el preprocesador. En cualquier caso hay que tomar algunas precauciones porque el paso de parámetros por nombre no conserva en todos los casos la semántica asociada a los parámemetros cuando se implementa el paso de parámetros por copia restauración o el otro procedimineto antes comentado. Como ejemplo vease el siguiente

```
  procedimiento intercambiar(ent/sal x,y:int)
   { int t;
     t:=x;
     x:=y;
     y:=t;
   }
```

cuando el procediminto de se sustituye por una macro como

```
macro intercambiar(x,y) {int t; t:=x; x:=y; y:=t} fin_macro
```

entonces la llamada a *intercambiar(i,a[i])* produce el código, despues de la expansión de la macro,

```
   { int t;
     t:=i;
     i:=a[i];
     a[i]:=t;
   }
```

que como se puede comprobar no realiza el trabajo deseado. Asi si el valor inicial de i es i1 y el del a[i1] a1, los valores finales no quedan intercambiados Asi i2, el valor final de i, es a[i0] como debe ser. Pero la casilla de a que ha quedado actualizada ha sido a[a[i0]]=i0 y no a[i0] como debia ser.

Procedimientos como parámetros: Vamos a ver el caso en que uno de los parámetros de llamada de una función es una llamada a un procedimiento.

Para permitir que los parámetros puedan ser procedimientos o incluso etiquetas suponemos que p pueda ser un nombre de procedimiento o una etiqueta en los cuartetos de tipo *param p*.

En general para pasar un procedimiento como parámetro hay que pasar el nombre del procedimiento y su número de parámetros. Según la máquina virtual que diseñemos habrá que pasar diferentes informaciones. El nombre del procedimiento se suele sustituir por la dirección de comienzo de su código.

Capítulo 14

Máquinas Virtuales

El objetivo de este capítulo es presentar los problemas que plantea la organización de la memoria en tiempo de ejecución y las soluciones a los mismos. Un modelo de la organización de la memoria en tiempo de ejecución debe estar bien definido para poder generar código adecuadamente. La organización de la pila, la estructura de los registros de activación, la implementación del paso de los parámetros a los procedimientos, la asignación estática de memoria, la manipulación del motón, son algunos de los problemas a los que se les da una solución.

Pero además definimos dos máquinas virtuales para las que vamos a realizar la generación de código en este tema.

La primera es la denominada máquina P. Es una simplificación del denominado código P. Esta es una máquina virtual especialmente pensada para la generación de código a partir de lenguajes tipo PASCAL. La segunda la hemos denominado máquina C. El lenguaje de esta maquina es un subconjunto del lenguaje C especialmente escogido para que la arquitectura de la máquina se asemeje en lo posible a una máquina de registros real. Este código intermedio tiene la ventaja de que es muy transportable. Puede ser convertido en ejecutable por un compilador de C que además podría optimizar el código resultante.

El tema puede prepararse a partir del capítulo siete de [ASU86], el capítulo nueve de [FiL88] y el capítulo seis de [Hol90]. La descripción de la arquitectura de la máquina P y su conjunto de instrucciones puede extraerse de un apéndice de [Ben82], de [Wir71a] y completar con [PeD82]. Una descripción elemental se puede encontrar en [Wir90]. La descripción de la máquina C se encuentra en [Hol90].

14.1 Máquinas Virtuales

La organización de la memoria en tiempo de ejecución presupone el diseño de una máquina virtual. Esta es en definitiva el conjunto de estructuras de datos que son necesarias para hacer ejecutable un determinado código intermedio. El diseño de una máquina virtual conlleva:

- Elección del tamaño de cada unidad de memoria de la máquina.

- La implementación de un conjunto de tipos básicos y de operaciones para manipularlos.
- La elección de un conjunto de memorias donde se ubicará el código y los datos. Estas memorias se gestionarán con distintas políticas. Entre ellas las más conocidas son: memoria de estática, de pila, memoria de código, montón, registros, etc.
- Escoger los modos de direccionamiento permitidos para los operandos en cada operación.
- Diseño de los registros de activación. Estos son el trozo de memoria que le corresponde a cada llamada de un procedimeinto o función. Este diseño conlleva ubicar los parámetros, las variables locales, temporales, valor de restorno, etc. El diseño supone tambien elegir la referencia para medir los desplazamientos dentro del registro de activación, asi como el sentido del desplazamiento.
- Diseño del conjunto de instrucciones que vamos a permitir en el lenguaje ejecutable en esa máquina.

Una vez diseñada la máquina virtual, sabemos la memoria ocupada por los tipos simples y por lo tanto calcular la memoria que necesitan los tipos compuestos. A partir de aqui es posible asignar una ubicación de memoria para cada variable y calcular el tamaño de los registros de activación asociados a cada procedimiento o función.

14.2 Implementación de los tipos en la Máquina Virtual

Suponemos que las memorias de la máquina constan de palabras de 4 bytes y que las direcciones de memoria toman como unidad la palabra. La representación de los diferentes tipos de datos puede ser:

Int: Ocupa una unidad de memoria de 4 bytes. Podemos usar enteros menores de dos bytes, desestimando el resto. Para enteros largos usaremos las unidades de memoria adecuadas.

Booleano: Lo representamos por un entero. Por tanto ocupa una unidad. *Float:* Ocupan una unidad de 4 bytes, estando determinada la precisión por estos 4 bytes.

Doble: Utilizaremos las unidades de memoria que necesitemos según la precisión que queremos para este tipo de datos.

Enumerados, Carácter: Podemos utilizar en los dos casos un entero para representarlos. Por lo tanto ocupan una unidad de memoria.

Conjunto de Enteros: Los representamos por arrays de bits. Si lo representamos por una palabra, no soportará mas que conjuntos de enteros de 0 a 31. Depende de la cardinalidad máxima que queramos escojeremos un número de palabras adecuado.

Hileras: se pueden implementar:

- Si la hilera es de longitud máxima n ocuparía n+2 / 4 unidades de memoria. Los dos primeros bytes mantendrían longitud de la hilera y los restantes ubicarán los caractes consecutivamente.

- Utilizar una memoria especifica de hileras de caracteres. En ella estarían las diferentes hileras colocadas consecutivamente. Cada hilera acabaría con un carácter especial (como en el lenguaje C). En las demás memorias las hileras se representan como un puntero a la memoria de hileras

Puntero: En general se representarán por un entero que ocupe una unidad de memoria. Si el tamaño total de unidades de memoria es mayor que el que se puede representar en 4 bytes se toman dos unidades para dicho puntero.

Arrays: Los arrays multidimensionales los reduciremos a arrays unidimensionales tal como vimos en el capítulo de código intermedio. Un array unidimensional declarado de la forma

```
T = array[0..N-1] de Tb;
a : T;
```

ocupa N*sizeof(Tb) unidades de memoria. La forma de calcular la direción de a[i] vendrá dada por:

```
 dir(a[i]) = dir(a) + i*sizeof(Tb);
```

Registros: Se ubicarán consecutivamente en la memoria los diferentes campos.

```
T = registro
       a1 : T1;
       a2 : T2;
       ...
       an : Tn;
     fin registro;
h : T;
```

Un registro ocupará la suma de lo que ocupen sus campos.

```
sizeof(T) = sizeof(T1)+...+sizeof(Tn);
```

La dirección de h.ai vendrá dada por:

```
dir (h.ai) = dir(h) + offset(ai)
```

Teniendo en cuenta que cada campo tiene un offset determinado que viene reflejado en la Tabla de Símbolos. Este offset es el desplazamiento del campo dentro del registro. El offset del primer campo es cero.

Uniones: Se ubicarán los diferentes campos en la misma zona de memoria.

```
T = union
       a1 : T1;
       a2 : T2;
```

```
       ...
       an : Tn;
     fin union;
h : T;
```

Una unión ocupará el máximo de lo que ocupen sus campos.

```
sizeof(T) = max(sizeof(T1),...,sizeof(Tn));
```

La dirección de h.ai vendrá dada por:

```
dir (h.ai) = dir(h)
```

Es decir todos los campos de la unión se ubican en la misma dirección de memoria.

Listas: Sea el tipo lista declarado de la forma

```
T = list(t);
a : T;
```

Este tipo se puede implementar por un registro con dos campos. Ambos campos son punteros. El primero señala a la cabeza de la lista. El segundo a la cola de la lista. La lista nula se representará por ambos punteros nulos. Una lista cualquiera se representará por su cabeza y su cola. La cola a su vez puede ser otra lista que se representará de la misma manera. Este tipo tendrá una parte en la memoria de pila o estática y otra en el montón. Como ejemplo vemos que una lista de enteros como:

```
[1,5,7]
```

se representará en la memoria por

```
[1,[5,[7]]]
```

La lista original se ha transformado en cabeza y cola, donde esta última es a su vez una lista y tiene que representarse como tal. La lista vacia se representa con los punteros nulos. En la lista con un elemento el primer puntero señala al elemento y el segundo es nulo.

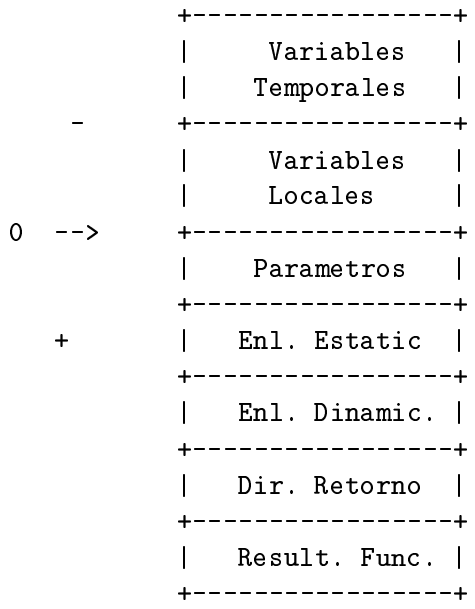
Árboles: Par los árboles binarios se reservan tres casillas: la primera es el puntero a la raíz del árbol y las otras dos se corresponden con los punteros a cada uno de los nodos hijos. Los árboles multicamino pueden reducirse a los binarios por técnicas conocidas. Igual que en el caso de las listas este tipo tendrá una parte en la memoria de pila o estática y el resto en el mónton. La representación tiene características similares a las de las listas.

14.3 Registros de activación asociados a los procedimientos

Un programa en sí constará del cuerpo principal y de una serie de procedimientos anidados o no. Cada vez que se invoque a un procedimiento, se deben reservar memoria para sus

parámetros, sus variables locales y temporales. Cada procedimiento, por tanto, llevará asociado un registro de activación. Este contiene la memoria necesaria para los objetos locales al procedimiento y para la información necesaria para hacer posible el retorno del procedimiento, el acceso a variables globales, etc.

En cada registro de activación incluiremos:



La ubicación de cada elemento puede ser diferente en cada caso. Otra cosa a decidir es la posición del desplazamiento 0 en el registro de activación y los signos positivos o negativos de los diferentes desplazamientos. Una vez diseñado el registro de activación y decididas las implementaciones de los tipos básicos de la máquina es posible calcular los desplazamientos (offset) de cada variable local, temporal y parámetro con respecto a la referencia establecida.

Cada variable está declarada dentro de un determinado procedimiento. Las variables globales pueden ubicarse en un resgistro de activación especifico. Esto hace que podamos asignar a cada variable un desplazamiento (offset) dentro de un registro de activación. Este cálculo puede hacerse en el momento de compilar.

14.4 Asignación de Memoria

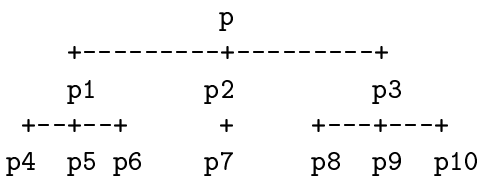
Hay varias formas de asignar memoria a las variables en tiempo de ejecución. Ya hemos visto que cada variable puede ser ubicada en un registro de activación. Ahora el problema reside en como asignamos memoria a los registros de activación. Una vez decidida la manera de hacer esto tendremos la forma de saber en tiempo de ejecución la direción absoluta de un registro de activación en particular. La dirección de una variable se obtendrá en tiempo de ejecución a partir de la dirección del registro donde está y el desplazamiento de la variable dentro de él.

Hay dos estrategias básicas: asignación estática y dinámica. En la primera a cada registro de activación se le ubica en tiempo de compilación en una posición fija de la

memoria. En la segunda los registros de activación se van apilando y desapilando en una memoria de pila. Esto se hace cuando se produce una llamada o un retorno de un procedimeinto.

14.4.1 Asignación estática

La asignación estática es adecuada para las variables globales de un lenguaje. Este tipo de asignación es adecuado cuando el número de llamadas a procedimiento es conocido en tiempo de compilación. Esto ocurre en lenguajes como el FORTRAN que no tienen recursividad. En este caso es posible determinar si dos procedimientos pueden compartir la misma memoria o no. Para hacer esto se calcula un arbol de secuencias de llamadas de un procedimiento a otro. En este arbol un determinado nodo estará asociado a un procedimiento. Si un nodo p llama a los procedimientos p1,p2,p3 entonces el nodo asociado a p tendrá como hijos nodos asociados a p1, p2, p3. A cada programa podemos asociarle un arbol de llamadas como este:



Si el lenguaje no tiene recursividad un nodo no puede estar asociado a mismo procedimiento que uno de sus descendientes. Cuando determinado nodo, por ejemplo p4, designa el procedimiento cuyo código se está ejecutando en un momento dado, entonces los procedimientos asociados a los nodos ascendentes no han terminado y por lo tanto sus varibles están activas tambien. Asi p, p1, p4 no pueden compartir memoria. Pero p1 y p2 si. Con estas ideas y teniendo en cuenta el arbol de llamadas es posible asignar memoria estáticamente a cada llamada de un procedimiento. La llamada asocida a p se la ubica en primer lugar, consecutivamente la de p1 y p4. Tambien p2 se ubica detras de p1 compartiendo memoria con p1, etc. Asi cada llamada recibirá la memoria adecuada para el registro de activación asociado.

14.4.2 Asignación dinámica

Cuando el lenguaje permite la recursividad la estrategia anterior no es adecuada. En este caso se usa una pila, de tal forma que se apila un registro de activación cuando se llama a un procedimiento y se desapila un registro cuando el procedimiento retorna. Puesto que los registros de activación son de tamaño variable hace falta un puntero que indique la base del registro de activación anterior cuando se desapila uno dado.

14.4.3 Anidamiento estático

En ciertos lenguajes (como por ejemplo PASCAL) otro problema a resolver es el anidamiento estático de procedimientos. Este problema se plantea cuando se permite declarar procedimientos anidados dentro de otro procedimiento, de tal forma que se las variables locales de un determinado procedimiento son visibles para todos sus procedimientos descendientes.

Para resolver este problema asociamos a cada procedimeinto su nivel de anidamiento estático. Este es un número que nos indica el nivel en el que un procedimiento está declarado dentro de otro. A los procedimientos más exteriores le asignamos el 1, a los declarados dentro de ellos el 2, etc. Cada variable local, temporal o parámetro vendrá identificada por dos números: el nivel del anidamiento estático del procedimiento donde está definida y su desplazamiento dentro del registro de activación de este procedimiento. Lo que tenemos que garantizar es que dos variables con esos dos números diferentes ocupan en cada momento posiciones de memoria diferentes. Dos variables, posiblemente distintas que tengan esos dos números iguales pueden compartir memoria. Esto es debido a que en un procedimiento dado no puede hacerse referencia mas que a sus variables locales o parámetros (nivel de anidamiento n como el procedimiento), las declaradas en el procedimiento que lo engloba (nivel de anidamiento n-1) y asi sucesivamente hasta las variables globales con nivel de anidamiento 0. Todas las variables accesibles desde una llamda particular reciben coordenas diferentes por este procedimiento.

Una vez asociado a cada variable su nivel y su desplazamiento, el problema que se plantea es calcular en tiempo de ejecución la dirección absoluta donde se ubicará la variable a partir de esos dos números. Eso puede hacerse con un mecanismo llamado *display*. Este es un array de punteros que mantene la siguiente información. Sea d el display, entonces d[1] señala la base del registro de activación de la última llamada a un procedimiento de nivel de anidamiento 1, asi con d[2], etc. La dirección absoluta de una variable x de coordenadas n, f (nivel, desplazamiento) es

```
dir(a) = d[n] + f
```

La atualización del display cuando se realiza la llamada a un procedimiento de nivel de anidamineto i puede hacerse de la siguiente manera:

- Se guarda d[i] en el nuevo registro de activación
- Se hace que d[i] apunte al nuevo registro de activación
- Justo antes de que acabe el procedimineto llamado se restaura d[i] al valor guardado.

14.5 Maquina C

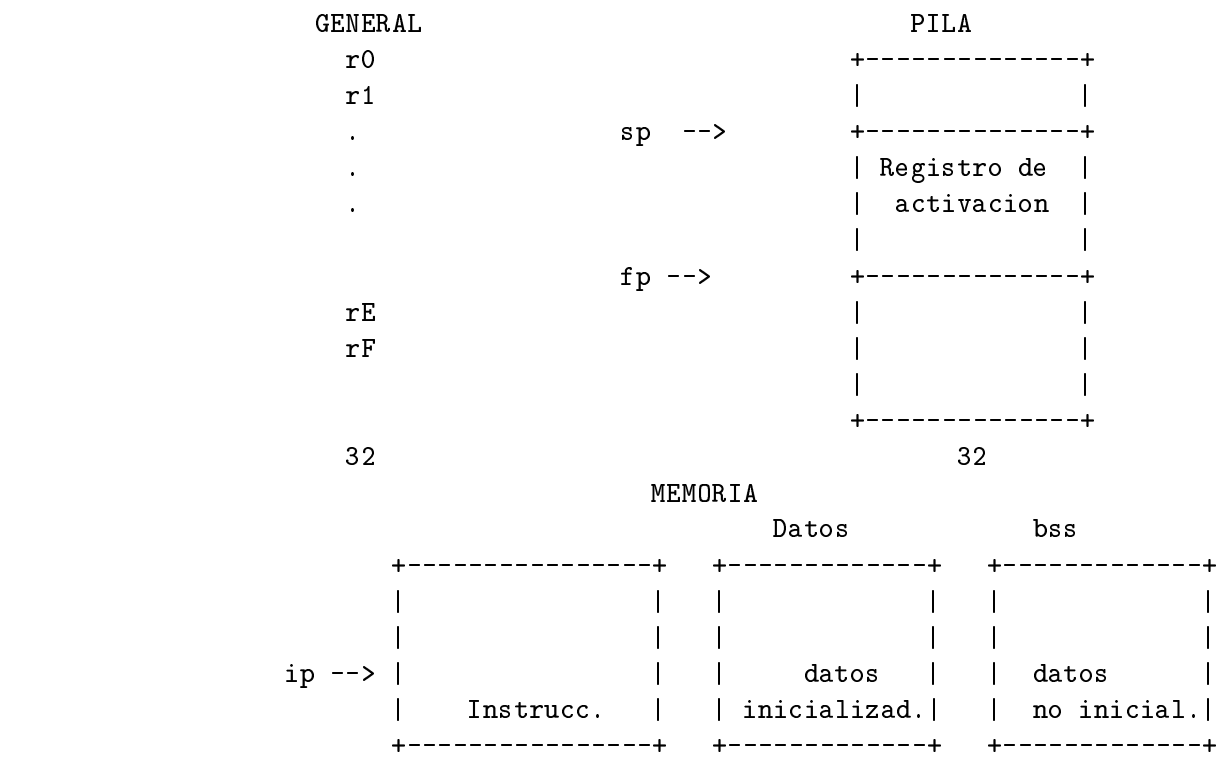
A diferencia de la máquina P, la máquina C utilizará REGISTROS para llevar a cabo las operaciones. Esto hace que la generación de código sea de una forma cercana a como se hace para una máquina real.

Las características de la máquina virtual quedarán recogidas en un fichero llamado "virtual.h".

14.5.1 Memorias de la máquina C

El esquema de la máquina C es:

```
REG. DE PROPOS.
```



La memoria de la máquina se divide en registros, pila y memoria.

Constará de 16 registros de propósito general (r0 a rF). Además disponemos de registros de propósito específico para hacer las funciones de contador de programa, cima de la pila, etc. Todos los registros serán de 32 bits, pudiendo ser accedidos como si fuesen de 32, 16 u 8 bits.

La pila es de 2048 entradas de 32 bits cada una. Además lleva asociados los registros sp y fp. El sp señala la cima de la pila y el fp señala el comienzo del registro de activación que está situado entre los parámetros y las variables locales.

Al simularlo con un compilador de C no tenemos que preocuparnos del contador de programa y no tenemos que reservar una zona donde escribir el código. El registro ip mantiene la dirección de la próxima instrucción a ejecutar.

Existe además una zona de memoria para la declaración de variables globales (en esta máquina sólo se usarán variables locales en la pila y globales en la zona de memoria).

Estas variables globales pueden ser variables globales inicializadas y no inicializadas. Las primeras no cambian de lugar durante la ejecución del programa o mantienen su valor estáticamente, y se almacenan en la zona de datos de la memoria para var. inicializadas. Las segundas, no tienen por qué estar presentes en el código ejecutable, pero sí deben incluirse.

Cada unidad de memoria puede contener un puntero, un entero largo, dos enteros cortos o cuatro caracteres. Las definiciones de tipos básicos irían dentro de virtual.h y serían:

virtual.h

```
 /******   definicion de tipos basicos    **********/

1 #include <c-code.h>   /* controla tama\~nos de tipos */
2
3 typedef char byte;
4 typedef short word;
5 typedef long lword;
6 typedef char *ptr;
```

Al declarar un registro, se hará una unión de los tipos básicos anteriores. El conjunto de registros y la pila también se especificarán en el fichero virtual.h

```
                    virtual.h
 /** definicion de los registros y de la pila **/

18 struct _words    {  word  high, low;  }
19 struct _bytes    {  byte  b0, b1, b2, b3; }
20
21 typedef union reg
22  {
23    char          *pp;
24    lword         l;
25    struct _words w;
26    struct _bytes b;
27  } reg;
28 typedef reg *pr;
29
30  reg   r0, r1, r2, r3, r4, r5, r6, r7;
31  reg   r8, r9, rA, rB, rC, rD, rE, rF;
32
33  reg   stack[ SDEPTH ];
34  reg   * __fp;
35  reg   * __sp;
36
37  #define fp ((char *) __fp)
38  #define sp ((char *) __sp)
```

Esto hace que si queremos acceder a una unidad de memoria m (registro, pila o dato estático) usemos el correspondiente caso de la union para extraer la información adecuada.

- puntero ..... m.pp
- palabra32-bit ..... m.l
- dos palabras16-bit .... m.w.high m.w.low
- 4 palabras de 8-bits .. m.b.b3 m.b.b2 m.b.b1 m.b.b0

Organización de memoria: segmentos

Dentro del programa de código_C podemos encontrarnos en el segmento de código, de datos inicializados y de datos no inicializados.

Con la directiva SEG(text), SEG(data) y SEG(bss) podemos pasar de un segmento a otro.

En virtual.h se define como:

```
                    virtual.h
/** definicion de directivas sobre segmentos */
39 #define SEG(segmento)     /* vacio */
```

Veamos una muestra de reserva de memoria para código y variables globales:

```
int x;  ----------->     SEG (bss)
                           word _x;

int y=1; ---------->     SEG (data)
                           word _y=1;
```

Las variables locales no se referenciarán por su nombre, sino que manejaremos los punteros de la pila según la posición del registro de activación.

Para variables y procedimientos se puden declarar diferentes modos de almacenamiento. Los modos de almacenamiento y su significado son:

- private: Se da espacio para la variable, pero esta no puede ser accedida desde otro fichero.
- public: Se da espacio para la variable y esta puede ser accedida desde cualquier fichero del programa.
- external: El espacio para esta variable se reserva en otro sitio.
- common: El espacio para esta variable lo reserva el enlazador de programas.

Para declarar distintos tipos de alamacenamiento usamos las macros

```
40 #define public
41 #define common
42 #define private static
43 #define external extern
```

Modos de direccionamiento

Algunos se basan en las directivas de acceso declaradas en virtual.h:

```
                virtual.h
 /******   definicion de directivas de acceso  *****/

45 #define W       * (word  *)
46 #define B       * (byte  *)
47 #define L       * (lword *)
48 #define P       * (ptr   *)
49 #define WP      * (word  *)
50 #define BP      * (byte  *)
51 #define LP      * (lword *)
52 #define PP      * (ptr   *)
```

Inmediato: Se especifica el número entero, hexadecimal (0xnum) o bien octal (0num).

Ejemplo: 92

Directo: Se nombra el contenido de una variable o registro y se obtiene su valor.

Ejemplo: x, ro.l

Indirecto: Se accede a un elemento especificando un puntero.

- B(p) : byte de la pila cuya dir. está en p
- W(p) : palabra cuya dirección está en p
- L(p) : pal. larga " " " "
- BP(p) : puntero a byte cuya dir. está en p

Doble Indirecto: De un puntero se extrae la dirección donde está el dato que nos interesa.

Ejemplo: *BP(p) : puntero que apunta a un puntero a byte, cuya dirección está en p.

Base Indirecto: Se usa para el acceso a los elementos de un array. Sabiendo la dirección base del array, especificamos el offset N.

Ejemplo:

- B(p+N) : byte cuya dirección es p+N
- W(p+N) : palabra cuya dirección está en p+N
- L(p+N) : palabra larga cuya dirección está en P+N
- P(p+N) : puntero cuyo valor está en p+N

Dirección efectiva:

Ejemplo:

- &nombre : dirección de variable o primer elemento del array.
- &W(p+N) : dirección de la palabra que se encuentra en la dirección que resulta de sumar un desplazamiento +N al puntero p.

14.5.2 Manejo de la Pila

Las directivas son:

- push(algo) : meter algo en la cima de la pila
- algo = pop(tipo) : sacar algo de la cima de la pila

Las operaciones se realizan sobre elementos de 32 bits (lword).

En el fichero virtual.h se definen:

```
                virtual.h
 /**  definici\'on de operaciones sobre pila **/

53 #define push(n)    (--__sp)->l = (lword)(n)
54 #define pop(t)     (t) ( (__sp++)->l )
```

14.5.3 Subrutinas

Las definiciones de subrutinas en virtual.h son:

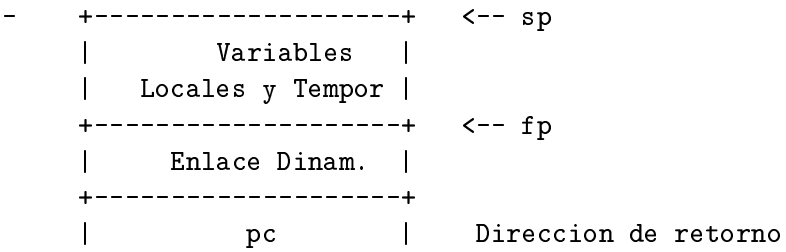
```
                  virtual.h
/* defin. de rutinas, llamadas y retornos   */
55 #define PROC(name,cls)    cls name() {
56 #define ENDP(name)        ret(); }
57
58 #define call(name)  --__sp; name()
59
60 #define ret()       __sp++; return
```

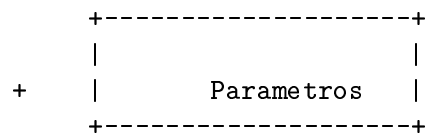
Valores de retorno de subrutinas

Cuando una función devuelve algo, se almacena en el registro rF, quedando los restantes registros para operaciones.

Estructura del registro de Activación

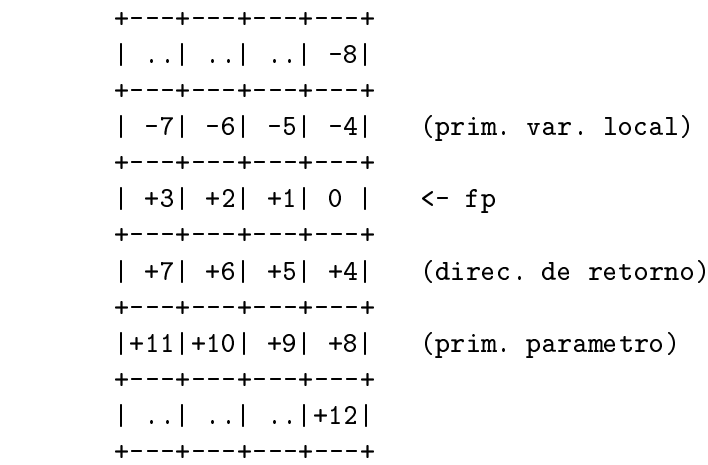
Un registro de activación consta de:





Los signos negativos están hacia arriba. Los parámetros tienen coordenadas positivas y se van apilando empezando por el último hasta el primero.

Para aclarar la forma de numerar las unidades de memoria vease el siguiente gráfico



Asi puede verse que el primer parámetro está en +8. La primera variable local en -4. El registro sp señala a la cima de la pila que crece hacia las coordenadas negativas. El registro fp señala a la referencia del registro de activación actual.

Para la reserva del espacio en la pila para el registro de activación se utilizan las instrucciones:

```
link(n);
unlink();
```

Estas se definen como:

```
#define link(n) {push(__fp);__fp=__sp; __sp -= n;}
#define unlink() {__sp=__fp; __fp = pop(pr);}
```

Cuando se realiza una llamada a un procedimiento (call), se siguen los siguientes pasos:

1. Se almacenan los parámetros en la pila.
2. Se realiza la llamada call(nombre)
3. Reserva (link) para variables locales y temporales.
4. Ejecución de la función.
5. Se deshace la reserva del link (unlink)
6. Se retorna de la llamada
7. Se liberan los parámetros.

14.5.4 Estructura del código

Un ejemplo típico de código_C sería:

```
SEG( text )
PROC( nombre, public )
     call( otro );
     ret();
ENDP( nombre )
```

El formato de las instrucciones que operan con datos es:

```
dest op fuente
```

Donde *dest* debe ser un registro. Este registro tiene un valor que operado con *fuente* produce otro valor. El resultado se coloca en el registro *dest*. Cada operando que aparezca en fuente puede tener un modo de direccionamiento de los explicados mas arriba. Se admiten los operadores del tipo:

```
=  +=  -=  *=  /=   ...
```

Control del flujo

Se pueden producir llamadas a procedimientos, saltos condicionales e incondicionales.

Las etiquetas se utilizan de la forma:

```
etiqueta:
     .
     .
     .
goto etiqueta;
```

Los saltos incondicionales se hacen con *goto et;*. Los saltos condicionales con una combinación del tipo

```
EQ(r4.b.b0,0)
goto et5;
```

que poduce el efecto de saltar a la etiqueta et5 si el del registro r4 es 0. En virtual.h se definen las directivas de comparación:

virtual.h

```
/*******  directivas de comparaci\'on  *****/
68 #define EQ(a,b)   if( (long)(a) == (long)(b) )
69 #define NE(a,b)   if( (long)(a) != (long)(b) )
70 #define LT(a,b)   if( (long)(a) <  (long)(b) )
```

```
71 #define LE(a,b)   if( (long)(a) <= (long)(b) )
72 #define GT(a,b)   if( (long)(a) >  (long)(b) )
69 #define GE(a,b)   if( (long)(a) >= (long)(b) )
```

Un ejemplo podría ser:

```
if ( expr )
  {
     sent1;
  }
else
  {
     sent2;
  }
```

En código_C sería:

```
     r4.b.b0 = expresion
     EQ(r4.b.b0,0)
     goto et5;
       sent1
     goto et6;
et5: sent2
et6: .......
```

Ejemplo de paso de código de entrada a código_C:

```
int x,y = 8;

int multiplica ( int i, int j)
  {
     int aux;
     aux = i*j;
     return(aux);
  }

main()
  {
     int a,b;
     x = 3;
     a = 17;
     b = (x+a)*y;
     y = multiplica(b,x);
   }
```

La ubicación en memoria de las diferentes variables y parámetros sería:

```
SEG( bss )
```

```
                x              -
SEG( data )
                y              -

Parametros  de Multiplicar
                i             +8
                j             +12

V. Locales de Multiplicar
               aux            -4

V. Locales de  Main
                a             -4
                b             -8
```

El código_C resultante sería:

```
#include "virtual.h"

#define T(x)        /* var\'{\i}ables temporales  */
SEG( data )
  word _y=8;

SEG( bss )
  word _x;

#define L0 8     /* L0 y L2 longitud de las locales */
#define L1 0
#define L2 4     /* L1 y L3 longitud de las temporales */
#define L3 0

SEG ( text )
PROC ( _multiplica )
 #undef T
  #define T(n) (fp-L2-((n)*4))
  link (L2+L3)
  r0.w.low = w(fp+8);
  r0.w.low *= w(fp+12);
  w(fp-4)=ro.w.low;
  rf.w.low = w(fp-4);
  unlink();
endp( _multiplica )
```

14.6 Cambios en la máquina C para tratar agregados

Los agregados de datos son estructuras complejas, bastante distantes de los datos elementales que suelen manipular las máquinas (virtuales o reales) para las que generamos código.

Por tanto, el problema de generar código para la manipulación de un agregado de datos se reduce a encontrar una representación de dicho agregado, combinando los elementos disponibles en la correspondiente máquina.

14.6.1 Clasificación de los agregados de datos

A la hora de analizar los problemas que plantea la generación de código para agregados, resulta adecuado establecer una clasificación de los distintos tipos de agregados que nos podemos encontrar. De esta forma podremos plantear la solución más idónea para cada tipo de agregado.

- Agregados estáticos
  - Tamaño constante
  - Tamaño variable
- Agregados dinámicos

Denominaremos *estáticos* a aquellos agregados de datos que desde el momento de su creación hasta su destrucción[1] mantienen siempre el mismo tamaño. Los arrays y los registros son ejemplos típicos de agregados estáticos.

Dentro de los agregados estáticos, podemos diferenciar entre los de tamaño *constante* y los de tamaño *variable*. Los de tamaño constante, son aquellos cuyo tamaño depende de cierta expresión constante, y por tanto podrá ser calculado en tiempo de compilación. Por su parte el tamaño de un agregado estático variable dependerá de cierta expresión variable, por lo que deberemos esperar al momento de la ejecución para conocer cual es el tamaño del agregado. Los arrays cuyas dimensiones dependan de expresiones variables son ejemplos de agregados estáticos variables, también lo serán los registros que tengan alguna componente de tamaño variable.

Denominaremos *dinámicos* a aquellos agregados de datos cuyo tamaño pueda variar a lo largo de su periodo de existencia. Ejemplos de estos tipos de agregados son las listas y los árboles.

14.6.2 Métodos de representación

Antes de pasar a los problemas que plantea la generación de código para agregados, necesitamos conocer las distintas formas de representación de las que disponemos. De esta manera seremos capaces de elegir la más adecuada dependiendo de las características del agregado que vayamos a manipular. Los métodos de representación que proponemos son los siguientes:

- Representación contigua

[1] Al periodo que va desde la creación hasta la destrucción de cualquier variable (sea agregado o no), lo denominaremos *periodo de existencia*. Dicho periodo coincide con la ejecución del programa en el caso de las variables globales, y con la ejecución de un procedimiento o función en el caso de las variables locales

- Representación dispersa

En la representación *contigua* hacemos que las distintas componentes de un determinado agregado de datos ocupen zonas de memoria consecutivas. Esto hace que el acceso a las componentes sea relativamente simple ya que sólo necesitamos conocer el desplazamiento de dicha componente dentro del bloque de memoria que ocupa el agregado. La tarea de creación y destrucción también es bastante simple ya que se reduce a la reserva y liberación, respectivamente, de un bloque de memoria. La limitación más importante de este método de representación es que no es aplicable a los agregados dinámicos ya que al no ser en éstos el tamaño fijo es imposible conocer a priori el tamaño del bloque de memoria necesario para representarlos.

En la representación *dispersa* las distintas componentes del agregado no tienen que ocupar posiciones de memoria consecutivas, sino que se utilizan bloques de memoria del montón que se reservan y se liberan de forma dinámica. Para mantener la *cohesión* del agregado necesitamos cierta información adicional (punteros) que nos permita relacionar a todas las componentes de un agregado. La ventaja de este método es su flexibilidad, ya que es válido para todo tipo de agregado. La desventaja es la complejidad que introduce la gestión de la memoria del montón, fundamentalmente con los problemas que plantea la recolección de basura.

14.6.3 Representación contigua de agregados de datos

Como ya hemos dicho anteriormente la representación contigua sólo es adecuada para los agregados estáticos, por lo que en esta sección sólo nos encargaremos de los problemas planteados al generar código para los registros y los arrays.

Estudiaremos distintas características de manipulación de agregados, y cómo la inclusión de cada una de estas características darán lugar planteamientos más o menos sofisticados. De esta forma, ante un determinado lenguaje, con unas características de manipulación de agregados concretas, sabremos que soluciones deberemos tomar.

Almacenamiento y acceso

El almacenamiento de agregados y el acceso a sus componentes son las cuestiones más básicas, y constituyen lo mínimo que un lenguaje que manipule agregados debe incluir.

Con respecto al acceso a una determinada componente, el problema se reduce a calcular la dirección de dicha componente. Para ello basta con calcular el desplazamiento dentro del bloque de memoria que representa al agregado y conocer la dirección de comienzo de dicho bloque de memoria (visto en la sección 13.3).

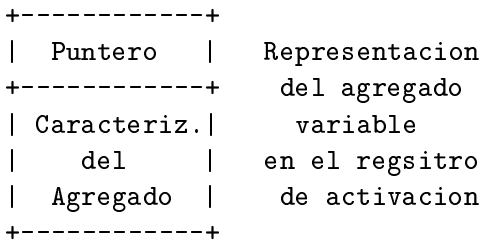
El almacenamiento debe tratarse de distinta forma según estemos ante un agregado constante o ante un agregado variable.

Almacenamiento para agregados constantes Los agregados constantes pueden ser ubicados sin ningún problema en el registro de activación del procedimiento correspondiente. De esta forma tendrán un tratamiento similar al resto de las variables, y de lo

único de lo que tendremos que preocuparnos será de reservar la cantidad adecuada de memoria, que por otra parte podrá ser calculada en tiempo de compilación.

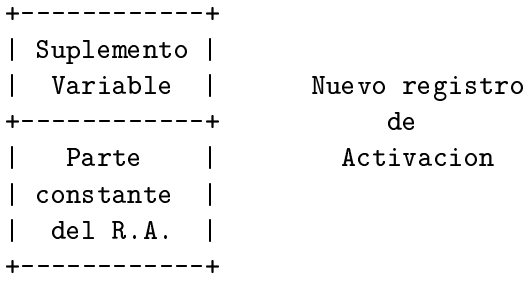
Almacenamiento para agregados variables Para los agregados variables el problema se complica un poco al no tener conocimiento del tamaño del agregado hasta el momento de la ejecución.

En este caso, lo que guardaremos en el registro de activación será una caracterización de la estructura del agregado variable (visto en la sección 13.3) y un puntero para indicar la dirección donde se encontrarán los datos, que se podrán ubicar en el montón:



Con esto conseguimos que la estructura del registro de activación sea fija, y que la variabilidad (datos) se desplaze a otra zona. Si no fuese así sería imposible caracterizar las variables locales de un determinado procedimiento con un desplazamiento (offset), conocido en tiempo de compilación.

Con respecto a los datos, podemos pensar incluso en almacenarlos en la parte alta del registro de activación, evitándonos así el uso del montón. De esta forma el registro de activación quedaría dividido en dos partes, la que hemos utilizado hasta ahora, de tamaño conocido en tiempo de compilación, y un suplemento variable que contendrá los datos de los agregados variables:



De esta forma la variabilidad de los agtregados no trastoca la estructura del registro de activación, y no es necesario acudir a la memoria del montón.

Asignación y otras operaciones entre agregados

Así como el almacenamiento y el acceso se pueden considerar problemas comunes a todo lenguaje que permita la manipulación de agregados, no ocurre lo mismo con la asignación y las operaciones entre agregados. Veamos un pequeño ejemplo que ilustre estas características:

```
Reg = Registro
        a: Entero;
        b: Real;
      Fin registro;
     ...
 r1, r2, r3: Reg;
     ...
 r1 := r2 + r3;
```

La asignación de agregados se puede reducir simplemente a una copia de un bloque de memoria sobre otro bloque de memoria, esta copia puede ser realizada por una rutina escrita en el código de la máquina.

Las operaciones entre agregados también se podrían resolver de esa forma, sin embargo puede llegar a ser complicado codificar esas operaciones, teniendo en cuenta que han de ser diseñadas para cualquier estructura de agregado. En este caso la solución más general es el aplanamiento, que además incluye como caso particular a la asignación.

Asignación mediante copia de bloques de memoria La copia de bloques de memoria, será una solución viable para lenguajes que permitan la asignación de agregados y no permitan ninguna otra operación entre agregados. La rutina `_cpmem` realizará dicha copia de memoria tomando tres argumentos, la dirección del bloque de memoria origen, la dirección del bloque de memoria destino y el tamaño de los bloques. En el caso de la máquina C, tendríamos la siguiente secuencia de llamada:

```
push(r0.pp);        /* direccion origen  */
push(r1.pp);        /* direccion destino */
push(r2.w.low);     /* tama\~no          */
call(_cpmem);
pop(word);
pop(ptr);
pop(ptr);
```

Como hemos comentado antes, en el caso de que el lenguaje permita además otro tipo de operaciones entre agregados, esta solución deja de ser interesante, siendo más general y adecuado el aplanamiento.

Aplanamiento El aplanamiento consiste en aprovechar el conocimiento de la estructura de un agregado, para traducir una operación entre agregados en una serie de operaciones entre componentes que sea equivalente.

Las operaciones entre registros se aplanan mediante una secuencia de operaciones entre campos. Así el fragmento:

```
r1,r2: Registro
       a: Entero;
       b: Registro c: Real; d: Real; Fin registro;
     Fin registro;
```

```
     ...
r1 := r2 * r1 + {1, {1.2, 4.4}};
```

daría lugar a la siguiente secuencia de operaciones elementales:

```
r1.a := r2.a * r1.a + 1;
r1.b.c := r2.b.c * r1.b.c + 1.2;
r1.b.d := r2.b.d * r1.b.d + 4.4;
```

Por su parte, las operaciones entre arrays se aplanan mediante la iteración de operaciones entre sus componentes. Así el fragmento:

```
t1, t2 : Tabla [1..5] de Real;
   ...
t1 := t2 * t1 + [0.1, 0.2, 0.3, 0.4, 0.5];
```

daría lugar a la siguiente secuencia de operaciones elementales:

```
cte: Tabla [1..5] de Real;
   ...
cte[1] := 0.1;
cte[2] := 0.2;
cte[3] := 0.3;
cte[4] := 0.4;
cte[5] := 0.5;
Desde i:= 1 hasta 5
      t1[i] := t2[i] * t1[i] + cte[i];
Fin desde;
```

En este caso nos hemos visto obligados a definir una zona de memoria para ubicar las constantes, ya que no existe otra forma de indexarlas. Esto se podía haber evitado expandiendo las operaciones entre arrays a través de una enumeración en lugar de a través de una iteración:

```
t1[1] := t2[1] * t1[1] + 0.1;
t1[2] := t2[2] * t1[2] + 0.2;
t1[3] := t2[3] * t1[3] + 0.3;
t1[4] := t2[4] * t1[4] + 0.4;
t1[5] := t2[5] * t1[5] + 0.5;
```

Sin embargo esta otra solución puede dar lugar a expansiones más extensas ya que no compacta en bucles las operaciones que no tienen constantes. Además, para arrays variables, la enumeración no nos serviría de solución ya que no sabemos a priori cuantos elementos tiene el array.

R-valores de agregados

Hasta ahora nos hemos preocupado de variables tipo agregado, es decir de objetos que ocupan un bloque de memoria de un determinado tamaño. Los mayores problemas a los que nos hemos enfrentado han sido la gestión y ubicación de esos bloques de memoria. Las variables sin embargo tienen una ventaja con respecto a las constantes, tienen L-valor, o en otras palabras, *tienen un sitio donde colocarlas.* ¿Que pasa con los agregados que no tienen L-valor? El problema se acentúa, además de tener bloques de datos, no tenemos una dirección donde ubicarlos.

Habitualemente los R-valores de los tipos simples los guardamos en registros o en variables temporales. Sin emabrgo en el caso de los agregados no es lógico almacenar una estructura de grandes dimensiones en registros o variables temporales.

Ya nos hemos tropezado con este problema, cuando en el proceso de aplanamiento de la asignación entre arrays intentábamos acceder a la componente de un agregado constante, en aquella ocasión la solución fue buscarle una zona de memoria a dicha constante declarando una variable y asignádola.

Hay otras situaciones en las que nos encontraremos con un problema similar, una de ellas es la devolución de agregados por parte de una función:

```
Reg = Registro a: Entero; b: Real; Fin registro;
   ...
Func f(): Reg;
   ...
```

En este caso la solución más adecuada es hacer que la función devuelva un puntero al montón donde se encontrarán los datos. De esta forma el R-valor existirá en el montón, y deberá ser destruido cuando haya sido utilizado. Si el agregado es constante, con los datos tendremos bastante, sin embargo si el agregado es variable (por ejemplo un array de dimensiones variables) necesitamos junto con los datos la caracterización de su estructura tal y como hemos visto ya.

Como conclusión podemos decir que una buena técnica para manipular R-valores de agregados consiste en ubicarlos en el montón y acceder a ellos a través de su dirección.

Paso de parámetros de entrada por referencia

En el caso de los agregados, debido a la cantidad de memoria que pueden llegar a ocupar, no es lógico que se haga una copia del parámetro real, sino que resulta más eficiente pasar una referencia, es decir, la dirección.

Esta solución conecta perfectamente con la propuesta anteriormente para representar R-valores de agregados, ya que siempre dispondremos de una referencia a un agregado, aunque conceptualmente no exista (como es el caso de los R-valores).

Esta técnica, sin embargo no es del todo general, ya que depende de la semántica que se de a los parámetros de entrada. Tenemos dos posibilidades:

1. *No se permite la modificación de los parámetros de entrada.* En este caso la solución

es perfectamente válida, quedando de parte del chequeador de la semántica estática la comprobación de que no se infringe la regla anterior.

2. *Se permite la modificación de los parámetros de entrada.* En realidad este tipo de parámetros no serían estrictamente de entrada, sino variables locales que reciben un valor inicial de un parámetro. En este caso tendríamos que detectar qué parámetros están en esa situación, y pasarlos por valor.

14.6.4 Representación dispersa de agregados de datos

Para los agregados dinámicos de datos la representación contigua es totalmente inadecuada. Esto se debe fundamentalmente a la posibilidad de cambio de tamaño del agregado durante su periodo de existencia, lo que hace imposible conocer la cantidad de memoria que debemos reservar en un principio. Por tanto, para poder utilizar la representación contigua tendríamos que acudir a la estimación, que puede dar lugar a alguna de estas dos situaciones:

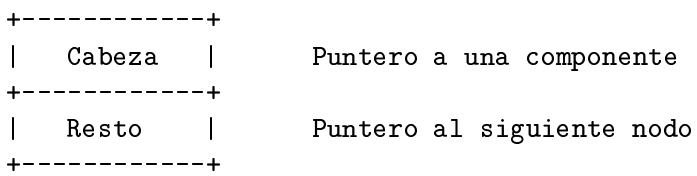
- Que reservemos más memoria de la necesaria, con lo que estaremos desperdiciando recursos, o
- que reservemos menos memoria de la necesaria, con lo que nos impediría representar adecuadamente el agregado.

Ninguna de las dos situaciones anteriores es deseable, por lo que necesitamos un método más flexible, con el que seamos capaces en cada momento de utilizar sólo la memoria que sea necesaria. Esta flexibilidad la conseguiremos con la representación dispersa. El precio que pagamos por ella es que además de almacenar los datos del agregado, tendremos que almacenar cierta información adicional necesaria para relacionar todos los componentes del agregado que se encuentran dispersos a lo largo de toda la memoria.

En este apartado estudiaremos la representación dispersa aplicada a las listas, uno de los agregados dinámicos más elementales, con las cuales resulta relativamente fácil modelar otros agregados dinámicos como conjuntos, pilas, colas, árboles o grafos.

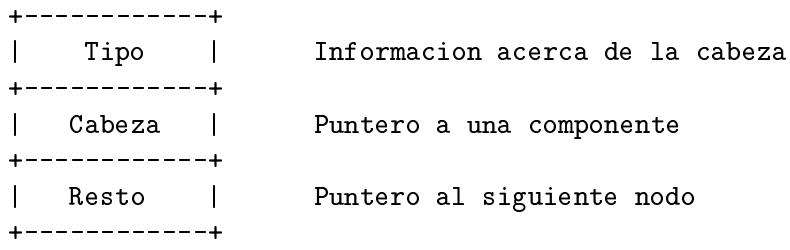
Estructura de los nodos

La representeción dispersa de listas, se basa en el encademiento de nodos, cada nodo tendrá dos partes, la *cabeza* y el *resto*. La *cabeza* contendrá un dato, es decir una componente de la lista (del tipo base de la lista), y el *resto* contendrá una dirección que nos indicará cuál es el siguiente nodo de la cadena. Con la idea de permitir cualquier tipo de dato en la cabeza, podemos guardar en ella una dirección a la zona de memoria donde se encuentra el dato, en lugar de guardar el dato en sí.



Con esta estructura tan simple, seremos capaces de representar listas de cualquier tipo de datos, incluso listas de listas.

En algunas ocasiones puede ser interesante, disponer de ciertas operaciones genéricas como borrar o copiar listas completas, que puedan ser aplicadas a cualquier tipo de lista. La estructura que hemos propuesto no soportaría este tipo de operaciones, ya que a la hora de borrar o copiar el dato apuntado por la cabeza de un nodo, necesitamos saber si es de un tipo estático o dinámico. Así en el caso del borrado, si fuera estático liberaríamos el bloque de memoria que ocupa, y si fuera dinámico volveríamos a aplicar de forma recursiva la función de borrado. Por tanto, si queremos permitir estas operaciones tendríamos que añadir dicha información al nodo.



En cualquiera de los dos casos, una lista se podrá representar por una sola dirección de memoria que apunte al primer nodo de la cadena. De esta forma en el registro de activación sólo tendremos que reservar una posición de memoria para las variables tipo lista, mientras que la memoria necesaria para albergar la lista en sí, será obtenida del montón.

Operaciones de manipulación

En este apartado estudiaremos distintas operaciones que nos serán de utilidad a la hora de generar código para listas, utilizando una representación dispersa.

Estas operaciones estarán orientadas a ampliar la máquina C. El mecanismo de ampliación consistirá en escribir un conjunto de rutinas (una por cada operación) en código C, que pasarán a formar parte de una librería. De esta forma podemos hacer la máquina más flexible, sin tener que cambiar su diseño. Al igual que cualquier otra rutina, los parámetros se pasarán a través de la pila, y el resultado será devuelto en el registro `rF`.

Operaciones de gestión de memoria Las operaciones `_pide_mem` y `_libera_mem`, nos permitirán reservar y liberar memoria del montón, respectivamente. Son por tanto las funciones a las que acudiremos cada vez que queramos incluir o borrar una componente de un agregado dinámico, en nuestro caso concreto de una lista.

- `_pide_mem` recibe como parámetro de entrada la cantidad de memoria requerida (a través de un `word`) y devuelve como resultado la dirección (`rF.pp`) del bloque de memoria reservado en el montón. Reservaremos una posición más de la solicitada que nos servirá para almacenar el tamaño del bloque, esta información nos será de utilidad a la hora de liberar el bloque.

- `_libera_mem` recibe como parámetro de entrada la dirección de un bloque de memoria del montón, que pasará de estar reservado a libre. El hecho de que el tamaño del bloque de memoria fuese almacenado en el momento de la petición nos evita tener que explicitarlo en el momento de la liberación, con lo que el usuario se podrá despreocupar de esta cuestión.

Operaciones básicas Definiremos un conjunto de operaciones, que nos servirán de base para la manipulación de listas. En primer lugar veremos las operaciones elementales `_inserta`, `_borra`, `_cabeza` y `_resto`. Posteriormente veremos operaciones más potentes, que se pueden definir combinando adecuadamente estas cuatro operaciones elementales.

En la descripción de estas operaciones, la expresión *dirección de una lista* se referirá o bien a la dirección de una variable tipo lista, o bien a la dirección de la componente *resto* de un nodo de una lista (que en realidad representa a la lista que resulta de quitar el primer elemento a otra lista).

- `_inserta` toma la dirección de una lista l y la dirección del dato d, colocando un nuevo nodo al principio de la lista l al que le asigna el dato d.
- `_borra` toma la dirección de una lista l y borra el primer nodo. Esta función no se encarga del borrado del dato asociado al nodo.
- `_cabeza` toma la dirección de una lista l y devuelve la dirección de la *cabeza* del primer nodo l.
- `_resto` toma la dirección de una lista l y devuelve la dirección del *resto* del primer nodo de l (que como hemos dicho antes, es la dirección de una lista).

Apoyándonos en estas funciones elementales, podemos construir otras funciones más potentes y por tanto más útiles. Ejemplos de estas funciones son `_longitud`, `_accede_iesimo` e `_inserta_iesimo`.

- `_longitud` toma la dirección de una lista y devuelve su longitud.
- `_accede_iesimo` toma la dirección de una lista l y un entero positivo i (`word`), y devuelve la dirección del nodo que ocupa la posición i en la lista l.
- `_inserta_iesimo` toma la dirección de una lista l, un entero positivo i, y la dirección de un dato d, y coloca un nuevo nodo en la posición i de la lista l al que le asigna el dato d.

Todas las operaciones básicas presentadas en este apartado son válidas para las dos propuestas de estructuras de nodo.

Operaciones genéricas Si utilizamos la segunda de las dos estructuras de nodo propuestas, será posible diseñar operaciones de más alto nivel que las presentadas en el apartado anterior, que puedan ser aplicadas a cualquier tipo de lista. A estas operaciones las denominaremos genéricas. Ejemplos de estas operaciones son `_borra` y `_copia`.

- `_borra` toma la dirección de una lista y la borra completamente.
- `_copia` toma la dirección de una lista y devuelve una copia exacta de dicha lista.

Para el correcto funcionamiento de las funciones anteriores, los agregados estáticos no podrán contener ninguna componente dinámica, ya que en ese caso no sería viable la manipulación *a ciegas* de bloques de memoria. Por ejemplo, sea una lista de registros que tienen a su vez un campo c de tipo lista, para borrar uno de esos registros no nos bastará con liberar la memoria que ocupa, sino que deberemos liberar también la memoria ocupada por la lista del campo c. Para poder hacer esto necesitaremos información acerca de la estructura del registro, cosa que puede llegar a ser bastante costosa. En estos casos es más razonable aplicar una solución más general, utilizar sólo representación dispersa tanto para los agregados dinámicos como para los estáticos.

Representación dispersa como método más general

Imaginemos un lenguaje en el que tenemos registros y listas, en este caso puede ser interesante aprovechar la estructura diseñada para soportar las listas, para representar también a los registros, en lugar de utilizar la representación contigua. Esto nos permitiría manipular ambas estructuras de una forma más homogénea y genérica, evitando así los problemas que aparecían en el diseño de operaciones genéricas.

En este caso vemos como la representación dispersa tiene la ventaja de ser más general que la contigua. Por contra, su manejo es más complicado y requiere un gasto adicional de memoria para almacenar los distintos enlaces forman la cadena de nodos.

Capítulo 15

Optimización local de código

La fase final de un compilador de varios pasos es la generación de código. Esta fase transforma el código intermedio generado por el generador de código intermedio y produce código para una máquina concreta. Es importante resaltar que el problema de generar código óptimo es indecidible lo que nos conduce al uso de heurísticas que puedan servirnos para generar un buen código aunque no necesariamente el óptimo. Entre los problemas con los que se enfrenta el generador de código están la selección de instrucciones y la asignación de registros.

El tema puede ser desarrollado a partir del capítulo nueve de [ASU86] y el capítulo quince de [FiL88].

15.1 Normalización de expresiones

Las transformaciones algebraicas mas usuales son:

- Sustituir 0+x por x
- Sustituir 1*x por x
- Sustituir 2.0*x por x+x
- Sustituir x/1 por x
- Sustituir $x\hat{2}$ por x*x
- Sustituir c1 op c2 por c, donde c1 y c2 son constantes y c el resultado de la operación.

Otras transformaciones importantes son el uso de la propiedad conmutativa de algunos operadores. Para eso debemos establecer un orden entre expresiones. Asi si `e1 op e2` en ese orden y `op` es conmutativo debemos sustituir `e1 op e2` por `e2 op e1`. Un orden adecuado es el que una constante es anterior a una variable, dos variables se ordenan lexicográficamente, etc.

Igualmente es importante aplicar la propiedad asociativa para posteriormente evidenciar las expresiones comunes. Esto lo haremos sustituyendo `(e1 op (e2 op e3))` por `((e1 op e2) op e3)`.

Todas esas transformaciones se reunirán en una función que tomando una expresión la devuelva normalizada.

15.2 Representación de bloques básicos

15.2.1 Construcción de un GDA

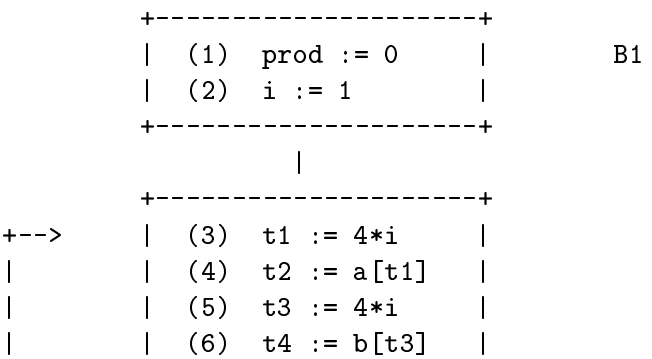
Dado el programa

```
{
 prod=0;
 i=1;
 do
  { prod=prod+a[i]*b[i];
    i=i+1;
  }
 while(i<=20)
}
```

tenemos el código intermedio asociado:

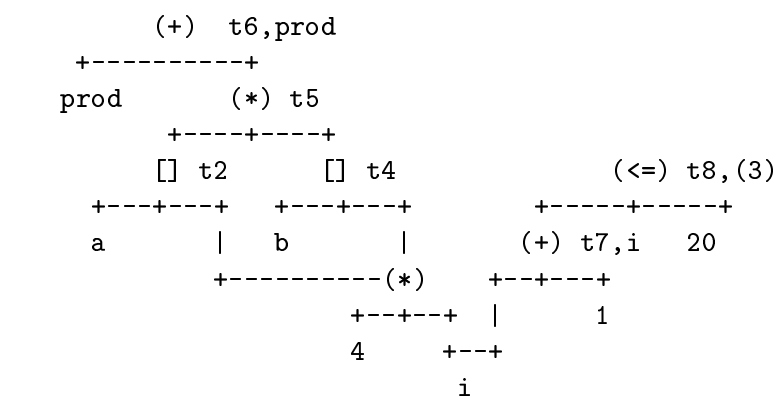
```
(1)  prod := 0
(2)  i := 1
(3)  t1 := 4*i
(4)  t2 := a[t1]
(5)  t3 := 4*i
(6)  t4 := b[t3]
(7)  t5 := t2*t4
(8)  t6 := prod+t5
(9)  prod := t6
(10)  t7 := i+1
(11)  i := t7
(12)  t8 := i<=20
(13) if t8 goto (3)
```

Los bloques básicos y el grafo de flujo del programa son:



```
|        |  (7)  t5 := t2*t4   |
|        |  (8)  t6 := prod+t5 |        B2
|        |  (9)  prod := t6    |
|        |  (10)  t7 := i+1    |
|        |  (11)  i := t7      |
|        |  (12)  t8 := i<=20  |
|        |  (13) if t8 goto (3)|
|        +---------------------+
|                   |
+-------------------+
```

GDA asociado al bloque básico B2:



Algoritmo para la construcción del GDA.

Suponemos que las tuplas son de una de las tres formas siguientes:

```
(1) x := y op z;
(2) x := op y;
(3) x := y;
```

Ademas suponemos la existencia de una función *nodo(id)* que dado el identificador id devuelve el nodo mas reciente creado asociado al identificador. Intuitivamente esta función devuelve el nodo que representa el valor del indentificador en el punto de construcción del *gda*

.

Algoritmo

```
Repetir para cada tupla t:
 Si nodo(y) esta indefinida entonces
   n1=crea_hoja(y);
   nodo(y)=n1;
 Fin Si
 Si la tupla t es de tipo (1) entonces
    Si nodo(z) esta indefinida  entonces
     n2=crea_hoja(z);
```

```
     nodo(y)=n2;
    Fin Si
 Fin Si

 Si la tupla t es de tipo (1) entonces
    Sea n1=nodo(y), n2=nodo(z);
    n=buscar_bin(op,n1,n2);
    Si n esta indefinida  entonces
     n=crea_nodo_interior_bin(op,n1,n2);
    Fin Si
 Fin Si
 Si la tupla t es de tipo (2) entonces
    Sea n1=nodo(y);
    n=buscar_un(op,n1);
    Si n esta indefinida  entonces
     n=crea_nodo_interior_un(op,n1,n2);
    Fin Si
 Fin Si
 Si la tupla t es de tipo (3) entonces
    Sea n=nodo(y);
 Fin Si

 Sea nx=nodo(x);
 Quitar(x,List_Id(nx));
 Poner(x,List_Id(n));
 nodo(x)=n;
Fin Repetir
```

Con el algoritmo anterior se crea un grafo dirigido acíclico a partir de la secuencia de tuplas. Los nodos del grafo pueden ser: hojas o interiores. Cada uno de ellos tiene un atributo que es una lista de identificadores. Para un nodo n este atributo viene dado por *List_Id(n)*. Esta lista de identificadores puede es actualizada mediante las operaciones *Poner, Quitar*. Un nodo hoja tiene un sólo identificador en su lista de identificadores asociados. Los nodos interiores pueden ser binarios o unarios. Los primeros tienen dos hijos y como atributo un operador binario. Los segundos tienen un hijo y como atributo un operador unario.

La función *nodo(x)*, donde x es un identificador devuelve el nodo n donde aparece x en su lista de identificadores o indefinido. Las funciones *crea_hoja(x), crea_nodo_interior_bin(op,n1,n2), crea_nodo_interior_un(op,n1)* crean respectivamente: un nodo hoja dado un identificador, un nodo interior binario dados sus dos hijos y un operador y un nodo interior unario dados su hijo y un operador.

Las funciones *buscar_bin(op,n1,n2), buscar_un(op,n1)* buscan un nodo en el grafo ya creado que tenga el operador op e hijos n1 y n2, en el primer caso o n1 solamente en el segundo caso.

En los tres tipos de tuplas vistos arriba se permite como operador binario el [] (indexación de array) y los operadores unarios *, & (contenido y dirección). Suponemos que las funciones que crean nodos, ademas de crearlos los insertan en una lista por orden de creación y quitan de esa lista los hijos del nodo creado. Asi esa lista mantine en cada

momento los nodos maximales del grafo y estos nodos ordenados por orden de creación. En lo que sigue llamaremos a esta lista *Lis_Graf*.

El algoritmo anterior se debe generalizar para las tuplas de los tipos

```
(4) x[y] := z;
(5) *x := y;
```

esto puede hacerse de una manera sencilla reescribiendo las tuplas anteriores en una manera funcional. Es decir haciendo explícitos los efectos laterales que producen esas tuplas. Asi las tuplas anteriores las reescribimos como

```
(4') x := act(x,y,z);
(5') a := cont(x,y);
```

En (4') aparece el operador ternario act(_,_,_). Este operador calcula un nuevo array a partir de su primer argumento, actualizando la casilla indicada por su segundo argumento con el valor del tercer argumento.

Si sabemos que el puntero x en (5) apunta a la variable *a* entonces la construcción del grafo puede hacerse sustituyendo como si (5) se sustituyera por (5'). El operador *cont(_,_)* actualiza el contenido del puntero que se le pasa como primer argumento con el valor dado en el segundo. La tupla (5') indica que el valor de a puede variar. Si no sabemos donde apunta x entonces (5) hay que sustituirlo por una asignación de ese tipo (5') por cada variable del bloque básico.

Simplificación del grafo obtenido: Si en el grafo obtenido un nodo maximal tiene vacia la lista de indentificadores asociada puede ser eliminado del grafo. La razón de ello es que el cálculo intermedio asociado a ese nodo no tiene que ser guardado en ningún sitio y por lo tanto es superfluo. Es posible que al eliminar un nodo maximal aparezcan otros nodos maximales. A estos se le aplicará la regla anterior. Esta simplificación permitirá eliminar el códido inútil por redundante.

Ordenación del Grafo: A partir del grafo podemos generar ahora una nueva lista de tuplas simplificada en varios sentidos: se ha eliminado el código inútil, se han detectado las expresiones comunes y ahora se va a minimizar el número de variables temporales necesarias. Para generar las tuplas a partir del grafo debemos primero hacer una ordenación topológica del mismo. El orden establecido debe cumplir una propiedad: la tupla asociada a un nodo tiene que ser generada despues de todos sus hijos. Para usar el mínimo de variables temporales se intentará generar la tupla asociada a un nodo inmediatamente despues que la de su hijo izquierdo. El algoritmo que numera los nodos del grafo desde el último al primero será

```
Mientras queden nodos sin numerar
   Sea n un nodo sin numerar y
      que todos sus padres estan numerados;
   Numerar(n);
   Mientras (m=Hijo_Izq(n) y
             m no tenga padres no numerados y
             m no sea una hoja)
       Numerar(m);
```

```
        n=m;
    Fin Mientras
 Fin Mientras
```

En el algoritmo anterior suponemos que existe para cada nodo n un atributo *Num(n)* que nos da su número de orden. La función *Numerar(n)* asigna el siguiente número de orden al atributo *Num* del nodo que recibe como parámetro. Suponemos que la función *Numerar* va actualizando una función *Nodo_Num(i)* que nos da el nodo que tiene el numero de orden i. Además la función *Numerar* ordena los padres de un nodo por el atributo *Num* de menor a mayor.

Asignación de registros a los nodos del grafo: Para generar la lista de tuplas a partir del grafo debemos asignar a cada nodo intermedio una posición de memoria donde se guarde el resultado de ese cálculo. Estas variables las vamos a designar por *r0,r1,r2,....* Serán nombradas así para reflejar que serán variables temporales que deberían guardarse en los registros de la máquina.

```
v=0;
Mientras (todos los nodos interiores
          no tengan asignado un registro)
    Sea n el nodo interior con el atributo Num(n) mas bajo;
    Bucle
       Asig_Reg(n,v);
       l=Hijo_Izq(n);
       Si (l no tiene asignado registro y
           n es el padre con atributo Num(n) mas bajo) entonces
          n=l;
       Sino
          salir;
       Finsi
    Fin Bucle
    v=v+1;
Fin Mientras
```

En el algoritmo anterior la acción *Asig_Reg(n,v)* asigna el registro número v al nodo n. Eso lo lleva a cabo actualizando el atributo *Reg(n)* que da el registro asignado a ese nodo. Igualmente esta acción actualiza una función *Vida_Reg(v)* que para un registro nos devuelve el conjunto de números de nodos en los que está vivo. Un registro está vivo, es decir es útil, en los nodos a los que está asignado, en los padres de los mismos y en todos los nodos que tengan un número comprendido entre otros en los que el registro está vivo. Para implementar lo anterior representamos el conjunto devuelto por un intervalo (m,M). La forma de actualizar un registro es: cuando se asigna el registro v al nodo n se calcula (m1,M1). Donde m1 el menor de los números asignados a n y a sus padres y M1 el mayor. La actualización de *Vida_Reg(v)* se hace de tomando como nuevo valor la envolvente convexa entre el intervalo que tenía y el intervalo (m1,M1) calculado. Así la actualización es *Vida_Reg(v)=EnvConv(Vida_Reg(v),(m1,M1))*. La operación *EnvConv* de dos intervalos se hace por la expresión

```
EnvConv((m1,M1),(m2,M2))=(min(m1,m2),max(M1,M2))
```

Una vez aplicado el algoritmo anterior veremos si hay registros cuyas vidas no tengan intersección. Si es así el segundo registro se sustituirá por el primero. Así si dos registros v1, v2 verifican *Inters(Vida_Reg(v1), Vida_Reg(v2))=Vacio* entonces sustituir el atributo *Reg(n)* de aquellos nodos que sea v2 por v1.

Generación de código a partir de grafos: El algoritmo de generación de tuplas a partir de grafos es ahora:

```
Para i=M hasta 1 con incremento -1
  Sea n el nodo cuyo atributo Num(n) es i;

  Para cada hijo n1 de n tal que no sea hoja y
       n sea el padre de n1 con el atributo Num(n) menor
     Sea r1=Reg(n1);
     Para cada identificador y asociado a n1;
      generar y := r1;
     FinPara
  FinPara

  Sea r=Reg(n);
  Sea op el operador de n;


  Si op es unario entonces
     Sea n1 el hijo de n;
     Si n1 hoja entonces
       Sea y su identificador;
       generar r := op y;
     Sino
       Sea r1=Reg(n1);
       generar  r := op r1;
     Finsi
  Finsi

  Si op es binario entonces
     Sea n1 el hijo izquierdo de n;
     Sea n2 el hijo derecho de n;
     Si n1 y n2 son hojas entonces
       Sea y, z sus identificadores;
       generar r := y op z;
     Finsi
     Si n1 no es hoja y n2 si entonces
       Sea r1=Reg(n1);
       Sea z el identificador de n2;
          generar  r := r1 op z;
     Finsi
     Si n1 es hoja y n2 no entonces
       Sea y el identificador de n1;
       Sea r2=Reg(n2);
       generar  r := y op r2;
```

```
     Finsi
     Si n1 no es hoja y n2 tampoco entonces
       Sea r1=Reg(n1);
       Sea r2=Reg(n2);
       generar  r := r1 op r2;
     Finsi
  Finsi

  Si n es un nodo maximal entonces
    Para cada identificador x
       de su lista de identificadores asociada;
    generar y := r;
  Finsi

FinPara
```

Parte V

Apéndices

Apéndice A

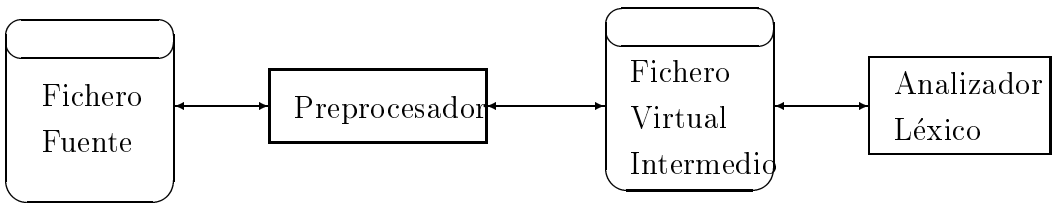
El preprocesador

A.1 Introducción

El preprocesado es una etapa previa al análisis léxico que proporciona al programador diversas utilidades que le permiten escribir más fácilmente sus programas. Entre estas utilidades se encuentran la inclusión de ficheros, la definición y expansión de macros y la compilación condicional. El hecho de que el preprocesado sea una etapa previa al análisis léxico, hace posible que podamos diseñar este último sin considerar la existencia de las facilidades incorporadas en el preprocesado. Tan sólo unas pocas instrucciones del preprocesador llegan al compilador y sirven para indicarle errores, números de línea dentro de los ficheros y para pasarle opciones.

Como hemos anticipado, el preprocesador cuenta para realizar estas tareas con unas instrucciones especiales a las que se le suele llamar *directivas de preprocesamiento*. La tarea del preprocesador es tratarlas convenientemente, de manera que el compilador no tenga que preocuparse en absoluto de ellas y que su analizador léxico se pueda diseñar como un módulo completamente separado.

Al utilizar este planteamiento, el analizador léxico del compilador no se enfrenta ya al fuente de entrada directamente, pues de ello se encargará el preprocesador. Este irá leyendo el fuente, interpretando adecuadamente las directivas y volcando su salida en un *fichero virtual transitorio*. El hecho de que sea virtual significa, como veremos al final del capítulo, que no tiene por qué tratarse de un fichero físico de los que residen en el sistema de archivos del sistema operativo, y transitorio significa que generalmente desaparece después de terminar la compilación.



Como vemos en el esquema, un preprocesador es un traductor, un transformador de lenguajes, exactamente en el mismo sentido que un compilador de C, Pascal o Eiffel, aunque más sencillo. Por lo tanto, internamente, su estructura es la misma que estudiamos en el capítulo anterior: habrá que especificar su léxico, su sintaxis y para implemen-

tarlo usaremos las mismas herramientas que estudiaremos a lo largo de este trabajo para implementar compiladores.

El resto del capítulo lo dedicaremos a examinar las características del preprocesador estándar para el lenguaje C, después estudiaremos cuáles son los problemas más importantes que plantea su implementación e intentaremos aportar soluciones adecuadas a cada uno de ellos. Una descripción completa del preprocesador de C se encuentra en el libro [**Kernighan**]. Existen muchos preprocesadores para lenguajes muy diferentes, pero el más extendido es el de C, razón por la que lo hemos escogido como ejemplo.

A.2 El preprocesador de C

El lenguaje de programación C, nos proporciona un claro ejemplo de lenguaje que puede ser extendido de una forma muy sencilla gracias al preprocesador.

Las directivas más importantes que incluye este preprocesador son las que se enumeran a continuación. Las estudiaremos por turno más adelante.

- Inclusión de ficheros.
- Definición y expansión de macros.
- Compilación condicional.
- Otras directivas que se reconocen pero no se filtran. Tan sólo son tres, completamente imprescindibles, y sirven para pasar cierta información al compilador.

A.2.1 Inclusión de ficheros

Esta característica del preprocesador de C permite insertar dentro de un fichero fuente referencias a otros ficheros que se incluirán en el original a la hora de compilar. Esta directiva admite los dos formatos siguientes:

1. `#include "nf"`
2. `#include <nf>`

Supongamos que en un texto fuente nos encontramos lo siguiente:

```
#include <stdio.h>
```

El efecto es incluir en tiempo de preprocesamiento el contenido del fichero con el nombre `stdio.h` dentro de aquel en el que aparece esta línea. Por lo tanto cada vez que se encuentra una directiva `#include`, el compilador dejará de procesar el fichero actual, continuará tratando el indicado en la directiva y volverá a leer del original cuando éste se acabe, pero de una forma completamente transparente.

Los dos formatos de inclusión no son exactamente iguales. La diferencia entre ambos está en relación a dónde busca el preprocesador los ficheros de inclusión. Generalmente el

preprocesador admite como opción una lista de nombres de directorios en los que puede localizar los ficheros de inclusión. Por ejemplo, en UNIX, el preprocesador estándar se llama `cpp` y la lista de directorios en que buscar se le indica con la opción `-I`. Por ejemplo, el comando

```
cpp -I/usr/include -I~/mis_includes fuente.c
```

instruye al preprocesador para que trate el fichero de nombre `fuente.c` buscando los ficheros de inclusión en los directorios `/usr/include` y `~/mis_includes`, en este orden. Cuando utilizamos el primer formato de inclusión, el preprocesador buscará el fichero de nombre `nf` primero en el directorio por defecto, y si no lo encuentra allí, en los directorios especificados con la opción `-I`. Si utilizamos el segundo formato, el preprocesador empezará a buscar siempre por los directorios indicados con `-I`, ignorando el fichero con el mismo nombre que pueda haber en el directorio actual.

La directiva `#include` se suele utilizar fundamentalmente para que diversos módulos puedan compartir trozos de código comunes, generalmente declaraciones de tipos y prototipos de funciones. Además, es la única manera en que C soporta la importación y exportación de símbolos entre módulos compilados por separado.

A.2.2 Definición y expansión de macros

La definición y expansión de macros es uno de los mecanismos más potentes que incluyen los preprocesadores, pues gracias a él se puede extender el lenguaje subyacente (de una forma limitada), sustituir una llamada a una subrutina por una expansión en línea o definir constantes simbólicas en lenguajes que como el C carecen de ellas.

Una macro se caracteriza por su nombre, que es una cadena de caracteres que la identifica, y por una descripción, que es otra cadena de caracteres a la que se considera equivalente el nombre de la macro. La expansión no es más que la sustitución del nombre de la macro por su descripción.

El preprocesador se encargará por un lado de reconocer la definición de macros, relacionando nombres con descripciones, y por otro lado de su expansión cada vez que se utilicen en el fuente de un programa.

Es posible la definición de macros con parámetros, de tal forma que la descripción de la macro expandida no sea una cadena constante, sino que tenga variables o huecos que se instanciarán en el momento de la expansión. Esta característica es muy útil cuando utilizamos las macros como sustitutas de las subrutinas, ya que nos da la posibilidad de pasarle argumentos "por nombre". Esta forma de paso se diferencia del paso por valor o por referencia en que lo que realmente se le pasa a la macro es la cadena de caracteres que representa al parámetro actual, con lo que al expandir la macro se produce una sustitución textual.

La sintaxis para definir macros es la siguiente:

- `#define N`
- `#define N T`

- `#define N(`p_1`, `p_2`, ..., `p_n`)`
- `#define N(`p_1`, `p_2`, ..., `p_n`) T`

`N` es el nombre de la macro y `T` es su texto de descripción. Como vemos éste es opcional, y en caso de omitirlo la macro se expandirá por una cadena vacía, esto es, una cadena sin caracteres. Por defecto se considera que la descripción llega hasta el primer retorno de carro no precedido por una barra invertida. La combinación barra invertida más retorno de carro se puede utilizar para definir macros largas dividiéndolas en varias líneas. Por ejemplo:

```
#define LARGA esta es una \
              macro larga que \
              ocupa varias líneas \
              de texto.
```

En el caso de definir parámetros, al utilizar la macro habrá que proporcionar exactamente el mismo número de parámetros actuales que formales con los que se haya declarado la macro. Por ejemplo, dada la definición:

```
#define MIN(a, b) ((a) > (b) ?  (b) :  (a))
```

se podrá usar como se muestra a continuación:

```
MIN(1, 2)
MIN((1, 2), 1)
MIN((esto no es C), (y esto tampoco))
```

En el último ejemplo se utilizan los paréntesis para agrupar los argumentos, y como éstos sugieren, no se trata de código C válido; pero esto no importa al preprocesador. Su misión es interpretar adecuadamente la macro y expandirla en la cadena

```
(((esto no es C)) > ((esto tampoco)) ?
          ((esto tampoco)) :  (esto no es C))
```

El ámbito de una macro en C va desde su definición hasta el final del preprocesamiento. Existe, sin embargo, la posibilidad de anular la definición de una macro con la directiva siguiente:

```
#undef N
```

A partir de la aparición de esta directiva, la macro identificada por `N` ya no existe, pudiendo incluso volver a definirse otra vez con una descripción distinta.

¿Qué ocurre cuando definimos una macro que ya estaba definida?. En este punto el comportamiento de los preprocesadores puede ser variado. Los hay que producen un error, indicando que una macro no se puede redefinir, y los hay que anulan temporalmente la definición anterior y volverán a utilizarla en el momento en que se encuentren una directiva `#undef`. El siguiente ejemplo es muy ilustrativo:

```
... /* Aquí N no es una macro */
#define N 1
... /* Aquí N se expande con la cadena 1 */
#define N 2
... /* Aquí N se expande con la cadena 2 */
#undef N
... /* N volverá a expandirse con la cadena 1 */
#undef N
... /* Aquí N ha dejado de ser una macro */
```

Un problema aparte es el hecho de que no hemos impuesto restricción alguna sobre qué puede constituir la descripción de una macro, con lo cual el siguiente conjunto de definiciones es perfectamente válido:

```
#define A(x) ((x) + B(x))
#define B(x) (A(x))
```

En este ejemplo hemos realizado una definición indirectamente recursiva. Si en el texto fuente encontramos la cadena `A(1)`, deberíamos expandirla como `((1) + B(1))`, al encontrar `B(1)` deberíamos expandirla como `(A(1))`, y así sucesivamente *ad infinitum*. Habitualmente los preprocesadores tratan estos casos cortando la expansión de una macro en el momento en el que se corre riesgo de producirse un ciclo. Por lo tanto, si en el texto fuente encontramos `A(1)`, se expandiría por `((1) + B(1))`, y cuando encontrásemos `B(1)` se expandiría por `A(1)`. En este punto en que en la expansión de la macro `A` vuelve a aparecer ella misma, se para la expansión.

A.2.3 Compilación condicional

La compilación condicional permite compilar selectivamente ciertas partes del fichero de entrada. De esta forma un mismo fichero fuente puede dar lugar a distintas compilaciones dependiendo de expresiones constantes que deberán ser evaluadas por el preprocesador, y de si han sido definidas o no ciertas macros.

Para conseguir la compilación condicional, se utilizan unas estructuras (similares a las de flujo de control) cuya sintaxis es la siguiente:

```
<IF>
  texto_1
#elsif exp_1
  texto_2
#elsif exp_2
  texto_3
...
#else
  texto_n
#endif
```

La cabecera <IF> puede tener cualquiera de los tres formatos que se recogen a continuación:

- `#if` $\texttt{exp}_0$
- `#ifdef ident`
- `#ifndef ident`

En el anterior esquema, $\texttt{texto}_i$ representa cualquier cadena de texto, `ident` es cualquier identificador y $\texttt{exp}_i$ cualquier expresión `int` o `char` construida con cualquiera de los literales y operadores que proporciona el lenguaje C. Llamaremos a estas expresiones o identificadores *guardas*.

El funcionamiento de esta directiva es bastante sencillo: se evalúa la primera guarda (veremos más adelante cómo se hace esto) y si es cierta (un valor distinto a 0), se produce en salida $\texttt{texto}_1$, ignorando las restantes piezas de texto hasta encontrar la directiva `#endif`. Si la primera guarda es falsa, se evalúa la de la primera rama `#elsif`. Si es cierta, se produce como salida $\texttt{texto}_2$, ignorando las restantes piezas de texto de la estructura. Si es falsa se continua con este esquema de evaluación hasta encontrar la primera guarda que sea cierta. Si ninguna lo es se produce en salida el texto asociado a la directiva `#else`; si esta se omite, la directiva completa no produce nada en la salida del preprocesador.

Las guardas que son expresiones constantes se evalúan de acuerdo con las reglas de evaluación del lenguaje C. Los otros dos tipos se avalúan como se muestra a continuación:

- `#ifdef N` se considera cierto si y sólo si `N` es el nombre de una macro.
- `#ifndef N` se considera cierto si y sólo si `N` no es el nombre de una macro.

Aquellos que conocen el preprocesador de C, quizá estén acostumbrados a ver directivas como la siguiente:

```
#if defined(N)
  ...
#endif
```

Muy a menudo se considera que el efecto de esta directiva es equivalente al de:

```
#ifdef N
  ...
#endif
```

Pero no es cierto, en general. `defined` es una macro predefinida por el preprocesador de C que, a falta de una definición explícita por parte del usuario, produce un 1 cuando se aplica sobre un nombre de macro y un cero cuando se aplica sobre otro identificador. Pero es posible cambiar la definición manualmente. Por ejemplo:

```
#define defined(N) 1
```

con lo que se aplique sobre el argumento que se aplique, la expresión `defined(N)` siempre se expandirá por la expresión `1`.

Una de las utilidades más frecuentes de la compilación condicional es prevenir la inclusión múltiples veces del mismo fichero. Por ejemplo, supongamos que reunimos todos los prototipos de varias funciones de utilidad en el fichero de nombre `util.h`. Si escribimos este fichero de la forma siguiente:

```
/* util.h */

int f1(void);
char *f2(int);
```

y se incluye varias veces a través de diversos caminos obtendremos errores de redeclaración. La forma de solucionarlo es utilizando la compilación condicional, como se muestra a continuación:

```
/* util.h */

#ifndef __UTIL_H
#define __UTIL_H

int f1(void);
char *f2(int);

#endif
```

La primera vez que se incluye este fichero, la macro `__UTIL_H` no está definida, por lo que inmediatamente se define en la línea siguiente a `#ifndef __UTIL_H`. De esta forma la siguiente vez que se incluya este fichero, la macro ya estará definida y no se dará en salida ninguna línea de texto hasta encontrar la directiva `#endif`. Como resultado, no se obtienen errores de redeclaración.

A.2.4 Otras directivas del preprocesador

En esta sección veremos tres directivas más del preprocesador de C que son fundamentales y que no se eliminan del fuente de los ficheros tratados. Son las únicas directivas que llegan a la fase de análisis léxico del compilador y será éste el encargado de tratarlas.

Se enumeran a continuación:

1. Directiva `#line` para números de línea y nombres de fichero.
2. Directiva `#error` para dar mensajes de error.
3. Directiva `#pragma` para pasar opciones al compilador.

Números de línea

La directiva `#line` es de tremenda utilidad para que el compilador pueda dar mensajes de error en los lugares adecuados. Hemos indicado en la introducción que el analizador léxico, en presencia del preprocesador, no lee directamente del fichero de entrada sino de un fichero virtual transitorio. Supongamos que tenemos los dos ficheros siguientes:

A:
```
/* Fichero A */
x
#include "B"
y
```
B:
```
/* Fichero B */
a
b
c
```

Cuando el preprocesador trata el fichero `A`, produce en su salida un fichero virtual intermedio que tiene aproximadamente el siguiente contenido:

Temporal:
```
/* Fichero A */
x
/* Fichero B */
a
b
c
y
```

Supongamos que la línea `c` contiene un error. ¿Dónde indicará el compilador que se encuentra este error?. Es evidente que si nos indica que el error está en la línea 6 del fichero virtual temporal `/tmp/cpp12345`, no nos servirá de mucha ayuda. El compilador necesita saber a qué línea en sus ficheros de origen corresponde cada línea en el fichero virtual, de forma que pueda informar de los mensajes de error adecuadamente. Para ello se puede utilizar la directiva `#line` que admite cualquiera de los cuatro formatos siguientes:

- `#line n`
- `#line n nf`
- `# n`
- `# n nf`

Cuando el compilador lea una de estás directivas del fichero virtual deberá considerar que la línea que viene a continuación se corresponde con la número `n` del fichero de nombre `nf`. En el caso de omitir el nombre del fichero, se considerará que se trata de la línea `n` del último fichero referenciado en una directiva `#line`.

Usando esta directiva, el contenido del fichero virtual para el ejemplo anterior debería quedar como se indica a continuación:

```
Temporal:
# 1 "A"
/* Fichero A */
x
# 1 "B"
a
b
c
# 3 "A"
y
```

Con lo que ahora el compilador puede saber en cada momento a qué línea corresponde cada una de las que lee del fichero virtual intermedio.

Errores

La directiva de errores nos permite informar al compilador de que el texto fuente no está preparado para ser tratado por él. Por ejemplo, supongamos que hemos aislado en un fichero una porción de código muy delicada que tan sólo funciona cuando la versión del compilador que vamos a utilizar es superior a la 7. Generalmente los compiladores definen automáticamente una macro, `__version` que se expande al número de su versión.

La directiva `#error` tiene el formato

```
#error texto_de_error
```

y se puede combinar con la directiva `#if` para conseguir el efecto deseado:

```
#if __version > 7
  texto delicado
#else
  #error "Para compilar este fichero \
        necesita una versión del \
        compilador superior a la 7"
#endif
```

Opciones para el compilador

Es posible incluir en el texto fuente de los programas algunas opciones para el compilador. Por ejemplo, algunos compiladores emiten un mensaje de aviso cuando encuentran una función en la que alguno de sus parámetros no se ha utilizado, como se muestra en el ejemplo siguiente.

```
f(int a) {
  return 1;
}
```

El mensaje de aviso suele ser parecido al siguiente:

```
Aviso:  El parámetro 'a' no se usa en la función 'f'
```

Como en muchas ocasiones este mensaje no nos es de interés, los compiladores suelen incluir una opción, `argused`, para evitar que lo muestre. Esta opción se puede incluir en el texto fuente mediante una directiva `#pragma` con el siguiente aspecto:

```
#pragma argused
```

No existen opciones estandarizadas para los compiladores de C, por lo que esta directiva se deberá utilizar con cuidado, ya que si el compilador que estamos utilizando no la reconoce, simplemente la ignora.

Algunas macros predefinidas

Como ya sabemos, el preprocesador de C define automáticamente la macro `defined`. También predefine las siguientes:

- `__LINE`, cuando se utiliza se sustituye por el número de línea en que aparece.
- `__FILE`, se sustituye por el nombre del fichero en que aparece.
- `__TIME`, se sustituye por una cadena con la hora actual.
- `__DATE`, se sustituye por una cadena con la fecha actual.

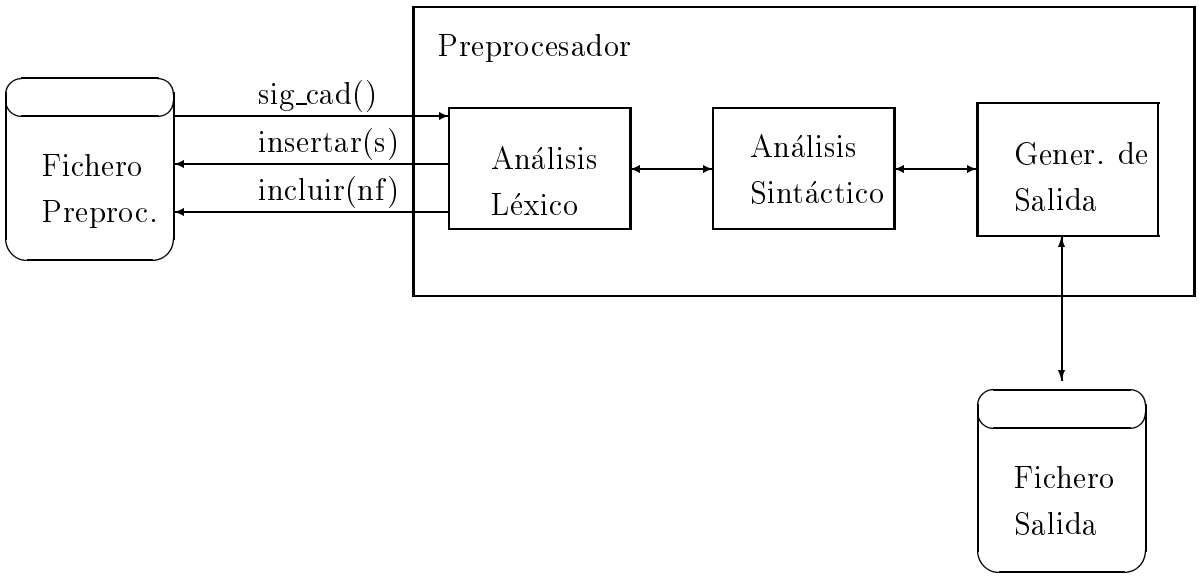
A.3 Implementación de un preprocesador de C

En este apartado estudiaremos los problemas que aparecen al abordar la implementación de un preprocesador de C y aportaremos algunas soluciones sencillas pero eficientes.

Comenzaremos mostrando un esquema general de bloques para el preprocesador y posteriormente trataremos el tema de cómo implementar cada uno de los grupos de directivas que hemos examinado con anterioridad utilizando ese esquema.

A.3.1 Esquema general

La estructura general del módulo preprocesador es la que se muestra en la siguiente figura:



Uno de los problemas fundamentales a los que cualquiera se enfrenta a la hora de construir un preprocesador es el hecho de que en algunas ocasiones es necesario incluir en la cadena o flujo de entrada cierto número de caracteres. Por ejemplo, cuando es necesario expandir un fichero referenciado en una directiva `#include` o cuando es necesario expandir una macro.

Por desgracia, la biblioteca estándar del lenguaje C, el que hemos elegido para explicar la implementación, no proporciona ningún modelo de fichero con capacidad para leer e insertar cadenas simultáneamente. Tendremos que modelarlo de forma explícita y lo hemos llamado *Fichero de Preprocesador*. Más adelante veremos de qué manera se podría implementar este tipo especial de fichero, pero de momento tan sólo estudiaremos sus características.

Los ficheros de preprocesador están siempre conectados con fichero fuente principal. Éste puede ser un fichero de texto de los que en C se representan utilizando el tipo `FILE *`. Para leer o devolver caracteres al mismo se pueden utilizar las funciones estándar de C `getc` y `ungetc`. Sobre este fichero base se construyen los ficheros del preprocesador que soportan las tres funciones principales siguientes:

1. `sig_cad()`, lee del fichero fuente y devuelve el prefijo más largo que o bien sea un identificador válido o bien no lo sea.

2. `insertar(s)`, inserta la cadena de caracteres `s` en el fichero del preprocesador, de forma que la siguiente llamada a `sig_cad()` consumirá estos caracteres.

3. `incluir(nf)`, inserta todo el contenido del fichero llamado `nf` en la posición actual de lectura. Los siguientes caracteres que devuelva la función `sig_cad()` serán los de este fichero. Cada vez que se incluye un fichero se toma nota de su nombre, de manera que si pretendemos volver a incluirlo recursivamente se produce un mensaje de error.

En C el prototipo de la implementación de este tipo abstracto de dato, incluyendo algunas funciones auxiliares de utilidad, podría tener el siguiente aspecto:

```
typedef ... FicheroPP;

FicheroPP abrir(const char *nf);
const char *sig_cad(FicheroPP fpp);
void insertar(FicheroPP fpp, const char *s);
void incluir(FicheroPP fpp, const char *nf);
int fin_fichero(FicheroPP fpp);
```

Las funciones indicadas son algunas de las que este tipo abstracto de dato podría tener. Tan sólo se trata de una sugerencia y la persona que implemente el preprocesador podría perfectamente necesitar de algunas funciones o procedimientos más. Lo que se ha mostrado es tan sólo una guía y veremos más adelante que no es suficiente para implementar satisfactoriamente un preprocesador.

A.3.2 Inclusión de ficheros

Lo primero que debemos hacer es determinar qué expresión regular describe el lenguaje de la directiva include. Para ello necesitamos de las siguientes definiciones regulares:

```
blanco      ([\ \t\f\v]|\\\n)
alm         (^{blanco}*"#"{blanco}*)
car         ([^\\\n]|\\(.|\n))
nom_f       ((\"([^\"]|\\\")*\")|(\<([^>]|\\\>)*\>))
id          ([a-zA-Z_][a-zA-Z0-9_]*)
nat         ([0-9]+)
```

`blanco` denota los caracteres que se consideran equivalentes al espacio en C, esto es: el espacio, el tabulador horizontal, el salto de página, el tabulador vertical y la secuencia barra invertida retorno de carro. `alm` es la expresión regular de la almohadilla con la que comienzan todas las directivas. Como vemos no es preciso que la almohadilla se coloque en la primera columna del texto, pero sí que se trate del primer carácter no blanco que aparece en una línea para que ésta se pueda considerar como una directiva del preprocesador. `nom_f` es la expresión regular de los nombres de ficheros encerrados entre comillas o ángulos, `id` la de los identificadores válidos en C y `nat` la de los números naturales.

Una vez estudiadas estas definiciones regulares de uso general, es posible escribir la de la directiva include tal y como se muestra a continuación:

```
include     {alm}"include"{blanco}+{nom_f}{car}*\n
```

Destacaremos el hecho de que en el estándar se admite cualquier secuencia de caracteres detrás del nombre del fichero. Por lo tanto, las directivas siguientes son equivalentes e igualmente correctas:

```
#include <stdio.h>     Entrada/salida básica
#include <stdio.h>     /* Entrada/Salida básica */
```

Una vez reconocida la aparición de esta directiva y aislado dentro de la misma el nombre del fichero a incluir, tenemos que conseguir que los siguientes caracteres que aparezcan en la salida del preprocesador sean el resultado de tratar los que se encuentran en el fichero referenciado por la directiva. Teniendo en cuenta las características de nuestro fichero especial de preprocesador, esto se reduce a una llamada a la rutina `incluir(nf)` proporcionando el nombre adecuado.

A.3.3 Las macros

Al igual que en la sección anterior, comenzaremos por escribir las expresiones regulares asociadas a esta directiva. Son las siguientes:

```
define      {alm}"define"{blanco}+{id}{param}?{def_m}?\n
undef       {alm}"undef"{blanco}+{id}{car}*\n
```

en donde `param` y `def_m` son las definiciones regulares de los parámetros formales de la macro y el texto de su definición.

```
param       ("()"|"("{blanco}*({id}{blanco}*"," \
                 {blanco}*)*{id}{blanco}*")")
def_m       ({car}*)
```

Las tareas fundamentales que el preprocesador tiene que realizar con respecto al tratamiento de las macros son:

1. Reconocer su definición, almacenarlas y anularlas.
2. Determinar dónde aparecen y expandirlas.

Estudiaremos cada una de estas tareas por turno.

Almacenamiento de la definición de una macro

Para almacenar las definiciones de las macros necesitaremos una estructura de datos que sea capaz de establecer qué macros están definidas y cuáles son sus definiciones. Esta estructura será manipulada exclusivamente por el preprocesador y la denominaremos *tabla de macros*. Su prototipo en C podría ser parecido al siguiente:

```
typedef ... Macro;
#define MACRO_NULA ...

Macro instala_macro(const char *nm, const char *par[],
                    const char *def);
Macro busca_macro(char *nm);
void borra_macro(Macro m);
const char *definicion(Macro m);
```

```
int numero_parametros(Macro m);
const char *expansion(Macro m, const char *par[]);
```

Con esta estructura de datos, cada vez que encontremos una directiva `#define` la instalaremos en la tabla y cada vez que aparezca una directiva `#undef`, la borraremos. Es interesante, como comentamos anteriormente, que al definir una macro por segunda vez, no se anule la primera definición y que ésta se pueda recuperar al utilizar la directiva `#undef`. Por lo tanto, cada entrada en la tabla no debe ser una definición de macro, sino una pila de definiciones en la que la única activa es la que ocupa la cima.

La constante `MACRO_NULA` puede ser el valor que devuelva la función `busca_macro(nm)` cuando no encuentra en la tabla ninguna macro de nombre `nm`.

Cómo guardar la definición de una macro es otro problema importante. Una posibilidad bastante evidente es guardarlas de la misma forma en que las escribe el usuario, esto es, literalmente. Este método tiene como ventaja el hecho de que guardar la definición es muy sencillo, pero utilizarla para expandir las macros resulta difícil, puesto que sería necesario analizarla carácter a carácter detectando en qué posiciones se encuentran los parámetros cada vez que expandamos la macro.

La segunda posibilidad es codificar los puntos en que aparecen los parámetros usando una secuencia especial de caracteres que resulte muy sencilla de identificar. Generalmente se elige una secuencia que empiece por un carácter especial (lo llamaremos $) y dos dígitos que indican cuál es el número de parámetro correspondiente. De esta forma, una macro como

```
#define MIN(a, b) ((a) > (b) ?  (b) :  (a))
```

se almacenaría utilizando el siguiente formato:

```
(($01) > ($02) ?  ($02) :  ($01))
```

Cada $ indica que los dos dígitos siguientes no forman parte de la macro y deben interpretarse como números de parámetros. Esta opción es mucho más eficiente que la primera puesto permite realizar la sustitución de parámetros de una forma mucho más simple.

Aún existe una tercera posibilidad que es bastante interesante puesto que permite expandir las macros de una forma mucho más rápida. Consiste en almacenar de la definición tan sólo aquellos caracteres que se deben volcar directamente en la salida, eliminando los nombres de los parámetros. De esta forma, la anterior macro se almacenaría así:

```
(() > () ?  () :  ())
```

Para saber en qué posición se debe insertar cada parámetro actual se utiliza una estructura auxiliar en forma de lista de pares $\{l, p\}$. l indica en número de caracteres que deben volcarse en la salida tomados literalmente de la definición de la macro. p el número del parámetro actual que se debe volcar a continuación. El número de parámetro 0 podría indicar que ha terminado la expansión de la macro. En el caso anterior, la lista asociada a la definición sería:

```
{ {2, 1}, {5, 2}, {5, 2}, {5, 1}, {2, 0} }
```

Esta lista significa que para expandir una macro como `MIN(a, b)` es preciso volcar los dos primeros caracteres de su representación en la salida y, a continuación, el primer parámetro actual. Con esto se obtiene:

```
((a
```

A continuación se vuelcan cinco caracteres de su representación y el segundo parámetro actual, obteniendo:

```
((a) > (b
```

El proceso continua hasta encontrar un parámetro con el número cero en la lista, o hasta que ésta se acabe.

El algoritmo de expansión

Ya sabemos cómo almacenar las macros y el único problema que nos queda por resolver es cómo expandirlas. Lo mejor es que el proceso de expansión sea anterior a cualquier otro análisis que realicemos del fuente, por lo que podríamos emplear un algoritmo como el que se muestra a continuación:

```
fpp ← abrir(nombre del fuente)
mientras no fin_fichero(fpp)
  c ← sig_cad(fpp)
  m ← buscar_macro(c)
  si c ≠ MACRO_NULA entonces
    pa ← leer los parámetros de c
    insertar(expansión(m, pa))
  si no
    dar en salida los caracteres de c
  fin si
fin mientras
```

Dada la forma especial en que hemos definido cuál es el comportamiento de la rutina `sig_cad()`, sabemos que ésta siempre devuelve el prefijo más largo que pueda localizar en la entrada que o bien sea un identificador o bien no lo sea. De esta forma, si en la entrada se encuentra una cadena como `1+a+b`, la troceará en las subcadenas `1+`, `a`, `+` y `b`, en las que los únicos candidatos a posibles nombres de macro son `a` y `b`.

En el caso en que la cadena devuelta por `sig_cad()` coincida con un nombre de macro previamente definido, lo expandiremos en un buffer intermedio (una cadena de caracteres de la longitud adecuada) y lo insertaremos en la entrada utilizando la función `insertar()` que ya conocemos.

Veremos a continuación un pequeño ejemplo para ilustrar el mecanismo de expansión de macros. Supongamos que el preprocesador se enfrenta a un fichero de entrada con el siguiente contenido:

```
#define UNO 1
#define DOS UNO + UNO
...
main()
{
  ...
  i = DOS;
  ...
}
...
```

Nos concentraremos en el análisis de la subcadena de entrada `i = DOS;`. Recogeremos la traza del preprocesador en una tabla en la que la primera columna muestra en contenido del fichero del preprocesador, la segunda columna la acción que se lleva a cabo, y la tercera la salida que se va produciendo.

Entrada	Acción	Caracteres dev.
"i = DOS;"	sig_cad() = "i"	"i"
" = DOS;"	sig_cad() = " = "	"i = "
"DOS;"	sig_cad() = "DOS" insertar("UNO + UNO")	"i = "
"UNO + UNO;"	sig_cad() = "UNO" insertar("1")	"i = "
"1 + UNO;"	sig_cad() = "1 + "	"i = 1 + "
"UNO;"	sig_cad() = "UNO" insertar("1")	"i = 1 +"
"1;"	sig_cad() = "1;"	"i = 1 + 1;"

Como se puede apreciar, gracias a las utilidades especiales que proporciona nuestro fichero de preprocesador, resulta muy sencillo expandir una macro. Pero hemos olvidado un problema que ya apuntamos a la hora de estudiar el preprocesador de C: ¿qué ocurre con las definiciones de macros recursivas?. Es evidente que si las tratamos de la forma en que hemos hecho en el anterior ejemplo, se producirían ciclos infinitos en caso de encontrarlas.

La solución a este problema pasa por modificar ligeramente nuestros requerimientos para el tipo abstracto de dato `FicheroPP`. Cada vez que insertemos en él una cadena deberemos indicar a la expansión de qué macro pertenece, y también necesitaremos de una nueva función de consulta que nos permita averiguar si una macro determinada está siendo o no expandida, para evitar ciclos.

El interface modificado del tipo abstracto de dato para tratar estas situaciones es el siguiente:

```
typedef ... FicheroPP;

FicheroPP abrir(const char *nf);
char *sig_cad(FicheroPP fpp);
void insertar(FicheroPP fpp, const char *s, const char *nm);
void incluir(FicheroPP fpp, const char *nf);
```

```
int fin_fichero(FicheroPP fpp);
int en_expansión(FicheroPP fpp, const char *nm);
```

Hemos modificado `insertar()` de forma que ahora además de indicar qué cadena queremos insertar podamos especificar a la expansión de qué macro corresponde. También hemos añadido la rutina `en_expansión()` que aplicada sobre un fichero de preprocesador y un nombre de macro `nm` determina si ésta está siendo expandida o no.

Utilizando este interface modificado, el algoritmo de expansión podríamos esquematizarlo de la forma que se muestra a continuación:

```
fpp ← abrir(nombre del fuente)
mientras no fin_fichero(fpp)
  c ← sig_cad(fpp)
  m ← buscar_macro(c)
  si c ≠ MACRO_NULA y no en_expansión(c) entonces
    pa ← leer los parámetros de c
    insertar(expansión(m, pa))
  si no
    dar en salida los caracteres de c
  fin si
fin mientras
```

A.3.4 Compilación condicional

Para poder implementar la compilación condicional, tal y como la hemos presentado anteriormente, tendremos que implementar un reconocedor y evaluador de expresiones constantes que nos servirá para determinar en tiempo de preprocesamiento qué parte de una estructura condicional hay que compilar, y por otro lado tenemos que implementar un reconocedor para el lenguaje asociado a las directivas de control que compruebe que cada directiva condicional termina con su correspondiente `#endif`, que las directivas `#else` no se utilizan fuera de lugar y que las expresiones que se utilizan en la directiva `#if` sean legales.

Desgraciadamente el lenguaje de estas directivas no es regular, por lo que para reconocerlas necesitaremos una herramienta más potente que las expresiones regulares y `lex`. Por lo tanto trataremos este tema en un capítulo posterior. De momento tan sólo podemos mostrar las expresiones regulares de las directivas.

```
if          {alm}"if"{car}+\n -- Ojo este lenguaje no es regular
ifdef       {alm}"ifdef"{blanco}+{id}{car}*\n
ifndef      {alm}"ifndef"{blanco}+{id}{car}*\n
elsif       {alm}"elseif"{car}*\n
else        {alm}"else"{car}*\n
endif       {alm}"endif"{car}*\n
```

A.3.5 Otras directivas

Sabemos que el preprocesador reconoce las directivas `#line`, `#error` y `#pragma`, pero no hace nada con ellas, por lo que tan sólo debemos preocuparnos de sus expresiones regulares. Son las siguientes:

```
line      {alm}("line"{blanco}+)?{nat}({blanco}+{nom_f})?{car}*\n
error     {alm}"error"{blanco}+{car}\n
pragma    {alm}"pragma"{blanco}+{car}\n
```

A.3.6 Implementación de `FicheroPP`

Abordaremos ahora el problema de cómo se realiza una implementación eficiente y sencilla de este tipo abstracto de dato que tan útil nos ha resultado.

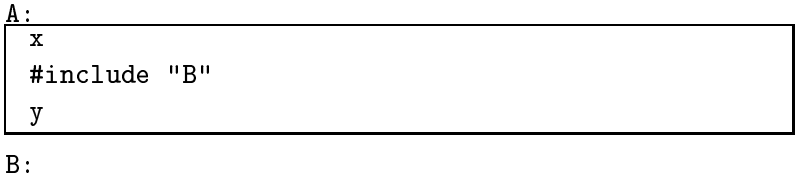
La mejor forma de implementarlo es utilizando una pila de elementos con la siguiente estructura:

```
typedef struct {
  FILE *f;
  enum {FICHERO, EXPANSION} tipo;
  char *n;
} elem_pila;
```

`f` será un puntero a un fichero estándar de C. Este puntero servirá para referenciar el fichero principal que se está tratando (el que se encontrará siempre en la base de la pila), un fichero de inclusión, o bien un fichero temporal que nos servirá para expandir macros. `tipo` es un enumerado que nos indica si un cierto elemento de la pila se corresponde con un fichero real o con uno para expandir macros, y `n` es el nombre de la macro que se está expandiendo en esta entrada de la pila o bien el nombre del fichero de inclusión que se está tratando en la misma.

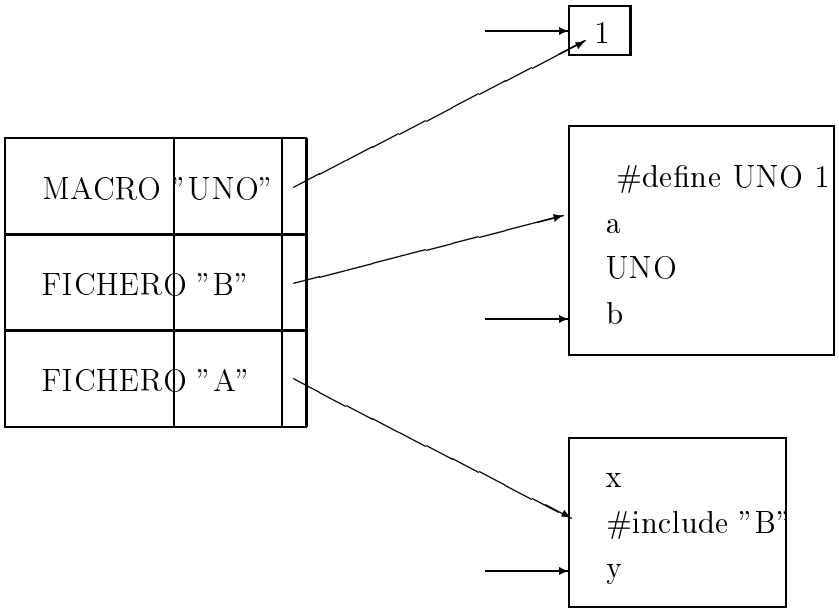
La rutina `sig_cad()` leerá siempre del fichero en la cima de la pila. La rutina `insertar()` lo que hará será abrir un nuevo fichero temporal, en el que volcará la cadena que se le pasa, y lo apilará en la cima de la pila, de forma que la siguiente llamada a `sig_cad()` consumirá estos caracteres antes que continuar con el fichero inmediatamente anterior en la pila. La rutina `incluir()` actuará de una manera similar, abriendo el fichero indicado, creando un nodo apropiado y colocándolo en la cima de la pila.

Como ejemplo gráfico podríamos ver en qué estado se encuentra esta pila cuando estamos leyendo la línea marcada con un asterisco en el siguiente trozo de código:



```
#define UNO 1
a
UNO *
b
```

El estado de la pila será el que se muestra a continuación:



En él se ve como el fichero principal es `A`, que el segundo se corresponde con el fichero de inclusión `B`, y que en la cima se encuentra el fichero temporal en el que estamos expandiendo la macro `UNO`. Las flechas al lado de los ficheros indican la posición del puntero de lectura de los mismos.

Este método de implementación es muy simple y mucho más homogéneo desde el punto de vista de su tratamiento que una estructura de buffers y ficheros mezclados, y además resulta muy eficiente. Alguien podría argumentar que abrir un fichero temporal para cada macro que expandamos puede resultar excesivamente costoso, pero no debemos olvidar que el la biblioteca del lenguaje C garantiza que todos los ficheros accedidos a través de estructuras del tipo `FILE *` tienen un buffer intermedio que minimiza el número de operaciones de entrada/salida reales que se deben llevar a cabo. Por lo tanto, en la mayoría de los casos, al expandir una macro lo que estaremos haciendo será escribir realmente en ese buffer intermedio, con lo que no se producirá ninguna entrada/salida real a disco. Por otra parte, este esquema nos permite expandir macros grandes sin necesidad de consumir gran cantidad de memoria principal, que puede ser mucho más útil para otros componentes del compilador.

A.4 Enlace con el análisis léxico

Para que el preprocesado sea realmente una etapa previa al análisis léxico, tenemos que asegurarnos de que existe una total independencia entre las dos fases, haciendo que el análisis léxico lea caracteres de un fichero virtual e ignorando por tanto los mecanismos de inclusión de ficheros, compilación condicional y expansión de macros. Esta transparencia se puede conseguir de dos formas distintas:

1. Implementando el preprocesador como un proceso separado y completamente aislado del compilador.

2. Implementándolo como un módulo de programa que se enlaza dentro del mismo ejecutable con el resto de los componentes del compilador

En las máquinas actuales, ambas formas son igual de eficientes, pero si tuviéramos que decantarnos por una elegiríamos la primera, puesto que en ella existe una separación mayor entre el preprocesador y el compilador, lo que nos permitirá desarrollarlos y ampliarlos por separado sin que uno interfiera con el otro.

En el primer caso, el preprocesador y el analizador léxico del compilador se comunicarán a través de un fichero virtual (un tubo en UNIX, por ejemplo), y en el segundo a través de las rutinas convenientes de lectura y devolución de carácter.

A.5 Revisión del preprocesador

Cuando tratamos el tema del preprocesador, indicamos que sólo utilizando expresiones regulares es imposible implementarlo de una forma sencilla y fácilmente modificable. Además vimos que ciertas partes del lenguaje del preprocesador no tenían una gramática regular, sino estrictamente de contexto libre, por lo que tendríamos que utilizar también la herramienta `yacc`.

En este tema trataremos todos los detalles que ya conocemos, otros que se nos quedaron en el tintero y explicaremos de qué forma es posible solucionarlos satisfactoriamente.

A.6 El léxico del preprocesador

Como todo lenguaje, el del preprocesador cuenta con un léxico bastante bien definido y rudimentario, puesto que tan sólo incluye la almohadilla, los nombres de las directivas, los identificadores y algunos elementos más.

El principal problema al que nos enfrentamos a la hora de especificar el léxico es el hecho de que las mismas cadenas de entrada pueden tener significados diferentes, en función a si nos encontramos en el entorno de una directiva o no. Por ejemplo, la cadena `123` no tiene significado alguno para el preprocesador cuando la lee en la definición de una macro, pero sí cuando la lee detrás de una directiva `#if`. En este último caso debe interpretarla como el número natural *123*, que sabemos para el preprocesador tiene el significado cierto en este contexto.

Otros dos problemas importantes están en relación con la forma en que se deben tratar los caracteres blancos y la forma en que se deben expandir las macros, que puede entrar en conflicto con la técnica de análisis de lenguajes que estemos utilizando.

El objetivo de esta sección es mostrar al lector dónde aparecen estos problemas, en qué condiciones y ofrecerle algunas ideas para su solución.

A.6.1 Dependencias del contexto en el léxico

El hecho de que una misma cadena de caracteres pueda tener significados distintos dentro del mismo texto, nos indica que en el léxico del preprocesador existen dependencias del contexto. Esto nos sugiere la idea de utilizar varios analizadores léxicos simultáneamente, dependiendo del contexto en el que nos encontremos, y ya conocemos que la forma de hacerlo es utilizando entornos léxicos.

En principio, esta idea puede parecer bastante buena, pero si la desarrollamos veremos que cada una de las directivas tiene dos o tres dependencias del contexto. Por ejemplo, dentro del ámbito de `#include` tan sólo cambia el significado de las cadenas de caracteres encerradas entre comillas dobles o entre ángulos, dentro de `#define` cambia el de los paréntesis, el de la coma y el de los identificadores en la parte que describe el nombre de la macro y sus parámetros, pero no en la parte de definición, etcétera.

Por lo tanto, el número de dependencias del contexto existentes dificultaría enormemente el desarrollo y penalizaría la eficiencia de nuestro analizador léxico. La mejor alternativa a este problema es considerar que todos los patrones que forman parte del léxico del preprocesador son significativos independientemente del contexto en que nos encontremos. Después, en la gramática, podremos deshacer las dependencias de una forma muy sencilla.

El léxico del preprocesador sería el siguiente:

Caracteres e identificadores

```
blanco      ([\ \t\f\v]|\\\n)
blancos     ({blanco}+)
alm         (^{blanco}*"#"{blanco}*)
car         ([^\\\n]|\\(.|\n))
id          ([a-zA-Z_][a-zA-Z0-9_]*)
```

Caracteres de puntuación

```
par_ab      "("
par_ce      ")"
coma        ","
retorno     "\n"
```

Literales

```
nom_f       ({cadena}|(\<([^>]|\\\>)*\>))
cadena      ((\"([^\"]|\\\")*\"))
lit_nat     ({dig_dec}+)
lit_carac   (\'({car_txt}|{car_oct}|{car_hex})\')
car_txt     (.|\.)
car_oct     (\{dig_oct}{dig_oct}?{dig_oct}?)
car_txt     (\x{dig_hex}{dig_hex}?)
dig_oct     ([0-7])
dig_hex     ([0-9a-fA-F])
dig_dec     ([0-9])
```

Operadores

```
or          "||"
and         "&&"
```

```
menor       "<"
...         ...
negación    "!"
```

Nombres de directivas

```
include     "include"
define      "define"
...         ...
pragma      "pragma"
```

Como en otras ocasiones supondremos que los *tokens* asociados a cada definición regular se identifican mediante el mismo nombre de la definición, pero escrito en mayúsculas. Por simplicidad, también supondremos que los *tokens* de las definiciones literales se pueden identificar a través de esa definición literal. Por ejemplo: ALM, LIT_NAT, '<', '||', etcétera.

A.6.2 Los espacios en blanco

El lenguaje del preprocesador es muy sensible a los espacios en blanco, pues hay ciertos lugares en que los ignora y otros en los que son fundamentales. Por ejemplo, en las expresiones que siguen a la directiva `#if` se ignoran, pero no así detrás del primer identificador que aparece en una directiva `#define`. Los dos ejemplos siguientes nos ayudarán a comprender esto mejor:

- `#define MIN(a, b) ((a) > (b) ? (b) : (a))`.
- `#define MIN (a, b) ((a) > (b) ? (b) : (a))`.

En el primer caso, `MIN` es una macro que acepta dos parámetros. En el segundo `MIN` es una macro sin parámetros que se expande cada vez que se encuentra por la cadena literal `(a, b) ((a) > (b) ? (b) : (a))`. La razón de esto es que para que la lista de parámetros se interprete como tal, el paréntesis de inicio debe estar inmediatamente detrás del identificador de la macro. Si no es así, se considera que es una macro sin parámetros y que todo lo que se haya escrito a partir del primer carácter no blanco es su definición.

Esta dependencia del contexto se puede resolver usando entornos léxicos, pero el resultado es un analizador bastante complejo e ineficiente. Lo más adecuado es dotar al analizador léxico con una rutina de nombre `ignorar_blancos(int)` que toma un parámetro booleano y decide si los blancos se deben ignorar o devolver como el *token* `BLANCOS`. Por defecto supondremos que después de encontrar una almohadilla se ignorarán los blancos, puesto que esto es lo normal en casi todas las directivas, pero en el analizador sintáctico podremos cambiar este comportamiento cuando lo necesitemos. También supondremos que tras la lectura de un retorno lo espacios volverán a ser significativos.

A.6.3 Expansión de las macros

Expandir las macros es un problema bastante complejo, puesto que en ocasiones puede interaccionar de forma incorrecta con la técnica de reconocimiento de lenguajes con la que estemos trabajando.

En principio, la alternativa más sencilla podría ser recoger la expansión de las macros dentro de la sintaxis del lenguaje del preprocesador, incluyendo algunas reglas de producción de la forma:

```
texto
   : llamada
   | ... otras cosas ...

llamada
   : ID                              { Expandir ID si es una macro }
   | ID '(' params_actuales ')'      { Ídibem }

params_actuales
   : ...
```

Esta posibilidad es una de las más sencillas y además nos permite detectar con facilidad errores de balanceo de paréntesis, pero presenta un inconveniente importante cuando el analizador utiliza un *token* de lectura adelantada o *lookahead*. Es el caso de los analizadores LL(1) y LALR(1), que son los más utilizados.

El problema se presenta a la hora de insertar la expansión de una macro en el fichero de entrada. Dado que el analizador siempre lleva un *token* de lectura adelantada para deshacer indeterminaciones, en el momento en que se produce la inserción efectiva del texto de la macro se hará siempre un *token* después de lo que corresponde. Este problema quedará más claro al examinar un ejemplo. Sea un fichero de entrada con el siguiente contenido:

```
A:
#define MSG "hola"
...
s=MSG;i=1; *
...
```

Examinaremos detenidamente la secuencia de acciones que un parser LALR(1), como los que genera `yacc`, llevaría a cabo al leer la línea marcada con el asterisco. Esta se descompone en la secuencia de *tokens* `ID`, `'='`, `ID`, `';'`, `ID`, `'='`, `'1'` y `';'`.

Entrada	Token actual	*Lookahead*	Acción
s=MSG;i=1;	ID = s	'='	salida("s")
MSG;i=1;	'='	ID = MSG	salida("=")
;i=1;	ID = MSG	';'	insertar("hola")
"hola"i=1;	';'	"hola"	...
...	...	...	...

En esta traza queda patente que dado que el parser lleva siempre un *token* de lectura adelantada, cuando insertamos la expansión de una macro lo haremos siempre después del carácter de *lookahead*, lo que producirá un comportamiento anómalo del preprocesador bastante difícil de depurar. En este caso, la salida que el preprocesador produciría sería la siguiente:

```
$$:
# 1 "A"
...
s=;"hola"i=1;
...
```

El error es aún más grave cuando la macro a expandir está justo al final del fichero de entrada, pues en este caso el *token* de lectura adelantada será el que indica el final del fichero. Por lo tanto, sea lo que sea lo que insertemos en el fichero de entrada a partir de ese momento, el parser ya no lo leerá.

Siempre que utilicemos un analizador basado en la lectura adelantada de *tokens* para deshacer ambigüedades [1] tendremos que expandir las macros antes de que el parser lea su *lookahead*. En la práctica la mejor forma de hacerlo es expandiendo las macros en el analizador léxico, utilizando el algoritmo que vimos en la sección ??.

A.7 La sintaxis del preprocesador

En esta sección ofreceremos una gramática BNF para el lenguaje del preprocesador. Supondremos que las macros son expandidas por el analizador léxico de la forma que hemos comentado antes y que es posible ignorar o tener en cuenta los blancos según convenga. Para ello utilizaremos dos reglas especiales con la siguiente estructura:

```
igb
   : λ              { ignorar_blancos(TRUE); }

no_igb
   : λ              { ignorar_blancos(FALSE); }
```

Las dos reglas son λ-producciones, por lo tanto se reducen inmediatamente cada vez que aparecen, instruyendo al analizador léxico para que ignore o no los blancos. Comentaremos con detalle los lugares de la gramática en los que se utilizan.

A.7.1 Estructura general

En las secciones siguientes iremos mostrando por bloques la sintaxis completa del lenguaje del preprocesador. En esta sección tan sólo mostraremos los constructores de más alto nivel en el lenguaje.

```
fuente
   : fuente línea
   | λ
```

[1] Existen analizadores como los LR(0) o los SLR(0) que no necesitan llevar un *token* de lectura adelantada, pero en la práctica son poco útiles pues el conjunto de gramáticas que permiten reconocer es bastante restringido.

```
línea
   : literales '\n'
   | ALM directiva '\n'

literales
   : literales literal
   | literal

literal
   : BLANCOS
   : CAR
   | ID
   | NOM_F
   | ...
   | '!'

comentarios
   : literales
   | λ

directiva
   : include | define | undef | if | line | error | pragma
```

Indicamos en una sección anterior que este lenguaje tiene varios problemas de dependencia del contexto en el léxico. La forma en que los hemos resuelto ha sido creando un símbolo no terminal, `literales`, que absorbe donde sea preciso secuencias de blancos, identificadores, nombres de ficheros, cadenas, literales, etcétera, sin darles el significado especial que tienen estos elementos dentro del ámbito de las directivas. De esta forma, en aquellos lugares en los que estos *tokens* carezcan de significado los reduciremos gracias a este no terminal, pero sin perder la posibilidad de utilizarlos individualmente allá donde sea preciso.

`comentarios` es similar a `literales`, pero permite la posibilidad de producir la cadena vacía λ. Esta producción se usará más adelante para absorber los literales que los usuarios pueden escribir libremente al final de muchas directivas.

También hemos añadido la regla necesaria para clasificar todas las directivas del preprocesador.

A.7.2 La directiva `#include`

Esta es una de las directivas más sencillas de describir.

```
include
   : INCLUDE NOM_F comentarios
```

Fíjese en que en esta regla no hemos tenido en cuenta los espacios en blanco que pueda haber en el fuente entre los *tokens* `INCLUDE` y `NOM_F`. Podemos hacerlo así puesto que como

indicamos anteriormente, cada vez que el analizador encuentra una almohadilla en el texto de entrada ignora automáticamente todos los blancos que aparezcan hasta encontrar el siguiente retorno de carro. Por lo tanto, podemos definir la gramática de esta directiva sin preocuparnos por la presencia de blancos en el fuente.

A.7.3 Las directivas `#define` y `#undef`

A continuación se muestra la sintaxis para las directivas `#define` y `undef`. En la primera de ella es necesario tener en cuenta los blancos entre el nombre de la macro y el primer paréntesis que aparezca detrás.

```
define
   : DEFINE no_igb ID BLANCOS no_igb descripción
   | DEFINE no_igb ID igb '(' params_form ')' no_igb descripción
   | DEFINE no_igb ID igb '(' ')' no_igb descripción

descripción
   : literales
   | λ

params_form
   : params_form ',' ID
   | ID

undef
   : UNDEF ID comentarios
```

En este caso, en el momento en que reconocemos el *token* `DEFINE` reducimos la regla `no_igb` con lo que se comenzará a tener en cuenta los espacios en blanco. De esta forma podremos distinguir fácilmente si lo que viene detrás forma parte de la descripción o de la lista de parámetros formales de la macro.

Salvo en este caso, no nos volveremos a encontrar con situaciones de este tipo en ninguna otra directiva, por lo que la solución tomada parece bastante adecuada.

A.7.4 Las directivas `#if`, `#ifdef`, etcétera

Las directivas de compilación condicional tienen una gramática que es estrictamente de contexto libre, por lo que resulta muy difícil de implementar en `lex`, al menos de una manera sencilla y fácilmente modificable.

Los problemas habituales con respecto a estas directivas son el hecho de que se exige que las expresiones que sirven de guarda a `#if` sean expresiones constantes correctas en el lenguaje C, que cada `#if` debe terminar con su correspondiente `#endif` y que no se permite la aparición de directivas `#else` o `#elsif` fuera de este ámbito. Como veremos en la gramática todos estos problemas se resuelven de una manera muy simple al utilizar `yacc`.

```
if
   : cab_if
       fuente
       ramas_elsif
       rama_else
     ALM ENDIF

cab_if
   : IF exp_cte comentarios
   | IFDEF ID comentarios
   | IFNDEF ID comentarios

ramas_elsif
   : ramas_elsif rama_elsif
   | λ

rama_elsif
   : ALM ELSEIF exp_cte comentarios '\n' fuente

rama_else
   : ALM ELSE comentarios '\n' fuente
   | λ
```

Expresiones constantes en la directiva `#if`

En esta sección mostraremos la gramática de las expresiones constantes que determinan el comportamiento de las directivas `#if`. La gramática no es ambigua y en ella la precedencia y asociatividad de los operadores se recoge de forma implícita en las reglas de producción.

```
exp_cte
   : exp_or

exp_or
   : exp_or '||' exp_and
   | exp_and

exp_and
   : exp_and '&&' exp_rel
   | exp_rel

exp_rel
   : exp_rel op_rel exp_adit
   | exp_adit

op_rel
   : '<' | '>' | '<=' | '>=' | '==' | '!='

exp_adit
   : exp_agit op_adit exp_mult
```

```
   | exp_mult

op_adit
   : '+' | '-'

exp_mult
   : exp_mult op_mult exp_base
   | exp_base

op_mult
   : '*' | '/' | '%' | '<<' | '>>' | '^' | '&' | '|'

exp_base
   : LIT_NAT
   | LIT_CAR
   | '(' exp ')
   | op_un exp_base

op_un
   : '+' | '-' | '!'  | '~'
```

A.7.5 Directivas `#line`, `#error` y `#pragma`

Terminamos esta revisión que hemos hecho de la implementación del preprocesador mostrando el último grupo de directivas, en el que hemos incluido todas aquellas que se deben reconocer pero ignorar.

```
line
   : LINE LIT_NAT NOM_F comentarios
   | LINE LIT_NAT comentarios
   | LIT_NAT NOM_F comentarios
   | LIT_NAT comentarios

error
   : ERROR no_igb literales

pragma
   : PRAGMA no_igb literales
```

Apéndice B

Implementación de tablas de símbolos

La tabla de símbolos proporciona un mecanismo para asociar valores (atributos) con nombres. Estos atributos son una representación del significado (semántica) de los nombres a los que están asociados, en este sentido, la tabla de símbolos desempeña una función similar a la de un diccionario. La tabla de símbolos aparece como una estructura de datos especializada que permite almacenar las declaraciones de los símbolos, que sólo aparecen una vez en el programa, y acceder a ellas cada vez que se referencie un símbolo a lo largo del programa, en este sentido la tabla de símbolos es un componente fundamental en la etapa del análisis semántico ya que será aquí donde necesitaremos extraer el significado de cada uno de los símbolos para llevar a cabo las comprobaciones necesarias.

Plantearemos el estudio de las tablas de símbolos intentando separar claramente los aspectos de especificación y los aspectos de implementación, de esta forma a través de la definición de una interface conseguiremos una visión abstracta de la tabla de símbolos en la que sólo nos preocuparemos de definir claramente las operaciones que la manipularán sin decir como se van a efectuar. Desde el punto de vista de la implementación, comentaremos diversas técnicas de distinta complejidad y eficiencia, lo que nos permitirá tomar decisiones con respecto a la implementación a elegir en función de las características del problema concreto que abordemos.

Definiremos la tabla de símbolos como un tipo abstracto de datos, es decir un tipo que contiene una serie de géneros y una serie de operaciones sobre esos géneros. A la hora de definir la interface debemos especificar los géneros utilizados, que van a ser conjuntos de valores (que en C se implementarán mediante tipos básicos o tipos definidos por el programador) y por otro lado debemos especificar el conjunto de operaciones que se aplicarán sobre dichos géneros, asociando una signatura o molde a cada nombre de operación que nos dirá que valores toma y que valores devuelve.

En muchos casos será necesario disponer de varias tablas de símbolos con la idea de almacenar de forma separada grupos de símbolos que tienen alguna propiedad en común (por ejemplo todos los campos de un registro o todas las variables de un procedimiento), de esta forma nuestra tabla de símbolos será una estructura que será capaz de albergar varias tablas, que estarán compuestas a su vez por una serie de entradas (una por cada símbolo que contenga la tabla) donde cada entrada se caracterizará por el lexema que representa al símbolo y por los atributos que representan el significado del símbolo.

```
entrada 1.1
entrada 1.2
    .
    .
    .
entrada 1.m

  tabla 1          tabla 2       . . .       tabla n

                   TABLA DE SIMBOLOS
```

Con este esquema de tabla de símbolos, los géneros necesarios serán *Tabla* para representar una tabla de símbolos, *Entrada* para representar una entrada dada de una tabla de símbolos, *Cadena* que será una cadena de caracteres que representará el lexema asociado a un símbolo y *Atributos* que será un registro donde cada uno de sus campos representará uno de los atributos del símbolo. En el caso de los atributos de un símbolo también se podría plantear la manipulación de cada uno de los atributos individualmente sin tener por que referenciarlos a todos, como ya veremos, esto influirá en la definición de las operaciones de recuperación y actualización de atributos.

La manera de representar los géneros en C será a través de la definición de tipos, en el siguiente fragmento de declaración en C representaremos la definición de estos géneros dejando en puntos suspensivos los detalles correspondientes a la implementación que no deben concretarse al hablar de interface.

```
typedef ... Cadena;
typedef ... Tabla;
typedef ... Entrada;
typedef ... Atributos;

#define TABLA_NULA ...
#define ENTRADA_NULA ...
```

Las constantes *TABLA_NULA* y *ENTRADA_NULA*, serán dos valores necesarios para detectar irregularidades acontecidas al aplicar ciertas operaciones sobre tablas de símbolos.

Veamos ahora cuales son las operaciones necesarias para manipular adecuadamente la estructura de datos que nos hemos planteado, en primer lugar comentaremos las funciones de creación y destrucción de tablas de símbolos. La función *crea_tabla* se encarga de crear la estructura de una nueva tabla de símbolos con todas sus entradas vacías devolviendo *TABLA_NULA* si se ha producido algún error en la creación de la nueva tabla. La función *destruye_tabla* se encarga de eliminar la estructura de una tabla de símbolos existentes despues de borrar todas sus entradas, los prototipos de estas funciones son los siguientes:

```
Tabla crea_tabla(void);
void destruye_tabla(Tabla);
```

El siguiente grupo de funciones se encarga de la gestión de entradas dentro de una tabla de símbolos. La función *busca_simbolo* busca un determinado lexema dentro de una tabla de símbolos devolviendo la entrada coorespondiente o *ENTRADA_NULA* si dicho símbolo no existe. La función *instala_simbolo* crea una entrada para un nuevo símbolo en una tabla dada devolviendo la referencia a la entrada o *ENTRADA_NULA* si se produce algún error en la creación de la nueva entrada. La función *borra_simbolo* se encarga de eliminar la entrada de un símbolo, los prototipos de estas funciones son:

```
Entrada busca_simbolo(Cadena, Tabla);
Entrada instala_simbolo(Cadena, Tabla):
void borra_simbolo(Entrada);
```

Algunas de las operaciones vistas anteriormente son implementaciones de las operaciones definidas para el tipo mapa. Pero esta implementación tiene en cuenta el tratameinto de errores y los correspondientes mensajes de error.

Las operaciones de manipulación de atributos se van a encargar de actualizar y recuperar los valores de los atributos de un determinado símbolo. Tenemos dos poibilidades, la primera es la de utilizar el género *Atributos* y disponer de dos funciones *actualiza_atributos* y *recupera_atributos* que tratan en bloque a todo el registro de atributos, debiéndose definir operaciones para construir y disgregar elementos del género *Atributos*:

```
void actualiza_atributos(Entrada, Atributos);
Atributos recupera_atributos(Entrada);
```

La segunda posibilidad de manipular los atributos no hace referencia al género *Atributos* definiéndose varias parejas de funciones que traten individualmente cada uno de los atributos, indicando el nombre y el tipo del atributo concreto:

```
void actualiza_<nombre>(Entrada, <tipo>);
<tipo> recupera_<nombre>(Entrada);
```

Con el conjunto de funciones planteado, disponemos de una interface para manipular varias tablas de símbolos, capaces de albergar varias símbolos y asociarles a cada uno de ellos un conjunto de atributos. Esta interface constituye un buen punto de partida y contiene las funciones necesarias para resolver los problemas planteados a la hora de procesar un lenguaje simple, sin embargo puede que en algunos casos necesitemos gestionar otras características adicionales propias del lenguaje a procesar. Podemos pues considerar la interface hasta ahora propuesta como el núcleo de cualquier tabla de símbolos que se podrá ampliar en el sentido impuesto por las características específicas de cada lenguaje.

B.1 Técnicas básicas de implementación

En esta sección vamos a proponer distintas técnicas de implementación para la interface anterior. Estas técnicas de implementación van a tener distintos grados de complejidad y distintos niveles de eficiencia, el hecho de escoger una u otra será una consideración de diseño y dependerá en gran medida del número de símbolos que se pretenda manejar, así presentaremos implementaciones simples como las secuencias no ordenadas, que

constituyen un solución aceptable para un número pequeño de símbolos y otras implementaciones más complejas como las tablas hash, pensadas para aplicaciones de mayor envergadura. Es de destacar la relevancia de la eficiencia de la implementación, fundamentalmente en las operaciones de búsqueda, ya que un programa puede contener un gran número de identificadores, y por lo tanto se pueden efectuar una gran cantidad de búsquedas al procesarlo.

B.1.1 Secuencias no ordenadas

Es la solución más simple y fácil de implementar, en contrapartida su eficiencia deja mucho que desear, al ser el algoritmo de búsqueda de orden n^2 en el tiempo. Podemos representarlas mediante una tabla o mediante una lista enlazada de punteros, la ventaja de la lista enlazada es que no se limita a priori el número máximo de símbolos, aunque no es aconsejable que sea muy alto.

B.1.2 Secuencias ordenadas

Incorpora una mejora con respecto al método anterior en el caso de la búsqueda, aunque el algoritmo de inserción se complica un poco. También tenemos la posibilidad de representarlas mediante tablas y listas enlazadas. En la implementación mediante tablas podemos efectuar una búsqueda dicotómica, con lo que la complejidad en el tiempo sería de orden $log_2(n)$, sin embargo el algoritmo de inserción es más costoso y mantenemos la limitación de símbolos a priori. En la implementación mediante listas enlazadas el algoritmo de inserción es más simple pero no podemos efectuar la búsqueda dicotómica, en este caso el algoritmo de búsqueda es de orden n. Las secuencias ordenadas son una buena solución para tablas de un tamaño mediano y en concreto su implementación mediante tablas constituye una buena solución para tablas donde no se realizan operaciones de inserción, como por ejemplo las tablas de palabras reservadas.

B.1.3 Arboles binarios ordenados

Combina las ventajas de las implementaciones mediante tablas y listas enlazadas de las secuencias ordenadas, ya se puede implementar un algoritmo de búsqueda dicotómica, y por otro lado se tiene la flexibilidad de tamaño y facilidad de inserción de las estructuras de datos enlazadas. La complejidad tanto del algoritmo de inserción como del de búsqueda es de orden $log_2(n)$ en el caso medio, aparece un pequeño problema para algunas secuencias de llegada de símbolos que hacen que el árbol no esté totalmente nivelado (por ejemplo si los símbolos llegan ordenados alfabéticamente se generaría un árbol binario desvirtuado en el que sólo se ocuparían las partes derechas de los árboles) y la búsqueda dicotómica pierda efectividad, una solución a este problema es utilizar un algoritmo de inserción que asegure que los árboles que se producen sean balanceados.

B.1.4 Tablas *hash*

Este método es el más usado ya que asegura que el algoritmo de búsqueda se efectúa en un tiempo prácticamente constante. La idea del método *hash* (término que en español

significa *embrollo*) es la de hacer corresponder el conjunto de todos los posibles nombres que puedan aparecer en un programa con un número fijo p de posiciones en la tabla de almacenamiento, esta correspondencia se formaliza con la función h, en un principio lo deseable es que esta función fuese inyectiva, es decir que dos nombres no tuviesen la misma posición asignada, pero eso implicaría que el conjunto de posiciones fuese al menos tan grande como el conjunto de nombres posibles. En lo referente a la implementación se debería habilitar espacio para todos los posibles nombres y sólo se utilizarían algunos, lo que hace esta solución inviable. Realmente lo que se hace es proponer un método para resolver adecuadamente los casos en los que dos o más identificadores tienen la misma posición asociada.

La efectividad del método hash depende en gran medida de la función h elegida, una buena función h debe cumplir las siguientes condiciones:

- La función h sólo dependerá del nombre n
- Debe ser fácilmente calculable
- Debe distribuir uniformemente los nombres en las posiciones disponibles

Ejemplos de funciones h pueden ser $(c_1 + c_2 + ... + c_k)\ mod\, p$ donde c_i es un número asociado al carácter i-ésimo del nombre n (por ejemplo su código ASCII), o $(c_1 * c_2 * ... * c_k)\, mod\, p$, en este tipo de funciones la uniformidad en la distribución depende del número p, ciertos números primos aseguran esa propiedad.

Una vez elegida la función h, el otro factor que influye en la eficiencia es el método elegido para la resolución colisiones, una colisión se presenta cuando al intentar insertar un nombre n_i en la tabla existe ya otro nombre n_j tal que $h(n_i) = h(n_j)$, en este caso se debe buscar una ubicación al nombre n_i, veamos dos métodos:

- Buscar una nueva posición, aplicando una nueva fución o la misma función h modificada, por ejemplo si $h(n)$ está ocupada probar en $(h(n)+1)\, mod p$, $(h(n)+2)\, mod p$, ..., este tipo de métodos tiene la desventaja de que se pueden generar largas cadenas de búsqueda cuando la tabla está bastante llena.
- Colocar todos los nombres en una misma posición, asociando a cada posición una estructura (secuencia, árbol o incluso otra tabla hash) que permita buscar eficientemente un nombre dentro de ella. La efectividad del método hash dependerá ahora de la adecuación de la estructura elegida al número medio de símbolos por cada posición.

B.1.5 Espacio de cadenas

Este método presenta una solución eficiente para almacenar los lexemas asociados a los símbolos. La facilidad que proporcionan algunos lenguajes de disponer de nombres de identificadores de longitud variable (y en algunos casos no acotada), hace que nos tengamos que plantear la forma en la que almacenamos dichos nombres, si optamos por dimensionar estáticamente el tamaño máximo con un valor relativamente grande estaremos desperdiciando memoria ya que casi nunca agotaremos ese espacio máximo, además de limitar a priori la longitud máxima de los identificadores. Mediante el método del espacio de

cadenas, seremos capaces de dimensionar dinámicamente el tamaño de los identificadores y además tendremos un mecanismo rápido de comparación de cadenas que acelerará notablemente el algoritmo de búsqueda. El método consiste en dimensionar un vector lo suficientemente grande como para albergar todos los nombres del programa y representar cada nombre por medio de dos números, la posición donde comienza en el espacio de cadenas y su longitud. El hecho de que todos los nombres compartan el espacio de cadenas hace que no se desperdicie memoria, simplemente tendremos que estimar la cantidad necesitada o reservar un nuevo espacio cuando el actual se llene. El disponer de la longitud del identificador hace que al buscar un nombre podamos descartar directamente aquellos cuya longitud no coincida con la del que buscamos sin tener que comparar las cadenas, lo que hace que el algoritmo de búsqueda sea más eficiente. El inconveniente de este método es que tenemos que implementar la gestión dinámica del espacio de cadenas con la problemática que puede traer el borrado selectivo de nombres.

Apéndice C

Bibliografía recomendada

ASU90 A. V. Aho, R. Sethi, J. D. Ullman. *Compiladores. Principios, Técnicas y Herramientas.* Addison-Wesley Iberoamericana, Wilmington, Delaware. 1990.

Bro89 J. G. Brookshear. *Theory of Computing.* Benjamin/Cummings. 1989.

FiL91 C. N. Fischer, R. J. Leblanc. *Crafting a Compiler with C.* Benjamin/Cummings Publishing Company, Inc. Redwood City, California. 1991.

Gro90 J. Grosch, H. Emmelmann. *A Tool Box for Compiler Construction.* Internal report No. 20. Karlsruhe. 1990.

Hol90 A. I. Holub. *Compiler Design in C.* Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1990.

KeP81 B. W. Kernighan, P. J. Plauger. *Software Tools in Pascal.* Addison-Wesley Publishing Company. Reading, Massachusetts. 1981.

Knu68 D. Knuth. *Semantics of context-free languages.* Mathematical Systems Theory 2. Pag. 127-145. 1968.

Mey91 B. Meyer. *Introduction to the Theory of Programming Languages.* Prentice-Hall International Series in Computer

Set92 R. Sethi. *Lenguajes de programación. Conceptos y constructores.* Addison-Wesley Iberoamericana, Wilmington, Delaware. 1992.

TrS85 J. P. Tremblay, P. G. Sorenson. *The Theory and Practice of Compiler Writing.* McGraw-Hill International Editions. 1985.

Wat93 D. A. Watt. *Programming Language Processors.* Prentice-Hall International Series in Computer Science. 1993.

Referencias bibliográficas

[Aho *et al.* 1986] A.V. Aho, V. Sethi, y J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison–Wesley, Menlo Park, California, 1.986.

[Aho y Johnson 1974] A.V. Aho y S.C. Johnson. LR parsing. *ACM Computing Surveys*, 6(2):99–124, June 1.974.

[Aho y Peterson 1972] A.V. Aho y T.G. Peterson. A minimal distance error correction parser for context free languages. En *SIAM J. Computing'72*, páginas 305–312, 1.972.

[Aho y Ullman 1972] A.V. Aho y J.D. Ullman. *The Theory of Parsing, Translation and Compiling*. Prentice Hall, Englewood Cliffs, New Jersey, 1.972.

[DeRemer y Pennello 1982] F.L. DeRemer y T. Pennello. Efficient computation of LALR(1) look–ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, October 1.982.

[DeRemer 1969] F.L. DeRemer. Simple LR(k) parsing. *Communications of the ACM*, 14(7):453–460, 1.969.

[Druseikis y Ripley 1976] D.C. Druseikis y G.D. Ripley. Error recovery for simple LR(k) parsers. En *Proceedings of the Annual Conference of the ACM*, páginas 20–22, Houston, Texas, October 1.976.

[Graham y Rhodes 1975] S.L. Graham y S.P. Rhodes. Practical syntactic error recovery. *Communications of the ACM*, 18(11):639–650, November 1.975.

[Horning 1976] J.J Horning. *Compiler Construction: An advanded Course*, capítulo What A Compiler Should Tell the User, páginas 525–548. Springer–Verlag, New York, 2^{nd} edición, 1.976.

[Knuth 1965] D.E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1.965.

[Korenjak 1969] A.J. Korenjak. A practical method for constructing LR(k) processors. *Communications of the ACM*, 12(11):612–623, 1.969.

[McKenzie *et al.* 1995] B.J. McKenzie, C. Yeatman, y L. De Vere. Error repair in shift–reduce parsers. *ACM Transaction on Programming Languages and Systems*, 17(4):672–689, July 1.995.

[Mickanus y Modry 1978] M.D. Mickanus y J.A. Modry. Automatic error recovery for LR parsers. *Communications of the ACM*, 21(6):465–495, June 1.978.

[Pennello y DeRemer 1978] T.J. Pennello y F. DeRemer. A forward move algorithm for LR error recovery. En *Conference Record of The Fifth Annual ACM Symposium of Principles of Programming Languages*, páginas 241–254, Tucson, Arizona, January 1.978.

[Rhodes 1973] S.P. Rhodes. *Practical Syntactic Error Recovery for Programming Languages*. PhD thesis, University of Berkeley, 1.973.

[Tremblay y Sorenson 1985] J. Tremblay y P.G. Sorenson. *The Theory and Practice of Compiler Writing*. MacGraw Hill, New York, 1.985.