CUADERNOS DIDACTICOS

UNIVERSIDAD DE OVIEDO
Departamento de Matemáticas

Análisis Sintáctico en Procesadores de Lenguaje

Cuaderno Nº 61

Juan Manuel Cueva Lovelle

Universidad de Oviedo

Oviedo, Abril 1995

DEPARTAMENTO DE MATEMÁTICAS
CUADERNO DIDÁCTICO Nº 61

ANÁLISIS SINTÁCTICO

EN

PROCESADORES DE LENGUAJE

Juan Manuel Cueva Lovelle

Catedrático de E.U. de Lenguajes y Sistemas Informáticos
Departamento de Matemáticas
Universidad de Oviedo
2ª Edición
Oviedo, Abril 1995

Tabla de contenidos

1 INTRODUCCION 1

2 GRAMATICAS LIBRES DE CONTEXTO 1
2.1 Derivaciones y árboles sintácticos 1
2.1.1 Ejemplo: gramática de expresiones aritméticas 1
2.2 Derivaciones más a la izquierda y más a la derecha 2
2.2.1 Ejemplo 3
2.3 Recursividad 6
2.3.1 Ejemplo 7
2.4 Gramáticas ambiguas 7
2.4.1 Ejemplo 7
2.5 Conversión de gramáticas ambiguas en no ambiguas 8
2.5.1 Ejemplo: representación de los arrays y llamada de funciones 8
2.5.2 Ejemplo: *else* danzante o ambiguo 8
2.5.3 Ejemplo: definición de precedencia y asociatividad 9
2.5.3.1 Nivel de precedencia 1 10
2.5.3.2 Nivel de precedencia 2 10
2.5.3.3 Nivel de precedencia 3 10
2.5.3.4 Nivel de precedencia 4 11
2.5.3.5 Nivel de precedencia 5 11
2.5.3.6 Gramática equivalente no ambigua 11
2.6 Gramáticas limpias y gramáticas sucias 12
2.7 Limpieza de gramáticas 12
2.7.1 Teorema de los símbolos vivos 13
2.7.2 Ejemplo 13
2.7.3 Teorema de los símbolos accesibles 13
2.7.4 Ejemplo 13
2.7.5 Análisis automático de la limpieza de gramáticas 14
2.8 Capacidad de descripción de las gramáticas libres de contexto 14
2.8.1 Ejemplo: llamada a funciones 14

3 FORMAS NORMALES DE CHOMSKY Y GREIBACH 15
3.1 Forma Normal de Chomsky (FNC) 15
3.1.1 Teorema de la forma normal de Chomsky 15
3.1.2 Ejemplo 15
3.2 Forma Normal de Greibach (FNG) 16
3.2.1 Teorema de la forma normal de Greibach 16
3.2.2 Ejemplo 16

4 RELACIONES BINARIAS 16
4.1 Definición de relación binaria 16
4.2 Representación mediante grafos 16
4.3 Representación mediante matrices 17
4.4 Ejemplo de relaciones binarias en gramáticas libres de contexto 17
a) Relación de dependencia simple 17
b) Relación de dependencia a izquierdas 17
4.5 Producto de relaciones 18
4.5.1 Ejemplos 18
4.5.2 Propiedades del producto de relaciones 18
4.5.3 Algoritmo para calcular el producto de relaciones 18
4.5.3.1 Algoritmo en Pascal 18
4.6 Relación reflexiva 19
4.6.1 Ejemplo 19
4.7 Relación simétrica 19
4.7.1 Ejemplo 19
4.8 Relación transitiva 19
4.8.1 Ejemplo 19
4.9 Relación de equivalencia 19
4.9.1 Ejemplos de relaciones de equivalencia 19
4.9.2 Propiedad de las relaciones de equivalencia 19
4.9.2.1 Ejemplo de clases de equivalencia 19
4.10 Relación irreflexiva 20
4.10.1 Ejemplo 20
4.11 Relación antisimétrica 20
4.11.1 Ejemplo 20

4.12 Inclusión de una relación 20
4.12.1 Ejemplo 20
4.13 Transpuesta de una relación 20
4.13.1 Ejemplos 20
4.14 Cierre transitivo de una relación 20
4.14.1 Teorema del cierre transitivo de una relación 20
4.14.2 Algoritmo de Warshall 20
4.14.3 Ejemplo 21
4.15 Cierre reflexivo transitivo de una relación 21

5 ANALISIS SINTACTICO DESCENDENTE 22
5.1 El problema del retroceso 22
5.1.1 Ejemplo de retroceso 22
5.2 La recursividad a izquierdas 25
5.2.1 Ejemplo de recursividad a izquierdas 26
5.3 Análisis sintáctico descendente con retroceso 27
5.3.1 Algoritmo de análisis sintáctico descendente con retroceso 27
5.3.2 Corolario 28
5.4 Análisis descendente sin retroceso 28
5.4.1 Gramáticas LL(k) 28
5.4.1.1 Teorema de la no ambigüedad de las gramáticas LL(k) 28
5.4.1.2 Teorema 28
5.4.2 Gramáticas LL(1) 28
5.4.2.1 S-gramáticas 28
5.4.2.1.1 Corolario de la definición de S-gramáticas 28
5.4.2.1.2 Contraejemplo de S-gramática 29
5.4.2.1.3 Ejemplo de S-grmática 29
5.4.2.1.4 Ejemplo de S-gramática 29
5.4.2.2 Conjunto de símbolos iniciales o cabecera 29
5.4.2.2.1 Ejemplo de cálculo del conjunto de Iniciales 30
5.4.2.3 Gramáticas LL(1) simples 30
5.4.2.3.1 Corolario de las gramáticas LL(1) simples 30
5.4.2.3.2 Teorema de equivalencia entre las gramáticas LL(1) y las S-gramáticas 30
5.4.2.3.3 Ejemplo de gramática LL(1) simple 30
5.4.2.3.4 Ejemplo de gramática LL(1) simple 31
5.4.2.4 Conjunto de símbolos seguidores o siguientes 31
5.4.2.4.1 Ejemplo de cálculo del conjunto de Seguidores 31
5.4.2.5 Conjunto de símbolos Directores 32
5.4.2.5.1 Ejemplo de cálculo del conjunto de símbolos Directores 32
5.4.2.6 Definición del las gramáticas LL(1) 32
5.4.3 Condiciones de las gramáticas LL(1) 32
5.4.3.1 Primera condición de *Knuth* 32
5.4.3.2 Segunda condición de *Knuth* 32
5.4.3.2.1 Tercera condición de *Knuth* 33
5.4.3.2.2 Cuarta condición de *Knuth* 33
5.4.4 Algoritmo de decisión 33
PASO 1: Encontrar los símbolos no terminales y producciones anulables 34
5.4.4.1.1 Ejemplo de cálculo de los símbolos anulables 34
PASO 2: Construcción de la relación EMPIEZA DIRECTAMENTE CON 36
5.4.4.2.1 Ejemplo de cálculo de la relación EMPIEZA DIRECTAMENTE CON 36
PASO 3: Construcción de la relación EMPIEZA CON 37
5.4.4.3.1 Ejemplo de cálculo de la relación EMPIEZA CON 37
PASO 4: Cálculo del conjunto de símbolos INICIALES de cada no terminal 38
5.4.4.4.1 Ejemplo de cálculo del conjunto de símbolos INICIALES de cada no terminal 38
PASO 5: Cálculo de los conjuntos INICIALES de cada producción 38
5.4.4.5.1 Ejemplo de cálculo de los símbolos INICIALES de cada producción 39
PASO 6: Construcción de la relación ESTA SEGUIDO DIRECTAMENTE POR 39
5.4.4.6.1 Ejemplo de cálculo de la relación ESTA SEGUIDO DIRECTAMENTE POR 39
PASO 7: Construcción de la relación ES FIN DIRECTO DE 41
5.4.4.7.1 Ejemplo de cálculo de la relación ES FIN DIRECTO DE 41
PASO 8: Construcción de la relación ES FIN DE 42
PASO 9: Construcción de la relación ESTA SEGUIDO POR 42
PASO 10: Calcular el conjunto de seguidores de cada no terminal anulable 43
PASO 11: Calcular el conjunto de símbolos directores 43
5.4.5 Transformación de gramáticas 43
5.4.5.1 Eliminación de la recursividad por la izquierda 44
5.4.5.1.1 Ejemplo de recursividad a izquierdas indirecta 45

5.4.5.1.2 Ejemplo de recursividad a izquierdas indirecta 45
5.4.5.2 Factorización y sustitución 46
5.4.5.2.1 Ejemplo del problema *if-then-else* 46
5.4.5.2.2 Ejemplo de una gramática de expresiones aritméticas 46
5.4.5.3 Transformación de gramáticas mediante aspectos semánticos 48
5.4.5.3.1 Ejemplo 48
5.5 Construcción de analizadores sintácticos descendentes 49
5.5.1 Métodos basados directamente en la sintaxis 49
5.5.1.1 Reglas de construcción de diagramas sintácticos 49
5.5.1.2 Traducción de reglas sintácticas a programas 50
5.5.1.2.1 Ejemplo 51
5.5.2 Construcción de analizadores sintácticos basados en autómatas de pila 52
5.5.2.1 Algoritmo de reconocimiento 53
5.5.3 Analizadores sintácticos recursivo descendentes 53
5.5.4 Analizadores sintácticos descendentes dirigidos por estructuras de datos 53
5.5.4.1 Traducción de reglas sintácticas a estructuras de datos 53
5.5.4.2 Construcción de analizadores sintácticos descendentes dirigidos por estructuras de datos 54
5.5.4.3 Construcción de analizadores sintácticos descendentes genéricos dirigidos por estructuras de datos 54
5.6 Tratamiento de errores sintácticos 55
5.6.1 Regla de la palabra clave 55
5.6.2 Regla del antipánico 55
5.6.3 Conclusión 56

6 ANALISIS SINTACTICO ASCENDENTE 57
6.1 Analizadores LR 57
6.1.1 Ejemplo de análisis ascendente con retroceso 57
6.2 Gramáticas y analizadores LR(k) 58
6.3 Estructura y funcionamiento de un analizador LR 58
6.3.1 Ejemplo 60
6.4 Algoritmo de análisis LR 61
6.5 Gramáticas LR 62
6.6 Construcción de tablas de análisis SLR 62
6.6.1 Función cierre 62
6.6.1.1 Ejemplo 63
6.6.2 Función goto 63
6.6.2.1 Ejemplo 63
6.6.3 Construcción de conjuntos de items 63
6.6.4 Tablas de análisis SLR 64
6.7 Construcción de tablas LR canónicas 65
6.7.1 Ejemplo 66
6.8 Construcción de tablas LALR 67
6.8.1 Algoritmo para construir tablas LALR 69
6.8.1.1 Ejemplo 70
6.9 Generadores de analizadores sintácticos: yacc, Bison, PCYACC y YACCOV 74
6.9.1 Utilización de yaccov 75
6.9.2 Símbolos 76
6.10 Estructura del fichero de entrada 76
6.10.1 Sección de declaraciones 76
6.10.2 Descripción de la gramática 79
6.10.3 Código C añadido. 80
6.11 Semántica 80
6.11.1 Valores semánticos 80
6.11.2 Acciones 80
6.11.3 Acciones en mitad de las reglas gramaticales 82
6.12 Recursividad 82
6.13 Conflictos, ambigüedad y precedencia 84
6.13.1 Conflictos shift/reduce (s/r) 85
6.13.2 Conflictos reduce/reduce (r/r) 86
6.13.3 Precedencia 86
6.14 Funciones que acompañan a *yyparse* 87
6.14.1 La función principal *main* 87
6.14.2 La función analizador léxico *yylex* 88
6.14.3 La función *yyerror* 88
6.15 Ficheros generados por *yaccov* 88
6.15.1 El fichero de salida 89
6.15.2 Funcionamiento de *yyparse* 89

6.15.3 Recuperación de errores 89
6.15.4 El fichero de cabecera 90
6.15.5 Depuración en tiempo de ejecución 92
6.15.6 El fichero de conflictos y estados 92
6.16 Diferencias entre YACCOV y PCYACC 97
6.17 Desarrollo de una minicalculadora con *yacc* 97
6.17.1 Calculadora elemental 97
6.17.2 Calculadora elemental con tratamiento de errores 106
6.17.3 Calculadora con variables 107
6.17.4 Calculadora con funciones 109
6.17.5 Calculadora HOC 112
6.18 Compilador de MUSIM/1 a ENSAMPOCO/1 117
6.19 YACCOV como entrada al yacc 119

7 EJERCICIOS PROPUESTOS 120
Ejercicio 7.1 120
Ejercicio 7.2 121
Ejercicio 7.3 121
Ejercicio 7.4 121
Ejercicio 7.5 121
Ejercicio 7.6 121
Ejercicio 7.7 122
Ejercicio 7.8 122
Ejercicio 7.9 122
Ejercicio 7.10 122
Ejercicio 7.11 122
Ejercicio 7.12 122

8 PRACTICAS DE LABORATORIO 122
8.1 Compilador de MUSIM/95 a ENSAMPIRE 122
8.2 Traductor de ENSAMPIRE a ENSAMBLADOR 80x86 122
8.3 Intérprete de ENSAMPIRE 123
8.4 Traductor de Eiffel/95 a C++ 123
8.4.1 Subconjunto del lenguaje Eiffel denominado Eiffel/95 123
8.4.2 Eiffel versus C++ en la generación de código 124
8.4.2.1 Declaración de clases 124
8.4.2.2 Declaración de funciones 125
8.4.2.3 Tratamiento de precondiciones 125
8.4.2.4 Tratamiento de postcondiciones 125
8.4.2.5 Herencia 126
Otros ejemplos 126

ANEXO I Lenguaje MUSIM/95 130
I.1 Especificación léxica 130
I.2 Especificación sintáctica 131
I.2.1 Estructuras de control de flujo siguiendo la sintaxis de Pascal estándar 131
I.2.2 Declaración de funciones prototipo delante de MAIN y construcción de funciones definidas por el usuario detrás de MAIN 131
I.2.3 Ficheros secuenciales de texto 131
I.2.4 Gestión de memoria dinámica heap y punteros al estilo C 131
I.3 Especificación semántica 131

ANEXO II Lenguaje Eiffel 131
II.1 ELEMENTOS BÁSICOS DE PROGRAMACIÓN EN EIFFEL 131
II.1.1 Objetos 132
II.1.2 Referencias 132
Creación dinámica 132
II.1.3 clases 132
Clases con atributos 132
Tipos y referencias 133
Usando las clases (clientes y proveedores) 133
Creación de objetos 134
Disociando una referencia de un objeto 134
Estados de una referencia 134
Inicialización 134
Acceso a campos 134
Rutinas 135

Variables locales 137
II.1.4 Entidades 137
Rutinas Predefinidas 137
Create explícito (no por defecto) 138
II.2 DE CLASES A SISTEMAS 138
II.3 CLASES VERSUS OBJETOS 140
II.4 GENERICIDAD 140
II.5 METODOS SISTEMÁTICOS PARA LA CONSTRUCCIÓN DE SOFTWARE 141
II.6 PRECONDICIONES Y POSTCONDICIONES 142
II.7 MÁS ASPECTOS DE EIFFEL 148
II.8 INTRODUCCIÓN A LA HERENCIA 154
Herencia múltiple 161
II.9 MÁS ACERCA DE LA HERENCIA 162
Herencia repetida 163
II.10 GRAMATICA DE EIFFEL 165
II.11 DIAGRAMAS SINTACTICOS DE EIFFEL 168

BIBLIOGRAFIA 174

Tabla de figuras

Fig. 1: Interfaces del analizador sintáctico 1
Fig. 2: Arbol sintáctico de -(a*9) 2
Fig. 3: Recursividad a izquierdas y a derechas 6
Fig. 4: Arboles sintácticos recursivos 7
Fig. 5: Ambigüedad. Primer árbol 7
Fig. 6: Ambigüedad. Segundo árbol 7
Fig. 7: 8
Fig. 8: 9
Fig. 9: 10
Fig. 10: 12
Fig. 11: Grafo de una relación binaria 16
Fig. 12: Grafo y matriz de la relación de dependencia simple 17
Fig. 13:Grafo y matriz de la relación de dependencia a izqda 18
Fig. 14: 23
Fig. 15: 23
Fig. 16: 23
Fig. 17: 24
Fig. 18: 24
Fig. 19: 24
Fig. 20: 25
Fig. 21 25
Fig. 22: 25
Fig. 23: 26
Fig. 24: 26
Fig. 25: 27
Fig. 26: 27
Fig. 27: 31
Fig. 28: vector de símbolos no terminales inicializado 35
Fig. 29: vector de símbolos no terminales 35
Fig. 30: vector de símbolos no terminales 35
Fig. 31: vector de símbolos no terminales 35
Fig. 32: Las gramáticas LL(1) como subconjunto particular 44
Fig. 33 44
Fig. 34: Recursividad a izquierdas indirecta 45
Fig. 35 49
Fig. 36: Diagráma sintáctico de una producción 49
Fig. 37: Diagrama sintáctico de repetición de una cadena 49
Fig. 38 50
Fig. 39: Diagrama de un símbolo no terminal 50
Fig. 40: Diagrama sintáctico de un símbolo terminal 50
Fig. 41 51
Fig. 42 51
Fig. 43 51
Fig. 44 51
Fig. 49 52
Fig. 45 53
Fig. 46: Regla alternativa 54
Fig. 47 54
Fig. 48 54

1 INTRODUCCION

El *análisis sintáctico* (*parser* en lengua inglesa) toma los *tokens* que le envia el analizador léxico y comprueba si con ellos se puede formar alguna sentencia válida del lenguaje. Recuérdese que se entiende por *sintaxis* como el conjunto de reglas formales que especifican como se construyen las sentencias de un determinado lenguaje.

La sintaxis de los lenguajes de programación habitualmente se describe mediante *gramáticas libres de contexto* (o gramáticas tipo 2) utilizando algún tipo de notación. Por ejemplo las notaciones BNF y EBNF. Esta especificación ofrece numerosas **ventajas** a los diseñadores de lenguajes y a los escritores de compiladores:

- Una gramática da una especificación sintáctica precisa, y fácil de comprender, de un lenguaje de programación.
- Para ciertas clases de gramáticas se puede construir automáticamente un analizador sintáctico, que determina si un programa está bien construido. Como beneficio adicional, el proceso de construcción del analizador sintáctico puede revelar ambigüedades sintácticas y otras dificultades para la construcción del *parser* que de otra forma no serían detectadas en la fase de diseño inicial de un lenguaje y que podrían se arrastradas a su compilador.
- Una gramática bien diseñada da una estructura al lenguaje de programación que se utiliza en la traducción del programa fuente en código objeto correcto y para la detección de errores. Existen herramientas que a partir de la descripción de la gramática se puede generar el código del analizador sintáctico.
- Los lenguajes de programación después de un periodo de tiempo, adquieren nuevas construcciones y llevan a cabo tareas adicionales. Estas nuevas construcciones se pueden añadir más fácilmente al lenguaje cuando hay una implementación basada en una descripción gramatical del lenguaje.

El papel del analizador sintáctico dentro de un procesador de lenguaje puede representarse según el esquema de la figura 1.

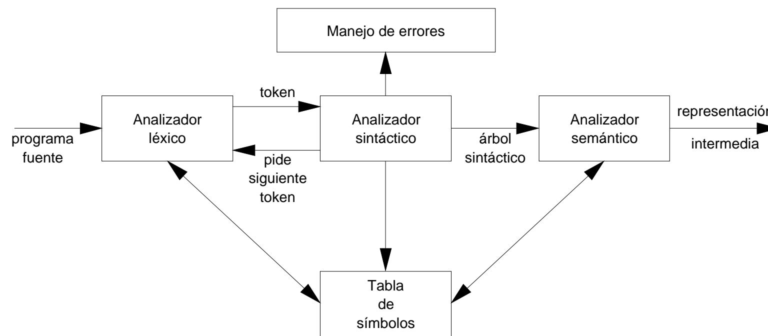


Figura 1: Interfaces del analizador sintáctico

2 GRAMATICAS LIBRES DE CONTEXTO

Habitualmente las estructura sintáctica de los lenguajes de programación se especifica sintácticamente mediante gramáticas libres de contexto o de tipo 2.

2.1 Derivaciones y árboles sintácticos

Analizar sintácticamente una cadena de tokens es encontrar para ella un **árbol sintáctico** o derivación, que tenga como raíz el símbolo inicial de la gramática libre de contexto y mediante la aplicación sucesiva de sus reglas de derivación se pueda alcanzar dicha cadena como hojas del árbol sintáctico. En caso de *éxito* la sentencia **pertenece al lenguaje** generado por la gramática, y puede proseguirse el proceso de compilación. *En caso contrario*, es decir cuando no se encuentra el árbol que genera dicha gramática, se dice entonces que la sentencia **no pertenece al lenguaje**, y el compilador emite un *mensaje de error*, pero el proceso de compilación trata de continuar. Algunos compiladores paran el proceso de compilación cada vez que encuentran un error, en ellos es necesario realizar tantas compilaciones como errores tendría el programa fuente.

2.1.1 Ejemplo: gramática de expresiones aritméticas

Supongamos que se desea construir una gramática G={VN, VT, S, P} que describe un lenguaje basado en la utilización de expresiones aritméticas con sumas, diferencias, productos, divisiones, potenciación, paréntesis, y menos unario. Este lenguaje también permitirá el uso de constantes e identificadores de variables. La gramática está compuesta por un vocabulario terminal

VT formado por los símbolos de los operadores, paréntesis y los tokens constante e identificador. El vocabulario no terminal VN sólo contendrá el símbolo no terminal <EXP>. El símbolo inicial S también será <EXP>. La gramática G queda tal y como se muestra a continuación:

VT= {+, *, /, ^, (,), -, identificador, constante}
VN= { <EXP> }
S= <EXP>

Las reglas de producción P que en un principio representarían al lenguaje se muestran a continuación numeradas del (1) al (9). La numeración se utilizará para indicar la regla que se aplica en cada derivación.

(1) <EXP> ::= <EXP> + <EXP>
(2) <EXP> ::= <EXP> * <EXP>
(3) <EXP> ::= <EXP> - <EXP>
(4) <EXP> ::= <EXP> / <EXP>
(5) <EXP> ::= <EXP> ^ <EXP>
(6) <EXP> ::= - <EXP>
(7) <EXP> ::= (<EXP>)
(8) <EXP> ::= identificador
(9) <EXP> ::= constante

Sea la expresión *- (a * 9)* se desea saber si pertenece o no al lenguaje, para lo cual se pretende alcanzar a partir de derivaciones desde el símbolo inicial de la gramática, construyéndose el árbol sintáctico de la figura 2. El árbol sintáctico no indica el orden seguido para alcanzar la cadena incognita. Para indicar el orden de aplicación de las derivaciones con las que se construyó el árbol se muestran las derivaciones siguientes, señalando encima de la flecha el número de la regla aplicada:

$$\langle EXP\rangle \xrightarrow{(6)} -\langle EXP\rangle \xrightarrow{(7)} -(\langle EXP\rangle) \xrightarrow{(2)} -(\langle EXP\rangle * \langle EXP\rangle) \xrightarrow{(8)} -(\text{ identificador } * \langle EXP\rangle) \xrightarrow{(9)} -(\text{identificador } * \text{ constante})$$

El analizador léxico se encarga de reconocer los símbolos terminales, así también indica que *a* es un *identificador* y que *9* es una *constante*.

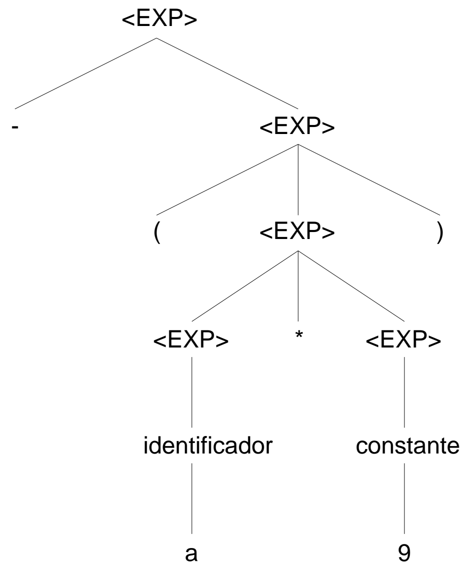


Figura 2: Arbol sintáctico de *-(a*9)*

2.2 Derivaciones más a la izquierda y más a la derecha

Las reglas de derivación que se aplican desde el símbolo inicial hasta alcanzar la sentencia a reconocer, pueden reemplazar los símbolos no terminales tomando el de más a la izquierda (derivaciones más a la izquierda) o tomando el no terminal de más a la derecha (derivaciones más a la derecha).

Habitualmente se trabaja con *derivaciones más a la izquierda*.

Se puede demostrar que todo árbol de análisis sintáctico tiene asociado una única derivación izquierda y una única derivación derecha. Sin embargo una sentencia de un lenguaje puede dar lugar a más de un árbol de análisis sintáctico tanto por la derecha como por la izquierda.

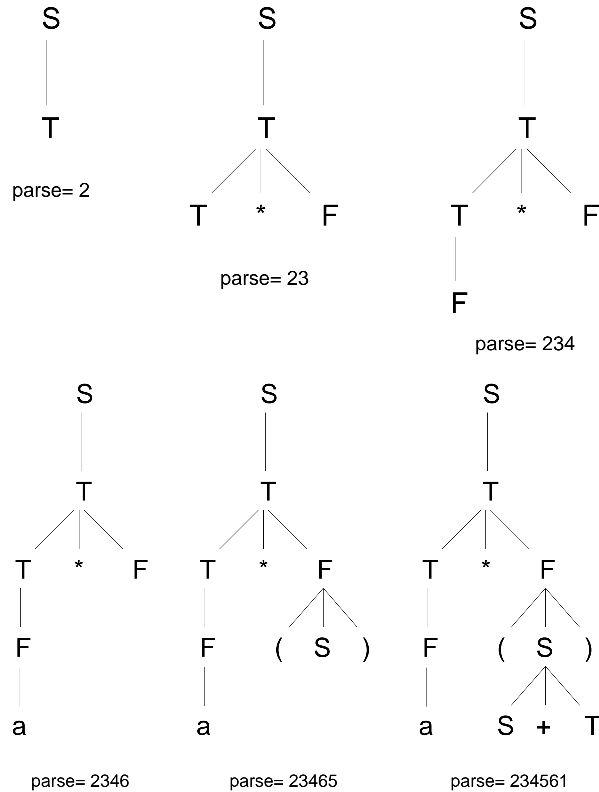
2.2.1 Ejemplo

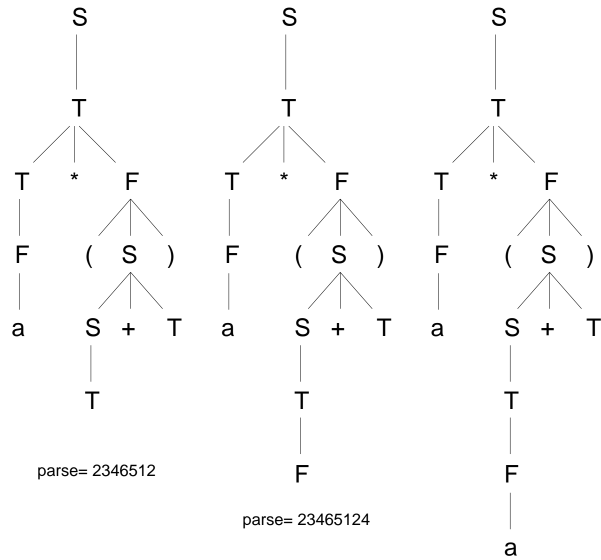
Sea la gramática de contexto libre G= (VN, VT, S, P) para al generación de expresiones aritméticas donde VN= {S, T, F}, VT= {a, b, +, *, (,)} y las reglas de producción P son:

(1)	S → S+T
(2)	S → T
(3)	T → T*F
(4)	T → F
(5)	F → (S)
(6)	F → (a)
(7)	F → (b)

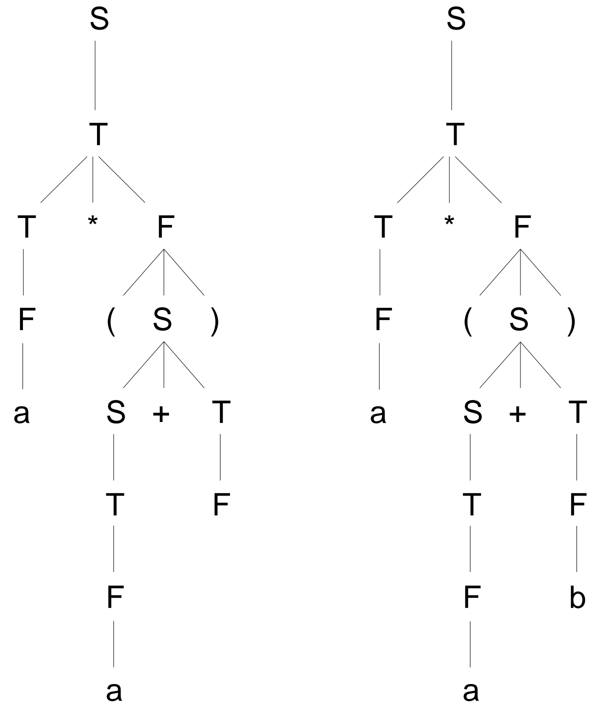
Determinar los árboles sintácticos más a la izquierda y más a la derecha que reconocen la cadena *a*(a+b)*.

Para determinar el *árbol más a la izquierda* se parte del símbolo inicial S y en la variable *parse* se van poniendo las producciones aplicadas, utilizando números para indicar los códigos de cada regla aplicada. Siempre se expansiona el no terminal más a la izquierda en el árbol hasta alcanzar un símbolo terminal. En las figuras siguientes se observa como se alcanza el árbol más a la izquierda aplicando las producciones por el órden indicado por los números 23465124647.



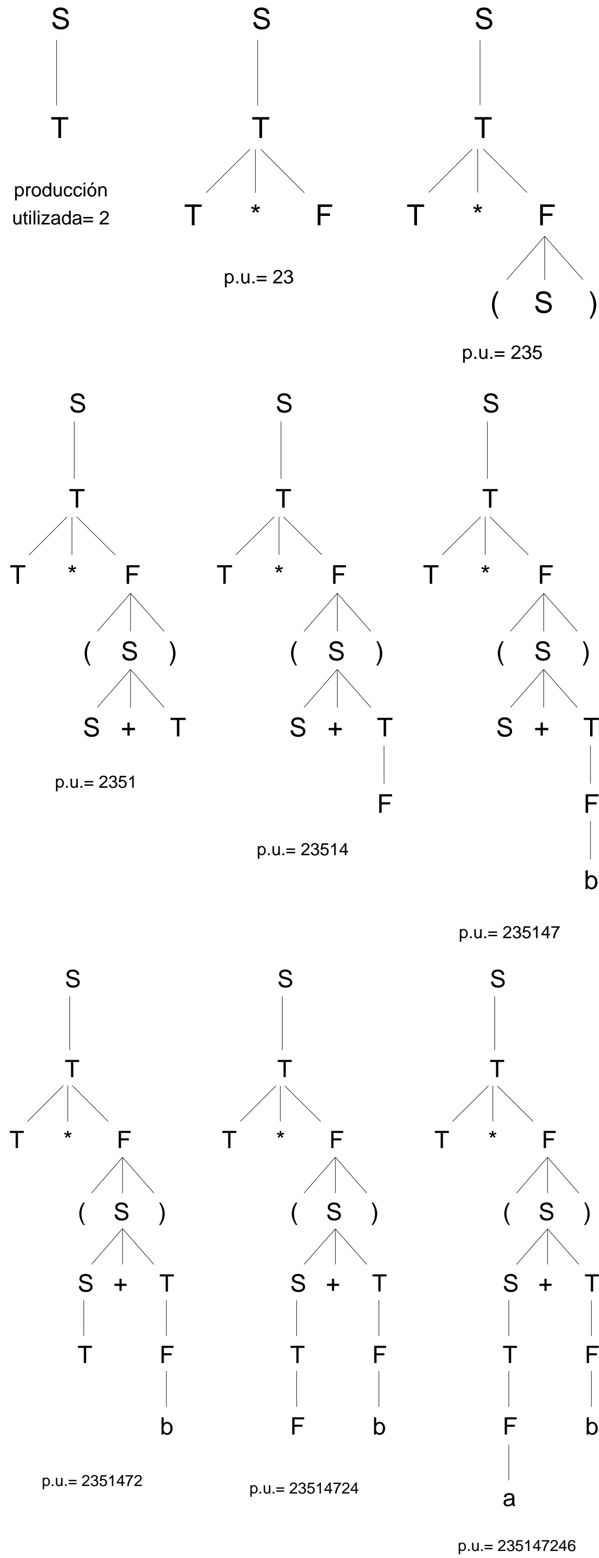


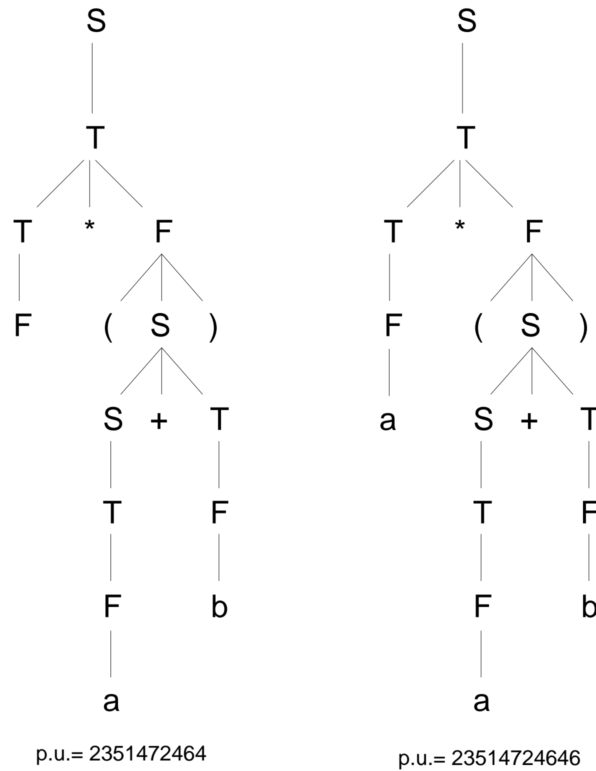
parse= 234651246



parse= 2346512464 parse Izdo= 23465124647

Para obtener el *árbol más a la derecha* se parte del símbolo inicial S y se van aplicando las producciones de forma que se eligen en primer lugar las expansiones del símbolo no terminal más a la derecha en el árbol. Así en la variable *producción utilizada (p.u.)* se van colocando los números de las producciones utilizadas. En las figuras siguientes puede observarse el orden de sustitución hasta alcanzar el árbol más a la derecha representado por los números 23514724646.


S
T
producción
utilizada= 2
T * F
p.u.= 23
(S)
p.u.= 235
S + T
p.u.= 2351
F
p.u.= 23514
b
p.u.= 235147
p.u.= 2351472
p.u.= 23514724
a
p.u.= 235147246



Puede observarse que los árboles finales alcanzados son los mismos aunque por distintos caminos. En general puede ocurrir que el árbol más a la izquierda y el árbol más a la derecha no sea el mismo, dando lugar a problemas de ambigüedad tal y como se estudia en el apartado 2.4.

2.3 Recursividad

Las reglas de producción de una gramática están definidas de forma que al realizar sustituciones dan lugar a recursividades. Entonces se dice que una regla de derivación es recursiva si es de la forma:

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

Si α es nula, es decir $A \rightarrow A\beta$ se dice que la regla de derivación es *recursiva a izquierda*. De la misma forma $A \rightarrow \alpha A$ es *recursiva a derecha* (véase figura 3).

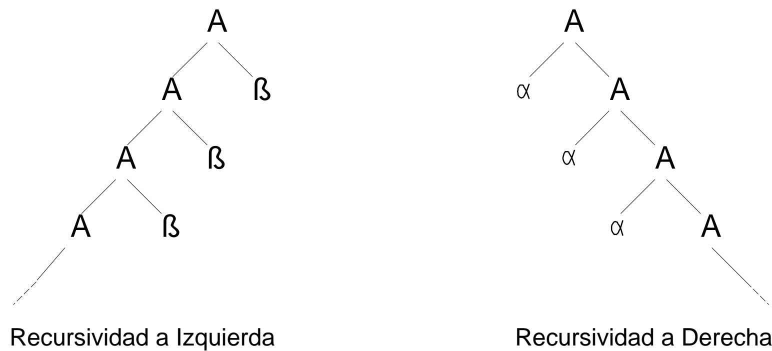


Figura 3: Recursividad a izquierdas y a derechas

Los analizadores sintácticos que trabajan con derivaciones más a la izquierda deben de evitar las reglas de derivación recursivas a izquierda, pues entrarían en un bucle infinito.

2.3.1 Ejemplo

Sea la gramática siguiente que describe números enteros e identificadores:

<entero> → <dígito> | <entero> <dígito>

<identificador> → <letra> | <identificador> <letra> | <identificador> <dígito>

sus árboles sintácticos son los de la figura 4.

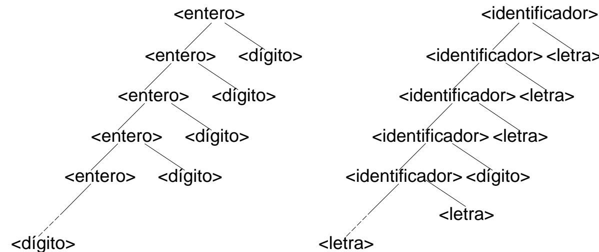


Figura 4: Arboles sintácticos recursivos

2.4 Gramáticas ambiguas

Una **sentencia generada por una gramática es ambigua** *si existe más de un árbol sintáctico para ella*. Una **gramática es ambigua** *si genera al menos una sentencia ambigua*, en caso contrario es **no ambigua**. Nótese que se llama a la gramática ambigua y no al lenguaje que genera dicha gramática. Hay muchas gramáticas equivalentes que pueden generar el mismo lenguaje, algunas son ambiguas y otras no. Sin embargo existen ciertos lenguajes para los cuales no pueden encontrarse gramáticas no ambiguas. A tales lenguajes se les denomina *ambiguos intrínsecos*. Por ejemplo el lenguaje {x^i, y^j, z^k / i=j o j=k} es un lenguaje ambiguo intrínseco.

El **grado de ambigüedad** de una sentencia se caracteriza por el *número de árboles sintácticos distintos que la generan*.

La ambigüedad de una gramática es una propiedad *indecidible*, lo que significa que no existe ningún algoritmo que acepte una gramática y determine con certeza y en un tiempo finito si la gramática es ambigua o no.

2.4.1 Ejemplo

Sea la gramática de manejo de expresiones mostrada en el apartado 2.1.1. Se puede comprobar que la sentencia: *5-C*6* tiene dos derivaciones más a la izquierda diferentes, y cuyos árboles sintácticos también son diferentes.

<EXP> → <EXP>-<EXP>
→ constante-<EXP>
→ 5-<EXP>*<EXP>
→ 5-identificador*<EXP>
→ 5-C*constante
→ 5-C*6

<EXP> → <EXP>*<EXP>
→ <EXP>-<EXP>*<EXP>
→ constante-<EXP>*<EXP>
→ 5-identificador*<EXP>
→ 5-C*constante
→ 5-C*6

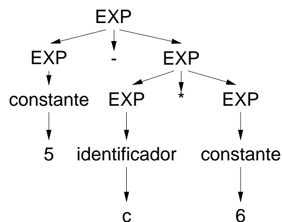


Figura 5: Arbol 1

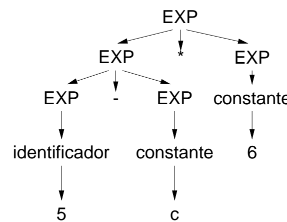


Figura 6: Arbol 2

El árbol de análisis sintáctico de la figura 5 refleja la precedencia habitual de Matemáticas de la operación sustracción sobre la multiplicación. Mientras que en el árbol de la figura 6 no se respeta esta precedencia.

2.5 Conversión de gramáticas ambiguas en no ambiguas

Se ha indicado anteriormente que la propiedad de ambigüedad es *indecidible*, es decir, no existe ni puede construirse un algoritmo que, tomando una gramática en BNF, determine con certeza, y en plazo finito de tiempo, si es ambigua o no. Sin embargo, se han desarrollado las **condiciones** que, de cumplirse, *garantizan la no ambigüedad*, aunque en caso de que no se cumplan no puede afirmarse nada. Son condiciones suficientes pero no necesarias. Así, las gramáticas del tipo **LL(k)** y **LR(k)**, que se estudiarán en otros apartados siguientes a éste, son no ambiguas, pero una gramática que no sea LL(k) ni LR(k) no puede decirse que sea ambigua.

Una gramática ambigua produce problemas al analizador sintáctico, por lo que interesa construir gramáticas no ambiguas. Sin embargo, en algunos lenguajes de programación, la **ambigüedad sintáctica** existe y se deberá eliminar con consideraciones semánticas.

2.5.1 Ejemplo: representación de los arrays y llamada de funciones

La representación de los elementos de un array en los lenguajes PL/I y FORTRAN da lugar a ambigüedades, pues se puede confundir la llamada a una función con un elemento de un array. Así la expresión M(I,J,K) es ambigua, puede considerarse como el elemento $M_{i,j,k}$ del array, y también la llamada a la subrutina M con los parámetros I,J,K. Si la declaración de variables de tipo array o de funciones es obligatoria, también se resuelve la ambigüedad, en caso contrario han de tenerse en cuenta consideraciones semánticas.

Los lenguajes Pascal, C, C++, y Algol 60 resuelven esta ambigüedad empleando corchetes para representar los arrays. El lenguaje FORTRAN no tiene declaración obligatoria de variables ni de subrutinas, pero resuelve la ambigüedad obligando a declarar las funciones no estándar. La llamada a subrutinas no constituye ambigüedad pues se deben hacer con la sentencia CALL. El lenguaje PL/I resuelve la ambigüedad con la declaración obligatoria de todas las variables.

2.5.2 Ejemplo: *else* danzante o ambiguo

Una sentencia alternativa simple es de la forma siguiente:

<sentencia> ::= **if** <expresión booleana> **then** <sentencia>
| **if** <expresión booleana> **then** <sentencia> **else** <sentencia>
| **otra_sentencia**

Puede darse el caso de que la sentencia que va después del **then** sea otra sentencia alternativa, es decir se produce una sentencia **if-then-else** anidada de la forma:

if E_1 **then if** E_2 **then** S_1 **else** S_2

En este caso se produce una ambigüedad pues la parte correspondiente al **else** puede asociarse a dos **if**, lo que da lugar a **dos árboles sintácticos**, que se muestran en las figuras 7 y 8.

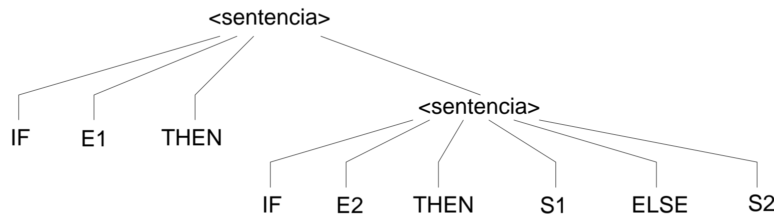


Figura 7: Arbol 1, el **else** se asocia con el **if** anterior más cercano

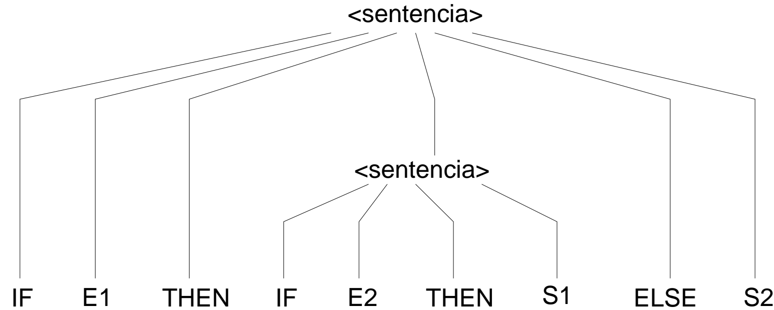


Figura 8: Arbol 2, el **else** se asocia con el primer **if**

La mayor parte de los lenguajes de programación con sentencias condicionales de este tipo utilizan el primer árbol sintáctico (fig. 7), es decir utilizan la regla *el* **else** *se empareja con el* **if** *anterior más cercano*. Esta regla elimina la ambigüedad y se puede incorporar a la gramática.

Se construye una gramática equivalente que elimine la ambigüedad. La idea es que una sentencia que aparezca entre un *THEN* y un *ELSE* debe estar emparejada, es decir no debe terminar con un *THEN* sin emparejar seguido de cualquier sentencia, porque entonces el *ELSE* estaría obligado a concordar con este *THEN* no emparejado. Así sentencia emparejada tan solo puede ser una sentencia alternativa *IF-THEN-ELSE* que no contenga sentencias sin emparejar o cualquier otro tipo de sentencia diferente de la alternativa. La nueva gramática equivalente será la siguiente:

```
               <sent>::= <sent_emparejada>
                         | <sent_no_emparejada>
    <sent_emparejada>::= IF <expr_bool> THEN <sent_emparejada> ELSE <sent_emparejada>
                         | OTRA_SENTENCIA
 <sent_no_emparejada>::= IF <expr_bool> THEN <sent>
                         | IF <expr_bool> THEN <sent_emparejada> ELSE <sent_no_emparejada>
```

El problema del *if-then-else* se volverá a tratar en el apartado 5.4.5.2.1.

2.5.3 Ejemplo: definición de precedencia y asociatividad

Una de las principales formas de evitar la ambigüedad es definiendo la **precedencia** y **asociatividad** de los operadores. Así el lenguaje C está basado en el uso intensivo de operadores, con una definición estricta de las precedencias y asociatividades de cada uno de ellos. Volvamos otra vez a la gramática de expresiones ya utilizada en los ejemplos 2.1.1 y 2.4.1. Dicha gramática era ambigüa debido a problemas de precedencia y asociatividad de sus operadores. En este ejemplo se mostrará como se refleja la precedencia y asociatividad en la gramática para evitar la ambigüedad.

- Se define el orden de *precedencia* de evaluación de las expresiones y los operadores. A continuación se presenta el orden de precedencia de mayor a menor precedencia:

  1º) ( ) identificadores constantes

  2ª) - (menos unario)

  3ª) ^ (potenciación)

  4ª) * /

  5ª) + -

- La *asociatividad* se define de forma diferente para el operador potenciación que para el resto de los operadores. Así el operador ^ es asociativo de derecha a izquierda:

  $$a \wedge b \wedge c = a \wedge (b \wedge c)$$

  mientras que el resto de los operadores binarios serán asociativos de izquierda a derecha, si hay casos de igual precedencia.

a-b-c= (a-b)-c
a+b-c= (a+b)-c
*a*b/c= (a*b)/c*

Estas dos propiedades, precedencia y asociatividad, son suficientes para convertir la gramática ambigua basada en operadores en no ambigua, es decir, que cada sentencia tenga sólo un árbol sintáctico.

Para introducir las propiedades anteriores de las operaciones en la gramática se tiene que escribir otra gramática equivalente en la cual **se introduce un símbolo no terminal por cada nivel de precedencia**.

2.5.3.1 Nivel de precedencia 1

Se introduce el no terminal <ELEMENTO> para describir una expresión indivisible, y con **máxima precedencia**, por lo tanto

<ELEMENTO> ::= (<EXP>) | identificador | constante

2.5.3.2 Nivel de precedencia 2

Se introduce el no terminal <PRIMARIO>, que describe el menos unario y el no terminal de precedencia superior.

<PRIMARIO> ::= - <PRIMARIO> | <ELEMENTO>

2.5.3.3 Nivel de precedencia 3

El siguiente no terminal que se introduce es <FACTOR> que describe los operadores del siguiente nivel de precedencia.

<FACTOR> ::= <PRIMARIO> ^ <FACTOR> | <PRIMARIO>

Hay que señalar que el *orden* es fundamental. Así <PRIMARIO> ^ <FACTOR> obliga a expresiones del tipo *a ^ b ^ c* a agruparse como *a ^ (b ^ c)* siendo su árbol sintáctico el representado en la figura 9.

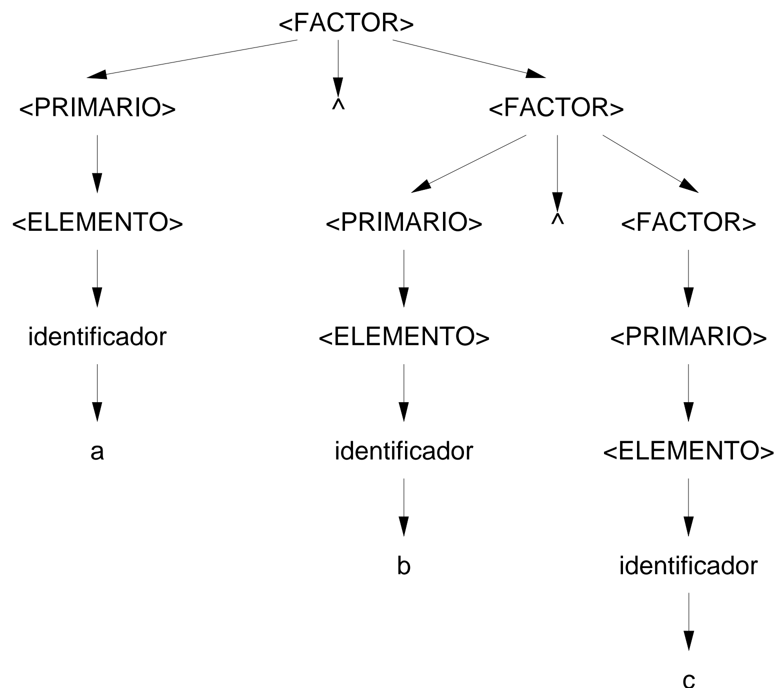


Figura 9:Arbol de *a^b^c*

2.5.3.4 Nivel de precedencia 4

El siguiente no terminal que se introduce es <TERMINO>, que es una secuencia de uno o más factores conectados con operadores de nivel de precedencia 4º, es decir: multiplicación y división.

<TERMINO> ::= <TERMINO> * <FACTOR> |
<TERMINO> / <FACTOR> |
<FACTOR>

El orden de esta producción obliga a que expresiones del tipo:

*a * b / c* significique *(a * b) / c*

2.5.3.5 Nivel de precedencia 5

Por último se introduce el no terminal <EXPRESION> que representará a <TERMINO> conectado a otro <TERMINO> con operadores de menor precedencia: adición y sustracción.

<EXP> ::= <EXP> + <TERMINO> |
<EXP> - <TERMINO> |
<TERMINO>

2.5.3.6 Gramática equivalente no ambigua

Entonces la nueva gramática no ambigua será GNA= (VN, VT, S, P) donde:

VN= {<ELEMENTO>, <PRIMARIO>, <FACTOR>, <TERMINO>, <EXP>}
VT= {identificador, constante, (,), ^, *, /, +, -}
S= <EXP>
y las producciones P:

<EXP> ::= <EXP> + <TERMINO> |
<EXP> - <TERMINO> |
<TERMINO>
<TERMINO> ::= <TERMINO> * <FACTOR> |
<TERMINO> / <FACTOR> |
<FACTOR>
<FACTOR> ::= <PRIMARIO> ^ <FACTOR> | <PRIMARIO>
<PRIMARIO> ::= - <PRIMARIO> | <ELEMENTO>
<ELEMENTO> ::= (<EXP>) | identificador | constante

Entonces el árbol sintáctico de la sentencia *a * c + d ^ 4* se muestra en la figura 10. Sin embargo esta gramática es recursiva a izquierdas, este inconveniente se resolverá en el ejemplo del apartado 5.4.5.2.2.

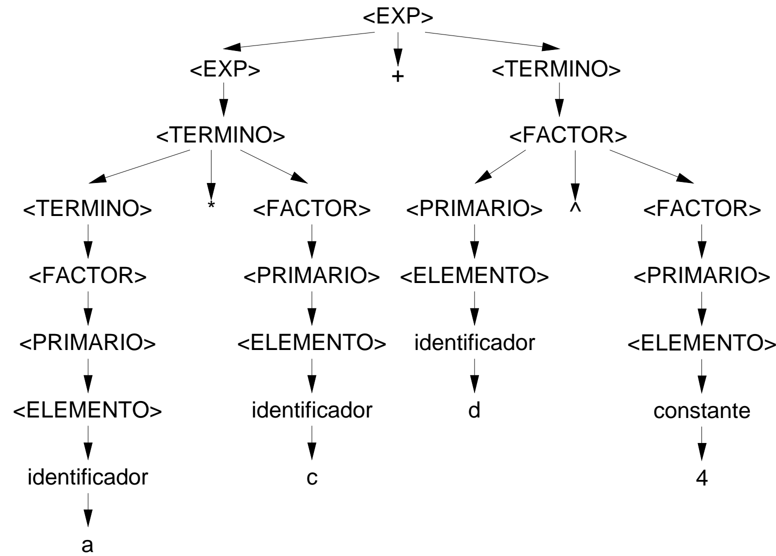


Figura 10:Arbol de a*c+d^4

2.6 Gramáticas limpias y gramáticas sucias

Las gramáticas de los lenguajes de programación están formadas por un conjunto de reglas BNF, cuyo número suele ser bastante amplio, lo cual incide en la ocultación de distintos problemas que pueden producirse, tales como tener reglas que produzcan símbolos que no se usen después, o que nunca se llegue a cadenas terminales. Todo esto se puede solventar realizando la transformación de la gramática inicial "sucia" a una gramática "limpia". A continuación se formalizarán estos conceptos por medio de definiciones.

- **Símbolo muerto**: *es un símbolo no terminal que no genera ninguna cadena de símbolos terminales*.
- **Símbolo inaccesible**: *es un símbolo no terminal al que no se puede llegar por medio de producciones desde el símbolo inicial*.
- **Símbolo extraño**: *se denomina así a todo símbolo muerto o inaccesible*.
- **Gramática sucia:** *es toda gramática que contiene símbolos extraños*.
- **Gramática limpia:** *es toda gramática que no contiene símbolos extraños.*
- **Símbolo vivo**: *es un símbolo no terminal del cual se puede derivar una cadena de símbolos terminales*. Todos los símbolos terminales son símbolos vivos. Es decir son símbolos vivos lo que no son muertos.
- **Símbolo accesible**: *es un símbolo que aparece en una cadena derivada del símbolo inicial*. Es decir, aquel símbolo que no es inaccesible.

2.7 Limpieza de gramáticas

En el apartado anterior se han definido algunos problemas que pueden presentarse en una gramática. Por lo tanto es norma general que toda gramática en bruto ha de limpiarse con el objetivo de eliminar todos los símbolos extraños.

Existen algoritmos para depurar y limpiar las gramáticas sucias. El método consiste en detectar en primer lugar todos los símbolos muertos, y a continuación se detectan todos los símbolos inaccesibles. Es importante seguir este orden, puesto que la eliminación de símbolos muertos puede generar nuevos símbolos inaccesibles.

Los algoritmos que se utilizan en la limpieza de gramáticas se basan en los teoremas que se enunciarán a continuación sobre los símbolos vivos y los símbolos accesibles.

2.7.1 Teorema de los símbolos vivos

Si todos los símbolos de la parte derecha de una producción son vivos, entonces el símbolo de la parte izquierda también lo es.

La demostración es obvia. Este teorema se utiliza para construir algoritmos de detección de símbolos muertos. El procedimiento consiste en iniciar una lista de no terminales que sepamos a ciencia cierta que son símbolos vivos, y aplicando el teorema anterior para detectar otros símbolos no terminales vivos para añadirlos a la lista. Dicho de otra forma, los pasos del **algoritmo** son:

1º) Hacer una lista de no-terminales que tengan al menos una producción sin símbolos no terminales en la parte derecha.
2º) Dada una producción, si todos los no-terminales de la parte derecha pertenecen a la lista, entonces podemos incluir al no terminal de la parte izquierda.
3º) Cuando no se puedan incluir más símbolos mediante la aplicación del paso 2º), la lista contendrá todos los símbolos vivos, el resto serán muertos.

2.7.2 Ejemplo

Sea la gramática expresada en BNF:

<INICIAL> ::= a <NOTA1> <NOTA2> <NOTA3> | <NOTA4> d
<NOTA1> ::= b <NOTA2> <NOTA3>
<NOTA2> ::= e | de
<NOTA3> ::= g <NOTA2> | h
<NOTA4> ::= <NOTA1> f <NOTA5>
<NOTA5> ::= t <NOTA4> | v <NOTA5>

Determinar los símbolos muertos.

Se aplican los pasos:

1º) Confección de la lista: sólo hay un símbolo no terminal con el cual comenzar la lista.

<NOTA2>

2º) Aplicando el teorema 2.7.1 se incluyen en la lista por el siguiente orden:

<NOTA3>
<NOTA1>
<INICIAL>

3º) No se puede aplicar el teorema más veces, por lo tanto la lista de símbolos vivos está completa y los símbolos <NOTA4> y <NOTA5> son no terminales muertos.

2.7.3 Teorema de los símbolos accesibles

Si el símbolo no terminal de la parte izquierda de una producción es accesible, entonces todos los símbolos de la parte derecha también lo son.

La demostración es obvia. El teorema se puede utilizar para construir algoritmos, consistiendo éstos en el siguiente procedimiento: se hace una lista inicial de símbolos no terminales que se sabe a ciencia cierta que son accesibles, y usando el teorema 2 para detectar nuevos símbolos accesibles para añadir a la lista, hasta que no se pueden encontrar más.

Los pasos a seguir son:

1º) Se comienza la lista con un único no terminal, el símbolo inicial.
2º) Si la parte izquierda de la producción está en la lista, entonces se incluyen en la misma a todos los no terminales que aparezcan en la parte derecha.
3º) Cuando ya no se puedan incluir más símbolos mediante la aplicación del paso 2º), la lista contendrá todos los símbolos accesibles, y el resto será inaccesible.

2.7.4 Ejemplo

Sea la siguiente gramática en BNF:

```
<INICIAL>    ::= a <NOTER1> <NOTER2> | <NOTER1>
<NOTER1>     ::= c <NOTER2> d
<NOTER2>     ::= e | f <INICIAL>
<NOTER3>     ::= g <NOTER4> | h <NOTER4> t
<NOTER4>     ::= x | y | z
```

Determinar los símbolos inaccesibles

Aplicando los pasos:

1º) Confección de la lista: <INICIAL>

2º) Aplicación del teorema 2.7.3: <NOTER1>
<NOTER2>

3º) No se puede aplicar más veces el paso, luego la lista de símbolos accesibles está completa, y los no terminales inaccesibles son:

<NOTER3>
<NOTER4>

2.7.5 Análisis automático de la limpieza de gramáticas

Los algoritmos de limpieza de gramáticas son fáciles de programar, y comprueban si las gramáticas son limpias. El primer paso para el tratamiento de cualquier gramática es la eliminación de los símbolos extraños. En la Universidad de Oviedo se ha desarrollado un analizador de gramáticas LL(1), que toma como entrada una gramática descrita en el formato BNF sin la opción alternativa e indica si la gramática es LL(1) o no señalando las reglas de la gramática que se lo impiden. El primer paso de este analizador es verificar si la gramática es limpia, así la gramática de los ejemplos anteriores debe introducirse en el formato siguiente:

```
<INICIAL>::= A <NOTER1> <NOTER2>
<INICIAL>::= <NOTER1>
 <NOTER1>::= C <NOTER2> D
 <NOTER2>::= E
 <NOTER2>::= F <INICIAL>
 <NOTER3>::= G <NOTER4>
 <NOTER3>::= H <NOTER4> T
 <NOTER4>::= X
 <NOTER4>::= Y
 <NOTER4>::= Z
```

El analizador indica que la gramática no es limpia. En el menú correspondiente se puede obtener la relación de símbolos no accesibles: NOTER3 y NOTER4.

2.8 Capacidad de descripción de las gramáticas libres de contexto

La sintaxis de la mayoría de los lenguajes de programación, tales como ALGOL 60, FORTRAN, C, C++ y Pascal no se puede describir completamente por gramáticas libres de contexto. Por este motivo, la definición sintáctica de los lenguajes de programación viene dada habitualmente en dos partes. La mayor parte de la sintaxis se describe por gramáticas libres de contexto, especificadas en notación BNF o EBNF. El resto de la sintaxis, que no se puede describir por medio de gramáticas libres de contexto, se presenta como un añadido en lenguaje natural, y se trata en el compilador en la parte de análisis semántico. También se amplian las gramáticas libres de contexto con atributos, condiciones y reglas semánticas, con el objeto de definir las especificaciones semánticas.

2.8.1 Ejemplo: llamada a funciones

En la llamada a *procedimientos* o *subrutinas* se ha de comprobar que el *número de argumentos* debe de coincidir con el *número de parámetros ficticios*. Esto no lo pueden describir las gramáticas libres de contexto. Esta verificación se suele hacer en la fase de análisis semántico. Otra solución es la que utiliza el lenguaje C ANSI obligando a utilizar la definición de funciones prototipo.

3 FORMAS NORMALES DE CHOMSKY Y GREIBACH

Sucede con frecuencia en lingüística matemática, que en algunas ocasiones es imprescindible que las gramáticas se hallen dispuestas de una forma especial. Es decir, se trata de obtener una gramática equivalente, que genera el mismo lenguaje, pero que debe cumplir unas especificaciones determinadas. A continuación se muestran las dos formas normalizadas más frecuentes, que se emplean en los lenguajes formales y sus aplicaciones.

3.1 Forma Normal de Chomsky (FNC)

Una gramática se dice que está en la *Forma Normal de Chomsky* si sus reglas son de una de estas formas:

$A \to BC$

$A \to a$

Siendo *A*, *B*, *C* no terminales y *a* un terminal.

3.1.1 Teorema de la forma normal de Chomsky

Toda gramática libre de contexto sin la cadena vacía tiene una gramática equivalente cuyas producciones están en la Forma Normal de Chomsky.

3.1.2 Ejemplo

Sea la gramática G= (VN={S, A, B}, VT={a, b}, P, S) cuyas producciones son:

$S \to bA \mid aB$

$A \to bAA \mid aS \mid a$

$B \to aBB \mid bS \mid b$

encontrar una gramática equivalente en FNC.

En primer lugar las reglas pueden reescribirse:

(1) $S \to bA$

(2) $S \to aB$

(3) $A \to bAA$

(4) $A \to aS$

(5) (*) $A \to a$

(6) (*) $B \to aBB$

(7) $B \to bS$

(8) $B \to b$

Solamente las señaladas con (*) están en forma FNC. La producción (1) puede sustituirse por dos:

$S \to C_b A$

$C_b \to b$

Igualmente la (2) puede sustituirse por

$S \to C_a B$

$C_a \to a$

Las producciones (3) y (4) se sustituyen por

$A \to C_b D_1$ $\qquad$ $D_1 \to AA$

$A \to C_a S$

y la (6) y la (7) por

$B \to C_a D_2$

$D_2 \to BB$

$B \to C_b S$

Entonces la gramática equivalente en FNC es:

$S \rightarrow C_b A$
$S \rightarrow C_a B$
$A \rightarrow C_a S$
$A \rightarrow C_b D_1$
$A \rightarrow a$
$B \rightarrow C_b S$
$B \rightarrow C_a D_2$
$B \rightarrow b$
$D_1 \rightarrow AA$
$D_2 \rightarrow BB$
$C_a \rightarrow a$
$C_b \rightarrow b$

3.2 Forma Normal de Greibach (FNG)

Se dice que una gramática está en la *Forma Normal de Greibach* si sus reglas de producción son de la forma:

$A \rightarrow a\alpha$
$A \rightarrow a$

donde *A* es un no terminal, *a* es un terminal y α es una cadena compuesta por terminales o no terminales.

3.2.1 Teorema de la forma normal de Greibach

Todo lenguaje de contexto libre sin la cadena vacía puede ser generado por una gramática cuyas reglas de producción son de la forma

$A \rightarrow a\alpha$
$A \rightarrow a$

donde *A* es un no terminal, *a* es un terminal y α es una cadena de terminales y n terminales.

3.2.2 Ejemplo

La gramática dada en el ejemplo 3.1.2.

$S \rightarrow bA \mid aB$
$A \rightarrow bAA \mid aS \mid a$
$B \rightarrow aBB \mid bS \mid b$

4 RELACIONES BINARIAS

El concepto de relación es uno de los más elementales de la Matemática. La idea que se tiene de relación en la vida real es la asociación de entes con alguna característica en común. Sólo se considerarán *relaciones binarias*, es decir, aquellas que se establecen entre un par de objetos. En este apartado se formalizará el concepto de relación, viéndose dos métodos para representarlas: mediante matrices y mediante grafos.

4.1 Definición de relación binaria

Sea un conjunto X= {x, y, ...} se dice que dos elementos están relacionados xRy por medio de la relación R, si cumplen una determinada propiedad dada por R. El conjunto de pares relacionados está definido por el producto escalar de X×X con la relación R.

4.2 Representación mediante grafos

Una relación binaria se puede representar mediante un grafo de la siguiente forma, si *xRy* entonces se representa el grafo de de la figura 11.

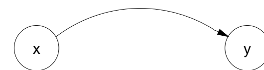


Figura 11: Grafo de una relación binaria

4.3 Representación mediante matrices

La relación en un conjunto X se puede representar mediante una matriz cuadrada A de dimensión cardinal(X), cuyos elementos son booleanos (o su representación por ceros y unos) de forma que:

$$a_{ij} = \begin{cases} 1 \quad si \quad x_i R x_j \quad (\textit{está relacionado}) \\ 0 \quad si \quad x_i \neg R x_j \quad (\textit{no está relacionado}) \end{cases}$$

R_1	x_1	x_2	...	x_n
x_1	0	0	...	1
x_2	0	1	...	0
...	...	...	...	...
x_n	0	1	...	0

4.4 Ejemplo de relaciones binarias en gramáticas libres de contexto

Sea la gramática G= (V_n, V_t, S, P) donde:

V_t= {a, b, +, (,), ⊣}
V_n= {E, 1, 1', P, P', S}
S= símbolo inicial
P: las reglas de producción

<E> ::= <1> <1'>
<1'> ::= + <1> <1'> | λ
<1> ::= <P> <P'>
<P'> ::= <P> <P'> | λ
<P> ::= a | b | (<E>)
<S> ::= <E> ⊣

A continuación se definen dos relaciones: la relación de dependencia simple y la relación de dependencia a izquierdas. Representar los grafos y determinar las matrices de ambas relaciones.

a) Relación de dependencia simple

Se dice que $\Sigma \in V_n$ y $\alpha \in V$ (siendo $V = V_n \cup V_t$) están relacionadas por una relación de dependencia simple $\Sigma R_s \alpha$ si $\exists \Sigma \to \sigma_1 \alpha \sigma_2 / \sigma_1, \sigma_2 \in V^*$

El grafo y la matriz de la relación de dependencia simple son los que se muestran en la figura 12.

R_s	S	E	1'	1	P'	P	a	b	(	)	+	⊣
S	0	1	0	0	0	0	0	0	0	0	0	1
E	0	0	1	1	0	0	0	0	0	0	0	0
1'	0	0	1	1	0	0	0	0	0	0	1	0
1	0	0	0	0	1	1	0	0	0	0	0	0
P'	0	0	0	0	1	1	0	0	0	0	0	0
P	0	1	0	0	0	0	1	1	1	1	0	0
a	0	0	0	0	0	0	0	0	0	0	0	0
b	0	0	0	0	0	0	0	0	0	0	0	0
(	0	0	0	0	0	0	0	0	0	0	0	0
)	0	0	0	0	0	0	0	0	0	0	0	0
+	0	0	0	0	0	0	0	0	0	0	0	0
⊣	0	0	0	0	0	0	0	0	0	0	0	0

Figura 12: Grafo de la relación de dependencia simple

b) Relación de dependencia a izquierdas

Se dice que $\Sigma \in V_n$ y $\alpha \in V$ cumplen $\Sigma R_I \alpha$ si y solamente si α aparece como primer símbolo (distinto de la cadena vacía) en una producción de la gramática en la que Σ es la parte izquierda. Es decir $\Sigma R_I \alpha \Leftrightarrow \exists \Sigma \to \Sigma_1 \Sigma_2 \cdots \Sigma_p \alpha \psi$ donde $p \geq 0$, $\Sigma_1, \Sigma_2, \ldots, \Sigma_p \in V_n$, $\psi \in V^*$, $\Sigma_1 \overset{*}{\to} \lambda$, $\Sigma_2 \overset{*}{\to} \lambda$, $\Sigma_3 \overset{*}{\to} \lambda, \cdots, \Sigma_p \to \lambda$.

En la figura 13 se representa el grafo y la matriz de la relación de dependencia a izquierdas.

R_l	S	E	1'	1	P'	P	a	b	(	)	+	┤
S	0	1	0	0	0	0	0	0	0	0	0	0
E	0	0	0	1	0	0	0	0	0	0	0	0
1'	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	1	0	0	0	0	0	0
P'	0	0	0	0	0	1	0	0	0	0	0	0
P	0	0	0	0	0	0	1	1	1	0	0	0
a	0	0	0	0	0	0	0	0	0	0	0	0
b	0	0	0	0	0	0	0	0	0	0	0	0
(	0	0	0	0	0	0	0	0	0	0	0	0
)	0	0	0	0	0	0	0	0	0	0	0	0
+	0	0	0	0	0	0	0	0	0	0	0	0
┤	0	0	0	0	0	0	0	0	0	0	0	0

(V_n: S, E, 1', 1, P', P; V_t: a, b, (,), +, ┤)

Figura 13: Grafo y matriz de la relación de dependencia a izquierdas

4.5 Producto de relaciones

Dadas dos relaciones R y P definidas en el mismo conjunto, se define una nueva relación Q, llamada producto de R y de P, a la siguiente:

$$c Q d \Leftrightarrow c R e \qquad y \qquad e P d$$

4.5.1 Ejemplos

Relación SER PADRE DE#2 ⎫ #2 Producto SER SUEGRO DE
Relación SER CONYUGE DE #2 ⎭

a ES PADRE de c ⎫ a ES SUEGRO DE b
c ES CONYUGE de b ⎭

4.5.2 Propiedades del producto de relaciones

§ El producto de relaciones tiene la propiedad asociativa.

P (QR) = (PQ) R

§ Empleando el producto de una relación, se define **potencia** de una relación R como:

$$R^1 \equiv R$$

$$R^2 \equiv RR$$

$$R^3 \equiv RRR$$

…

$$R^n \equiv R^{n-1}R = RR^{n-1} \forall n > 1$$

§ Se define R^0 como la relación de **identidad**, es decir, si

a R^0 b   si y sólo si   a = b

4.5.3 Algoritmo para calcular el producto de relaciones

Dado que las relaciones se representan por matrices booleanas, una forma de calcular el producto de relaciones es por medio del producto booleano de matrices. Así se define producto de dos relaciones representadas por las matrices A y B a otra matriz C resultado del producto booleano de las dos anteriores.

C = A * B

4.5.3.1 Algoritmo en Pascal

Dado que las matrices que representan una relación sólo están compuestas por 0 y 1, la matriz producto se puede calcular por el siguiente fragmento de programa Pascal, siendo n el tamaño de la matriz.

```
FOR j:= 1 TO n DO
  FOR i:= 1 TO n DO
      IF A[i,j]=1 THEN
         (* Bucle para inspeccionar la fila j de la matriz B si y solo si A[i,j]= 1 *)
            FOR k:=1 TO n DO
                 IF B[j,k]=1 THEN C[i,k]:= 1
                    (* si y solo si A[i,j]=B[j,k]=1 *)
```

4.6 Relación reflexiva

Una relación R se dice que es reflexiva, si se cumple que para todo x

$$x \, R \, x$$

La matriz de una relación reflexiva tiene la diagonal principal con unos.

4.6.1 Ejemplo

La relación *menor o igual* es reflexiva con el conjunto de los números reales.

4.7 Relación simétrica

Una relación R se dice que es simétrica, si se cumple para todo x e y que

$$xRy \Leftrightarrow yRx$$

La matriz de una relación simétrica es simétrica respecto de la diagonal principal.

4.7.1 Ejemplo

La relación *igual* es simétrica con los números reales.

4.8 Relación transitiva

Se dice que una relación R es transitiva si se cumple que

$$\left.\begin{matrix} xRy \\ xRz \end{matrix}\right\} entonces \quad xRz$$

4.8.1 Ejemplo

La relación *MENOR QUE* es transitiva en el conjunto de los números reales.

4.9 Relación de equivalencia

Una relación aplicada a un conjunto que sea simultaneamente reflexiva, simétrica y transitiva se denomina una relación de equivalencia.

4.9.1 Ejemplos de relaciones de equivalencia

Relación	**Aplicada al conjunto**
Igualdad	Números
Semejanza	Triángulos
Tener el mismo nº de patas	Animales
Tener el mismo nº de letras	Palabras
Acabar con la misma letra	Palabras
Tiene el mismo resto que	Enteros

4.9.2 Propiedad de las relaciones de equivalencia

Las relaciones de equivalencia dividen al conjunto al que se aplican en **clases de equivalencia**, de forma que los elementos del conjunto sólo se relacionan con los de su clase de equivalencia, y sólo con ellos.

4.9.2.1 Ejemplo de clases de equivalencia

La relación *ser de la misma paridad* clasifica a los enteros en dos clases de equivalencia: pares e impares.

4.10 Relación irreflexiva

Una relación es irreflexiva si para todos los elementos del conjunto se cumple

$$x \neg R\ x$$

4.10.1 Ejemplo

Relación *mayor que* en el conjunto de los números reales.

4.11 Relación antisimétrica

Una relación es antisimétrica si para todos los elementos del conjunto se cumple

$$x\ R\ y \Leftrightarrow y \neg R\ x$$

4.11.1 Ejemplo

Relación *mayor que* en el conjunto de números reales.

4.12 Inclusión de una relación

Se dice que una relación P incluye a otra relación R, si aRb implica necesariamente aPb.

4.12.1 Ejemplo

La relación *menor o igual* incluye la relación *menor* en el conjunto de los números reales.

4.13 Transpuesta de una relación

La transpuesta de una relación *R* es otra relación *Q* denominada *TRANSPUESTA(R)* tal que

$$Q = TRANSPUESTA(R) \text{ si } cQd \Leftrightarrow dRc$$

4.13.1 Ejemplos

a) En el conjunto de los números reales, la relación *TRANSPUESTA("mayor que")* es "*menor que*".
b) La relación *TRANSPUESTA("ser hijo de")* es "*ser padre de*".

4.14 Cierre transitivo de una relación

Sea un conjunto finito X, que contiene *n* elementos, y una relación *R* en X, que permite obtener las relaciones R^m (m= 1, 2,...). Se puede construir una relación R^+ en X denominada *cierre transitivo de R en x* y dada por

$$R^+ = R \cup R^2 \cup R^3 \cup \ldots$$

Naturalmente, esta construcción necesitará solamente un número finito de potencias de *R* para ser calculada, y se puede realizar fácilmente usando la representación matricial de *R* y la multiplicación booleana de estas matrices.

4.14.1 Teorema del cierre transitivo de una relación

El cierre transitivo R^+ de una relación *R* en un conjunto finito X es transitivo. Es decir, para cualquier otra relación transitiva *P* en X tal que $R \subseteq P$, se puede afirmar que $R^+ \subseteq P$. En este sentido R^+ es la relación transitiva más pequeña contenida en *R*.

4.14.2 Algoritmo de Warshall

El cierre transitivo de la matriz de relación, se puede calcular fácilmente por medio del siguiente algoritmo debido a Warshall. A continuación se muestra una implementación en Pascal. Toma la matriz A(n,n) de la relación *R*, y determina la nueva matriz A^+(n,n) de la relación R^+.

```
FOR j:= 1 TO n DO
      FOR i:= 1 TO n DO
            IF a[i,j]= 1 THEN
              FOR k:= 1 TO n DO
                    IF a[j,k]=1 THEN a[i,k]:= 1
                    (* si y solo si a[i,j]= a[j,k]= 1 *)
```

4.14.3 Ejemplo

Calcular la matriz de la relación cierre transitiva de la relación de dependencia a izquierdas, que se definió en el ejercicio 4.4. Aplicando el algoritmo de *Warshall* se obtiene la matriz siguiente:

		V_n						V_t					
	R^+	S	E	1’	1	P’	P	a	b	(	)	+	⊣
V_n	S	0	1	0	1	0	1	1	1	1	0	0	0
	E	0	0	0	1	0	1	1	1	1	0	0	0
	1’	0	0	0	0	0	0	0	0	0	0	1	0
	1	0	0	0	0	0	1	1	1	1	0	0	0
	P’	0	0	0	0	0	1	1	1	1	0	0	0
	P	0	0	0	0	0	0	1	1	1	0	0	0
V_t	a	0	0	0	0	0	0	0	0	0	0	0	0
	b	0	0	0	0	0	0	0	0	0	0	0	0
	(	0	0	0	0	0	0	0	0	0	0	0	0
	)	0	0	0	0	0	0	0	0	0	0	0	0
	+	0	0	0	0	0	0	0	0	0	0	0	0
	⊣	0	0	0	0	0	0	0	0	0	0	0	0

A la vista de la matriz anterior, se puede construir la siguiente tabla:

Símbolos no terminales	Símbolos terminales por los que se accede desde el no terminal de la izquierda
S	a, b, (
E	a, b, (
1’	+
1	a, b, (
P’	a, b, (
P	a, b, (

4.15 Cierre reflexivo transitivo de una relación

Se llama cierre reflexivo transitivo de una relación R a otra relación R^* tal que

$$aR^*b \quad si\ y\ sólo\ si \begin{cases} a = b \\ o \\ aR^+b \end{cases}$$

Es decir, es una extensión del cierre transitivo, de la forma:

$$R^+ = R^0 \cup R^2 \cup R^3 \cup \ldots$$

Para realizar el cierre reflexivo transitivo de una matriz que representa una relación R, se puede emplear el algoritmo de Warshall con una modificación, que consiste en obligar a todos los términos de la diagonal principal a que sean la unidad. A continuación se muestra el algoritmo de Warshall modificado para el cierre reflexivo transitivo de una relación (en Pascal):

```
FOR j:= 1 TO n DO
  BEGIN
      FOR i:= 1 TO n DO
            IF a[i,j]= 1 THEN
              FOR k:= 1 TO n DO
                    IF a[j,k]= 1 THEN a[i,k]:= 1
                    (* si y solo si a[i,j]= a[j,k]= 1 *)
      a[j,i]:= 1  (* esta modificación *)
  END;
```

5 ANALISIS SINTACTICO DESCENDENTE

El análisis sintáctico (en inglés *parser*) es la fase del procesador de lenguaje que toma como entrada un conjunto de *tokens* enviados por el analizador léxico (*scaner*) y determina si con ellos puede formar una instrucción del lenguaje. Para verificar si una instrucción pertenece al lenguaje construye el árbol sintáctico de la instrucción a partir de los tokens recibidos y que constituirán las hojas del árbol sintáctico. Si el analizador sintactico no puede formar una sentencia produce un mensaje de error, tratando de indicar las causas por las cuales no puede formar la sentencia. Existen dos tipos básicos de analizadores sintácticos para las gramáticas libres de contexto: los **descendentes** (*top-down*) y los **ascendentes** (*bottom-up*). Los ascendentes intentan construir el árbol desde las hojas hasta la raíz, mientras que los descendentes comienzan por la raíz y bajan hasta las hojas.

Uno de los métodos de reconocimiento para las gramáticas de contexto libre son los **analizadores sintácticos descendentes** o también llamados **predictivos** y **orientados hacia un fin**, debido a la forma en que trabajan y construyen el árbol sintáctico. Los analizadores sintácticos descendentes son lo que construyen el árbol sintáctico de la sentencia a reconocer de una forma descendente, comenzando por el símbolo inicial o raíz, hasta llegar a los símbolos terminales que forman la sentencia. El análisis descendente es un procedimiento que crea objetivos y subobjetivos al intentar relacionar una sentencia con su entorno sintáctico. Al comprobar los subobjetivos, se verifican y descartan las salidas falsas y las ramas impropias hasta que se alcanza el subobjetivo propio que son los símbolos terminales que forman la sentencia.

El procedimiento de análisis, en cada encuentro con la estructura sintáctica, tiene que examinar los subobjetivos. Si se cumplen los subobjetivos requeridos, entonces, por definición, se logra el objetivo de mayor rango. En caso contrario, se descarta dicho objetivos superior. Esta secuencia de comprobar, descartar y por último efectuar los objetivos, se programa hacia abajo siguiendo el árbol sintáctico hasta que todas las componentes básicas se hayan acumulado en componentes de nivel superior o hasta que se compruebe que la sentencia es errónea.

Los algoritmos que realizan el análisis descendente deben de cumplir al menos dos condiciones: a) el algoritmo debe saber en todo momento dónde se encuentra dentro del árbol sintáctico; y b) debe poder elegir los subobjetivos (es decir la regla de producción que aplicará).

Históricamente, los compiladores dirigidos por sintaxis, en la forma de análisis recursivo descendente fue propuesta por primera vez en forma explícita por *Lucas (1961)*, para describir un compilador simplificado de ALGOL 60 mediante un conjunto de subrutinas recursivas, que correspondían a la sintaxis BNF. El problema fundamental para el desarrollo de este método fue el retroceso, lo que hizo que su uso práctico fuera restringido. *La elegancia y comodidad de la escritura de compiladores dirigidos por sintaxis fue pagada en tiempo de compilación por el usuario*.

La situación cambió cuando comenzó a realizarse análisis sintáctico descendente sin retroceso, por medio del uso de gramáticas LL(1), obtenidas independientemente por *Foster (1965)* y *Knuth (1967)*. Generalizadas posteriormente por *Lewis*, *Rosenkrantz* y *Stearns* en *1969*, dando lugar a las gramáticas LL(k), que pueden analizar sintácticamente sin retroceso, en forma descendente, examinando en cada paso todos los símbolos procesados anteriormente y los *k* símbolos más a la derecha.

5.1 El problema del retroceso

El primer problema que se presenta con el análisis sintáctico descendente, es que a partir del nodo raíz, el analizador sintáctico no elija las producciones adecuadas para alcanzar la sentencia a reconocer. Cuando el analizador se da cuenta de que se ha equivocado de producción, se tienen que deshacer las producciones aplicadas hasta encontrar otras producciones alternativas, volviendo a tener que reconstruir parte del árbol sintáctico. A este fenómeno se le denomina *retroceso, vuelta atrás* o en inglés *backtracking.*

El proceso de retroceso puede afectar a **otros módulos** del compilador tales como tabla de símbolos, código generado,... Teniendo que deshacerse también los procesos desarrollados en estos módulos.

5.1.1 Ejemplo de retroceso

Para explicar mejor el problema del retroceso, se estudiará el siguiente ejemplo, donde se utiliza la gramática G=(VN, VT, S, P) donde:

VN={<PROGRAMA>, <DECLARACIONES>, <PROCEDIMIENTOS>}
VT={module, d, p, ;, end}
S=<PROGRAMA>

las reglas de producción P son las siguientes:

<PROGRAMA> ::= module <DECLARACIONES>; <PROCEDIMIENTOS> end
<DECLARACIONES> ::= d | d; <DECLARACIONES>
<PROCEDIMIENTOS> ::= p | p; <PROCEDIMIENTOS>

Se desea analizar la cadena de entrada siguiente: `module d ; d ; p ; p end`. A continuación se construye el árbol sintáctico de forma descendente:

a) Se parte del **símbolo inicial**.

<PROGRAMA>

b) Aplicando la **primera regla** de producción de la gramática:

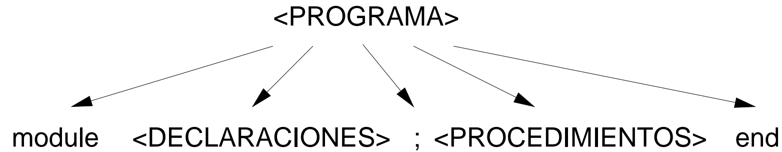


Figura 14

c) Aplicando las **derivaciones más a la izquierda**, se tiene que:

c1) *module* es un símbolo terminal, que coincide con el primero de la cadena a reconocer.

c2) se deriva *<DECLARACIONES>* con la **primera alternativa**.

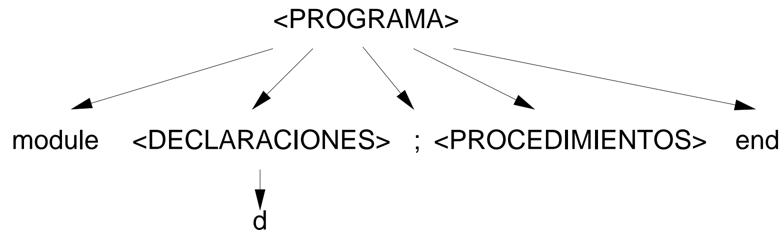


Figura 15

Se comprueba que el nuevo terminal generado *d* coincide con el segundo token de la cadena de entrada.

c3) *;* coincide con el tercer token de la cadena de entrada.

c4) Se deriva *<PROCEDIMIENTOS>* con la **primera alternativa**.

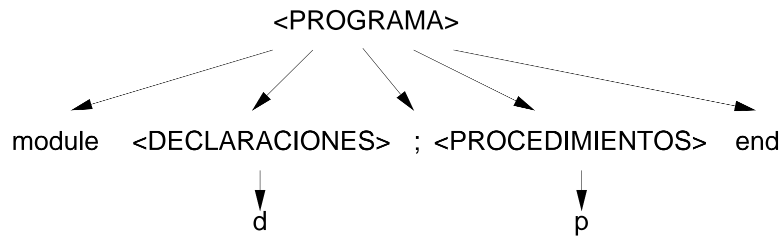


Figura 16

y se llega a un terminal que no es el que tiene la cadena.

c4.1) Se deriva con la **segunda alternativa**.

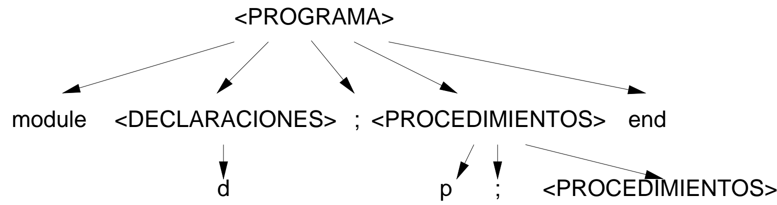


Figura 17

Se observa que el siguiente terminal generado, tampoco coincide con el token de la cadena de entrada.

Entonces el analizador sintáctico debe de **volver atrás**, hasta encontrar la última derivación de un no terminal, y comprobar si tiene alguna **alternativa** más. En caso afirmativo se debe de elegir la siguiente y probar. En caso negativo, **volver más atrás** para probar con el no terminal anterior. Este fenómeno de vuelta atrás es el que se ha definido anteriormente como *retroceso*.

Si en este proceso de marcha atrás se llega al símbolo inicial o raíz, se trataría de una cadena errónea sintácticamente.

Dado que no se ha encontrado el terminal de la cadena de entrada, se ha de volver atrás hasta el no-terminal analizado anteriormente, que en este caso es *<DECLARACIONES>*, es decir, se pasa a c2) y en vez de tomar la primera alternativa se toma la **segunda alternativa**.

c2.bis)

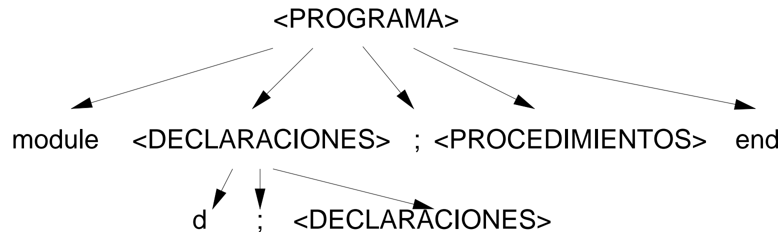


Figura 18

Llegados a este punto, también se debe de retroceder en la cadena de entrada hasta la primera *d*, ya que en el proceso de vuelta atrás lo único válido que nos ha quedado del árbol ha sido el primer token *module*.

c2bis1.1) Si se deriva *<DECLARACIONES>* nuevamente con la primera alternativa, se tiene

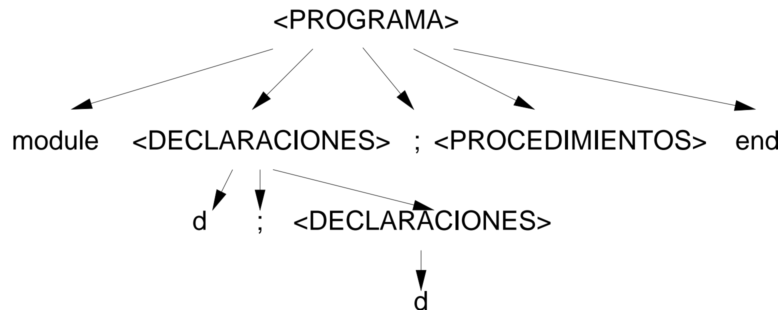


Figura 19

c3.bis) En este momento se tienen reconocidos los primeros 5 tokens de la cadena.

c4.bis) Se deriva el no terminal *<PROCEDIMIENTOS>*, con la primera alternativa, y el árbol resultante se muestra en la figura 20. Se ha reconocido el sexto token de la cadena (*p*).

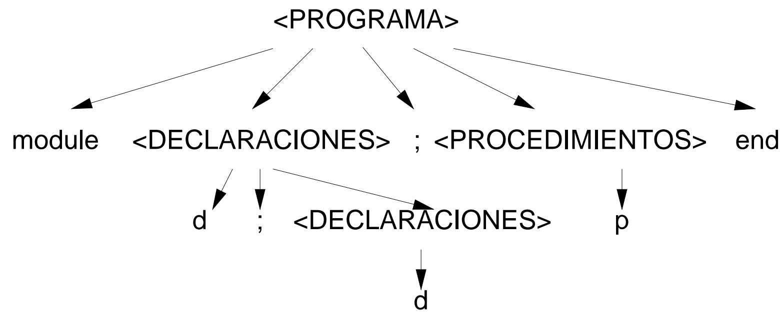


Figura 20: Reconocimiento de *module d ; d ; p ...*

c5) El siguiente token de la cadena es *;*, mientras que en el árbol se tiene *end*, por lo tanto habrá que **volver atrás** hasta el anterior no terminal, y mirar si tiene alguna otra alternativa.

c4bisbis) El último no terminal derivado es *<PROCEDIMIENTOS>*, si se deriva con su otra alternativa se tiene el árbol de la figura 21. Con esta derivación se ha reconocido la parte de la cadena de entrada *module d ; d ; p ;*

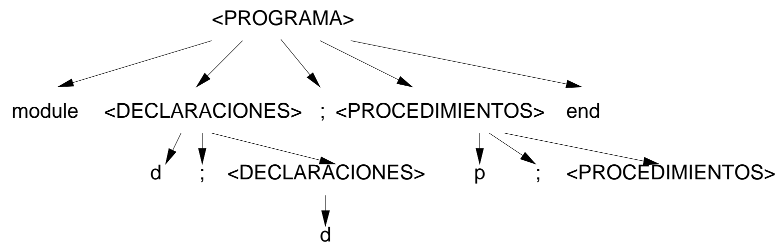


Figura 21: Reconocimiento de *module d ; d ; p ; ...*

c4bisbis.1) Se deriva *<PROCEDIMIENTOS>* con la primera alternativa y se obtiene el árbol sintáctico de la figura 22, reconociéndose *module d ; d ; p ; p...*

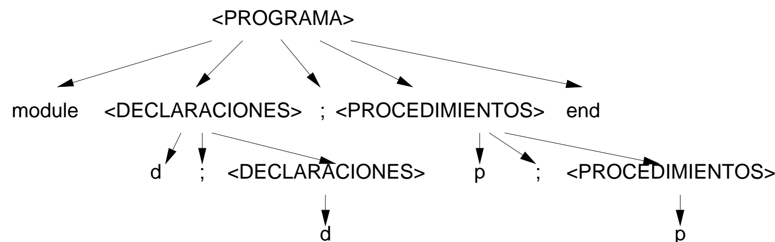


Figura 22: Reconocimiento de *module d ; d ; p ; p end*

c5bis) El árbol sintáctico de la figura 22 ya acepta el siguiente token de la cadena ('*end*') y por tanto la cadena completa.

Puede concluirse, que los tiempos de reconocimiento de sentencias de un lenguaje pueden dispararse a causa del retroceso, por lo tanto los analizadores sintácticos deben eliminar las causas que producen el retroceso.

5.2 La recursividad a izquierdas

Se dice que una gramática tiene recursividad a izquierdas ("*left-recursive*"), si existe un no terminal A, tal que para algún $\alpha \in V^*$ existe una derivación de la forma:

$$A \overset{+}{\rightarrow} A\alpha$$

donde el signo + encima de la flecha indica que al menos existe una producción.

Si el analizador sintáctico toma el primer no terminal (el más a la izquierda), el árbol sintáctico es el de la figura 23. La recursividad a izquierdas deja al analizador sintáctico en un bucle infinito, ya que al expandir A, se encontraría con otro A, y así sucesivamente, sin avanzar en la comparación del token de entrada. Es preciso eliminar la recursividad a izquierdas con técnicas que se estudiarán más adelante.

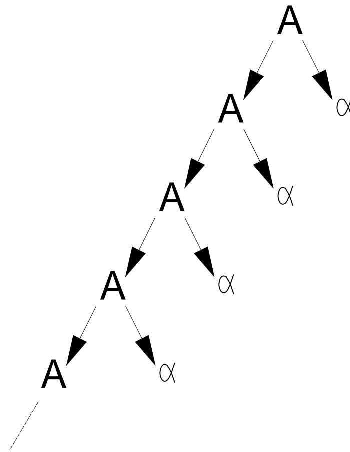


Figura 23: Recursividad a izquierda

5.2.1 Ejemplo de recursividad a izquierdas

Sea la gramática G= (VN, VT, S, P) donde VN= {A, S}, VT= {a, b, c}, y las reglas de producción P son las siguientes:

$S \to aAc$
$A \to Ab \mid \lambda$

el lenguaje reconocido por esta gramática es L= $\{ab^n c \ / \ n \geq 0\}$. Supongamos que el analizador sintáctico desea reconocer la cadena: *abbc*, se obtienen los árboles sintácticos de la figura 24. No saliendo el analizador sintáctico de este bucle.

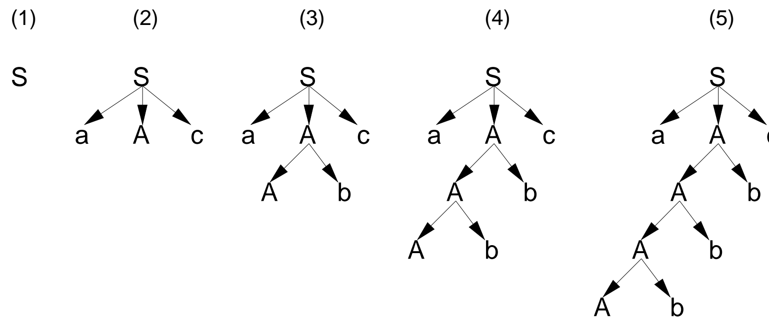


Figura 24: Bucle infinito tratando de reconocer *abbc*.

a) Una forma de resolver este problema es volviendo a escribir la segunda producción de la gramática de esta otra manera:

$$A \to \lambda \mid Ab$$

entonces se construyen los árboles sintácticos que se muestran en la figura 25.

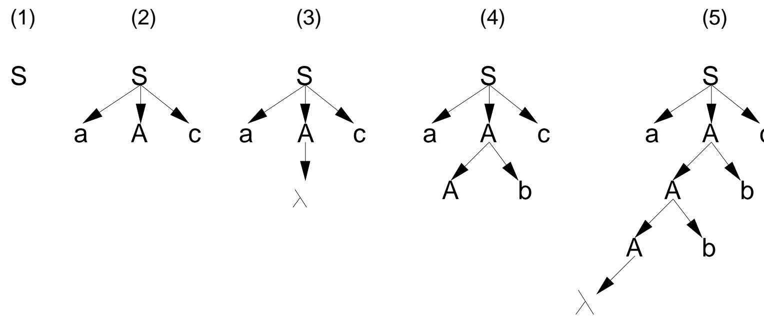


Figura 25: Reconocimiento de la cadena *abbc*.

b) Otra forma de resolver el problema es cambiando en la regla conflictiva el orden *Ab* a *bA*, que a su vez cambia la gramática, pero se obtiene una gramática equivalente, es decir que genera el mismo lenguaje.

$$A \rightarrow bA \mid \lambda$$

Para reconocer *abbc* se obtienen los árboles sintácticos de la figura 26.

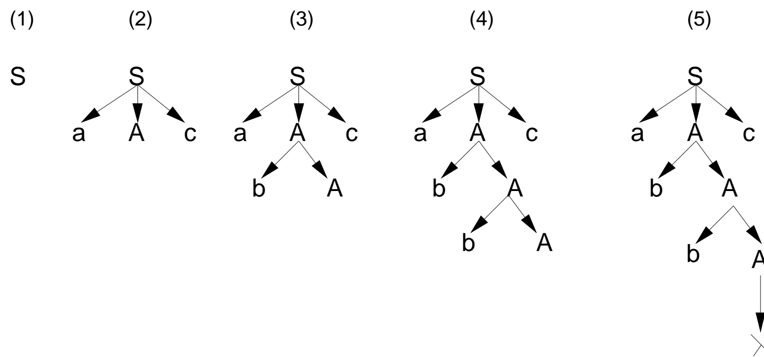


Figura 26: Reconocimiento de la cadena *abbc*.

No siempre es tan fácil eliminar la recursividad a izquierdas, en el apartado 5.4.5 *Transformación de gramáticas*, se estudian otros métodos para eliminar la recursividad a izquierdas.

5.3 Análisis sintáctico descendente con retroceso

La forma en que se hace el análisis descendente con retroceso ya se ha visto en el ejemplo del apartado 5.1, aquí se expone el algoritmo general para construir un analizador sintáctico descendente con retroceso.

5.3.1 Algoritmo de análisis sintáctico descendente con retroceso

1.- Se colocan las reglas de la gramática según un orden preestablecido para cada uno de los no terminales de la que está compuesta.

2.- Se comienza la construcción del árbol sintáctico a partir del símbolo inicial, y aplicando las siguientes reglas en forma recursiva. Al nodo en proceso de expansión en un determinado momento se le llamará **nodo activo**.

3.- Cada nodo activo escoge la primera de sus alternativas posibles, por ejemplo, para el *no terminal* A con la regla$A \rightarrow x_1 x_2 \ldots x_n$ crea *n* descendientes directos, y el nodo activo en ese momento pasa a ser *el primer descendiente por la izquierda*. Cuando se deriven todos los descendientes, pasará a ser nodo activo el más inmediato derecho de A susceptible de ser derivado. En el caso de que la regla fuese:$A \rightarrow x_1 \mid x_2 \mid \ldots \mid x_n$ se elegirá en un principio la alternativa de más a la izquierda.

4.- Si el nodo activo es un **terminal** deberá entonces compararse el símbolo actual de la cadena a analizar con dicho nodo. *Si son iguales* se avanza un token de entrada y el nuevo símbolo actual será el situado más a la derecha del terminal analizado.

Si *no son iguales* se **retrocede** hasta un nodo *no terminal* y se reintenta eligiendo la **siguiente alternativa**. Si aún así no existe ninguna alternativa posible se retrocede al no terminal anterior, y así sucesivamente. Si se llega al símbolo inicial la cadena no pertenece al lenguaje.

5.3.2 Corolario

Una gramática de contexto libre, que es una gramática limpia y no es recursiva a izquierdas, para cualquier cadena de símbolos de su alfabeto terminal existe un *número finito* de posibles análisis a izquierda desde el símbolo inicial para reconocerla o no.

A partir de todo lo expresado anteriormente, se pueden construir analizadores sintácticos descendentes con retroceso. Su principal problema es el tiempo de ejecución.

5.4 Análisis descendente sin retroceso

Para eliminar el retroceso en el análisis descendente, se ha de elegir correctamente la producción correspondiente a cada no terminal que se expande. Es decir que el análisis descendente ha de ser **determinista**, y sólo se debe de dejar tomar una opción en la expansión de cada no terminal.

5.4.1 Gramáticas LL(k)

Las gramáticas LL(k) son un subconjunto de las gramáticas libres de contexto. Permiten un análisis descendente determinista (o sin retroceso), por medio del reconocimiento de la cadena de entrada de izquierda a derecha ("*Left to right*") y que va tomando las derivaciones más hacia la izquierda ("*Leftmost*") con sólo mirar los *k* tokens situados a continuación de donde se halla.

5.4.1.1 Teorema de la no ambigüedad de las gramáticas LL(k)

Toda gramática LL(k) es no ambigua.

Demostración: Por definición de gramática LL(k).

5.4.1.2 Teorema

Una gramática LL(k) no es recursiva a izquierdas.

Demostración: Por definición de gramática LL(k).

5.4.2 Gramáticas LL(1)

Antes de definir completamente las gramáticas LL(1), se definirán otros tipos de gramáticas más sencillas que son LL(1). La introducción de estas gramáticas permitirá una aproximación paso a paso hacia la definición completa de las gramáticas LL(1).

5.4.2.1 S-gramáticas

Las S-gramáticas son un subconjunto muy restringido de las gramáticas LL(1), debido a dos condiciones muy fuertes. Una S-gramática es una gramática libre de contexto que debe complir las siguientes dos condiciones:

1ª) Todas las partes derechas de cada producción comienzan con un símbolo terminal.

2ª) Si dos producciones tienen la misma parte izquierda, entonces su parte derecha comienza con diferentes símbolos terminales. Es decir las distintas producciones de cada no terminal $A \in VN$, deben comenzar por distinto símbolo terminal. Así si se tienen la producciones del no terminal A: $A \rightarrow a_1\alpha_1 \mid a_2\alpha_2 \mid \dots \mid a_m\alpha_m$ se debe cumplir que $a_i \neq a_j \quad \forall i \neq j \qquad a_i \in VT \quad \alpha_i \in V^* \quad 1 \leq i \leq m$

La primera condición es similar a decir que la gramática está en la *Forma Normal de Greibach* (FNG), excepto que los terminales de comienzo de la parte derecha de cada producción, pueden estar seguidos por simbolos no terminales y/o terminales. La segunda condición ayudará a escribir los analizadores sintácticos descendentes sin retroceso (o deterministas), ya que permitirán siempre elegir la derivación correcta, con sólo mirar un token hacia adelante.

5.4.2.1.1 Corolario de la definición de S-gramáticas

Toda S-gramática es LL(1), la inversa no es cierta. Toda S-gramática es una gramática LL(1) dado que una vez que analiza un *token* es posible determinar que regla de producción se debe aplicar.

5.4.2.1.2 Contraejemplo de S-gramática

Sea la gramática:

(1) <S> → a <T>
(2) <S> → <T> b <S>
(3) <T> → b <T>
(4) <T> → b a

No es una s-gramática, ya que la parte derecha de la producción (2) no comienza por un terminal como exige la *primera condición*. Además las producciones (3) y (4) no cumplen la *segunda condición*.

5.4.2.1.3 Ejemplo de S-grmática

Sea la gramática:

<S> → a b <R>
<S> → b <R> b <S>
<R> → a
<R> → b <R>

Obviamente *es una s-gramática*, y por tanto es una gramática LL(1).

5.4.2.1.4 Ejemplo de S-gramática

Sea la gramática:

<S> → p <X>
<S> → q <Y>
<X> → a <X> b
<X> → x
<Y> → a <Y> d
<Y> → y

Obviamente *es una s-gramática*, y por tanto es una gramática LL(1).

5.4.2.2 Conjunto de símbolos iniciales o cabecera

Se define el conjunto de *símbolos iniciales o cabecera* (en inglés *FIRST*) de un símbolo no terminal A, como los símbolos terminales que pueden aparecer en primer lugar de cualquier cadena generada por A. La definición anterior se puede expresar como:

$$INICIALES(A) = \{x / A \Rightarrow x \ldots \quad siendo \quad x \in VT\}$$

Otra forma de definir el conjunto iniciales, es por medio de la relación PRIMERO. Se define la *relación PRIMERO*, de la forma siguiente *"Dos símbolos A y x están relacionados por la relación binaria primero, y se escribe A PRIMERO x si y sólo si existe una regla de producción de la gramática de la forma: A→ x..."*. El cierre transitivo de la relación primero sería A PRIMERO$^+$ x si y sólo si existe una cadena de reglas de producción de la forma:

$A \to A_1$
$A_1 \to A_2$
....
$A_N \to x$

Lo que implica que A PRIMERO$^+$ x si $A \overset{+}{\Rightarrow} x$

A partir del cierre transitivo de la relación PRIMERO se puede definir el conjunto de símbolos iniciales de un no terminal A como:

$$INICIALES(A) = \{x / (A, x) \in PRIMERO^+, \quad x \in VT\}$$

También puede extenderse la definición de conjunto de INICIALES de un símbolo no terminal a conjunto de INICIALES de una cadena de símbolos $\alpha \in V$.

5.4.2.2.1 Ejemplo de cálculo del conjunto de Iniciales

Sea la gramática

S' → S#
S → ABe
A → dB
A → aS
A → c
B → AS
B → b

INICIALES(A)={d, a, c}, pues existen las reglas:

A → dB
A → aS
A → c

INICIALES(S')=INICIALES(A)={d, a, c}, pues existen las reglas:

S'→ S#
S → ABe

INICIALES(B)=INICIALES(A) ∪ {b} ={d, a, c, b}.

5.4.2.3 Gramáticas LL(1) simples

Las gramáticas LL(1) simples son un subconjunto de las gramáticas LL(1), con las dos restricciones siguientes:

- No se permiten símbolos no terminales que deriven a vacio. Es decir no se permiten **producciones vacías** o reglas-λ, cuya parte derecha es la cadena vacía λ.
- Las distintas producciones de cada no terminal $A \in VN\ A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ deben cumplir los conjuntos INICIALES(α_1), INICIALES(α_2),..., INICIALES(α_n) son disjuntos entre sí, es decir $INICIALES(\alpha_i) \cap INICIALES(\alpha_j) = \emptyset \quad \forall i \neq j$

5.4.2.3.1 Corolario de las gramáticas LL(1) simples

Toda gramática LL(1) simple es LL(1), la inversa no es cierta

5.4.2.3.2 Teorema de equivalencia entre las gramáticas LL(1) y las S-gramáticas

Dada una gramática LL(1) simple siempre es posible encontrar una S-gramática equivalente.

5.4.2.3.3 Ejemplo de gramática LL(1) simple

Sea la gramática cuyas reglas de producción son:

S' → S#
S → ABe
A → dB
A → aS
A → c
B → AS
B → b

Se desea verificar si es o no LL(1) simple.

a) Dadas las producciones A → dB | aS | c

$$INICIALES(dB) \cap INICIALES(aS) = \{d\} \cap \{a\} = 0$$

$$INICIALES(dB) \cap INICIALES(c) = \{d\} \cap \{c\} = 0$$

$$INICIALES(aS) \cap INICIALES(c) = \{a\} \cap \{c\} = 0$$

b) Dadas las producciones B → AS | b

$INICIALES(AS) \cap INICIALES(b) = \{a, c, d\} \cap \{b\} = 0$

Luego *esta gramática es LL(1) simple.*

5.4.2.3.4 Ejemplo de gramática LL(1) simple

Sea la gramática cuyas de reglas de producción son:

(0) <S'> → <S> #
(1) <S> → a <S>
(2) <S> → b <A>
(3) <A> → d
(4) <A> → c c <A>

Claramente es una gramática LL(1) simple, que permite reconocer una cadena sin retroceso. Así con esta gramática se desea reconocer la cadena: *aabccd#*. Realizando derivaciones más a la izquierda se obtiene:

<S>' → <S># → aa <S># → aab <A># → aabcc <A># → aabccd#

siendo el árbol sintáctico el que se muestra en la figura 27.

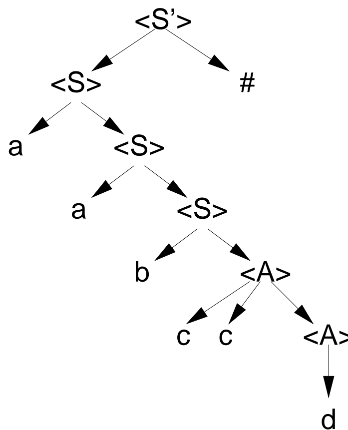


Figura 27: Reconocimiento sin retroceso de *aabccd#*

5.4.2.4 Conjunto de símbolos seguidores o siguientes

Dada una gramática libre de contexto con un símbolo inicial *S*, para un símbolo no terminal *A*, se dice que el conjunto de símbolos *SEGUIDORES(A)* es el conjunto de símbolos terminales que pueden seguir inmediatamente a una cadena derivada de *S*. La definición formal se puede expresar de la siguiente forma:

$$SEGUIDORES(A) = \{a / S \rightarrow \beta A \delta, \beta, \delta \in (VT \cup VN)^*, A \in VN, a \in INICIALES(\delta)\}$$

Los símbolos seguidores o siguientes (en inglés se denominan FOLLOW) de un símbolo no terminal también se pueden definir como los símbolos iniciales del símbolo que le sigue.

5.4.2.4.1 Ejemplo de cálculo del conjunto de Seguidores

Sea la gramática:

<PROGRAMA>	::= module <DECLARACIONES> <PROCEDIMIENTOS> end
<DECLARACIONES>	::= d <A>
<A>	::= <vacío> | ; <DECLARACIONES>
<PROCEDIMIENTOS>	::= p <B>
<B>	::= <vacío> | ; <PROCEDIMIENTOS>

Cálculo del conjunto de seguidores del símbolo no terminal *A*.

<PROGRAMA> → module d <A> <PROCEDIMIENTOS> end

INICIALES (<PROCEDIMIENTOS>) = { p }

Cálculo de los símbolos seguidores de <B>

<PROGRAMA> → module <DECLARACIONES> p <B> end

SEGUIDORES(<B>)= { end }

5.4.2.5 Conjunto de símbolos Directores

Los símbolos directores (en inglés *SELECT*) de una producción como su nombre indica son los que dirigen al analizador sintactico para elegir la alternativa adecuada, se pueden definir como el conjunto de símbolos terminales que determinarán que expansión de un no terminal se ha de elegir en un momento dado, con solo mirar un símbolo hacia adelante. La definición formal se puede enunciar de la siguiente forma, dada una producción $A \rightarrow \alpha$ donde A es un símbolo no terminal, y α es una cadena de símbolos terminales y no terminales. Entonces se define **conjunto de símbolos directores** SD(A,α) de una producción $A \rightarrow \alpha$ como:

$$SD(A,\alpha)\begin{cases} INICIALES(\alpha) & si\ \alpha\ es\ no\ anulable \\ INICIALES(\alpha) \cup SEGUIDORES(A) & si\ \alpha\ es\ anulable \end{cases}$$

5.4.2.5.1 Ejemplo de cálculo del conjunto de símbolos Directores

Sea la gramática:

<PROGRAMA>	::= module <DECLARACIONES> <PROCEDIMIENTOS> end
<DECLARACIONES>	::= d <A>
<A>	::= <vacío> | ; <DECLARACIONES>
<PROCEDIMIENTOS>	::= p <B>
<B>	::= <vacío> | ; <PROCEDIMIENTOS>
<vacío>	::= λ

a) Calcular el conjunto de símbolos directores de la producción: $< A > \rightarrow < vacío >$

<vacío> es anulable, luego:

SD (<A>,<vacío>) = INICIALES(<vacío>) ∪ SEGUIDORES (<A>) = ∅ ∪ { p } = { p }

b) Calcular el conjunto de símbolos directores de la producción: $< A > \rightarrow\ ; < DECLARACIONES >$

SD(<A>, ; <DECLARACIONES>) = { ; }

5.4.2.6 Definición del las gramáticas LL(1)

La condición necesaria y suficiente para que una gramática limpia sea LL(1), es que los símbolos directores correspondientes a las diferentes expansiones de cada símbolo no terminal sean conjuntos disjuntos.

La **justificación** de esta condición es simple. La **condición es necesaria**, puesto que si un símbolo aparece en dos conjuntos de símbolos directores, el analizador sintáctico descendente no puede decidir (sin recurrir a información posterior) qué expansión aplicar. La **condición es suficiente**, puesto que el analizador siempre puede escoger una expansión como un símbolo dado, y esta alternativa será siempre correcta. Si el símbolo no está contenido en ninguno de los conjuntos de los símbolos directores, la cadena de entrada no pertenecerá al lenguaje y se tratará de un error.

5.4.3 Condiciones de las gramáticas LL(1)

La definición original dada por *D.E. Knuth*, de gramáticas LL(1) consta de cuatro condiciones equivalentes a la definición dada en el epígrafe anterior.

5.4.3.1 Primera condición de *Knuth*

No se permitirán producciones de la forma $A \overset{*}{\rightarrow} A\alpha$ donde $A \in VN$ y $\alpha \in V^*$. Esta condición equivale a no admitir la recursividad a izquierdas.

5.4.3.2 Segunda condición de *Knuth*

Los símbolos terminales que pueden encabezar las distintas alternativas de una regla de producción deben formar conjuntos disjuntos. Es decir, si

$$A \overset{*}{\rightarrow} B\beta \mid C\gamma \qquad A,B,C \in VN \quad \beta,\gamma \in V^*$$

no debe ocurrir que

$$B \overset{*}{\rightarrow} dS \qquad d \in VT$$
$$C \overset{*}{\rightarrow} d\psi \qquad S,\psi \in V^*$$

Esto implica que en todo momento, el símbolo terminal que estamos examinando señala sin ambigüedad, que alternativa se ha de escoger sin posibilidad de error y, en consecuencia, sin retroceso.

5.4.3.2.1 Tercera condición de *Knuth*

Si una alternativa de una producción de un símbolo no terminal origina la cadena vacía, los símbolos terminales que pueden seguir a dicho no-terminal en la producción donde aparezca, han de formar un conjunto disjunto con los terminales que pueden encabezar las distintas alternativas de dicho terminal. Expresado de otra forma, sea la cadena A_1... A_2... $A_3A_4A_5$ y sea A_3 el símbolo que se está analizando, además se tienen las producciones:

$$A_3 \rightarrow ax \mid \lambda$$
$$A_4 \rightarrow A_3ay$$

Dado que A_3 puede derivar a la cadena vacía, puede darse el caso de que:

INICIALES(A_3)= { a }
INICIALES(A_4)= { a }

y no puede determinarse si se ha de elegir la producción de A_3 o de A_4.

5.4.3.2.2 Cuarta condición de *Knuth*

Ningún símbolo no terminal puede tener dos o más alternativas que conduzcan a la cadena vacía. Esta condición deriva de la anterior. Así por ejemplo no se permite:

$$X \rightarrow A \mid B$$
$$A \rightarrow \lambda \mid C$$
$$B \rightarrow \lambda \mid D$$

5.4.4 Algoritmo de decisión

Existe un algoritmo diseñado por *Lewis et al. [LEWI76, pág. 262-276]*, traducido y adaptado por *Sánchez Dueñas et al. [SANC84, pág 86-95]*, que permite decidir si una gramática es o no LL(1). Es decir, si con el diseño de una gramática para un lenguaje de programación, se puede construir un analizador sintáctico descendente determinista, con solo examinar el siguiente token de entrada. Los distintos pasos del algoritmo se encaminan a construir los conjuntos de símbolos directores para cada expansión de un símbolo no terminal, y aplicar la condición necesaria y suficiente para que la gramática sea LL(1).

Para construir los conjuntos citados se ejecutan los siguientes pasos:

PASO 1: Encontrar los símbolos no terminales y producciones que sean anulables.
PASO 2: Construcción de la relación EMPIEZA DIRECTAMENTE CON.
PASO 3: Calcular la relación EMPIEZA CON.
PASO 4: Calcular el conjunto de INICIALES de cada no terminal.
PASO 5: Calcular el conjunto de INICIALES de cada producción.
PASO 6: Construcción de la relación ESTA DIRECTAMENTE SEGUIDO POR.
PASO 7: Construcción de la relación ES FIN DIRECTO DE.
PASO 8: Construcción de la relación ES FIN DE.
PASO 9: Construcción de la relación ESTA SEGUIDO POR.
PASO 10: Calcular el conjunto de SEGUIDORES de cada no terminal anulable.
PASO 11: Calcular el conjunto de SIMBOLOS DIRECTORES.

PASO 1: Encontrar los símbolos no terminales y producciones anulables

Consiste en aplicar un algoritmo que comprueba qué no terminales generan la cadena vacía. El algoritmo se basa en utilizar un **vector** cuyos índices son elementos no terminales, y los elementos del vector solamente pueden tomar estos tres valores:

- **SI** (el símbolo no terminal *SI* genera λ)
- **NO** (el no terminal *NO* genera λ)
- **INDECISO** (no se sabe si genera λ)

Los pasos del *algoritmo* son:

1.- Inicializar todos los elementos del vector con *INDECISO*.

2.- Efectuar las siguiente acciones en la primera pasada de la gramática:

a) Si una producción de un no terminal es la cadena vacía, el correspondiente elemento del vector toma el valor *SI*, y se eliminan de la gramática todas las producciones de dicho no terminal.

b) Se eliminan de la gramática todos los no terminales cuyas producciones contengan un símbolo terminal. Si la acción elimina todas las producciones de un no-terminal su correspondiente valor del vector toma el valor *NO*.

3.- En este momento, la gramática está limitada a las *producciones* cuya *parte derecha contiene sólo símbolos no terminales*. En las siguientes pasadas se examinan cada uno de los símbolos de la parte derecha.

a) Si para un no terminal de la parte derecha su elemento en el vector vale *SI*, se elimina ese símbolo de la parte derecha. Si este deja como parte derecha la cadena vacía, el elemento correspondiente del vector al no terminal de la parte izquierda *toma el valor SI*, y se eliminan todas las producciones correspondientes a dicho no terminal.

b) Si para un no terminal de la parte derecha su elemento del vector vale *NO*, se elimina la producción. Si todas las producciones de un no terminal se eliminan de esta manera, su entrada en el vector *toma el valor NO*.

4.- Si durante una *pasada completa de la gramática* no se cambia ninguna entrada del vector, y *existen todavía elementos del vector con el valor INDECISO, termina el algoritmo* y *la gramática no es LL(1)*.

Si una gramática no pasa de este primer paso, es a la vez *recursiva a izquierdas* y *sucia*.

a) Es **sucia** ya que existen producciones que constan sólo de símbolos no terminales que no pueden generar cadenas compuestas sólo por terminales. Es decir, hay símbolos no terminales muertos.

b) Es **recursiva a izquierdas**, puesto que son un conjunto finito y aún quedan los elementos del vector como INDECISOS, por tanto, los miembros más a la izquierda deben de formar un bucle.

Si la **gramática es limpia**, siempre se podrá generar un vector cuyos índices son los no terminales y los elementos son todos SI o NO.

5.4.4.1.1 Ejemplo de cálculo de los símbolos anulables

Sea la gramática G= (VT, VN, P, S) donde:

VN= {<E>, <A>, <B>, <C>, <D>}
VT= {+, id}
S= <E>

y las reglas de producción P son:

<E> ::= <E>+<E> | <A> + <B> | id

<B> ::= <C><A>

<C> ::= <B>

<D> ::= <E>

<A> ::= λ

Se trata de aplicar el algoritmo expuesto en el epígrafe anterior.

1. Construcción del vector cuyos índices son no terminales, y se inicializa a indeciso.

<E>	<A>	<B>	<C>	<D>
IND	IND	IND	IND	IND

Figura 28: vector de los símbolos no terminales inicializados a INDECISO

2. a) La producción <A> ::= <VACIO>, hace que el elemento correspondiente a <A> tome el valor *SI* en el vector, y se eliminan todas las producciones de <A> que en este caso sólo es una.

<E>	<A>	<B>	<C>	<D>
IND	SI	IND	IND	IND

Figura 29: vector de símbolos no terminales

En la gramática sólo quedan las producciones:
<E> ::= <E>+<E> | <A>+<B> |id
<B> ::= <C><A>
<C> ::= <B>
<D> ::= <E>

b) Se eliminan de la gramática los no terminales cuyas producciones tengan un símbolo terminal. En este caso es el <E>, y su correspondiente elemento del vector toma el valor *NO*.

<E>	<A>	<B>	<C>	<D>
NO	SI	IND	IND	IND

Figura 30: vector de símbolos no terminales

En la gramática sólo quedan las producciones:
<B> ::= <C><A>
<C> ::= <B>
<D> ::= <E>

3. En este momento sólo hay producciones cuya parte derecha sólo contiene símbolos no terminales. Se examina a continuación cada uno de los símbolos de la parte derecha de cada producción:
   a) Se toma la producción B:= <C><A> dado que <A> ya tiene el valor si se elimina con lo que queda la gramática:
      <B>:= <C>
      <C>:= <B>
      <D>:= <E>
      el *vector no sufre cambios*.
   b) Se toma la producción <D>::= <E> y dado que <E> tiene valor NO, se elimina la producción, y en el vector el elemento correspondiente a <D> toma el valor NO y la gramática queda:
      <B>:= <C>
      <C>:= <E>

<E>	<A>	<B>	<C>	<D>
NO	SI	IND	IND	NO

Figura 31: vector de símbolos no terminales

4. Si se da otra pasada a la gramática, el vector no cambia, y quedan <B> y <C> como indecisos. Luego la gramática es *sucia* y *recursiva a izquierdas* como puede observarse.

También se puede utilizar el programa analizador de gramáticas LL(1), en el cual la gramática debe introducirse en el formato siguiente pues no admite el símbolo alternativa.

```
<E> ::= <E> + <E>
<E> ::= <A> + <B>
<E> ::= id
<A> ::=
<B> ::= <C> <A>
<C> ::= <B>
<D> ::= <E>
```

El resultado que se obtiene en el programa es:

```
RELACION DE SIMBOLOS INDECISOS: B C
RELACION DE SIMBOLOS ANULABLES: A
RELACION DE SIMBOLOS NO ANULABLES: D E
RELACION DE SIMBOLOS MUERTOS: B C
RELACION DE SIMBOLOS NO ACCESIBLES: D
```

PASO 2: Construcción de la relación EMPIEZA DIRECTAMENTE CON

Se define que *A EMPIEZA DIRECTAMENTE CON B, si derivando A se puede obtener B como principio de cadena derivada de A*. Los no terminales anulables se reemplazan por el vacío. $A \in VN$ y $B \in V^+$. Es decir, si y sólo si existe una producción de la forma $A \to \alpha B \beta$ siendo:

$A \in VN$
α una cadena anulable, es decir que puede derivar a vacio
$B \in V^+$
β una cadena cualquiera

5.4.4.2.1 Ejemplo de cálculo de la relación EMPIEZA DIRECTAMENTE CON

Sea una gramática elemental del lenguaje MUSIM.

```
<PROGRAMA>              ::= <BLOQUE>.
<BLOQUE>                ::= <SENTENCIA> <OTRA SENTENCIA>
<OTRA SENTENCIA>        ::= <SENTENCIA> <OTRA SENTENCIA> | <vacío>
<SENTENCIA>             ::= <ASIGNACION> | <LECTURA> | <ESCRITURA>
<ASIGNACION>            ::= <VARIABLE> = <EXPRESION>
<EXPRESION>             ::= <TERMINO><MAS TERMINOS>
<MAS TERMINOS>          ::= + <TERMINO><MAS TERMINOS> |
                             - <TERMINO><MAS TERMINOS> |
                             <vacío>
<TERMINO>               ::= <FACTOR><MAS FACTORES>
<MAS FACTORES>          ::= * <FACTOR><MAS FACTORES> |
                             / <FACTOR><MAS FACTORES> |
                             % <FACTOR><MAS FACTORES> |
                             <vacío>
<FACTOR>                ::= (<EXPRESION>) | <VARIABLE> | <CONSTANTE>
<LECTURA>               ::= R <VARIABLE>
<ESCRITURA>             ::= W <VARIABLE>
<CONSTANTE>             ::= dígito
<VARIABLE>              ::= letra_minúscula
<vacío>                 ::= λ
```

Determinar la relación EMPIEZA DIRECTAMENTE POR.

<PROGRAMA>	*empieza directamente por*	<BLOQUE>
<BLOQUE>	*empieza directamente por*	<SENTENCIA>
<OTRA SENTENCIA>	*empieza directamente por*	;
<SENTENCIA>	*empieza directamente por*	<ASIGNACION> <LECTURA> <ESCRITURA>
<ASIGNACION>	*empieza directamente por*	<VARIABLE>
<EXPRESION>	*empieza directamente por*	<TERMINO>
<MAS TERMINOS>	*empieza directamente por*	+

		-
<TERMINO>	*empieza directamente por*	<FACTOR>
<MAS FACTORES>	*empieza directamente por*	*
		/
		%
<FACTOR>	*empieza directamente por*	(
		<VARIABLE>
		<CONSTANTE>
<LECTURA>	*empieza directamente por*	R
<ESCRITURA>	*empieza directamente por*	W
<CONSTANTE>	*empieza directamente por*	dígito
<VARIABLE>	*empieza directamente por*	letra_minúscula

Esta relación se puede representar mediante la siguiente matriz EMPIEZA DIRECTAMENTE CON.

	ASIGNACION	BLOQUE	CONSTANTE	ESCRITURA	EXPRESION	FACTOR	LECTURA	MAS_FACTORES	MAS_TERMINOS	OTRA_SENTENCIA	PROGRAMA	SENTENCIA	TERMINO	VARIABLE	R	W	%	(	*	+	-	.	/	;	=	)	dígito	letra_minúscula
ASIGNACION														1														
BLOQUE												1																
CONSTANTE																											1	
ESCRITURA																1												
EXPRESION													1															
FACTOR			1											1				1										
LECTURA															1													
MAS_FACTORES																	1		1				1					
MAS_TERMINOS																				1	1							
OTRA_SENTENCIA																								1				
PROGRAMA		1																										
SENTENCIA	1			1			1																					
TERMINO						1																						
VARIABLE																												1
R																												
W																												
%																												
(																												
*																												
+																												
-																												
.																												
/																												
;																												
=																												
)																												
dígito																												
letra_minúscula																												

Matriz "EMPIEZA DIRECTAMENTE CON"

PASO 3: Construcción de la relación EMPIEZA CON

Se define que *A EMPIEZA CON B si y sólo si existe una cadena derivada de A que empiece con B*. También se admite que A EMPIEZA CON A. Se puede demostrar que la relación EMPIEZA CON es el **cierre reflexivo transitivo de la relación EMPIEZA DIRECTAMENTE CON**. Para determinar la nueva matriz de relación se puede utilizar el *algoritmo de Warshall modificado* (apartado 4.15).

5.4.4.3.1 Ejemplo de cálculo de la relación EMPIEZA CON

Continuando con el ejemplo del paso anterior (apartado 5.4.4.2.1), se obtiene la matriz EMPIEZA CON aplicando el algoritmo de *Warshall modificado*.

	ASIGNACION	BLOQUE	CONSTANTE	ESCRITURA	EXPRESION	FACTOR	LECTURA	MAS_FACTORES	MAS_TERMINOS	OTRA_SENTENCIA	PROGRAMA	SENTENCIA	TERMINO	VARIABLE	R	W	%	(	*	+	-	.	/	;	=	)	dígito	letra_minúscula
ASIGNACION	1													1														1
BLOQUE	1	1		1			1					1		1	1	1												1
CONSTANTE			1																								1	
ESCRITURA				1												1												
EXPRESION			1		1	1							1	1				1									1	1
FACTOR			1			1								1				1									1	1
LECTURA							1								1													
MAS_FACTORES								1									1		1				1					
MAS_TERMINOS									1											1	1							
OTRA_SENTENCIA										1														1				
PROGRAMA	1	1		1			1				1	1		1	1	1												1
SENTENCIA	1			1			1					1		1	1	1												1
TERMINO			1			1							1	1				1									1	1
VARIABLE														1														1
R															1													
W																1												
%																	1											
(																		1										
*																			1									
+																				1								
-																					1							
.																						1						
/																							1					
;																								1				
=																									1			
)																										1		
dígito																											1	
letra_minúscula																												1

Matriz "EMPIEZA CON"

PASO 4: Cálculo del conjunto de símbolos INICIALES de cada no terminal

Se pueden calcular por el método visto en los apartados anteriores (5.4.2.2), o *tomando los símbolos terminales marcados con un 1 en cada fila de la matriz de la relación EMPIEZA CON*.

5.4.4.4.1 Ejemplo de cálculo del conjunto de símbolos INICIALES de cada no terminal

Continuando con el ejemplo de los apartados anteriores se obtiene:

INICIALES(<ASIGNACION>)= { letra_minúscula }
INICIALES(<BLOQUE>) = { R W letra_minúscula }
INICIALES(<CONSTANTE>) = { dígito }
INICIALES(<ESCRITURA>) = { W }
INICIALES(<EXPRESION>) = { (dígito letra-minúscula }
INICIALES(<FACTOR>) = { (dígito letra_minúscula }
INICIALES(<LECTURA>) = { R }
INICIALES(<MAS FACTORES>) = { % * / }
INICIALES(<MAS TERMINOS>) = { + - }
INICIALES(<OTRA SENTENCIA>) = { ; }
INICIALES(<PROGRAMA>) = { R W letra_minúscula }
INICIALES(<SENTENCIA>) = { R W letra_minúscula }
INICIALES(<TERMINO>) = { (dígito letra_minúscula }
INICIALES(<VARIABLE>) = { letra_minúscula }

PASO 5: Cálculo de los conjuntos INICIALES de cada producción

Se calculan los conjuntos de INICIALES de la cadena formada por la parte derecha de cada producción.

5.4.4.5.1 Ejemplo de cálculo de los símbolos INICIALES de cada producción

Continuando con los ejemplos anteriores se obtiene:

INICIALES(<VARIABLE> = <EXPRESION>) = { letra_minúscula }
INICIALES(<SENTENCIA> <OTRA SENTENCIA>) = { R W letra_minúscula }
INICIALES(dígito) = { dígito }
INICIALES(W <VARIABLE>) = { W }
INICIALES(<TERMINO> <MAS TERMINOS>) = { (dígito letra_minúscula }
INICIALES((<EXPRESION>) = { (}
INICIALES(<VARIABLE>) = { letra_minúscula }
INICIALES(<CONSTANTE>) = { dígito }
INICIALES(<R <VARIABLE>) = { R }
INICIALES(* <FACTOR> <MAS FACTORES>) = { * }
INICIALES(/ <FACTOR> <MAS FACTORES>) = { / }
INICIALES(% <FACTOR> <MAS FACTORES>) = { % }
INICIALES(+ <TERMINO> <MAS TERMINOS>) = { + }
INICIALES(- <TERMINO> <MAS TERMINOS>) = { - }
INICIALES(; <SENTENCIA> <OTRA SENTENCIA>) = { ; }
INICIALES(<BLOQUE>) = { R W letra_minúscula }
INICIALES(<ASIGNACION>) = { letra_minúscula }
INICIALES(<LECTURA>) = { R }
INICIALES(<ESCRITURA>) = { W }
INICIALES(<FACTOR> <MAS FACTORES>) = { (dígito letra_minúscula }
INICIALES(letra_minúscula) = { letra_minúscula }
INICIALES(<vacío>) = { }

PASO 6: Construcción de la relación ESTA SEGUIDO DIRECTAMENTE POR

Se define que A *está-seguido-directamente-por* B si y sólo si existe una producción de la forma:

$$D \rightarrow \alpha A \beta B \gamma$$

donde:

$A, B \in V^+$
β es una cadena anulable
α, γ son cadenas cualesquiera

5.4.4.6.1 Ejemplo de cálculo de la relación ESTA SEGUIDO DIRECTAMENTE POR

Entonces continuando el ejemplo de los epígrafes anteriores se tiene que:

<PROGRAMA> ::= <BLOQUE> .

<BLOQUE> *está-seguido-directamente-por* .

<BLOQUE> ::= <SENTENCIA> <OTRA SENTENCIA>

<SENTENCIA> *está-seguido-directamente-por* <OTRA SENTENCIA>

<OTRA SENTENCIA> ::= ; <SENTENCIA> <OTRA SENTENCIA> | <vacío>

; *está-seguido-directamente-por* <SENTENCIA>
<SENTENCIA> *está-seguido-directamente-por* <OTRA_SENTENCIA>

<ASIGNACION> ::= <VARIABLE> = <EXPRESION>

<VARIABLE> *está-seguido-directamente-por* =
= *está-seguido-directamente-por* <EXPRESION>

<EXPRESION> ::= <TERMINO> <MAS TERMINOS>

<TERMINO> *está-seguido-directamente-por* <MAS TERMINOS>

<MAS TERMINOS> ::= + <TERMINO> <MAS TERMINOS> |
- <TERMINO> <MAS TERMINOS> |
<vacío>

+ *está-seguido-directamente-por* <TERMINO>
- *está-seguido-directamente-por* <TERMINO>
<TERMINO> *está-seguido-directamente-por* <MAS TERMINOS>

<TERMINO> ::= <FACTOR> <MAS FACTORES>

FACTOR *está-seguido-directamente-por* <MAS FACTORES>

<MAS FACTORES> ::= * <FACTOR> <MAS FACTORES> |
/ <FACTOR> <MAS FACTORES> |
% <FACTOR> <MAS FACTORES> |
<vacío>

* *está-seguido-directamente-por* <FACTOR>
/ *está-seguido-directamente-por* <FACTOR>
% *está-seguido-directamente-por* <FACTOR>
<FACTOR> *está-seguido-directamente-por* <MAS FACTORES>

<FACTOR> ::= (<EXPRESION>) | <VARIABLE> | <CONSTANTE>

(*está-seguido-directamente-por* <EXPRESION>
<EXPRESION> *está-seguido-directamente-por*)

<LECTURA> ::= R <VARIABLE>

R *está-seguido-directamente-por* <VARIABLE>

<ESCRITURA> ::= W <VARIABLE>

W *está-seguido-directamente-por* <VARIABLE>

Representándolo matricialmente:

	ASIGNACION	BLOQUE	CONSTANTE	ESCRITURA	EXPRESION	FACTOR	LECTURA	MAS_FACTORES	MAS_TERMINOS	OTRA_SENTENCIA	PROGRAMA	SENTENCIA	TERMINO	VARIABLE	R	W	%	(	*	+	-	.	/	;	=	)	dígito	letra_minúscula
ASIGNACION																												
BLOQUE																						1						
CONSTANTE																												
ESCRITURA																												
EXPRESION																										1		
FACTOR								1																				
LECTURA																												
MAS_FACTORES																												
MAS_TERMINOS																												
OTRA_SENTENCIA																												
PROGRAMA																												
SENTENCIA										1																		
TERMINO									1																			
VARIABLE					1																				1			
R														1														
W														1														
%						1																						
(					1																							
*						1																						
+													1															
-													1															
.																												
/						1																						
;												1																
=					1																							
)																												
dígito																												
letra_minúscula																												

Matriz "ESTA SEGUIDO DIRECTAMENTE POR"

PASO 7: Construcción de la relación ES FIN DIRECTO DE

Se define A *ES FIN DIRECTO DE* B, si y sólo si existe una producción de la forma:

$$B \rightarrow \alpha A \beta$$

donde A y B son símbolos terminales o no terminales, β es anulable, y α es una cadena cualquiera.

5.4.4.7.1 Ejemplo de cálculo de la relación ES FIN DIRECTO DE

Continuando con el ejemplo de los epígrafes anteriores se obtiene que:

<PROGRAMA> no es fin directo de ningún símbolo
<BLOQUE> no es fin directo de ningún símbolo
<OTRA SENTENCIA> *es fin directo de* <BLOQUE> y <OTRA SENTENCIA>
<SENTENCIA> *es fin directo de* <BLOQUE> y <OTRA SENTENCIA>
<ASIGNACION> *es fin directo de* <SENTENCIA>
<EXPRESION> *es fin directo de* <ASIGNACION>
<MAS TERMINOS> *es fin directo de* <EXPRESION> y <MAS TERMINOS>
<TERMINO> *es fin directo de* <EXPRESION> y <MAS TERMINOS>
<MAS FACTORES> *es fin directo de* <MAS FACTORES> y <TERMINO>
<FACTOR> *es fin directo de* <MAS FACTORES> y <TERMINO>
<LECTURA> *es fin directo de* <SENTENCIA>
<ESCRITURA> *es fin directo de* <SENTENCIA>
<CONSTANTE> *es fin directo de* <FACTOR>
<VARIABLE> *es fin directo de* <FACTOR>, <LECTURA> y <ESCRITURA>
. *es fin directo de* <PROGRAMA>
) *es fin directo de* <FACTOR>
dígito *es fin directo de* <CONSTANTE>
letra_minúscula *es fin directo de* <VARIABLE>
Los demás símbolos terminales no son fin directo de ningún símbolo.

	ASIGNACION	BLOQUE	CONSTANTE	ESCRITURA	EXPRESION	FACTOR	LECTURA	MAS_FACTORES	MAS_TERMINOS	OTRA_SENTENCIA	PROGRAMA	SENTENCIA	TERMINO	VARIABLE	R	W	%	(	*	+	-	.	/	;	=	)	dígito	letra_minúscula
ASIGNACION												1																
BLOQUE																												
CONSTANTE						1																						
ESCRITURA												1																
EXPRESION	1																											
FACTOR								1					1															
LECTURA												1																
MAS_FACTORES								1					1															
MAS_TERMINOS					1				1																			
OTRA_SENTENCIA		1								1																		
PROGRAMA																												
SENTENCIA		1								1																		
TERMINO					1				1																			
VARIABLE				1		1	1																					
R																												
W																												
%																												
(																												
*																												
+																												
-																												
.											1																	
/																												
;																												
=																												
)						1																						
dígito			1																									
letra_minúscula														1														

Matriz "ES FIN DIRECTO DE"

PASO 8: Construcción de la relación ES FIN DE

Se define la relación A *ES FIN DE* B, si y sólo si existe una cadena que pueda derivarse de B y que termina con A. La relación ES FIN DE *es el cierre reflexivo de la relación ES FIN DIRECTO DE*. Para calcular la matriz que representa esta relación se aplica el **algoritmo de Wharsall** modificado a la matriz que representa la **relación ES FIN DIRECTO DE**.

	ASIGNACION	BLOQUE	CONSTANTE	ESCRITURA	EXPRESION	FACTOR	LECTURA	MAS_FACTORES	MAS_TERMINOS	OTRA_SENTENCIA	PROGRAMA	SENTENCIA	TERMINO	VARIABLE	R	W	%	(	*	+	-	.	/	;	=	)	dígito	letra_minúscula
ASIGNACION	1	1								1		1																
BLOQUE		1																										
CONSTANTE	1	1	1		1	1		1	1	1		1	1															
ESCRITURA		1		1						1		1																
EXPRESION	1	1			1					1		1																
FACTOR	1	1			1	1		1	1	1		1																
LECTURA		1					1			1		1																
MAS_FACTORES	1	1			1			1	1	1		1	1															
MAS_TERMINOS	1	1			1				1	1		1																
OTRA_SENTENCIA		1								1																		
PROGRAMA											1																	
SENTENCIA		1								1		1																
TERMINO	1	1			1				1	1		1	1															
VARIABLE	1	1		1	1	1	1	1	1	1		1	1	1														
R															1													
W																1												
%																	1											
(																		1										
*																			1									
+																				1								
-																					1							
.											1											1						
/																							1					
;																								1				
=																									1			
)	1	1			1	1		1	1	1		1	1													1		
dígito	1	1	1		1	1		1	1	1		1	1														1	
letra_minúscula	1	1		1	1	1	1	1	1	1		1	1	1														1

Matriz "ES FIN DE"

PASO 9: Construcción de la relación ESTA SEGUIDO POR

Se define que A *ESTA SEGUIDO POR* B, si y sólo si existe una cadena que pueda derivarse del símbolo inicial de la gramática, en la que aparece inmediatamente seguida por B. La relación ESTA SEGUIDO POR es el producto de las relaciones ES FIN DE, ESTA SEGUIDO DIRECTAMENTE POR y EMPIEZA CON. La matriz que representa la relación ESTA SEGUIDO POR se obtiene como **producto de las matrices** de las relaciones indicadas anteriormente.

	ASIGNACION	BLOQUE	CONSTANTE	ESCRITURA	EXPRESION	FACTOR	LECTURA	MAS_FACTORES	MAS_TERMINOS	OTRA_SENTENCIA	PROGRAMA	SENTENCIA	TERMINO	VARIABLE	R	W	%	(	*	+	-	.	/	;	=	)	dígito	letra_minúscula
ASIGNACION										1												1		1				
BLOQUE																						1						
CONSTANTE								1	1	1							1		1	1	1	1	1	1		1		
ESCRITURA										1												1		1				
EXPRESION										1												1		1		1		
FACTOR								1	1	1							1		1	1	1	1	1	1		1		
LECTURA										1												1		1				
MAS_FACTORES									1	1										1	1	1		1		1		
MAS_TERMINOS										1												1		1		1		
OTRA_SENTENCIA																						1						
PROGRAMA																												
SENTENCIA										1												1		1				
TERMINO									1	1										1	1	1		1		1		
VARIABLE			1		1	1		1	1	1			1	1			1	1	1	1	1	1	1	1	1	1	1	1
R														1														1
W														1														1
%			1			1								1				1									1	1
(			1		1	1							1	1				1									1	1
*			1			1								1				1									1	1
+			1			1							1	1				1									1	1
-			1			1							1	1				1									1	1
.																												
/			1			1								1				1									1	1
;	1			1			1					1		1	1	1												1
=			1		1	1							1	1				1									1	1
)								1	1	1							1		1	1	1	1	1	1		1		
dígito								1	1	1							1		1	1	1	1	1	1		1		
letra_minúscula			1		1	1		1	1	1			1	1			1	1	1	1	1	1	1	1	1	1	1	1

Matriz "ESTA SEGUIDO POR"

PASO 10: Calcular el conjunto de seguidores de cada no terminal anulable

El conjunto de seguidores de cada no terminal anulable, se obtiene examinando las filas correspondientes a los no terminales anulables (determinados en el paso 1), de la matriz de la relación **ESTA SEGUIDO POR**, obtenida en el paso anterior.

PASO 11: Calcular el conjunto de símbolos directores

El conjunto de símbolos directores se calcula a partir de su definición, por medio de los conjuntos de símbolos Iniciales (obtenido en el paso 5) y Seguidores (obtenido en el paso 10). Por último sólo queda comprobar que los conjuntos de símbolos directores de la expansión de cada no terminal, son conjuntos disjuntos.

El algoritmo de decisión, está implementado en la Universidad de Oviedo en el programa ANALIZADOR DE GRAMATICAS LL(1), disponible en disquete para ordenadores bajo MS-DOS.

5.4.5 Transformación de gramáticas

En el apartado anterior se ha visto la forma de determinar si una gramática es LL(1) o no. Sin embargo, la cuestión de que si un lenguaje posee una gramática LL(1) es *indecidible*. Es decir, no se puede saber se un determinado lenguaje puede ser generado o no por una gramática LL(1), hasta que no se encuentre esta gramática. *No existe ningún algoritmo general que transforme una gramática a LL(1)*. Pero en algunos casos se puede obtener una gramática equivalente por medio de las transformaciones que se estudiarán en los siguientes epígrafes. Ha de tenerse en cuenta que las gramáticas LL(1) son un subconjunto muy particular de las gramáticas libres de contexto, tal como se muestra en el diagrama de la figura 32.

Figura 32: Las gramáticas LL(1) como subconjunto de las gramáticas libres de contexto

5.4.5.1 Eliminación de la recursividad por la izquierda

Una gramática con producciones recursivas a izquierda no cumple las condiciones necesarias para que sea LL(1). *Greibach*, en *1964*, demostró teóricamente que es posible eliminar la recursividad a izquierdas de cualquier gramática. A continuación se muestra el caso más simple, sea una gramática con el par de producciones, presentando la primera una *recursividad a izquierdas inmediata*:

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

donde β no comienza por *A*, se puede eliminar la recursividad a izquierdas con sólo reemplazar este par por:

$$A \rightarrow \beta C$$
$$C \rightarrow \alpha C \mid \lambda$$

Se puede observar que si se desea reconocer la cadena βαα por medio de producciones más a la izquierda se obtienen los árboles sintácticos de la figura 33.

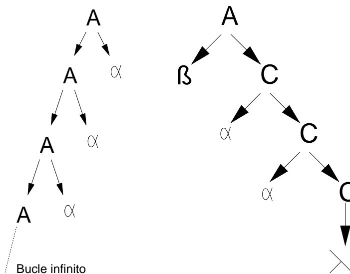


Figura 33: Arboles sintácticos con y sin recursividad a izquierdas

En general, para eliminar la recursividad a izquierdas de todas las producciones de A, lo que se hace es agrupar dichas producciones de la forma siguiente:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \dots \mid \beta_n$$

donde ningún β_i comienza por *A*. Seguidamente, se reemplazan las producciones de *A* por:

$$A \rightarrow \beta_1 C \mid \beta_2 C \mid \dots \mid \beta_n C$$
$$C \rightarrow \alpha_1 C \mid \alpha_2 C \mid \dots \mid \alpha_n C \mid \varepsilon$$

Este proceso elimina todas las *recursividades a izquierda inmediatas* (siempre que ningún α_i sea la cadena vacía), pero no se eliminan las recursividades a izquierda indirectas. Se dice que existe *recursividad indirecta* cuando no se observan recursividades directas inmediatas, pero se llegan a recursividades a izquierda inmediatas por medio de dos o más derivaciones de las reglas. El método para resolver las recursividades indirectas es convertirlas a recursividades inmediatas por medio de sustituciones.

5.4.5.1.1 Ejemplo de recursividad a izquierdas indirecta

Sea la gramática, en la que <EXPRESION> presenta una recursividad a izquierda indirecta:

(1) <EXPRESION> ::= <LISTA>
(2) <LISTA> ::= identificador | <EXPRESION> + identificador

Para eliminarla se realiza la sustitución de la producción (2) en (1), quedando la nueva gramática equivalente con una recursividad a izquierdas inmediata:

<EXPRESION> ::= identificador | <EXPRESION> + identificador

Ahora se puede aplicar el método estudiado anteriormente para eliminar la recursividad a izquierda inmediata, obteniéndose la siguiente gramática no recursiva a izquierdas:

(1) <EXPRESION> ::= identificador <OTRA_EXP>
(2) <OTRA_EXP> ::= + identificador | λ

5.4.5.1.2 Ejemplo de recursividad a izquierdas indirecta

Sea la siguiente gramática que presenta una recursividad a izquierdas indirecta:

<PROGRAMA> ::= <BLOQUE> .
<BLOQUE> ::= <SENTENCIAS> ;
<SENTENCIAS> ::= <PROCEDURE> *return* | *end*
<PROCEDURE> ::= <BLOQUE> *exit*

Esta gramática es recursiva a izquierdas indirectamente tal como se muestra en el árbol sintáctico de la figura 34.

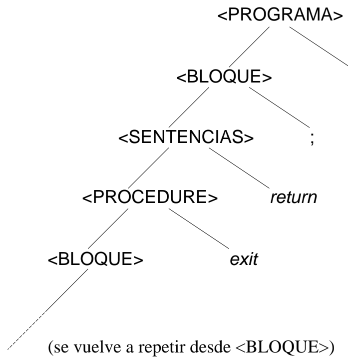


Figura 34: Recursividad a izquierdas indirecta

La producción que hace que sea recursiva a izquierdas es la siguiente:

<PROCEDURE> ::= <BLOQUE> *exit*

Se puede sustituir <BLOQUE> por su parte derecha en la 2ª producción quedando:

<PROCEDURE> ::= <SENTENCIAS> ; *exit*

Pero aún así sigue siendo recursiva a izquierdas, entonces se sustituye <SENTENCIAS> por sus dos partes derechas, con lo que quedan las producciones:

<PROCEDURE> ::= <PROCEDURE> *return* ; *exit*
<PROCEDURE> ::= *end* ; *exit*

Con lo que se ha llegado a una recursividad a *izquierdas inmediata*. Aplicando el método visto anteriormente para eliminar la recursividad a izquierdas inmediata, las producciones quedan:

<PROCEDURE> ::= *end* ; *exit* <C>
<C> ::= *return* ; *exit* <C> | λ

La gramática equivalente a la dada sin recursividad a izquierdas es la siguiente:

<PROGRAMA> ::= <BLOQUE> .
<BLOQUE> ::= <SENTENCIAS> ;
<SENTENCIAS> ::= <PROCEDURE> *return* | *end*
<PROCEDURE> ::= *end* ; *exit* <C>
<C> ::= *return* ; *exit* <C> | λ

5.4.5.2 Factorización y sustitución

Una gramática puede no ser LL(1) y no ser recursiva a izquierdas. El motivo puede ser que los conjuntos de símbolos directores no sean conjuntos disjuntos, por empezar varias reglas de producción por el mismo símbolo no anulable. No existe ningún algoritmo genérico que pueda convertir las gramáticas, tan solo hay recomendaciones. El método de factorización y sustitución lleva implícito el concepto matemático de *sacar factor común*. Es decir, se trata de *agrupar las producciones que comienzan por el mismo símbolo no anulable*, *realizar sustituciones de reglas o incluir nuevos símbolos no terminales*. Dada una producción de la forma:

$$\langle A \rangle ::= \alpha\beta \mid \alpha\gamma$$

evidentemente no es LL(1) si α no es anulable. Se puede transformar de la siguiente forma para que sea LL(1):

$$\langle A \rangle ::= \alpha \langle C \rangle$$
$$\langle C \rangle ::= \beta \mid \gamma$$

El *orden* en que se realizan las sustituciones es importante tenerlo en cuenta. En general, se debe de sustituir primero el no terminal que necesita el mayor número de expansiones, antes de sacar el factor común. *Bauer (1974)* da en su libro [BAUE74, pág. 75] un algoritmo para elegir el orden de sustitución.

5.4.5.2.1 Ejemplo del problema *if-then-else*

La instrucción *if-then-else* podría describirse gramaticalmente de la siguiente forma:

<SENT> ::= *if* <COND> *then* <SENT> *else* <SENT>
| *if* <COND> *then* <SENT>

Presenta el problema que impediría que la gramática fuese LL(1), además tal como se estudió en el apartado 2.5.2, puede dar lugar a ambigüedades. Sin embargo aplicando factorización y sustitución:

<SENT> ::= *if* <COND> *then* <SENT> <RESTO_IF>
<RESTO_IF>::= *else* <SENT>
| λ

Obsérvese que esta sustitución también elimina la ambigüedad, dado que sólo se genera un árbol sintáctico para cada sentencia. Nótese además que se utiliza el mismo símbolo no terminal <SENT> en la parte izquierda de la primera producción y en las partes derechas de las dos producciones, esto es importante dado que en caso contrario puede dar lugar a que la gramática no sea LL(1). En general se aconseja construir las gramática de la siguiente forma para especificar las distintas sentencias de un lenguaje:

<SENT>::= *if* <COND> *then* <SENT> <RESTO_IF>
| *while* <COND> *do* <SENT>
| <VARIABLE> *:=* <EXPRESION>
| *repeat* <SENT> *until* <EXPRESION>
| *for* <VARIABLE> *:=* <EXPRESION> *do* <SENT>
| *goto constante*
| *case* <EXPRESION> *of* <LISTA_CASE> *end*
<RESTO_IF>::= *else* <SENT>
| λ

5.4.5.2.2 Ejemplo de una gramática de expresiones aritméticas

Sea la gramática no ambigua GNA= (VN, VT, S, P), obtenida en el apartado 2.5.3.6, que describe expresiones aritméticas teniendo en cuenta precedencia y asociatividad. Sin embargo *no es LL(1)*. En primer lugar se observa varias recursividades a izquierdas inmediatas.

VN= {<ELEMENTO>, <PRIMARIO>, <FACTOR>, <TERMINO>, <EXP>}
VT= {identificador, constante, (,), ^, *, /, +, -}
S= <EXP>

y las producciones P:

```
<EXP>          ::=  <EXP> + <TERMINO> |
                    <EXP> - <TERMINO> |
                    <TERMINO>
<TERMINO>      ::= <TERMINO> * <FACTOR> |
                    <TERMINO> / <FACTOR> |
                    <FACTOR>
<FACTOR>       ::= <PRIMARIO> ^ <FACTOR> | <PRIMARIO>
<PRIMARIO>     ::= - <PRIMARIO> | <ELEMENTO>
<ELEMENTO>     ::= ( <EXP> ) | identificador | constante
```

Para resaltar la recursividad a izquierdas y manejar reglas más compactas se aplicará factorización y sustitución, utilizando los no terminales <N1>, <N2> y <N3>:

```
<N1>::= + <TERMINO> | - <TERMINO>
<N2>::= * <FACTOR> | / <FACTOR>
<N3>::= ^ <FACTOR> | <vacío>
```

Entonces las reglas de producción anteriores quedan de la forma:

```
<EXP>          ::= <EXP> <N1> | <TERMINO>
<TERMINO>      ::= <TERMINO> <N2> | <FACTOR>
<FACTOR>       ::= <PRIMARIO> <N3>
<PRIMARIO>     ::= - <PRIMARIO> | <ELEMENTO>
<ELEMENTO>     ::= ( <EXP> ) | identificador | constante
```

Ahora pueden verse claramente *dos recursividades a izquierda inmediatas* que aplicando los métodos dados se convierten en:

```
<EXP>          ::= <TERMINO> <MAS_TERMINOS>
<MAS_TERMINOS>::= <N1> <MAS_TERMINOS> | <VACIO>
<TERMINO>      ::= <FACTOR> <MAS_FACTORES>
<MAS_FACTORES>::= <N2> <MAS_FACTORES> | <VACIO>
```

El resto de las producciones quedan como están, y si se vuelve a deshacer la sustitución de <N1>, <N2> y <N3> se obtiene la siguiente gramática de expresiones aritméticas con precedencia y asociatividad, que es LL(1).

```
<EXPRESION> ::= <TERMINO> <MAS_TERMINOS>
<MAS_TERMINOS> ::= + <TERMINO> <MAS_TERMINOS>
<MAS_TERMINOS> ::= - <TERMINO> <MAS_TERMINOS>
<MAS_TERMINOS> ::= <VACIO>
<TERMINO> ::= <FACTOR> <MAS_FACTORES>
<MAS_FACTORES> ::= * <FACTOR> <MAS_FACTORES>
<MAS_FACTORES> ::= / <FACTOR> <MAS_FACTORES>
<MAS_FACTORES> ::= <VACIO>
<FACTOR> ::= <PRIMARIO> <EXP>
<EXP> ::= ^ <FACTOR>
<EXP> ::= <VACIO>
<PRIMARIO> ::= - <PRIMARIO>
<PRIMARIO> ::= <ELEMENTO>
<ELEMENTO> ::= ( <EXPRESION> )
<ELEMENTO> ::= identificador
<ELEMENTO> ::= constante
<VACIO> ::= λ
```

Esta gramática se puede utililizar para construir la gramática del lenguaje MUSIM/11, que se adjunta a continuación:

```
<PROGRAMA> ::= MAIN  {  <BLOQUE>  }
<BLOQUE> ::= <SENTENCIA> <OTRA_SENTENCIA>
<OTRA_SENTENCIA> ::= ; <SENTENCIA> <OTRA_SENTENCIA>
<OTRA_SENTENCIA> ::= <VACIO>
<SENTENCIA> ::= <ASIGNACION>
<SENTENCIA> ::= <LECTURA>
<SENTENCIA> ::= <ESCRITURA>
<ASIGNACION> ::= <VARIABLE> = <EXPRESION>
<EXPRESION> ::= <TERMINO> <MAS_TERMINOS>
<MAS_TERMINOS> ::= + <TERMINO> <MAS_TERMINOS>
<MAS_TERMINOS> ::= - <TERMINO> <MAS_TERMINOS>
<MAS_TERMINOS> ::= <VACIO>
<TERMINO> ::= <FACTOR> <MAS_FACTORES>
<MAS_FACTORES> ::= * <FACTOR> <MAS_FACTORES>
<MAS_FACTORES> ::= / <FACTOR> <MAS_FACTORES>
<MAS_FACTORES> ::= % <FACTOR> <MAS_FACTORES>
<MAS_FACTORES> ::= <VACIO>
<FACTOR> ::= <PRIMARIO> <EXP>
<EXP> ::= ^ <FACTOR>
<EXP> ::= <VACIO>
<PRIMARIO> ::= - <PRIMARIO>
<PRIMARIO> ::= <ELEMENTO>
<ELEMENTO> ::= ( <EXPRESION> )
<ELEMENTO> ::= <VARIABLE>
<ELEMENTO> ::= <CONSTANTE>
<LECTURA> ::= READ <VARIABLE>
<ESCRITURA> ::= WRITE <VARIABLE>
<CONSTANTE> ::= entera
<VARIABLE> ::= letra_minuscula
<VACIO> ::=
```

5.4.5.3 Transformación de gramáticas mediante aspectos semánticos

En algunas construcciones de los lenguajes de programación es necesario conocer más información que la estrictamente sintáctica, para ayudar a elegir el símbolo director en cada momento, en esos casos se puede tomar información semántica, que habitualmente proporcionará la tabla de símbolos o los atributos semánticos de los símbolos de la gramática. En el caso de los procesadores de lenguaje de una sola pasada, esto es fácil de implementar dado que en cada instante se tiene toda la información del símbolo que se está procesando.

5.4.5.3.1 Ejemplo

Sea un lenguaje de programación tipo ALGOL con un fragmento de gramática de la forma:

<BLOQUE>::= *begin* <ETIQ_SENT> *end*
<ETIQ_SENT>::= <ETIQUETA> <SENT>
<ETIQUETA>::= *identificador* :
| λ
<SENT>::= *if* <COND> *then* <SENT> <RESTO_IF>
| *while* <COND> *DO* <SENT>
| <VARIABLE> *:=* <EXPRESION>
| *repeat* <SENT> *until* <EXPRESION>
| *for* <VARIABLE> *:=* <EXPRESION> *do* <SENT>
| *goto constante*
| *case* <EXPRESION> *of* <lista_case> *end*
<RESTO_IF>::= *else* <SENT>
| λ
<VARIABLE>::= *identificador*
...

Puede observarse que, en este lenguaje, las sentencias pueden llevar etiquetas, que son identificadores. Supongamos que se tienen las sentencias (1) y (2).

(1) *yy: x:= 5*
(2) *yy:= 5*

Cuando se analiza una sentencia y se encuentra el identificador *yy* puede haber duda al elegir la alternativa correcta en la producción <ETIQUETA>, si se elige *identificador* se supone que se está en el caso (1). Si se elige vacío se supone que se está en el caso (2), eligiéndose posteriormente <SENT>, <VARIABLE> e *identificador*. Esto produce que el lenguaje no sea LL(1). Para resolver este problema el lenguaje de programación puede exigir la declaración obligatoria de los identificadores de tipo etiqueta, y por medio de una consulta a la tabla de símbolos el analizador puede decidir si el identificador es una etiqueta o una variable de otro tipo, con lo que discrimina entre instrucción con etiqueta (1) y sentencia de asignación (2). Otras soluciones son dar un símbolo terminal diferente para las etiquetas, dado que la declaración obligatoria permitirá fácilmente al analizador léxico reconocerlas, y se evita que la gramática no sea LL(1).

El lenguaje Pascal estándar subsana esto definiendo *etiqueta* como un entero sin signo con declaración obligatoria, sin embargo las extensiones del Turbo Pascal de Borland permiten el uso de identificadores como etiquetas. Los lenguajes C y C++ permiten el uso de identificadores como etiquetas que no se declaran.

5.5 Construcción de analizadores sintácticos descendentes

En este apartado se tratan las distintas técnicas de construcción de analizadores sintácticos descendentes, que se pueden clasificar en cuatro grandes grupos:

a) Métodos basados directamente en la sintaxis
b) Análisis sintáctico *descendente o predictivo no recursivo*, basado en *máquinas de tipo 2 o de pila*.
c) Análisis sintáctico *recursivo descendente*.
d) Análisis sintáctico *descendente o predictivo dirigido por estructura de datos*

5.5.1 Métodos basados directamente en la sintaxis

En este epígrafe se explicarán someramente métodos para transcribir directamente las reglas gramaticales a progrmas que realizan el análisis sintáctico. Son métodos sencillos que lo único que pretenden es ayudar al lector a dar los primeros pasos entre teoría e implementación.

5.5.1.1 Reglas de construcción de diagramas sintácticos

Todo lenguaje de programación puede especificarse de una manera completa y rigurosa mediante un metalenguaje, tal como BNF o diagramas sintácticos. Las reglas para la construcción de diagramas sintácticos a partir de las producciones de una gramática son: secuencial, alternativa y repetitiva.

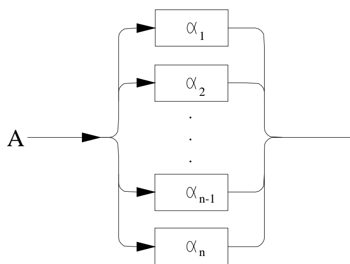


Figura 35: Diagrama sintactico de $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \ldots \mid \alpha_n$

Figura 36: Diagrama sintactico de $B \rightarrow \alpha_1\alpha_2\alpha_3\ldots\alpha_n$

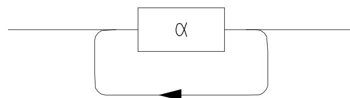


Figura 37: Diagrama sintáctico de la repetición de una cadena.

5.5.1.2 Traducción de reglas sintácticas a programas

Una vez que la gramática se ha descrito mediante un metalenguaje (BNF o diagramas sintácticos), llega el momento de traducir esas reglas sintácticas a programas:

a) Los símbolos no terminales son *procedimientos, funciones o métodos.*
b) Los símbolos terminales son *tokens* enviados por el analizador léxico.
c) Las reglas de producción se traducen en *estructuras de control.*

1. La regla sintáctica $A \rightarrow B_1 B_2 B_3 \ldots B_n$ corresponde a la sentencia compuesta

```
BEGIN
   B1;
   B2;
   B3;
   ...
   Bn
END;
```

2. La regla de producción $A \rightarrow S_1 \mid S_2 \mid S_3 \mid \ldots \mid S_n$ se traduce a

```
CASE token OF
   l1: S1;
   l2: S2;
   ...
   ln: Sn
END;
```

donde $l_i \in$ Símbolos Directores(A, $S_1 \mid S_2 \mid \ldots \mid S_n$)

3. El diagrama sintáctico de la figura 38 se traduce por

```
WHILE l IN L DO S;
```

donde l $\in$ SD(A,{S}) y L= SD(A,{S}).

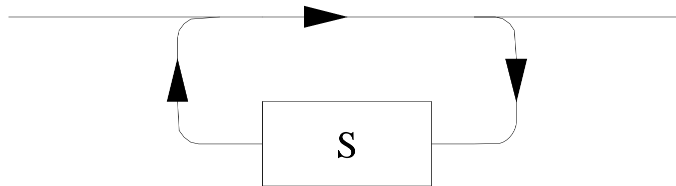


Figura 38: Diagrama sintáctico repetitivo

4.El símbolo no terminal de la figura 39 se traduce como una llamada a un procedimiento, función o método.

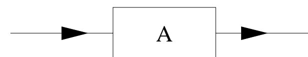


Figura 39: Diagrama sintáctico de un símbolo no terminal

5.El símbolo terminal de la figura 40 se traduce

```
IF token= token7
   THEN
      Lee(token)   (* Pide el siguiente *)
   ELSE
      Error(token7);
```

Figura 40: Diagrama sintáctico de un símbolo terminal

5.5.1.2.1 Ejemplo

Sea la gramática G= (VN, VT, P, S) donde VN= {A, B, C}, VT= {+, x, (,)}, S= A y P son las reglas:

<A> ::= x | (<B>)
<B> ::= <A> <C>
<C> ::= { + <A> }

Determinar:
a) Los diagramas sintácticos.
b) Ejemplos de cadenas del lenguaje
c) Escribir un programa en C que, dada una cadena, indique si es o no del lenguaje.

a) Los diagramas sintácticos se construyen en la figuras 41, 42 y 43.

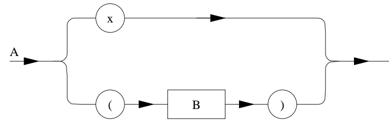


Figura 41: Diagrama sintáctico de <A> ::= x | (<B>)

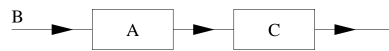


Figura 42: Diagrama sintáctico <B> ::= <A> <C>

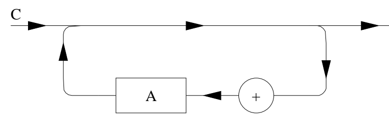


Figura 43: Diagrama sintáctico <C> ::= { + <A> }

En este caso particular, los diagramas sintácticos pueden combinarse para formar un diagrama sintáctico que representa a todo el lenguaje:

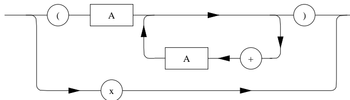


Figura 44: Diagrama sintáctico que representa todo el lenguaje

b) Ejemplos de cadenas pertenecientes al lenguaje: *x, (x), (x+x), (((x))), (x+x+x), (x+x+x+x+x).*

c) En este lenguaje los componentes léxicos son de una letra, por lo tanto la programación de un analizador sintáctico es muy simple.

```
/* Análisis sintáctico */

#include <stdio.h>
char token, cadena[80];
int i=0;

void main(void)
{
 printf("Introduce la cadena a reconocer \n");
 printf("=>");
 scanf("%s",cadena);
 token= cadena[0];
 if (a()) printf("\nCADENA RECONOCIDA");
   else printf("\nCADENA NO RECONOCIDA");
}
/*******************************/
int a(void)
{
 if (token== 'x') { i+=1; token= cadena[i]; return(1); }
```

```
  else if (token== '(') if (b())
                           {
                            if (token== ')') return(1);
                             else return(0);
                           {
                           else return(0);
         else return(0);
{
/*******************************/
int b(void)
{
 i+=1;
 token= cadena[i];
 if (a())
  if (c()) return(1);
  else return(0);
 else return(0);
}
/*******************************/
int c(void)
{
 while (token== '+')
 {
  i+=1;
  token= cadena[i];
  if (!a()) return(0);
 }
 return(1);
}
```

5.5.2 Construcción de analizadores sintácticos basados en autómatas de pila

Los autómatas de tipo 2 reconocen los lenguajes de tipo 2, por lo tanto las sentencias de un lenguaje de programación descrito por una gramática de tipo 2 pueden ser reconocidos por un autómata de pila. Un autómata de pila se puede representar como en la figura 49. Si la gramática es LL(1) el autómata de pila será determinista.

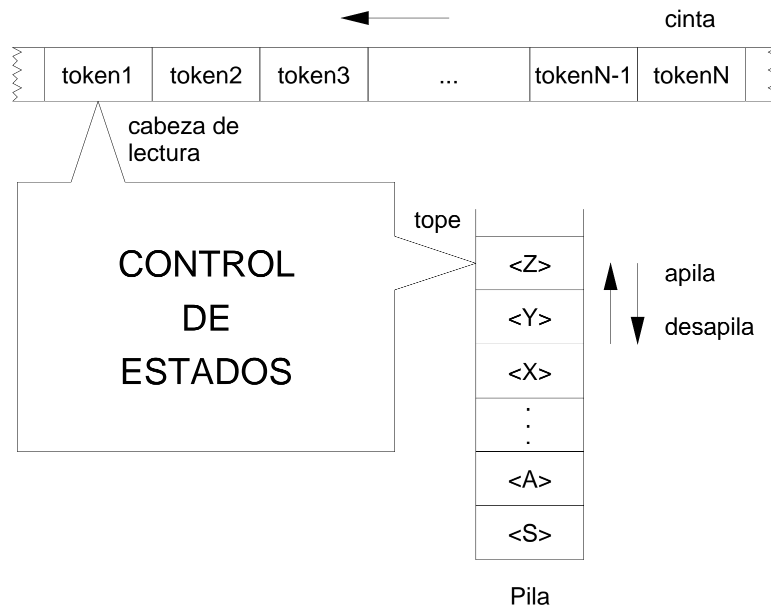


Figura 49: Autómata de pila

5.5.2.1 Algoritmo de reconocimiento

1.- Inicialmente la pila contiene el símbolo inicial de la gramática S.

2.- Si la pila contiene en su tope un símbolo terminal a, entonces el símbolo leido por la cabeza de lectura debe de ser *a*, sino ERROR. Si el tope de la pila y la cabeza de lectura contienen el mismo símbolo, entonces la cabeza de lectura avanza un símbolo, y desapila el símbolo terminal de la pila.

3.- Si en el tope de la pila hay un símbolo no terminal <*A*>, entonces se examina el símbolo de la cabeza de lectura, y consultando los símbolos directores de la gramática LL(1) para elegir la expansión a aplicar al no terminal <*A*>; una vez elegida se desapila <*A*> de la pila, y se sustituye por su expansión en la pila.

4.- La máquina para cuando la pila queda vacía, reconociéndose la sentencia.

Para implementar este algoritmo, el lenguaje no tiene que ser recursivo obligatoriamente.

5.5.3 Analizadores sintácticos recursivo descendentes

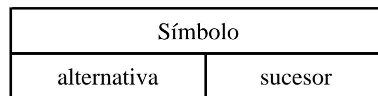
La condición necesaria para que un analizador recursivo descendente opere correctamente es que la gramática del lenguaje fuente sea LL(1). En el apartado 5.5.1.2 se estudió como se traducen las reglas de la gramática a sentencias de programa. La construcción del analizador sintáctico recursivo descendente se realiza desde el símbolo inicial hasta las hojas (sentencia a reconocer), utilizando los esquemas del apartado anterior. El lenguaje de implementación ha de permitir recursividad. El análisis recursivo descendente es la forma más intuitiva y didáctica de análisis sintáctico descendente.

5.5.4 Analizadores sintácticos descendentes dirigidos por estructuras de datos

Fueron propuestos por *Wirth* [WIRT76, capítulo 5] con el objetivo de construir analizadores genéricos.

5.5.4.1 Traducción de reglas sintácticas a estructuras de datos

Se puede suponer que una gramática está formada por un conjunto determinista de grafos sintácticos. Cada nodo se puede representar por



El símbolo es un registro variante:

a) Símbolo terminal

b) Símbolo no terminal, que es un puntero a la estructura de datos que representa el símbolo terminal.

La declaración en lenguaje Pascal:

```
TYPE
     puntero= ↑nodo
     nodo= RECORD
                succ, alt: puntero;
                CASE terminal: boolean OF
                      true: (tablaToken: STRING[12]);
                      false: (tablaNterm: puntero)
                END
```

Las reglas equivalentes a las producciones se muestran en las figuras 45, 46 y 47.

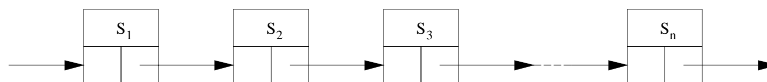


Figura 45: Regla secuencial

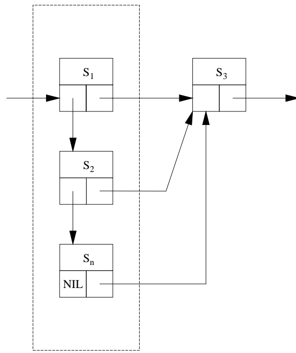


Figura 46: Regla alternativa

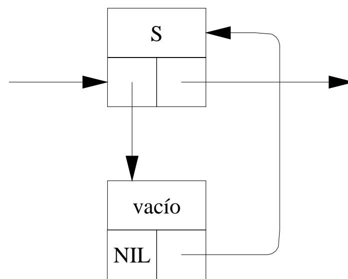


Figura 47: Regla repetitiva

5.5.4.2 Construcción de analizadores sintácticos descendentes dirigidos por estructuras de datos

Aplicando las reglas de traducción anteriores al ejemplo del apartado 5.5.1.2.1, la estructura de datos es la que se muestra en la figura 48.

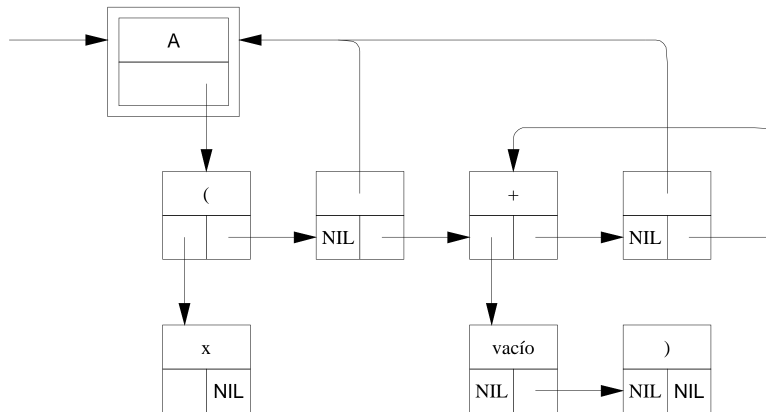


Figura 48

5.5.4.3 Construcción de analizadores sintácticos descendentes genéricos dirigidos por estructuras de datos

Para llevar a cabo esta tarea, es preciso escribir un **traductor** de *reglas sintácticas a estructuras de datos* por lo cual hay que hacer:

a) Definir una nueva notación equivalente a la BNF para introducir los datos al ordenador.

b) Traducir las producciones según las reglas del apartado 5.5.4.1.

c) Construcción del analizador sintáctico.

5.6 Tratamiento de errores sintácticos

Hasta ahora, el analizador sintáctico sólo comprueba si una sentencia pertenece al lenguaje o no. Si la sentencia no pertenece al lenguaje se da un mensaje de error. En un compilador real, *el mensaje de error debe de proporcionar al programador suficiente información para hacerle intuir rápidamente la forma correcta que espera el compilador*. Para poder continuar el análisis a partir de la detección de un error, habrá que estimar:

1º) la **naturaleza del error**

2º) la **intención del programador** cuando cometió el error.

El primer punto determinar la naturaleza del error es complejo, y no siempre es posible, pero el segundo adivinar la intención del programador es todavía más difícil de estimar pues, en un principio, no se conocen los factores que influyen en una persona a la hora de codificar un programa.

Otra consideración a tener en cuenta en el tratamiento de errores es la gran *dependencia* del diseño del tratamiento de errores del *lenguaje específico* que se está procesando, con lo cual no se puede generalizar a todos los lenguajes libres de contexto.

Una **solución rápida** (e ineficiente en muchos lenguajes) sería *saltar el texto fuente hasta encontrar una nueva sentencia*. Esto dependerá de la estructura sintáctica del lenguaje, así se puede hablar de dos grandes tipos de lenguajes desde este punto de vista:

a) Lenguajes estructurados sintácticamente en líneas (por ejemplo las definiciones originales de FORTRAN, COBOL y BASIC). Dado que cada sentencia comienza con una nueva línea, es fácil encontrar la siguiente sentencia.

b) Lenguajes no estructurados sintácticamente en líneas (por ejemplo PASCAL, C, y C++). No es tan fácil encontrar la siguiente sentencia, pues puede haber varias sentencias en la misma línea, y no siempre separadas por ";". Recuérdese que el punto y coma (";") en Pascal es un separador de sentencias, mientras que en C y C++ es un evaluador de expresiones.

La solución anterior depende, en gran medida, de la estructura y sintaxis del lenguaje en cuestión, en resumen, habrá que buscar **reglas** que amplíen esta primera solución.

5.6.1 Regla de la palabra clave

Dice que para poder saltar o ignorar una parte del texto fuente cuando se detecta un error, será conveniente que el lenguaje tenga palabras clave o reservadas, cuyo uso inadecuado sea muy improbable. Así por ejemplo en PASCAL se tienen las palabras clave: *BEGIN, IF, WHILE*,... y lo mismo ocurre con las declaraciones *TYPE, CONST, VAR.*

5.6.2 Regla del antipánico

Esta segunda regla se refiere, en forma más directa, a la construcción del analizador sintáctico. Según se ha estudiado en los analizadores sintácticos descendentes, los objetivos generales se dividen en objetivos parciales, que se traduce en que unos procedimientos llaman a otros para que analicen esos objetivos parciales.

La segunda regla establece que, cuando el analizador detecta un error, no sólo debe pararse e informar del error, sino que *debe seguir examinando el texto hasta llegar a un punto estable*. Ello supone que no se podrá salir del analizador, a no se por su punto de terminación normal.

La aplicación práctica de esta regla consiste en saltar texto a partir de la detección del error, y no parar hasta encontrar *un símbolo que pueda seguir correctamente la estructura que se está analizando*. Ello implica que cada analizador debe conocer el símbolo o conjunto de posibles símbolos que puedan seguir a la estructura sintáctica que se está analizando. Entonces *se han de establecer el símbolo o conjunto de símbolos seguidores a una sentencia o estructura dada*, y *pasar a cada procedimiento analizador un parámetro con los seguidores correspondientes a la estructura que analiza dicho procedimiento.*

Por lo tanto, se suministrará a cada procedimiento analizador un parámetro explícito que especificará los posibles símbolos sucesores. Al final de cada procedimiento se pone una comprobación explícita de que el símbolo que viene a continuación en la secuencia de entrada está, de hecho, dentro del conjunto de posibles sucesores.

Sería, sin embargo, una *decisión de diseño muy pobre*, saltar el texto de entrada en todas las circunstancias, hasta que se encontrara uno de tales símbolos sucesores. Después de todo, el programador puede haber omitido un sólo símbolos (por ejemplo un ";", un *END*, o cualquier otro símbolo seguidor esperando) y saltar texto hasta encontrar un seguidor puede derivar en que se salten varias sentencias (que no se sabrá si son correctas o no).

Es preciso, por tanto, *ampliar los controles de detección* y se hará introduciendo un nuevo tipo de símbolos, los **símbolos de parada**, que serán el conjunto de símbolos compuesto por las *posibles palabras reservadas que expresan explícitamente el comienzo de una estructura*.

El número de procedimientos que analizan sintácticamente sentencias de un lenguaje de programación, son del orden de un centenar o más, y sería engorroso el tener que andar *pasando parámetros* a cada uno de ellos, preguntar por los símbolos contenidos en dichos parámetros, etc... Para realizar esta tarea se puede introducir un *procedimiento adicional* **test**, el cual se va a encargar de hacer este trabajo.

```
PROCEDURE test (S1, S2: tokenSet; numError: integer)
  BEGIN
     IF NOT (nuevoToken IN S1) THEN
       BEGIN
         error(numError);
         S1:= S1+S2;
         WHILE NOT (nuevoToken IN S1) DO scanner;
       END
  END;   (* fin de test *)
```

Cada procedimiento leerá el siguiente token del texto fuente y llamará a **test** pasándole los símbolos seguidores y de parada correspondientes. *Test* comprobará si el token pertenece a dichos conjuntos, y si no pertenece emite el mensaje de error correspondiente, saltando texto hasta que llegue un token que si pertenezca a dichos conjuntos.

El procedimiento anterior tiene **tres parámetros**:

1) **S1**, *conjunto de símbolos sucesores válidos*; si el símbolo en curso no está entre ellos, se tiene un error.

2) **S2**, *conjunto de símbolos de parada adicionales*; cuya presencia es, desde luego, un error, pero que no deben, en ningún caso, ser ignorados o saltados.

3) **numError**, *el número del mensaje de error.*

Se puede conseguir una mayor eficiencia del proceso descrito, incorporando un tercer conjunto de símbolos, los *iniciales de cada sentencia o estructura*, y preguntar por ellos a la ENTRADA de cada procedimiento, para comprobar si el token en curso es válido o no. Para verificar los símbolos iniciales se utiliza el procedimiento **testIniciales**.

```
PROCEDURE testIniciales(iniciales, pararSet: tokenSet; numerror: integer);
  BEGIN
     test(iniciales,pararSet,numError)
  END;
```

Es recomendable hacer esto en *todos los casos que se llama a un procedimiento analizador incondicionalmente*.

Ejemplo:

Llamada al procedimiento *X* en

```
if nuevoToken= a1 then SENT1 ELSE
if
...
if nuevoToken= an then SENTn ELSE X;
```

que es el resultado de traducir la regla sintáctica A::= a_1 $SENT_1$ | a_2 $SENT_2$ | ... | a_n $SENT_n$ | X.

Toda la teoría desarrollada hasta ahora tiene la propiedad de intentar recuperar el analizador, ignorando uno o más símbolos de entrada. Pero esto será insuficiente cuando el error cometido sea por *omisión*, y ocurre un gran porcentaje de errores de este tipo. La experiencia enseña que tales errores se restringen, casi completamente, a símbolos que tienen una función meramente sintáctica, sin representar una acción. Por ejemplo el ";" o el *END*.

Por tanto, es preciso calibrar bien qué símbolos y, sobre todo, qué *palabras clave* incluir en los conjuntos de símbolos seguidores o de parada, para que el analizador deje de ignorar símbolos cuanto antes.

5.6.3 Conclusión

Wirth especifica las características de un buen analizador sintáctico:

1.- Ninguna sentencia debe dar lugar a que el analizador sintáctico pierda el control.

2.- Todos los errores sintácticos deben de ser detectados y señalados.

3.- Los errores muy frecuentes e imputables a verdaderos fallos de comprensión o descuido del programador, habrán de ser diagnosticados correctamente y habrán de evitar tropiezos adicionales del compilador (los llamados mensajes no genuinos o de rebote). Esta tercera característica es la más difícil de lograr, ya que incluso compiladores de gran calidad emiten dos o más mensajes para un determinado error.

6 ANALISIS SINTACTICO ASCENDENTE

Se denominan analizadores sintácticos ascendentes (en inglés *botton-up*) porque pretenden construir un árbol sintáctico para una determinada cadena de entrada, empezando por las hojas y constituyendo el árbol hasta llegar a la raíz.

También se puede considerar este proceso como la *reducción* (en contraste con producción) de una cadena de símbolos al símbolo inicial de la gramática, es decir, una derivación en sentido inverso. En cada paso del proceso, una cadena que coincida con la parte derecha de la producción se reemplaza por el símbolo no terminal de la parte izquierda de dicha producción.

6.1 Analizadores LR

Los analizadores del tipo reducción/desplazamiento (en inglés *shift/reduce*), reconocen lenguajes realizando estas dos operaciones (cargar y reducir). Lo que hacen es leer los tokens de la entrada e ir cargándolos en una pila, de forma que se puedan explorar los *n* tokens superiores que contiene ésta y ver si se corresponden con la parte derecha de alguna de las reglas de la gramática. Si es así se realiza una *reducción*, consistente en sacar de la pila esos *n* tokens y en su lugar colocar el símbolo que aparezca en la parte izquierda de esa regla. En caso contrario, se carga en la pila el siguiente token y una vez hecho esto se vuelve a intentar una reducción.

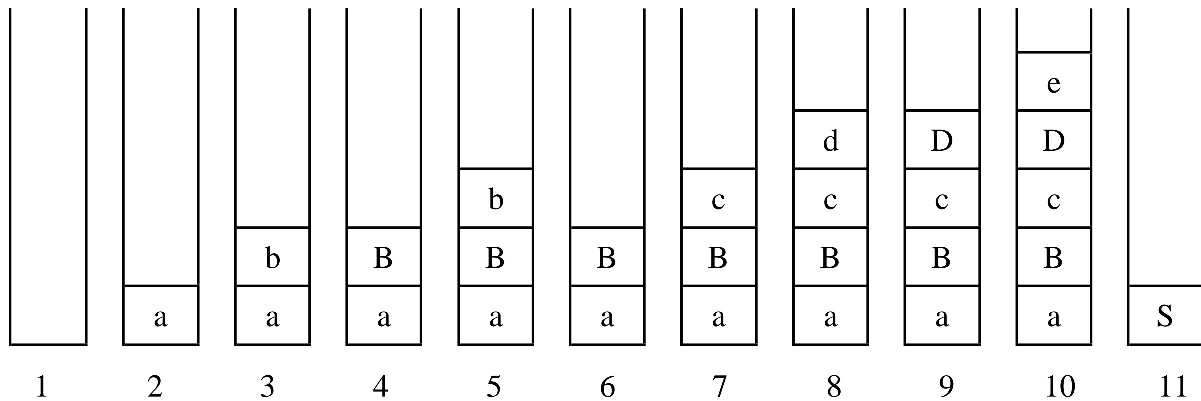
6.1.1 Ejemplo de análisis ascendente con retroceso

Dada la gramática

S → a B c D e [1]
B → B b [2]
B → b [3]
D → d [4]

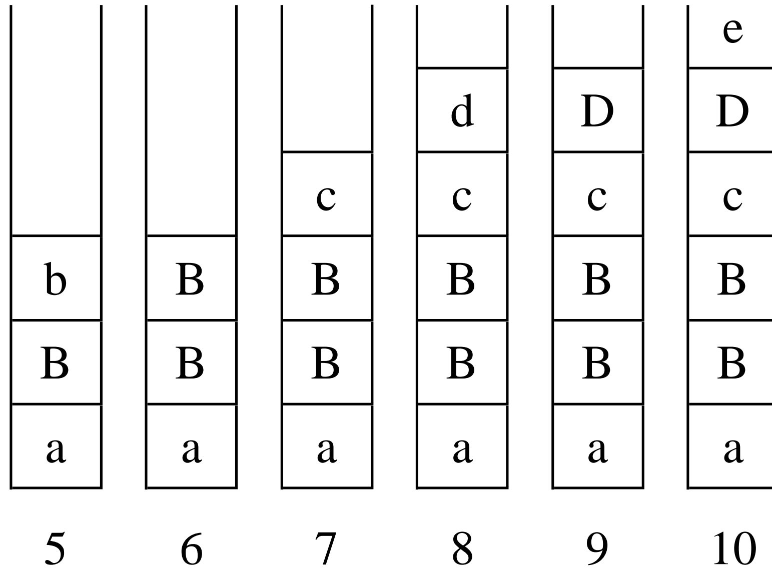
comprobar si la cadena *abbcde* pertenece al lenguaje que describe.

Los contenidos de la pila en las sucesivas etapas del análisis serán:



1 - Inicialmente la pila está vacía.
2 - Se lee el token *a* y se carga en la pila.
3 - Se lee el token *b* y se carga en la pila
4 - Se reduce *b* a *B* utilizando la regla [3] de la gramática.
5 - Se lee el token *b* y se carga en la pila.
6 - Se reduce *Bb* a *B* utilizando la regla [2].
7 - Se carga *c* en la pila.
8 - Se carga *d* en la pila.
9 - Se reduce *d* a *D* utilizando la regla [4].
10 - Se carga *e* en la pila.
11 - Se reduce *aBcDe* a *S* utilizando la regla [1].

Se ha conseguido reducir la cadena de entrada al símbolo inicial *S*, por lo tanto, la sentencia pertenece al lenguaje generado por la gramática dada. A la hora de hacer una reducción, hay que tener cuidado al elegir la regla a utilizar. Si en el paso 6 del ejemplo anterior se hubiera reducido utilizando la regla [3] en lugar de la [2] nos habría quedado en la pila:



y no se podría seguir adelante a partir de aquí porque en la gramática no existe ninguna regla que se ajuste a esa combinación de símbolos.

Para que no se presente este problema, cada vez que vaya a hacer una reducción, el analizador sintáctico ascendente tiene que saber reconocer las reglas de producción que no bloquean el análisis y producen el retroceso (*backtracking*). Las gramáticas LR poseen la propiedad de realizar el análisis ascendente sin retroceso.

6.2 Gramáticas y analizadores LR(k)

Las gramáticas LR(k) realizan eficientemente el análisis ascendente sin retroceso. Se define *gramática LR(k) como aquella que puede ser reconocida por analizador LR(k)*. Los analizadores LR(k) pueden ser generados para la mayoría de las gramáticas libres de contexto. La *L* de su nombre indica que se lee la entrada de izquierda a derecha (*Left-to-right*); la *R*, que el análisis se realiza por derivaciones más a la derecha en sentido inverso (*Rightmost*) y *k* es el número de símbolos de entrada por delante (*lookaheads*) que mira el analizador para tomar decisiones. Si no se especifica *k* se asume que es 1.

Se puede construir un analizador LR para cualquier lenguaje que pueda ser representado por una gramática libre de contexto. Este es el más general de todos los métodos *shift/reduce* que no utilizan backtraking (vuelta a atrás o retroceso) y puede ser implementado tan eficientemente como otros métodos de este tipo. Los analizadores que utilizan la técnica LR pueden analizar un número mayor de gramáticas que los analizadores descendentes y además pueden detectar un error sintáctico tan pronto como es posible hacerlo cuando se evalúa la entrada de izquierda a derecha.

Su principal desventaja es que cuesta demasiado trabajo construirlos. Se necesitan herramientas especializadas, generadores de analizadores LR como *yacc*, que a partir de la descripción de una gramática libre de contexto, produzcan automáticamente el analizador para el lenguaje descrito por esa gramática.

Las tres técnicas más comunes para construir tablas de análisis LR son:

- *SLR (LR simple)*: Es la más fácil de implementar, pero la menos potente de las tres. Puede no ser capaz de producir la tabla para algunas gramáticas que los otros métodos sí pueden tratar.
- *LR canónico*: Es la más potente y la más cara de las tres.
- *LALR (Lookahead LR)*: Tiene un costo y una potencia intermedia entre los dos anteriores.

6.3 Estructura y funcionamiento de un analizador LR

Un analizador LR consta de un *programa analizador LR*, una *tabla de análisis*, y una pila en la que se van cargando los estados por los que pasa el analizador y los símbolos de la gramática que se van leyendo. Se le da una entrada para que la analice y produce a partir de ella una salida.

Lo único que cambia de un analizador a otro es la *tabla de análisis*, que consta de dos partes: una función *accion* que se ocupa de las acciones a realizar (decide qué hacer a continuación) y un función *goto*, que se ocupa de decidir a qué estado se pasa a continuación. El resto es igual para todos los analizadores LR. El programa de análisis lee caracteres del buffer de entrada, de uno en uno. Utiliza una pila para almacenar una cadena de la forma $S_0X_1S_1X_2S_2...X_mS_m$, donde S_m es lo que está en su parte más alta. Cada X_i es un símbolo de la gramática y cada S_i un símbolo que representa un estado. Los símbolos de estado resumen toda la información contenida en la pila por debajo de ellos. La combinación del estado situado en lo más alto de la pila y el símbolo de entrada a leer (aquel en el que está situada la cabeza de lectura) se usan como índices en la tabla de análisis para determinar qué se debe hacer a continuación.

De una forma esquemática, un analizador LR puede representarse en la figura 50.

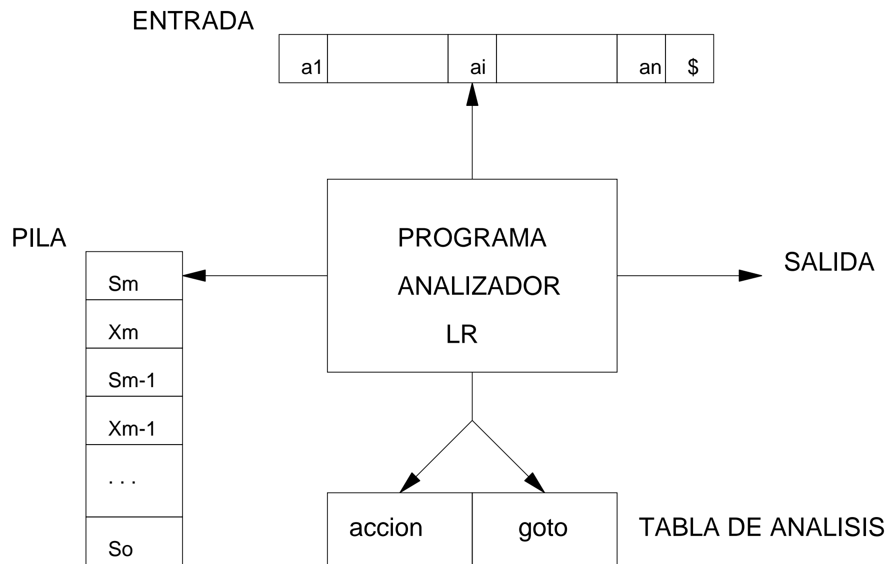


Figura 50: Esquema de un analizador LR

El símbolo *$* señala el final de la cadena de entrada.

El programa de análisis determina qué estado está en la parte superior de la pila y cuál es el símbolo de entrada a leer. Entonces consulta la acción correspondiente de la tabla $accion[S_m, a_i]$, que es la entrada en la parte de acción de la tabla de análisis para el estado S_m y el símbolo de entrada a_i. Puede tener cuatro valores:

1. cargar (shift) *S*, siendo *S* un estado.
2. reducir (reduce) utilizando una regla de producción.
3. aceptar (accept).
4. error.

La función *goto* toma como argumentos un estado y un símbolo de la gramática y produce un estado. Es la función de transición de un autómata finito (AF) que reconoce los *prefijos viables* de la gramática. El estado inicial de este AF es el que está puesto inicialmente en la pila.

Para explicar lo que son los *prefijos viables* de una gramática, es necesario definir antes otros términos:

Una *sentencia o frase* es una secuencia de símbolos terminales a la que se puede llegar partiendo del símbolo inicial y haciendo sucesivas derivaciones aplicando las reglas de la gramática. Una *forma de frase* de una gramática *G* es cualquier combinación de símbolos terminales y no terminales que se pueda derivar en sucesivos pasos a partir del símbolo inicial (es decir, lo mismo que una frase pero conteniendo símbolos terminales y no terminales). Si aplicamos derivaciones más a la derecha será una *forma de frase derecha* y si aplicamos derivaciones más a la izquierda será una *forma de frase izquierda*.

Ahora podemos definir *prefijos viables* de una gramática como los prefijos de las formas de frase derecha que pueden aparecer en la pila de un analizador shift/reduce.

Una *configuración* de un analizador LR es un par cuyo primer componente es el contenido de la pila y el segundo el trozo de entrada que queda sin leer:

$$(S_0X_1S_1X_2S_2...X_mS_m,\ a_ia_{i+1}...a_n\$)$$

El siguiente movimiento del analizador se determina mediante a_i y S_m. Las configuraciones que resultan de cada uno de los cuatro tipos de movimiento son:

1. Si $accion[S_m, a_i] = cargar\ S$, el analizador ejecuta un movimiento de carga, dando como resultado la configuración:

   $(S_0X_1S_1X_2S_2...X_mS_ma_iS,\ a_{i+1}...a_n\$)$

   Se ha cargado tanto el símbolo de entrada a_i como el siguiente estado, S (que es $goto[S_m, a_i]$) en la pila. a_{i+1} pasa a ser el símbolo de entrada a leer. El analizador está ahora en el estado S.

2. Si $accion[S_m, a_i] = reducir\ por\ A \rightarrow b$, entonces el analizador ejecuta una reducción utilizando la regla $A \rightarrow b$, dando la configuración:

   $(S_0X_1S_1X_2S_2...X_{m-r}S_{m-r}AS,\ a_ia_{i+1}...a_n\$)$

   donde S será $goto[S_{m-r}, A]$ y r la longitud de la cadena de símbolos b. El analizador primero saca de la pila $2r$ símbolos, r estados ($S_{m-r+1}...S_m$) y r símbolos de la gramática ($X_{m-r+1}...X_m$), que tendrán que ser los mismos que componen la cadena b (parte derecha de la regla por la que se reduce) y después coloca en la pila el no terminal A (parte izquierda de la regla, que pasará a sustituir a la parte derecha en la pila) y S, que será el estado al que pase el analizador tras reducir. No cambia el símbolo de entrada a leer.

   Una vez que se ha llevado a cabo la reducción se deben ejecutar las *acciones semánticas* correspondientes a la regla por la que se redujo, que son trozos de programa que producen la salida del analizador LR para la entrada que se está leyendo. Cada regla de producción puede llevar una o varias acciones asociadas, que se ejecutarán cada vez que se reduzca utilizándola, es decir, cada vez que en la entrada aparezca una secuencia de símbolos que se ajuste a su parte derecha.

3. Si $accion[S_m, a_i] = aceptar$, el análisis se ha terminado con éxito.

4. Si $accion[S_m, a_i] = error$, se ha descubierto un error. El analizador entonces llamará a la rutinas de recuperación de errores.

6.3.1 Ejemplo

Sea la gramática G:

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

la tabla de análisis LR para ella será:

ESTADO	accion						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

si indica cargar y apilar el estado *i*, *rj* reducir mediante la producción *j* y *acc* aceptar. Los cuadros en blanco corresponden a situaciones erróneas.

En la parte *goto* de la tabla sólo aparecen las transiciones para los símbolos no teminales. Esto es debido a que las transiciones de los terminales se producen inmediatamente después de cargarlos, es decir, cada carga de la parte *accion* de la tabla lleva consigo una acción *goto*, ya que, tras cargar el símbolo, el analizador pasa a otro estado. Por ejemplo, *accion[1, +] = s6* indica que si el analizador está en el estado 1 y aparece en la entrada un signo +, se deberá cargar éste en la pila y a continuación pasar al estado 6, por lo que se podría decir que *goto[1, +] = 6*.

6.4 Algoritmo de análisis LR

Recibe como entrada una cadena *w* y una tabla de análisis LR con las funciones *accion* y *goto* para una gramática determinada *G* y produce como salida un análisis ascendente para la cadena *w* si ésta pertenece al lenguaje generado por *G*, o error si no es así.

Inicialmente, el analizador tiene en su pila S_0, el estado inicial, y *w$* en el buffer de entrada. Entonces ejecuta el programa siguiente hasta llegar a *error* o *accept*.

```
hacer que p apunte al primer símbolo de w$;
repeat forever
 begin
  Sea S el estado situado en la parte superior de la pila y a
  el símbolo al que apunta p;
  if accion[S,a] = cargar S' then
    begin
     Cargar a y luego S' en la pila;
     Hacer que p apunte al siguiente símbolo;
    end
  else if accion[S,a] = reducir por A→β then
    begin
     Sacar 2*|β| símbolos de la pila;
     Sea S' el estado situado en la parte alta de la pila;
     Poner A y luego goto[S',A] en la pila;
     Ejecutar las acciones semánticas de A→β;
    end
  else if accion[S,a] = aceptar then
```

```
        return
    else error()
  end
```

6.5 Gramáticas LR

Una gramática para la que se puede construir una tabla de análisis LR se dice que es una *gramática LR*. Una gramática que puede ser analizada por un analizador LR mirando hasta *k* símbolos de entrada por delante (*lookaheads*) en cada movimiento, se dice que es una *gramática LR(k)*.

Existen otro tipo de gramáticas, las denominadas *gramáticas LL(k)*, que también se utilizan para crear analizadores sintácticos. Se pueden ver algunas diferencias significativas entre éstas y las gramáticas LR(k). En primer lugar, las LL(k) dan lugar a analizadores descendentes, mientras que los que utilizan la técnica LR son ascendentes. Además, para que una gramática sea LR(k), tiene que ser posible reconocer la parte derecha de una regla de producción cuando aparece en la pila, después de haber leído toda esa parte derecha y *k* símbolos más de entrada (*lookaheads*). Esta restricción es mucho menor que la de las gramáticas LL(k), en las que tenemos que ser capaces de reconocer el uso de una regla de producción viendo sólo los primeros *k* símbolos de su parte derecha. Por lo tanto, las gramáticas LR pueden describir un mayor número de lenguajes que las gramáticas LL.

6.6 Construcción de tablas de análisis SLR

Esta es la más débil de las tres técnicas LR que se verán en lo que se refiere al número de gramáticas a las que se puede aplicar, pero también es la más fácil de implementar. Se basa en la construcción de los conjuntos de items LR(0) de la gramática SLR. Una *gramática SLR* es aquella para la que se puede construir un *analizador SLR*. Un *item LR(0)* de una gramática *G* es una de sus producciones, con un punto en algún lugar de su parte derecha. El 0 indica que no utilizamos *lookaheads*.

Por ejemplo, de la producción $A \rightarrow XYZ$ podemos sacar cuatro items:

$A \rightarrow \cdot XYZ$
$A \rightarrow X \cdot YZ$
$A \rightarrow XY \cdot Z$
$A \rightarrow XYZ \cdot$

La producción $A \rightarrow \lambda$ genera un solo item: $A \rightarrow \cdot$. El punto separa la parte de la regla que ya ha sido analizada (antes de él) de la que aún queda por analizar (tras él). Los items se agrupan en *conjuntos de items*, cada uno de los cuales se correponde con un estado del AF que vamos a generar. Para construirlos, a partir de una gramática *G* definimos una gramática aumentada *G'* y dos funciones: *cierre* y *goto*. Cada conjunto de items se genera aplicando la función *cierre* a otro de ellos. La gramática aumentada *G'* se consigue añadiéndole a *G* la producción $S' \rightarrow S$, (siendo *S* el símbolo inicial de *G*) y haciendo que *S'* pase a ser el nuevo símbolo inicial. Se hace esto para poder indicarle al analizador de alguna forma que la entrada ha sido aceptada, lo que ocurrirá solamente cuando el analizador esté a punto de reducir utilizando la regla $S' \rightarrow S$.

6.6.1 Función cierre

Sea I un conjunto de items, *cierre(I)* es el conjunto de items construido a partir de I haciendo:

```
function cierre(I)
begin
  J = I;
  repeat
    for cada item A → α·Bβ de J y cada producción B → γ de
      la gramática / B → ·γ no esté en J do
        Añadir B → ·γ a J;
  until no se puedan añadir más items a J;
  return J
end
```

6.6.1.1 Ejemplo

Sea la gramática aumentada:

$E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

si I es el conjunto de un item $\{[E' \rightarrow \cdot E]\}$, entonces *cierre(I)* será:

(1)	$E' \rightarrow \cdot E$	(5)	$T \rightarrow \cdot F$
(2)	$E \rightarrow \cdot E + T$	(6)	$F \rightarrow \cdot (E)$
(3)	$E \rightarrow \cdot T$	(7)	$F \rightarrow \cdot id$
(4)	$T \rightarrow \cdot T * F$		

$E' \rightarrow \cdot E$ se pone en *cierre(I)* inmediatamente por la inicialización $J = I$. Como hay una E inmediatamente a la derecha del punto, añadimos todas las reglas de producción con parte izquierda E (*E-producciones*) con un punto al principio de la parte derecha de la producción, obteniendo (2) y (3). De la misma forma, como T está en (3) inmediatamente a la derecha del punto, añadiremos todas las *T-producciones* obteniendo (4) y (5), y así sucesivamente.

Hay que notar que si una *X-producción* es añadida a *cierre(I)* con el punto en el extremo izquierdo de su parte derecha, entonces todas las demás *X-producciones* serán añadidas también a él.

6.6.2 Función goto[1]

goto(I, X) es la otra función. I es un conjunto de items y B es un símbolo gramatical. Se define como el *cierre* del conjunto de todos los items $[A \rightarrow aB \cdot b]$ tales que $[A \rightarrow a \cdot Bb]$ esté en I. De una forma informal, se puede decir que si I es el conjunto de items permitidos para un prefijo viable γ, entonces, *goto(I, X)* es el conjunto de items permitidos para el prefijo viable γX.

6.6.2.1 Ejemplo

Dada la gramática vista en el ejemplo anterior, si I es el conjunto de dos items $\{[E' \rightarrow E \cdot], [E \rightarrow E \cdot + T]\}$, entonces *goto(I, +)* es:

$E \rightarrow E + \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$

Computamos *goto(I, +)* buscando en I los items que tengan un + justo a la derecha del punto. Esto no sucede en $E' \rightarrow E \cdot$, pero sí en $E \rightarrow E \cdot + T$, por lo tanto movemos el punto al otro lado del signo y hacemos el cierre de lo que obtenemos: $[E \rightarrow E + \cdot T]$

6.6.3 Construcción de conjuntos de items

Ahora ya podemos ver el algoritmo que construye C, la colección canónica de conjuntos de items LR(0) para una gramática aumentada G'. Es:

```
procedure items(G');
begin
  C = {cierre({[S'→ ·S]})}
  repeat
    for cada conjunto de items I de C y cada símbolo X /
      goto(I, X) no esté vacío y no pertenezca a C do
```

1 - Esta función *goto()* no es la misma que la *goto[]* de la tabla de análisis. Esta última se generará posteriormente a partir de los datos calculados por ella.

Añadir *goto(I, X)* a *C*;

until no se puedan añadir más conjuntos de items a *C*

end

Podemos aplicar este algoritmo a la gramática aumentada que teníamos:

$E' \to E$
$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid id$

y obtendremos:

I_0:
$E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

I_1:
$E' \to E \cdot$
$E \to E \cdot + T$

I_2:
$E \to T \cdot$
$T \to T \cdot * F$

I_3:
$T \to F \cdot$

I_4:
$F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

I_5:
$F \to id \cdot$

I_6:
$E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

I_7:
$T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

I_8:
$F \to (E \cdot)$
$E \to E \cdot + T$

I_9:
$E \to E + T \cdot$
$T \to T \cdot * F$

I_{10}:
$T \to T * F \cdot$

I_{11}:
$F \to (E) \cdot$

La función *goto* se puede expresar como un diagrama de transiciones de un AF. Si cada uno de sus estados es final e I_0 es el estado inicial, entonces éste reconoce exactamente los prefijos viables de la gramática.

6.6.4 Tablas de análisis SLR

El siguiente algoritmo construye una tabla de análisis SLR con las funciones *accion* y *goto* a partir del AF que reconoce los prefijos válidos. Este algoritmo no sive para crear la tabla de análisis de cualquier gramática, pero sí para muchas.

Dada una gramática *G*, construimos la gramática aumentada *G'* y a partir de ella la colección de conjuntos de items, *C*. Necesitamos conocer *SIGUIENTES(A)* para cada no terminal *A* de la gramática. Construimos las funciones *acción* y *goto* a partir de *C* haciendo:

1. Construir $C = \{I_0, I_1, ..., I_n\}$, la colección de conjuntos de items LR(0) para *G'*.
2. El estado *i* se construye a partir de I_i. Las acciones para el estado *i* se determinan:

    a) Si $[A \to a{\cdot}ab]$ está en I_i y $goto(I_i, a) = I_j$, entonces *accion[i, a] = cargar j*. *a* debe ser un terminal.

    b) Si $[A \to a\cdot]$ está en I_i, entonces *accion[i, a] = reducir por* $A \to a$ para cualquier *a* que pertenezca a *SIGUIENTES(A)*; siendo *A* distinto de *S'*.

    c) Si $[S' \to S\cdot]$ está en *I*, entonces *accion[i, $] = aceptar*.

Si se genera alguna acción conflictiva al aplicar las reglas anteriores, decimos que la gramática no es SLR(1). El algoritmo no puede producir un analizador en este caso.

3. Las transiciones *goto* para el estado *i* se construyen usando la regla:

   if $goto(I_i, A) = I_j$, **then** $goto[i, A] = j$

   para todos los no terminales *A*.
4. Todas las entradas no definidas por (2) y (3) serán *error*.
5. El estado inicial del analizador es el que se construye a partir del conjunto de items que contiene $[S' \rightarrow \cdot S]$.

A la tabla determinada utilizando este algoritmo se la llama *tabla SLR(1) para G*. Un analizador LR que utilice la tabla SLR(1) de G, se llama *analizador SLR(1) para G* y una gramática que tenga una tabla de análisis SLR(1) se dice que es una *gramática SLR(1)*[2]. Cualquier gramática SLR(1) es no ambigua.

Si queremos construir la tabla SLR para la gramática *G* vista , tenemos que considerar primero los conjuntos de items LR(0).

Si consideramos I_0, el item $F \rightarrow \cdot (E)$ hace que *accion[0,(]= cargar 4* y el item $F \rightarrow \cdot id$ hace que *accion[0, id] = cargar 5*. Los otros items de I_0 no dan lugar a ninguna acción.

En I_1, el primer item hace que *accion[1, $] = aceptar* y el segundo que *accion[1, +] = cargar 6*.

Considerando I_2, como *SIGUIENTES(E) = { $, +,) }*, el primer item hace que *accion[2, $] = accion[2, +] = accion[2,)] = reducir por* $E \rightarrow T$. El segundo item hace que *accion[2, *] = cargar 7*. Siguiendo de esta forma obtendremos las tablas correspondientes a *accion* y *goto*.

La tabla de análisis obtenida será la que se presentó como ejemplo en el apartado 6.3.1.

6.7 Construcción de tablas LR canónicas

Esta es la técnica más general para construir una tabla de análisis LR para una gramática.

Es posible conseguir que los estados lleven más información que en la técnica SLR, de forma que podamos regular algunas de las reducciones no válidas permitidas por esta técnica. Dividiendo los estados cuando sea necesario, podemos conseguir que cada estado de un analizador LR indique exactamente qué símbolos de entrada pueden seguir a a para los cuales hay una reducción posible a *A* utilizando $A \rightarrow \alpha$.

La información adicional se incorpora a los estados redefiniendo los items, de forma que incluyan un símbolo terminal como segundo componente. La forma general de un item pasa a ser $[A \rightarrow a \cdot b, a]$, donde $A \rightarrow ab$ es una producción y *a* es un terminal o $ (que marca el final de la cadena). Esto es lo que llamamos item LR(1). El *1* se refiere a la longitud del segundo componente del item (donde aparecen los *lookaheads*, en este caso sólo hay uno). El *lookahead* no hace nada en un item de la forma $[A \rightarrow a \cdot b, a]$, donde *b* no es la cadena vacía, pero un item de la forma $[A \rightarrow a \cdot, a]$ hace que se reduzca mediante $A \rightarrow a$ sólo si el siguiente símbolo de entrada es *a*. De esta forma, sólo estamos obligados a reducir para cada regla $A \rightarrow a$ con aquellos símbolos de entrada para los cuales $[A \rightarrow a \cdot, a]$ es un item LR(1) del estado que se encuentra en la parte superior de la pila del analizador. El conjunto de tales terminales *a* será siempre un subconjunto de *SIGUIENTES(A)*, pero podría ser un subconjunto propio.

El método para construir los conjuntos de items LR(1) es esencialmente el mismo utilizado para construir los LR(0). Sólo hay que modificar los dos procedimientos *cierre* y *goto*.

```
function cierre(I)
begin
  repeat
    for cada item [A→α·Bβ, a] de I, cada producción B→γ de G'
    y cada terminal b de INICIALES(βa) / [B→·γ, b] ∉ I do
        Añadir [B → ·γ, b] a I;
```

2 - Normalmente se omitirá el (1), ya que no vamos a tratar ningún analizador que utilice más de un *lookahead*.

until no se puedan añadir más items a *I*;
return *I*
end;

function goto(I,X);
begin
sea *J* el conjunto de items $[A \rightarrow \alpha B \cdot \beta, a]$ /
$[A \rightarrow \alpha \cdot B\beta, a]$ está en *I*;
return cierre(*J*)
end;

procedure items(G');
begin
C = {cierre({[*S'*→ ·*S*]})}
repeat
for cada conjunto de items de *C* y cada símbolo gramatical
X / *goto(I,X)* no esté vacío y no pertenezca a *C* **do**
Añadir *goto(I,X)* a *C*;
until no se puedan añadir más conjuntos de items a *C*
end

El método para construir la tabla de análisis será:

1. Construir $C = \{I_0, I_1, ..., I_n\}$, la colección de conjuntos de items LR(1) para *G'*.
2. El estado *i* se construye a partir de I_i. Las acciones para el estado *i* se determinan:
   a) Si $[A \rightarrow a \cdot ab, b]$ está en I_i y $goto(I_i, a) = I_j$, entonces *accion[i, a] = cargar j*. *a* debe ser un terminal.
   b) Si $[A \rightarrow a \cdot, a]$ está en I_i, entonces *accion[i, a] = reducir por A → a*, siendo *A* distinto de *S'*.
   c) Si $[S' \rightarrow S \cdot]$ está en *I*, entonces *accion[i, $] = aceptar*.

   Si se produce algun conflicto al aplicar las reglas anteriores, decimos que la gramática no es LR(1). El algoritmo no puede producir un analizador en este caso.
3. Las transiciones *goto* para el estado *i* se construyen usando la regla:
   if $goto(I_i, A) = I_j$, **then** $goto[i, A] = j$
   para todos los no terminales *A*.
4. Todas las entradas no definidas por (2) y (3) serán *error*.
5. El estado inicial del analizador es el que se construye a partir del conjunto de items que contiene [*S'*→ ·*S, $*]

Toda gramática SLR(1) es LR(1), pero para una gramática SLR(1) el analizador LR canónico puede tener más estados que el analizador SLR de esa misma gramática.

6.7.1 Ejemplo

Siguiendo con la gramática vista anteriormente

E'→ *E*
E → *E* + *T* / *T*
T → *T* * *F* / *F*
F → (*E*) / *id*

Los conjuntos de items LR(1) serán:

I_0:	*E'*→ · *E*,	*$*
	E → · *E* + *T*,	+ / $
	E → · *T*,	+ / $
	T → · *T* * *F*,	+ / * / $
	T → · *F*,	+ / * / $
	F → · (*E*),	+ / * / $
	F → · *id*,	+ / * / $
	E → · *T*,	+ /)
	T → · *T* * *F*,	+ / * /)
	T → · *F*,	+ / * /)
	F → · (*E*),	+ / * /)
	F → · *id*,	+ / * /)
I_5:	*F* → *id* ·,	+ / * / $

Estado	Ítem	Símbolos
I_1:	$E' \to E\,\cdot,$	$\$$
	$E \to E\,\cdot + T,$	$+ / \$$
I_2:	$E \to T\,\cdot,$	$+ / \$$
	$T \to T\,\cdot * F,$	$+ / * / \$$
I_3:	$T \to F\,\cdot,$	$+ / * / \$$
I_4:	$F \to (\,\cdot E\,),$	$+ / * / \$$
	$E \to \cdot\, E + T,$	$+ /)$
I_8:	$F \to (\,E\,\cdot\,),$	$+ / * / \$$
	$E \to E\,\cdot + T,$	$+ /)$
I_9:	$E \to T\,\cdot,$	$+ /)$
	$T \to T\,\cdot * F,$	$+ / * /)$
I_{10}:	$T \to F\,\cdot,$	$+ / * /)$
I_{11}:	$F \to (\,\cdot E\,),$	$+ / * /)$
	$E \to \cdot\, E + T,$	$+ /)$
	$E \to \cdot\, T,$	$+ /)$
	$T \to \cdot\, T * F,$	$+ / * /)$
	$T \to \cdot\, F,$	$+ / * /)$
	$F \to \cdot\,(\,E\,),$	$+ / * /)$
	$F \to \cdot\, id,$	$+ / * /)$
I_{12}:	$F \to id\,\cdot,$	$+ / * /)$
I_{13}:	$E \to E + T\,\cdot,$	$+ / \$$
	$T \to T\,\cdot * F,$	$+ / * / \$$
I_{14}:	$T \to T * F\,\cdot,$	$+ / * / \$$
I_6:	$E \to E + \cdot\, T,$	$+ / \$$
	$T \to \cdot\, T * F,$	$+ / * / \$$
	$T \to \cdot\, F,$	$+ / * / \$$
	$F \to \cdot\,(\,E\,),$	$+ / * / \$$
	$F \to \cdot\, id,$	$+ / * / \$$
I_7:	$T \to T * \cdot\, F,$	$+ / * / \$$
	$F \to \cdot\,(\,E\,),$	$+ / * / \$$
	$F \to \cdot\, id,$	$+ / * / \$$
I_{15}:	$F \to (\,E\,)\,\cdot,$	$+ / * / \$$
I_{16}:	$E \to E + \cdot\, T,$	$+ /)$
	$T \to \cdot\, T * F,$	$+ / * /)$
	$T \to \cdot\, F,$	$+ / * /)$
	$F \to \cdot\,(\,E\,),$	$+ / * /)$
	$F \to \cdot\, id,$	$+ / * /)$
I_{17}:	$T \to T * \cdot\, F,$	$+ / * /)$
	$F \to \cdot\,(\,E\,),$	$+ / * /)$
	$F \to \cdot\, id,$	$+ / * /)$
I_{18}:	$F \to (\,E\,\cdot\,),$	$+ / * /)$
	$E \to E\,\cdot + T,$	$+ /)$
I_{19}:	$E \to E + T\,\cdot,$	$+ /)$
	$T \to T\,\cdot * F,$	$+ / * /)$
I_{20}:	$T \to T * F\,\cdot,$	$+ / * /)$
I_{21}:	$F \to (\,E\,)\,\cdot,$	$+ / * /)$

La tabla de análisis LR(1) resultante es la que se muestra en la figura 51.

6.8 Construcción de tablas LALR

Este método se usa mucho en la práctica porque las tablas que se obtienen a través de él son considerablemente más pequeñas que las LR canónicas, y pueden expresar convenientemente las construcciones sintácticas más comunes de los lenguajes de programación. Se diferencian de las SLR en que éstas no pueden manejar convenientemente algunas construcciones.

Respecto al tamaño del analizador, las tablas SLR y LALR para una gramática tienen el mismo número de estados, muy inferior al número de los que tendría una tabla LR canónica para ese mismo lenguaje. Por eso es mucho más fácil y económico construir tablas SLR y LALR que tablas LR canónicas.

ESTADO	accion						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7			r2			
3		r4	r4			r4			
4	s12			s11			8	9	10
5		r6	r6			r6			
6	s5			s4				13	3
7	s5			s4					14
8		s16			s15				
9		r2	s17		r2				
10		r4	r4		r4				
11	s12			s11			18	9	10
12		r6	r6		r6				
13		r1	s7			r1			
14		r3	r3			r3			
15		r5	r5			r5			
16	s12			s11				19	10
17	s12			s11					20
18		s19			s21				
19		r1	s17		r1				
20		r3	r3		r3				
21		r5	r5		r5				

Figura 51

Observando los conjuntos de items LR(1), vemos que algunos de ellos tienen idéntica su primera parte y sólo se diferencian en la segunda, es decir, en sus *lookaheads*. Esto ocurre, por ejemplo, con los estados I_4 e I_{11} del ejemplo anterior:

I_4: $F \rightarrow (\cdot E),\quad + / * / \$$
$E \rightarrow \cdot E + T,\quad + /)$
$E \rightarrow \cdot T,\quad + /)$
$T \rightarrow \cdot T * F,\quad + / * /)$
$T \rightarrow \cdot F,\quad + / * /)$
$F \rightarrow \cdot (E),\quad + / * /)$
$F \rightarrow \cdot id,\quad + / * /)$

I_{11}: $F \rightarrow (\cdot E),\quad + / * /)$
$E \rightarrow \cdot E + T,\quad + /)$
$E \rightarrow \cdot T,\quad + /)$
$T \rightarrow \cdot T * F,\quad + / * /)$
$T \rightarrow \cdot F,\quad + / * /)$
$F \rightarrow \cdot (E),\quad + / * /)$
$F \rightarrow \cdot id,\quad + / * /)$

Vamos a llamar *núcleo* de un conjunto de items a las primeras partes de éstos (o lo que es lo mismo, a lo que serían los items LR(0) completos). Por ejemplo, el núcleo de I_4 sería:

$F \rightarrow (\cdot E)$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$

Podemos buscar los conjuntos de items LR(1) que tengan el mismo núcleo y unirlos en uno solo. La acción de este estado será reducir con cualquiera de los *lookaheads* de los conjuntos de items que han sido unidos. De esta forma obtendremos un analizador distinto que se comportará esencialmente como el LR canónico, aunque podría reducir en circunstancias en las que el otro habría dado error. El error será visto con el tiempo; será detectado antes de que se cargue ningún símbolo de entrada más, por lo tanto, este nuevo analizador es prácticamente tan eficiente como el LR canónico, pero tiene menos estados, es decir, ocupa menos espacio.

Como el núcleo de *goto(I, X)* depende solamente del núcleo de *I*, las acciones *goto* de conjuntos unidos también pueden ser unidas. La función *accion* será modificada, de forma que refleje todas las acciones que no serían error para alguno de los conjuntos de items de la unión.

6.8.1 Algoritmo para construir tablas LALR

1. Construir los conjuntos de items LR(1), $C = \{I_0, I_1, ..., I_n\}$
2. Realizar la unión de los conjuntos de items LR(1) que tengan el mismo núcleo.
3. Sea $C' = \{J_0, J_1, ..., J_m\}$ la nueva colección de conjuntos de items. Las acciones se construyen a partir de J_i de la misma forma que se hacía para construir la tabla LR(1). Si se produce algún conflicto, el algoritmo no puede producir un analizador. Se dice entonces que la gramática no es LALR(1).
4. La tabla *goto* se construye: si *J* es la unión de uno o más conjuntos de items, es decir, $J = I_1$ U I_2 U ... U I_k, entonces los núcleos de $goto(I_1, X)$, $goto(I_2, X)$, ..., $goto(I_k, X)$ son los mismos ya que I_1, I_2, ..., I_k tienen el mismo núcleo. Sea K la unión de todos los conjuntos de items con el mismo núcleo que $goto(I_1, X)$. Entonces $goto(J, X) = K$.

La tabla de análisis producida mediante este método se llama *tabla de análisis LALR* para G. Si no hay conflictos, se dice entonces que la gramática dada es una *gramática LALR(1)*.

Este algoritmo es fácil de implementar, pero ocupa mucho espacio.

Ejemplo:

Siguiendo con el ejemplo anterior, podemos agrupar los conjuntos de items LR(1) como sigue:

I_2 e $I_9 \rightarrow I_{29}$	I_7 e $I_{17} \rightarrow I_{717}$
I_3 e $I_{10} \rightarrow I_{310}$	I_8 e $I_{18} \rightarrow I_{818}$
I_4 e $I_{11} \rightarrow I_{411}$	I_{13} e $I_{19} \rightarrow I_{1319}$
I_5 e $I_{12} \rightarrow I_{512}$	I_{14} e $I_{20} \rightarrow I_{1420}$
I_6 e $I_{16} \rightarrow I_{616}$	I_{15} e $I_{21} \rightarrow I_{1521}$

Uniendo los conjuntos de items con el mismo núcleo nos quedará:

I_0:	$E' \rightarrow \cdot E,$	$\$$	I_{310}:	$T \rightarrow F \cdot,$	$+ / * /) / \$$
	$E \rightarrow \cdot E + T,$	$+ / \$$			
	$E \rightarrow \cdot T,$	$+ / \$$			
	$T \rightarrow \cdot T * F,$	$+ / * / \$$	I_{411}:	$F \rightarrow (\cdot E),$	$+ / * /) / \$$
	$T \rightarrow \cdot F,$	$+ / * / \$$		$E \rightarrow \cdot E + T,$	$+ /)$
	$F \rightarrow \cdot (E),$	$+ / * / \$$		$E \rightarrow \cdot T,$	$+ /)$
	$F \rightarrow \cdot id,$	$+ / * / \$$		$T \rightarrow \cdot T * F,$	$+ / * /)$
				$T \rightarrow \cdot F,$	$+ / * /)$
I_1:	$E' \rightarrow E \cdot,$	$\$$		$F \rightarrow \cdot (E),$	$+ / * /)$
	$E \rightarrow E \cdot + T,$	$+ / \$$		$F \rightarrow \cdot id,$	$+ / * /)$
I_{29}:	$E \rightarrow T \cdot,$	$+ /) / \$$			
	$T \rightarrow T \cdot * F,$	$+ / * /) / \$$	I_{512}:	$F \rightarrow id \cdot,$	$+ / * /) / \$$
I_{616}:	$E \rightarrow E + \cdot T,$	$+ /) / \$$			
	$T \rightarrow \cdot T * F,$	$+ / * /) / \$$			
	$T \rightarrow \cdot F,$	$+ / * /) / \$$			
	$F \rightarrow \cdot (E),$	$+ / * /) / \$$			
	$F \rightarrow \cdot id,$	$+ / * /) / \$$			

I_{717}: $T \to T * \cdot F,\quad + / * /) / \$$
$F \to \cdot (E),\quad + / * /) / \$$
$F \to \cdot id,\quad + / * /) / \$$

I_{818}: $F \to (E \cdot),\quad + / * /) / \$$
$E \to E \cdot + T,\quad + /)$

I_{1319}: $E \to E + T \cdot,\quad + /) / \$$
$T \to T \cdot * F,\quad + / * /) / \$$

I_{1420}: $T \to T * F \cdot,\quad + / * /) / \$$

I_{1521}: $F \to (E) \cdot,\quad + / * /) / \$$

La tabla LALR será:

	accion						goto		
ESTADO	id	+	*	(	)	$	E	T	F
0	s512			s411			1	2	3
1		s616				acc			
29		r2	s717		r2	r2			
310		r4	r4		r4	r4			
411	s512			s411			8	2	3
512		r6	r6		r6	r6			
616	s512			s411				9	3
717	s512			s411					10
818		s616			s1521				
1319		r1	s717		r1	r1			
1420		r3	r3		r3	r3			
1521		r5	r5		r5	r5			

Se pueden hacer varias modificaciones en el algoritmo visto para evitar el tener que construir todos los conjuntos de items LR(1).

Podemos llamar *kernel* de un conjunto de items a un subconjunto de éste formado a partir de él, eliminando todos los items que lleven un punto en el extremo izquierdo de su parte derecha (solamente se pueden suprimir los items que no forman parte de los kernels, una vez que se han generado todos los conjuntos de items, si no, no llegarían a producirse todos).

6.8.1.1 Ejemplo

Los kernels de los conjuntos de items de la gramática del ejemplo serán:

I_0: $E' \to \cdot E$

I_1: $E' \to E \cdot$
$E \to E \cdot + T$

I_2: $E \to T \cdot$
$T \to T \cdot * F$

I_3: $T \to F \cdot$

I_4: $F \to (\cdot E)$

I_5: $F \to id \cdot$

I_6: $E \to E + \cdot T$

I_7: $T \to T * \cdot F$

I_8: $F \to (E \cdot)$
$E \to E \cdot + T$

I_9: $E \to E + T \cdot$
$T \to T \cdot * F$

I_{10}: $T \to T * F \cdot$

I_{11}: $F \to (E) \cdot$

Es posible calcular la tabla de análisis utilizando solamente los kernels de los conjuntos de items. El procedimiento a seguir es:

Una vez que tenemos los kernels, tenemos que añadirles los *lookaheads* que les corresponden. Estos pueden generarse espontáneamente o ser propagados de unos items a otros. Primero habrá que buscar aquellos que se generan espontáneamente y luego dejar que se propaguen lo más posible. El algoritmo que aparece a continuación se ocupa de ello.

for cada item $B \rightarrow \gamma \cdot \delta$ de K **do**
begin
J' = cierre($\{[B \rightarrow \gamma \cdot \delta, \#]\}$);
if $[A \rightarrow \alpha \cdot X\beta, a] \in J'$, $a <> \#$ **then**
El *lookahead* a se genera espontáneamente para el item
$A \rightarrow \alpha X \cdot \beta$ de $goto(I, X)$;
if $[A \rightarrow \alpha \cdot X\beta, \#] \in J'$ **then**
Los *lookaheads* se propagan de $B \rightarrow \gamma \cdot \delta$ en I
a $A \rightarrow \alpha X \cdot \beta$ en $goto(I, X)$;
end

K es el kernel de un conjunto de items LR(0) I. El símbolo # se utiliza para detectar las situaciones en las que los *lookaheads* se propagan.

Habrá que aplicar este algoritmo a los kernels de todos los conjuntos de items para cada símbolo de la gramática X.

Si cogemos el kernel de I_0 del ejemplo, como $I_0 = \{[E' \rightarrow \cdot E]\}$, lo primero que tenemos que hallar es $cierre(\{[E' \rightarrow \cdot E, \#]\})$. Partimos de $[E' \rightarrow \cdot E, \#]$ y le aplicamos la función *cierre* del algoritmo que construye los conjuntos de items LR(1), vista anteriormente. Como $INICIALES(\#) = \#$ y el punto está a la izquierda de E, añadimos todas las producciones de la gramática que tienen el símbolo E en su parte izquierda, con el punto en el extremo izquierdo de su parte derecha y con el *lookahead #*. Es decir, ya tenemos:

$E' \rightarrow \cdot\ E, \#$
$E \rightarrow \cdot\ E + T, \#$
$E \rightarrow \cdot\ T, \#$

Cogemos ahora el item $[E \rightarrow E+T, \#]$. Como E vuelve a estar a la derecha del punto, e $INICIALES(+T\#) = +$, añadimos a J' todas las *E-producciones*, con un punto en el extremo izquierdo de su parte derecha y con + como *lookahead*. J' contendrá ahora:

$E' \rightarrow \cdot\ E, \#$
$E \rightarrow \cdot\ E + T, \#$
$E \rightarrow \cdot\ T, \#$
$E \rightarrow \cdot\ E + T, +$
$E \rightarrow \cdot\ T, +$

Ahora pasaríamos a considerar el siguiente item y así sucesivamente. Finalmente obtendremos que $J' = cierre(\{[E' \rightarrow \cdot E, \#]\})$ es:

$E' \rightarrow \cdot\ E,$	#
$E \rightarrow \cdot\ E + T,$	# / +
$E \rightarrow \cdot\ T,$	# / +
$T \rightarrow \cdot\ T * F,$	# / + / *
$T \rightarrow \cdot\ F,$	# / + / *
$F \rightarrow \cdot\ (E)$	# / + / *
$F \rightarrow \cdot\ id$	# / + / *

Siguiendo con el algoritmo, buscamos ahora los items de J' que tengan *lookaheads* distintos de #. $E \rightarrow \cdot E+T$ presenta el *lookahead* +, por tanto, éste se genera espontaneamente para el item $E \rightarrow E \cdot +T$ de $goto(I_0, E) = I_1$.

$E \rightarrow \cdot T$ también tiene + como *lookahead*, así que éste se genera espontáneamente para el item $E \rightarrow T \cdot$ de $goto(I_0, T) = I_2$. Haciendo esto para todos los items de J' se obtiene que + se genera espontáneamente para los items:

$E \rightarrow E \cdot + T \quad (I_1)$
$E \rightarrow T \cdot \quad (I_2)$
$T \rightarrow T \cdot * F \quad (I_2)$

$T \rightarrow F \cdot$ (I_3)
$F \rightarrow (\cdot E)$ (I_4)
$F \rightarrow id \cdot$ (I_5)

y * se genera espontáneamente para:

$T \rightarrow T \cdot * F$ (I_2)
$T \rightarrow F \cdot$ (I_3)
$F \rightarrow (\cdot E)$ (I_4)
$F \rightarrow id \cdot$ (I_5)

Ya tenemos los *lookaheads* generados espontáneamente en I_0. Ahora tenemos que ver como se propagan. Para cada item que tenga *#* como *lookahead*, los del item de K con el que estamos trabajando (aquel del que hicimos el cierre, $E \rightarrow \cdot E$ en este caso), se propagarám hasta él, es decir, los *lookaheads* se propagan desde $E \rightarrow \cdot E$ (I_0) hasta:

$E' \rightarrow E \cdot$ (I_1)
$E \rightarrow E \cdot + T$ (I_1)
$E \rightarrow T \cdot$ (I_2)
$T \rightarrow T \cdot * F$ (I_2)
$T \rightarrow F \cdot$ (I_3)
$F \rightarrow (\cdot E)$ (I_4)
$F \rightarrow id \cdot$ (I_5)

Con esto hemos terminado de calcular los *lookaheads* del item $E \rightarrow \cdot E$ de K, kernel de I_0. Como K no tiene mas items, pasaremos ahora a hacer lo mismo con cada uno de los dos items del kernel de I_1, $[E \rightarrow E\cdot]$ y $[E \rightarrow E\cdot+T]$.

Al hacer el *cierre* de $[E' \rightarrow E\cdot, \#]$ no se obtiene ningún item nuevo, ya que el punto está al final de la parte dercha. *cierre(*$\{[E \rightarrow E\cdot+T, \#]\}$*)* tampoco añadirá ningún item nuevo a J', pero en este caso, como el punto no está al final de la parte derecha de la regla, se propagarán los *lookaheads* desde $E \rightarrow E\cdot+T$ (I_1) hasta $E \rightarrow E+\cdot T$ (I_6).

Se continúa con el proceso hasta acabar todos los kernels Obtendremos:

I_2: Los *lookaheads* se propagan desde $T \rightarrow T\cdot*F$ (I_2) hasta $T \rightarrow T*\cdot F$ (I_7).

I_4: + y *)* se generan espontáneamente para los items:

$E \rightarrow E \cdot + T$ (I_1)
$E \rightarrow T \cdot$ (I_2)
$T \rightarrow T \cdot * F$ (I_2)
$T \rightarrow F \cdot$ (I_3)
$F \rightarrow (\cdot E)$ (I_4)
$F \rightarrow id \cdot$ (I_5)

y *:

$T \rightarrow T \cdot * F$ (I_2)
$T \rightarrow F \cdot$ (I_3)
$F \rightarrow (\cdot E)$ (I_4)
$F \rightarrow id \cdot$ (I_5)

Los *lookaheads* se propagan desde $F \rightarrow (\cdot E)$ a $F \rightarrow (E\cdot)$ (I_8)

I_6: * se genera espontáneamente para:

$T \rightarrow T \cdot * F$ (I_2)
$T \rightarrow F \cdot$ (I_3)
$F \rightarrow (\cdot E)$ (I_4)
$F \rightarrow id \cdot$ (I_5)

Los *lookaheads* se propagan desde $E \rightarrow E+\cdot T$ a:

$E \to E + T \cdot$ (I_9)
$T \to T \cdot * F$ (I_2)
$T \to F \cdot$ (I_3)
$F \to (\cdot E)$ (I_4)
$F \to id \cdot$ (I_5)

I_7: Los *lookaheads* se propagan desde $T \to T^* \cdot F$ a:

$T \to T * F \cdot$ (I_{10})
$F \to (\cdot E)$ (I_4)
$F \to id \cdot$ (I_5)

I_8: Los *lookaheads* se propagan:

de	$F \to (E \cdot)$	a	$F \to (E) \cdot$ (I_{11})
	$E \to E \cdot + T$		$E \to E + \cdot T$ (I_6)

I_9: Los *lookaheads* se propagan desde $T \to T \cdot *F$ a $T \to T^* \cdot F$ (I_7)

En I_3, I_5, I_{10}, e I_{11} ni se generan ni se propagan *lookaheads*.

Ahora podemos crear una tabla que contenga la información calculada acerca de la propagación de los *lookaheads* entre items. Sería:

DESDE	HASTA
I_0: $E' \to \cdot E$	I_1: $E' \to E \cdot$ I_1: $E \to E \cdot + T$ I_2: $E \to T \cdot$ I_2: $E \to T \cdot * F$ I_3: $T \to F \cdot$ I_4: $F \to (\cdot E)$ I_5: $F \to id \cdot$
I_1: $E \to E \cdot + T$	I_6: $E \to E + \cdot T$
I_2: $T \to T \cdot * F$	I_7: $T \to T * \cdot F$
I_4: $F \to (\cdot E)$	I_8: $F \to (E \cdot)$
I_6: $E \to E + \cdot T$	I_3: $T \to F \cdot$ I_4: $F \to (\cdot E)$ I_5: $F \to id \cdot$ I_9: $E \to E + T \cdot$ I_9: $T \to T \cdot * F$
I_7: $T \to T * \cdot F$	I_4: $F \to (\cdot E)$ I_5: $F \to id \cdot$ I_{10}: $T \to T * F \cdot$
I_8: $F \to (E \cdot)$	I_{11}: $F \to (E) \cdot$
I_8: $E \to E \cdot + T$	I_6: $E \to E + \cdot T$
I_9: $T \to T \cdot * F$	I_7: $T \to T * \cdot F$

El siguiente paso será crear otra tabla, que inicialmente contendrá los *lookaheads* que fueron generados espontáneamente para cada uno de los items que componen los kernels. Al item que contiene el símbolo inicial se le pone inicialmente como *lookahead $*.

Una vez inicializada la tabla, habrá que "propagar" los *lookaheads*. Para cada item *i* se mira a qué otros items propaga sus *lookaheads*, utilizando la información contenida en la tabla creada anteriomente y a todos ellos se les añaden como todos los que en este momento tiene el item *i*. Una vez hecho esto con todos los items, se volverá a repetir esta operación empezando por el primero otra vez, y se seguirá haciendo hasta que uno de los pasos no añada ningún *lookahead* más a ninguno de los items. Para el ejemplo, la tabla será:

CONJ	ITEM	LOOKAHEADS			
		INICIAL	PASO 1	PASO 2	PASO 3
I_0:	E'→ · E	$	$	$	$
I_1:	E'→ E ·		$	$	$
I_1:	E → E · + T	+ /)	$ / + /)	$ / + /)	$ / + /)
I_2:	E → T ·	+ /)	$ / + /)	$ / + /)	$ / + /)
I_2:	T → T · * F	+ / * /)	$ / + / * /)	$ / + / * /)	$ / + / * /)
I_3:	T → F ·	+ / * /)	$ / + / * /)	$ / + / * /)	$ / + / * /)
I_4:	F → (· E)	+ / * /)	$ / + / * /)	$ / + / * /)	$ / + / * /)
I_5:	F → id ·	+ / * /)	$ / + / * /)	$ / + / * /)	$ / + / * /)
I_6:	E → E + · T		+ /)	$ / + /)	$ / + /)
I_7:	T → T * · F		+ / * /)	$ / + / * /)	$ / + / * /)
I_8:	F → (E ·)		+ / * /)	$ / + / * /)	$ / + / * /)
I_8:	E → E · + T				
I_9:	E → E + T ·			+ /)	$ / + /)
I_9:	T → T · * F			+ /)	$ / + /)
I_{10}:	T → T * F ·			+ / * /)	$ / + / * /)
I_{11}:	F → (E) ·			+ / * /)	$ / + / * /)

En la columna *INICIAL* aparecen los *lookaheads* que habíamos visto que se generaban espontáneamente. En el paso 1, I_0 propaga los suyos a los dos items de I_1 e I_2 y al item de I_3, I_4 e I_5. El segundo item de I_1 ($E \rightarrow E{\cdot}{+}T$), los propaga a I_6, por lo tanto I_6 tendrá ahora como *lookaheads* + y *)*. Se hace lo mismo con el resto de los items de los kernels. Una vez hecho esto, se empieza de nuevo el proceso, propagando en este caso los *lookaheads* que aparecen para cada item en la columna *PASO 1*, dando como resultado la columna *PASO 2* y así sucesivamente. Al hacer la cuarta pasada se obtendría la columna *PASO 4*, exactamente igual a la anterior, *PASO 3*, (por lo que ya no aparece en la tabla). Esto indica que ya no se pueden propagar más los *lookaheads*, es decir, el proceso se ha terminado.

Ahora sólo falta obtener la tabla de análisis. El procedimiento es básicamente el mismo que para obtener las tablas SLR a partir de los items LR(0). La diferencia es que para los items del tipo $A \rightarrow a\cdot$ que producen reducciones, en lugar de reducir para todos los símbolos pertenecientes a *SIGUIENTES(A)*, se hará para los que estén en el apartado correspondiente a ese item de la última columna (en este caso *PASO 3*) de la tabla anterior.

Para el ejemplo visto, la tabla producida por el método LALR es exactamente igual a la SLR, pero en casos más complicados no sucederá así, especialmente cuando se trate de gramáticas con conflictos shift/reduce.

6.9 Generadores de analizadores sintácticos: yacc, Bison, PCYACC y YACCOV

Yacc es un generador de analizadores sintácticos, es decir, un programa que, a partir de la descripción de una gramática, genera un analizador sintáctico para el lenguaje descrito por ella. YACCOV es una versión de *yacc* creada en la Universidad de Oviedo (*YACCOViedo*) basado en otro generador BISON (GNU versión de *yacc*).

A la hora de diseñar YACCOV, se ha tratado de hacerlo compatible con *yacc* (generador de analizadores sintácticos incorporado al sistema operativo UNIX) y con PCYACC (implementación de *yacc* para PC's de *Abraxas Software Inc.*), de forma que la utilización de ambos programas sea similar. Esto hará posible que los usuarios de YACC puedan utilizar YACCOV sin ningún problema y de la misma forma, que los usuarios de YACCOV puedan utilizar YACC o PCYACC.

Existen algunas diferencias entre ellos que se indicarán más adelante, pero en general, todo lo que se diga en este manual respecto a la utilización de YACCOV, se puede aplicar también a YACC y PCYACC. Cualquier gramática descrita de la forma adecuada debe funcionar igual en uno y otro programa.

La gramática que se le da como entrada tiene que ser libre de contexto e ir expresada en una notación especial, que es bastante parecida a la BNF.

El analizador generado es LR y por tanto ascendente. No es un programa completo sino una función escrita en C llamada *yyparse* a la que el programador tendrá que añadir otras funciones para conseguir un compilador.

Para utilizar este programa es necesario conocer el lenguaje de programación C porque el fichero de salida estará escrito en ese lenguaje y también las acciones que acompañan a las reglas gramaticales y las funciones de apoyo a *yyparse*.

En adelante, siempre que en este manual se hable de *el analizador* o *el analizador sintáctico* se estará haciendo referencia al fichero producido por YACCOV (a menos que se indique otra cosa).

6.9.1 Utilización de yaccov

Para construir un compilador de un determinado lenguaje utilizando YACCOV, habrá que seguir los siguientes pasos:

1 - Definir el lenguaje para el que se va a construir el compilador,

2 - utilizando gramáticas libres de contexto. La gramática debe aparecer en el fichero de entrada a YACCOV escrita de forma que pueda ser entendida por él.
Escribir el fichero de entrada, cuya estructura se estudiará posteriormente y en el que va incluida la descripción de la gramática. Las reglas de producción pueden ir acompañadas por *acciones*, que son trozos de programa escritos en C que se ejecutarán cada vez que el analizador haga una reducción utilizando la regla a la que acompañan.

3 - Escribir las rutinas de apoyo que deberán acompañar a *yyparse* para formar el compilador o intérprete. Deben ser al menos tres: *main* (función de control), *yylex* (analizador léxico) e *yyerror* (función que se encarga de la recuperación de errores). *main* es la función principal del programa, que se encarga de llamar a *yyparse* entre otras cosas. *yyparse* supone que existen las funciones *yylex* e *yyerror* y las llama (utilizando esos nombres) cuando las necesita.

4 - Ejecutar YACCOV dándole como entrada el fichero que contiene la descripción de la gramática, para obtener el analizador sintáctico (*yyparse*).

5 - Compilar el fichero de salida obtenido junto con los otros ficheros (si los hay) que van a formar parte del compilador o intérprete y construir un programa ejecutable con todo ello.

Para llamar a YACCOV, se debe escribir en la línea de comandos:

yaccov [opciones] nombre_de_fichero

donde *nombre_de_fichero* es el nombre del fichero que contiene la descripción de la gramática. No existe ninguna restricción respecto a su nombre, pero normalmente se utiliza para él la extensión ".y". El nombre del fichero de salida será el del fichero de entrada cambiando su extensión por ".c" (si no se indica otra cosa en las opciones).

Al llamar a YACCOV se pueden indicar una, ninguna o varias de las siguientes opciones:

-*c*: Hace que el fichero generado por YACCOV se llame "yytab.c", en lugar de llevar el nombre del fichero de entrada con la extensión ".c". Este es el nombre que YACC da a los ficheros que produce.

-*C<nombre>*: Asigna al fichero generado por YACCOV el nombre que se indica en su argumento.

-*d*: Produce un fichero de cabecera con las *#define* de los tokens de la gramática. Su nombre será "yytab.h".

-*D<nombre>*: Hace lo mismo que -*d* pero le da al fichero producido el nombre indicado en su argumento si lo lleva o, si no es así, el nombre del fichero de entrada cambiando su extensión por ".h".

-*t*: Hace que se ponga la macro *#define YYDEBUG* en el fichero de salida, lo que hará que al ejecutar éste se saque por pantalla información acerca de las operaciones que se están realizando.

-*T*: Hace lo mismo que -*t*.

-*v*: Produce un fichero, de nombre "yy.lrt" que contiene información acerca del AF generado que puede resultar útil para localizar y corregir conflictos entre otras cosas.

-*V<nombre>*: Hace lo mismo que -*v*, pero le da al fichero producido el nombre que aparece en su argumento, o, si no lo lleva, el nombre del fichero de entrada cambiando su extensión por ".lrt".

-*l*: Evita que aparezcan macros *#line* en el fichero de salida.

-*L*: Hace lo mismo que -*l*.

-*p<fichero>*: Utiliza como estructura para el analizador que se va a producir el fichero que se indica en su argumento.

-*P<fichero>*: Hace lo mismo que -*p*.

-*h*: Saca una pantalla con información acerca de las opciones permitidas en YACCOV y su utilización.

-*H*: Hace lo mismo que -*h*.

Las opciones *D* y *V* llevan argumento opcional, es decir, pueden llevarlo o no. Si lo llevan, no puede haber espacios entre él y el carácter de opción. Por ejemplo, si queremos que el fichero producido a partir de *fich.y* por la opción *D* se llame *fichero.cab* tendríamos que poner:

yaccov ... -Dfichero.cab ... fich.y

Si después del carácter de opción (*D* en este caso) hubiera un espacio, YACCOV consideraría que la opción no lleva argumento. Esto no sucede con las opciones *p*, *P* y *C*, ya que al llevar argumento obligatorio, si después del carácter de opción aparece un espacio, se buscará el argumento en lo que venga a continuación.

No es necesario que las opciones vayan delante del nombre del fichero, podrán ir en cualquier orden, porque la función que trata la línea de comandos se encarga de ordenarlos de forma correcta.

Si se llama a YACCOV sin especificar el fichero de entrada, se sacará la misma pantalla de ayuda producida por la opción *-h*.

6.9.2 Símbolos

En la descripción de una gramática, podemos encontrarnos *símbolos terminales* o *tokens* y *símbolos no terminales* o *variables*. Pero a la hora de construir un analizador sintáctico hay que tener en cuenta dos tipos de *tokens*: por una parte están los de los ficheros de entrada a nuestro analizador, y por otra, los que aparecen en la descripción de la gramática, que representan a los anteriores. Los de los ficheros de entrada serán trozos de estos ficheros que la función *yylex* le irá pasando a *yyparse* para que los agrupe en sentencias. Cada uno de los tokens de la gramática representa a un token del fichero de entrada, o bien a un conjunto de ellos sintácticamente equivalentes. Por ejemplo: el token '*' de la gramática representará solamente a los símbolos ***** del fichero a analizar, pero el token *INTEGER* representará a todos los números enteros que aparezcan en él.

Todos los tokens utilizados en la gramática hacen que aparezca al principio del fichero de salida una *#define* que les asignará un código numérico. De esta forma, *yylex*, que cada vez que es llamada por *yyparse* tiene que devolverle el código de lo que acaba de leer, puede utilizar los nombres de los tokens en lugar de sus códigos. Los literales de un solo carácter no generan *#define*. Su código será el número ASCII que les corresponda.

El símbolo *error* es un terminal reservado para la recuperación de errores y no debe ser usado para ninguna otra cosa. En particular, *yylex* no debe devolver nunca este valor.

Todos los símbolos terminales de la gramática deben ser declarados, excepto los literales de un solo carácter. Los símbolos como '<>', '>=' o '<=', que son literales de más de un carácter, tienen que ser representados por un nombre (por ejemplo *NE*, *GE*, *LE*) y declarados en la sección de declaraciones (y por lo tanto, generarán una *#define* en el fichero de salida). Por el contrario, los símbolos no terminales se declaran implícitamente en las reglas de la gramática, no es necesario declararlos de forma explícita a menos que se quiera indicar el tipo que van a tomar sus valores semánticos.

6.10 Estructura del fichero de entrada

El fichero de entrada a YACCOV debe tener la forma:

```
DECLARACIONES
%%
DESCRIPCION DE LA GRAMATICA
%%
CODIGO C AÑADIDO
```

Las secciones de *declaraciones* y *código C adicional* son opcionales, pero la *descripción de la gramática* tiene que aparecer siempre. El símbolo *%%* es un delimitador especial que señala el principio y el final de estas secciones. El primero de ellos debe aparecer aunque no exista la sección de declaraciones, en cambio no es necesario poner el segundo si no hay código C adicional.

6.10.1 Sección de declaraciones

En esta sección pueden aparecer dos tipos de declaraciones: *declaraciones C* y *declaraciones específicas de YACCOV*. Se suele utilizar el formato:

```
%{
   DECLARACIONES C
%}

DECLARACIONES YACCOV
```

También pueden aparecer varios grupos de declaraciones C (cada uno con sus correspondientes *%{* y *%}*) intercalados con el resto de las declaraciones YACCOV, pero la utilización del formato indicado hará que el fichero de entrada presente una forma más estructurada.

Cualquier cosa encerrada entre *%{* y *%}* pasa directamente al principio del fichero de salida, sin ser tocado para nada por YACCOV, por tanto, si el texto escrito en estas zonas no es correcto, los errores tendrán que ser detectados posteriormente por el compilador de C. Como es colocado al inicio del fichero de salida, lo que suele aparecer en esta zona son declaraciones de tipos que serán utilizados en *yyparse*, macros como *#include* o cualquiera de las cosas que pueden estar al principio de un programa.

Tanto en esta sección como en el resto del fichero de entrada, se pueden incluir comentarios, que tendrán la misma estructura que en C, es decir, /*...*/

Las *declaraciones específicas de YACCOV* son una serie de cláusulas para definir los tokens y símbolos no terminales de la gramática y algunas de sus propiedades. Son:

%union -> Es una estructura (similar a la *union* de C) en la que se indican los tipos de los distintos valores que van a pasar por la pila del analizador generado (*yyparse*) durante el análisis sintáctico, es decir, los tipos de los valores semánticos de los tokens y no terminales que intervienen en la gramática. Por ejemplo:

```
%union
{
   tipo1  nombre1;
   tipo2  nombre2;
   tipo3  nombre3;
}
```

indica que la pila de *yyparse* contendrá valores de los tipos *tipo1*, *tipo2* y *tipo3*, que ocuparán los campos *nombre1*, *nombre2* y *nombre3* de la unión respectivamente. Esto pasa al fichero de salida transformado en:

```
typedef union
{
   tipo1  nombre1;
   tipo2  nombre2;
   tipo3  nombre3;
} YYSTYPE;
```

Si el usuario escribiese esto directamente al principio del fichero de salida o en las declaraciones C (*%{ ... %}*) y suprimiese la cláusula *%union* del fichero de entrada a YACCOV, el resultado sería exactamente el mismo. YYSTYPE es el nombre del tipo de la pila de *yyparse*. Si va a contener valores de un solo tipo, no es necesario utilizar la cláusula *%union*. Basta con poner:

```
%{
   :
   #define YYSTYPE tipo
   :
%}
```

en la sección de declaraciones, o bien poner la cláusula *#define* directamente al principio del fichero de salida. Si no se define ningún tipo de datos, por defecto el tipo de la pila será *entero*.

%token -> Esta cláusula indica que todos los símbolos que vienen a continuación de ella (hasta que aparezca una nueva declaración o *%%*) son tokens. Los nombres de los tokens pueden ir separados por comas o por blancos. También se puede indicar su tipo poniendo:

```
%token <tipo> token1 token2 token3 ...
```

siendo *tipo* uno de los campos definidos en *%union* (en el ejemplo anterior *nombre1*, *nombre2* o *nombre3*). Sólo se puede indicar un tipo por cada declaración *%token*.

YACCOV asigna automáticamente un número entero a cada token, que será el que le corresponda según el código ASCII si es un literal de un solo carácter o cualquier número que se le asigne si no lo es. En este caso, si queremos asignarle un número concreto, podemos indicarlo en esta declaración, poniendo detrás del nombre del token el código que queramos que tenga, que no debe coincidir con el de ningún otro. Generalmente es mejor dejar que sea el programa quien elija los códigos numéricos para los tokens, ya que se ocupará de que sean todos distintos y de que no coincidan con los de los caracteres ASCII, liberando de esta carga al programador.

%type -> Se utiliza para declarar el tipo de los símbolos no terminales. Su estructura es similar a la de *%token* con la diferencia de que *%type* tiene que llevar obligatoriamente un nombre de tipo detrás:

```
%type <tipo> variable1 variable2 variable3 ...
```

%left, %right, %nonassoc -> Estas cláusulas se utilizan para declarar un token y a la vez especificar su precedencia y asociatividad. Su sintaxis es la misma que la de *%token*. Indican que todos los tokens nombrados a continuación de ellas son asociativos de izquierda a derecha, de derecha a izquierda o no asociativos respectivamente. También pueden especificar su tipo y su código numérico como ocurría con la cláusula *%token*.

La *asociatividad* de un operador OP determina cómo se asocian los usos repetidos de éste, es decir, dada la expresión *X OP Y OP Z*, si se debe analizar agrupando primero X con Y o Y con Z. *%left* indica asociatividad de izquierda a derecha (primero se agrupa X con Y, es decir, se ejecutará *(X OP Y) OP Z* y *%right* de derecha a izquierda (primero se agrupa Y con Z, es decir, se ejecutará *X OP (Y OP Z))*. *%nonassoc* indica que no hay asociatividad, es decir, *X OP Y OP Z* se considerará un error de sintaxis.

La *precedencia* de un operador determina cómo se asocia éste con otros operadores. Si en la sección de declaraciones aparecen varias declaraciones de asociatividad, cada una de ellas tendrá una precedencia un nivel superior a la que la precede, es decir, si en el fichero de entrada aparece:

```
%left '+' '-'
%left '*' '/'
%right '^'
```

la precedencia más baja la tienen los tokens '+' y '-', la de '*' y '/' será superior a la de los anteriores y la precedencia más alta de todas será la de '^'. Dentro de una misma declaración de asociatividad, todos los tokens tienen la misma precedencia.

%start -> Esta cláusula indica que lo que viene a continuación de ella es el símbolo inicial de la gramática. Tiene que ser necesariamente un no terminal, si no lo es se producirá un error. Si no aparece esta cláusula en el fichero de entrada, se toma como símbolo inicial el no terminal que aparece en la parte izquierda de la primera regla especificada en la descripción de la gramática.

%expect -> YACCOV, al final de su ejecución saca un mensaje indicando el número de conflictos shift/reduce y reduce/reduce que encontró en la gramática (si los había). Si ya sabemos cuántos conflictos shift/reduce tiene la gramática y queremos evitar que salga el mensaje, basta con poner en la sección de declaraciones esta cláusula seguida del número de conflictos de este tipo que esperamos que se produzcan. Si en la gramática aparece un número de conflictos shift/reduce distinto al que indicamos, el mensaje saldrá de todas formas.

%pure_parser -> Indica que se quiere obtener un analizador reentrante. Normalmente, el analizador producido por YACCOV no lo es, porque utiliza dos variables estáticamente localizadas para la comunicación con *yylex*, que son *yylval* e *yylloc*. Al poner esta cláusula en la sección de declaraciones, se consigue que estas dos variables sean punteros pasados como argumentos a *yylex*, donde deberá aparecer:

```
yylex (yylval, yylloc)
YYSTYPE *yylval;
YYLTYPE * yylloc;
{
  ...

  *yylval = valor;   /* Poner valor en la pila de YACCOV */
  return INT;        /* Devolver el tipo del token.      */

  ...
}
```

El resto del fichero de salida no se modifica.

%semantic_parser -> Esta cláusula hace que se utilice como estructura para el analizador que se va a generar el fichero "SEM.PRS", en lugar de "SIMPLE.PRS". Este fichero no define las mismas macros que "SIMPLE.PRS" por lo que un fichero de entrada que fuera válido para este último, podría requerir algunos cambios para que pueda llevar como estructura el contenido de "SEM.PRS"

Todas estas declaraciones pueden ir separadas por blancos o por ';'.

Cualquier símbolo que aparezca en una regla gramatical y no haya sido declarado en esta sección se considera no terminal (excepto los tokens literales de un solo carácter).

6.10.2 Descripción de la gramática

Esta parte del fichero de entrada es la única que tiene que aparecer necesariamente. Contiene la descripción de la gramática, que deberá ir expresada de forma que YACCOV pueda entenderla.

Los símbolos, tanto terminales como no terminales, se representan solamente con su nombre. Por convenio se escriben en mayúsculas los tokens y en minúsculas los no terminales, para distinguirlos. Los símbolos terminales de un sólo carácter (signos de puntuación, paréntesis, etc.) pueden ser representados por ese mismo carácter encerrado entre comillas, por ejemplo, el token ***** se representará mediante '*'.

Las reglas de producción serán de la forma:

```
parte_izquierda : parte_derecha  ;
```

parte_izquierda es el símbolo no terminal descrito por esa regla y *parte_derecha* es un conjunto de símbolos terminales y/o no terminales. Por ejemplo:

```
exp : exp '+' exp  ;
```

indica que si en la entrada del analizador aparece una *expresión* seguida de un signo '+' y otra *expresión*, todo esto debe ser agrupado para formar una nueva *expresión*.

Los espacios en blanco (incluyendo tabuladores y marcas de fin de línea) en las reglas se utilizan sólo para separar símbolos. Se pueden añadir tantos como se quiera. El signo ':' separa las dos partes de la regla y el ';' indica el fin de esta. Si existe más de una regla para describir a un no terminal, se puede utilizar la siguiente notación:

```
parte_izquierda :  componentes de la regla 1
                | componentes de la regla 2
                | ...
                ;
```

Aunque vayan expresadas de esta forma, YACCOV internamente las trata como reglas independientes. No es necesario poner una regla en cada línea, pero normalmente se utiliza esta notación para hacer la gramática más legible.

Si en la parte derecha de una regla no hay nada, significa que el no terminal que aparece en su parte izquierda acepta la cadena vacía. Por ejemplo, para definir una secuencia de cero o más expresiones separadas por comas, se podría escribir:

```
exps  :  /* vacío */
       | exps1
       ;
exps1 :  exp
       | exps1 ',' exp
       ;
```

Cuando la parte derecha de una regla está vacía suele escribirse el comentario */* vacío */* para hacer más legible la descripción de la gramática.

Acompañando a las reglas pueden aparecer *acciones*, que determinan su semántica. Normalmente sólo existe una acción para cada regla y va al final de ella, detrás de todos sus componentes y antes del ';'.

Se pueden poner comentarios en cualquier parte de la decripción de la gramática.

Para que una gramática sea correcta hay que tener en cuenta dos cosas: todos los símbolos que aparecen en la parte izquierda de alguna regla tienen que ser no terminales y todos los no terminales que aparecen en la descripción de la gramática deben estar en la parte izquierda de al menos una regla de producción.

6.10.3 Código C añadido.

Todo lo incluido en esta sección se copia literalmente al final del fichero de salida de YACCOV. Este es el lugar más apropiado para poner cualquier cosa que queramos que acompañe al analizador y que no tenga que estar delante de la definición de *yylex*. Los códigos de *yylex* e *yyerror* suelen estar aquí, aunque también pueden ir en ficheros independientes.

El fichero de salida contiene muchas variables estáticas, macros y funciones cuyos nombres comienzan por *yy* o *YY*, por lo que sería recomendable evitar que los nombres de las variables, funciones, etc. que aparezcan en esta sección empiecen por estas letras.

En programas pequeños, suele incluirse todo el código que acompaña a *yyparse* en esta sección, pero para proyectos más complejos es mejor distribuir el texto fuente en varios ficheros para facilitar la localización y corrección de los errores.

Si esta sección está vacía, se puede omitir el *%%* que la separa de las reglas gramaticales.

6.11 Semántica

Las reglas de producción solamente determinan la sintaxis, es decir, la estructura de las sentencias del lenguaje. La semántica viene determinada por los *valores semánticos* asociados a los tokens y no terminales de la gramática y por las acciones que aparecen en las reglas.

6.11.1 Valores semánticos

Una gramática formal define qué tokens pueden aparecer en cada sitio, por ejemplo, si una regla menciona el token *constante_entera*, quiere decir que cualquier constante entera es sintácticamente válida en esa posición, independientemente de su valor. Pero este valor es muy importante a la hora de ejecutar la entrada una vez que ésta ha sido analizada. Por esta razón cada token de la gramática lleva asociado un valor semántico, que puede ser su valor si representa a un número, su nombre si representa a un identificador, etc. Los signos de puntuación y operadores aritméticos no tienen valor semántico.

Los símbolos no terminales pueden llevar también un valor semántico asociado.

Las reglas gramaticales no utilizan los valores semánticos para nada, éstos son utilizados en las acciones que las acompañan.

En un programa simple, puede ser suficiente con un solo tipo de datos para los valores semánticos de todos los símbolos de la gramática. Por defecto se usa el tipo entero para todos ellos. Para especificar cualquier otro tipo se define YYSTYPE en la sección de declaraciones.

Pero en la mayoría de los programas se necesitarán diferentes tipos de datos, por ejemplo, una constante numérica puede ser de tipo *int* o *long*, mientras que una cadena de caracteres necesita el tipo *(char *)* y el valor semántico de un identificador podría ser un puntero a una entrada en la tabla de símbolos. Para poder utilizar valores semánticos de varios tipos son necesarias dos cosas: declarar todos los tipos posibles utilizando la cláusula *%union* y elegir uno de esos tipos para cada símbolo que vaya a utilizar valores semánticos, asignándoselos en las declaraciones *%token* y *%type* de la forma vista.

6.11.2 Acciones

Para que sea útil, nuestro compilador además de analizar la entrada, deberá producir una salida. Conseguir esto es la principal función de las *acciones* que acompañan a las reglas gramaticales.

Las *acciones* son trozos de programa escritos en C que van asociados a algunas reglas gramaticales. Cada vez que el analizador semántico efectúe una reducción utilizando una de las reglas gramaticales, inmediatamente después ejecutará la acción correspondiente a esa regla. La mayor parte de las veces, el propósito de una acción es hallar el valor semántico del símbolo que aparece en la parte izquierda de la regla a partir de los valores semánticos de los componentes de su parte derecha. Por ejemplo, la siguiente regla:

```
expr :  expr '+' expr  { $$ = $1 + $3; }
```

indica que se deben combinar las dos expresiones que aparecen en la parte derecha separadas por el signo más, para dar lugar a otra expresión de orden superior. La acción indica que el valor semántico de la nueva expresión debe ser calculado sumando los valores semánticos de las otras dos expresiones.

Si la acción no asigna un valor semántico a *$$*, el valor del símbolo que está en la parte izquierda de la regla será impredecible y podría ocasionar problemas si se utilizara posteriormente en otras acciones.

El código C de una acción puede hacer referencia a los valores semánticos de los componentes de la regla, utilizando la construcción *$N* (siendo *N* un número entero) para representar el valor del símbolo que aparece en la posición *N* de la parte derecha de la regla y *$$* para referirse al valor semántico del símbolo de la parte izquierda de la regla.

Por ejemplo en:

```
expr1 : expr2 '+' expr3  { $$ = $1 + $3;  }
```

$$ se refiere al valor semántico de *expr1*, *$1* al de *expr2* y *$3* al de *expr3*.

Estas construcciones serán transformadas en referencias a elementos de un array al pasar al fichero de salida (*yyparse*).

En *$N*, *N* puede ser 0 o negativo, para referenciar símbolos que hayan sido introducidos en la pila del analizador antes que los de la regla que está siendo analizada. No es recomendable hacer esto, a menos que se conozca con certeza el contexto en que es aplicada la regla. Un ejemplo de utilización de esta construcción podría ser:

```
expr1:  expr previa '+' expr  {...}
      | expr previa '-' expr  {...}
                              ;
previa: /* vacío */  { expr_previa = $0; }
                              ;
```

En este caso, *$0* se refiere siempre al valor de la expresión que va antes de *previa*.

Si se ha elegido un solo tipo de datos para los valores semánticos, las construcciones *$$* y *$N* tendrán siempre ese tipo. Pero si se han especificado varios mediante la cláusula *%union*, entonces se debe indicar para cada símbolo que vaya a tener un valor semántico, cuál va a ser su tipo. Cada vez que se use *$$* o *$N*, el tipo de la construcción vendrá determinado por el del símbolo al que se refiere.

También se puede especificar el tipo en las construcciones *$$* o *$N*, insertando el nombre del tipo entre *$* y lo que venga detrás (*$* o *N*) de la forma:

```
$<tipo>N     o     $<tipo>$
```

Por ejemplo, si tenemos:

```
%union
{
   int     tipoi;
   double  tipod;
}
```

$<tipoi>1 indicará que el primer símbolo que aparece en la parte derecha de la regla es de tipo *int* y de la misma forma *$<tipod>1* indicará que es de tipo *double*.

Los símbolos (tanto terminales como no terminales) pueden ser tratados como variables ordinarias dentro de las acciones, pueden aparecer en expresiones como operandos, ser pasados como argumentos a subrutinas, se les pueden asignar valores, etc. La única diferencia es que deben ser accedidos utilizando las notaciones *$$* y *$N*. Como los valores semánticos de los símbolos tienen tipos asignados, el programador debe ocuparse de que en estas manipulaciones, las acciones se ajusten a las reglas de la programación C. YACCOV no mira lo que hay dentro de las acciones, por tanto no podrá detectar errores de programación dentro de ellas.

Si no se especifica ninguna acción para una regla, por defecto se le adjudica *{ $$ = $1 ; }*, es decir, se hace que el símbolo de la parte izquierda de la regla tome el valor semántico del primer componente de la parte derecha de la misma.

Además de las construcciones vistas, en las acciones puede aparecer:

- *YYERROR* .- Causa un inmediato error de sintaxis. Esto hace que se llame a *yyerror* y que se activen los mecanismos de recuperación de errores.
- *YYACCEPT* .- Hace que *yyparse* devuelva el control inmediatamente indicando que el análisis ha sido válido.
- *YYABORT* .- Hace que *yyparse* devuelva el control inmediatamente indicando que se ha producido un error.

- *yyclearin* .- Hace que se deseche el *lookahead* actual. Se utiliza para la recuperación de errores.
- *yyerrok* .- Hace que se vuelvan a generar mensajes de error.
- *@N* .- Actúa como una variable estructurada que contiene información sobre los números línea y columna del componente que ocupa la posición *N* en la regla que se está analizando, almacenada en los campos *first_line*, *first_column*, *last_line* y *last_column*. Por ejemplo, para obtener el número de la primera línea en la que aparece el segundo componente de una regla, en la acción que lleva asociada tendrá que poner:

  ... @2.first_line ...

  Esta información debe ser colocada en la variable *yylloc* por *yylex*.

6.11.3 Acciones en mitad de las reglas gramaticales

A veces puede ser útil o necesario escribir una acción entre los símbolos de una regla, es decir, antes del fin de ésta. Estas acciones son exactamente iguales que las convencionales; la única diferencia es que se ejecutan antes de que el analizador reconozca la regla completa y por tanto, antes de que se realice la reducción.

Una acción de este tipo puede hacer referencia a los símbolos que la preceden en la parte derecha de la regla de la forma habitual, utilizando las construcciones *$N*, pero no a los que la siguen, ya que es ejecutada antes de que éstos sean analizados. La acción se cuenta como un componente más de la regla, de tal forma que si aparece otra más adelante en esa misma regla, las construcciones *$N* de ésta deberán tenerla en cuenta como un símbolo más. Incluso podrán referirse a su valor semántico (el valor que toma *$$* dentro de ella una vez que es ejecutada) utilizando *$N* (*N* será el valor que le corresponda según el lugar que ocupe dentro de la regla). Como su valor semántico no va asociado a ningún símbolo gramatical, no se puede declarar su tipo, por lo que habrá que utilizar *$<...>N* cada vez que este valor sea utilizado en acciones posteriores.

Un ejemplo muy simple de utilización de este tipo de acciones podría ser:

```
exp : factor '*' factor     { $<int>$ = $1 * $3 ; }
      '+' sumando           { $$ = $<int>4 + $6 ; }   ;
```

La utilización de acciones en mitad de las reglas gramaticales puede ocasionar problemas en el análisis, por lo que es recomendable evitarlas. Cualquier acción de ese tipo puede pasar a ser una acción normal (al final de la regla). Considerando el ejemplo anterior podemos hacer:

```
producto : factor '*' factor  { $$ = $1 * $3 ; }  ;
exp      : producto '+' sumando  { $$ = $1 + $3 ; }  ;
```

con lo que habremos evitado la acción en mitad de la regla.

YACCOV, cada vez que se encuentra una de estas acciones, genera un no terminal falso que pasa a sustituir a la acción en mitad de la regla y añade una regla a la gramática, que tendrá como parte izquierda el no terminal generado y la parte derecha vacía y con esa acción al final. Para el ejemplo visto sería:

```
@1 : /* vacío */     { $<int>$ = $1 * $3 ; }  ;
exp      : factor '*' factor @1
             '+' sumando  { $$ = $<int>4 + $6 ; }  ;
```

El nombre del no terminal generado será siempre @*numero* (*numero* se incrementa cada vez que se produce uno de estos símbolos, para que no pueda haber dos con el mismo nombre).

6.12 Recursividad

Cuando en una regla de producción el símbolo de la parte izquierda aparece también entre los componentes de la parte derecha, se dice que la regla es *recursiva*. Casi todas las gramáticas libres de contexto utilizan la recursividad para definir una secuencia de 0 o más apariciones de algo. Por ejemplo:

```
exprs :  expr
       | exprs ',' expr
       ;
```

define una secuencia de una o más expresiones separadas por comas. Al menos una de las reglas debe llevar fuera de la recursión, si no se entraría en un bucle infinito.

Como *exprs* aparece al principio de la parte derecha de la regla, se dice que la regla es *recursiva a la izquierda*. Por el contrario:

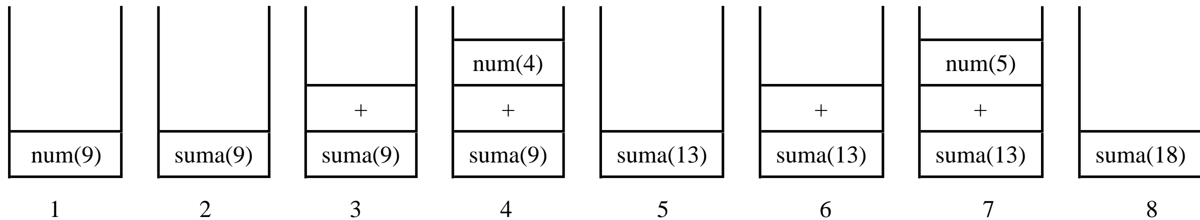
```
exprs :  expr
       | expr ',' exprs
       ;
```

será *recursiva a la derecha*. Los dos tipos de recursividad hacen lo mismo pero de distinta forma.

Cualquier tipo de secuencia puede ser definida utilizando recursividad a la izquierda o a la derecha. En los analizadores LR (y en todos los ascendentes en general) se debe utilizar siempre recursividad a la izquierda porque ocupa menos espacio en la pila del analizador. Por ejemplo dadas las siguientes reglas:

```
suma :  num                 [1]
      | suma  '+' num       [2]
      ;
(suponemos num : 0 | 1 | .. | 9  ; )
```

si queremos reconocer la expresión *9 + 4 + 5*, los contenidos de la pila en las sucesivas etapas del análisis serán:



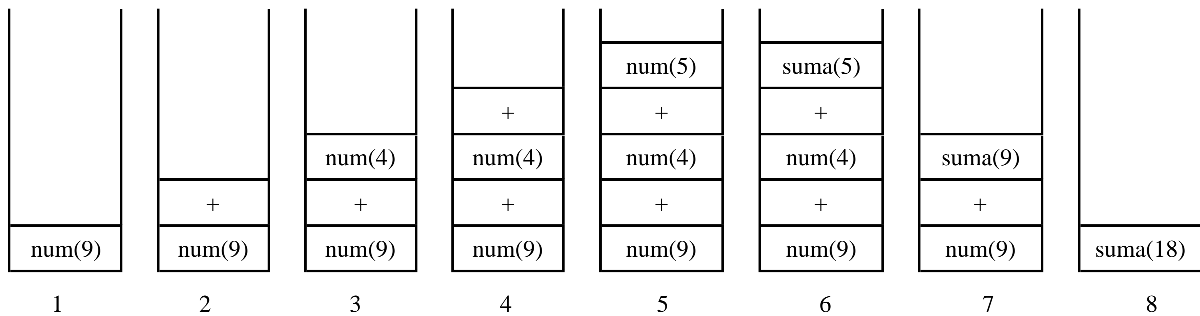
1 - Se carga *num* (9) en la pila.
2 - Se reduce *num* (9) a *suma* (9) utilizando la regla [1].
3 - Se carga '+' en la pila.
4 - Se carga *num* (4) en la pila.
5 - Se reduce *num* (4) + *suma* (9) a *suma* (13) utilizando la regla [2].
6 - Se carga '+' en la pila
7 - Se carga *num* (5) en la pila.
8 - Se reduce *num* (5) + *suma* (13) a *suma* (18) utilizando la regla [2].

Los números entre paréntesis indican el valor semántico de los símbolos que aparecen delante de ellos.

La pila nunca llega a contener más de tres elementos. En cambio, si la regla fuera:

```
suma :  num               [1]
      | num '+' suma      [2]
      ;
```

los contenidos de la pila serían:



1- Se carga *num* (9) en la pila

2- Aquí aparece un conflicto shift/reduce. El analizador tiene que elegir entre cargar el símbolo siguiente ('+') o reducir *num* a *suma* utilizando la regla [1]. Si no se le indica otra cosa, elige siempre cargar. Pero aunque quisiéramos no podríamos hacer la reducción porque no tenemos ninguna regla cuya parte derecha empiece con *suma* así que nunca llegaríamos a reconocer la cadena completa por este camino. Por lo tanto se carga '+' en la pila.
3- Se carga *num* (4) en la pila.
4- Se carga '+' en la pila (aquí ocurre lo mismo que en el apartado 2).
5- Se carga *num* (5) en la pila.
6- Esta vez ya no nos queda nada que cargar, así que habrá que reducir *num* (5) a *suma* (5) utilizando la regla [1].
7- Se reduce *num* (4) + *suma* (5) a *suma* (9) utilizando la regla [2].
8- Se reduce *num* (9) + *suma* (9) a *suma* (18) utilizando la regla [2].

En esta caso la pila llega a tener hasta cinco elementos.

En este ejemplo simplísimo, no tiene mucha importancia la diferencia entre una y otra recursividad, pero en situaciones más complicadas, el utilizar la recursividad a la derecha podría incluso llegar a agotar la capacidad de la pila del analizador.

Por razones similares, en analizadores descendentes, en los que se utilizan derivaciones en lugar de reducciones, se debe evitar la recursividad a la izquierda, utilizando siempre recursividad a la derecha.

Hasta ahora sólo hemos visto *recursividad directa*. Pero también existe la *recursividad indirecta*, que es la que se produce cuando el símbolo de la parte izquierda no aparece en la parte derecha de su misma regla, pero sí en las partes derechas de otras reglas que definen símbolos no terminales que sí aparecen en la parte derecha de su regla. Por ejemplo:

```
exp:  termino
    | termino '+' termino
    ;

termino :  constante
         | '(' exp ')'
         ;
```

en la parte derecha de la definición de *exp* aparece *termino* y en la parte derecha de la definición de *termino* aparece *exp*, por tanto este será un caso de recursividad indirecta.

6.13 Conflictos, ambigüedad y precedencia

A la hora de elaborar un analizador sintáctico, siempre es mejor trabajar con gramáticas no ambiguas, porque si es posible reconocer una sentencia mediante distintas secuencias de reglas gramaticales (es decir, produciendo varios árboles sintácticos distintos), aunque desde el punto de vista sintáctico todos estos árboles producen el mismo resultado, semánticamente no es así. Se producirán diferentes resultados semánticos dependiendo de qué acciones y en qué orden se ejecuten. Por ejemplo si tenemos la gramática:

```
%token NUMERO
%%
expr :  expr '+' expr  { $$ = $1 + $3; }    [1]
      | expr '*' expr  { $$ = $1 * $3; }    [2]
      | NUMERO         { $$ = $1; }         [3]
      ;
```

dada la expresión *3 + 2 * 5*, existen al menos dos árboles sintácticos que la reconocen:

```
NUMERO(3) + NUMERO(2) * NUMERO(5)    (aplicando [3] se pasa a:)
 expr(3)  + NUMERO(2) * NUMERO(5)    (aplicando [3]:)
 expr(3)  +  expr(2)  * NUMERO(5)    (aplicando [1]:)
       expr(5)        * NUMERO(5)    (aplicando [3]:)
       expr(5)        *  expr(5)     (aplicando [2]:)
                expr(25)
```

y

```
NUMERO(3)  + NUMERO(2)  * NUMERO(5)     (aplicando [3]:)
 expr(3)   + NUMERO(2)  * NUMERO(5)     (aplicando [3]:)
 expr(3)   +  expr(2)   * NUMERO(5)     (aplicando [3]:)
 expr(3)   +  expr(2)   *  expr(5)      (aplicando [2]:)
 expr(3)   +        expr(10)            (aplicando [1]:)
             expr(13)
```

En ambos casos se reconoce la expresión como sintácticamente correcta, pero en el primer caso se produce 25 como resultado y en el segundo 13, por lo tanto sólo es válido el segundo árbol.

En este ejemplo es fácil eliminar la ambigüedad, bastará con asignarles la precedencia adecuada al producto y a la suma. Pero no siempre es así de sencillo, incluso hay casos en los que no se la puede eliminar del todo.

El tratar de analizar sintácticamente sentencias de una lenguaje generado por una gramática ambigua puede producir problemas. En los analizadores LR, y en concreto en *yyparse*, la ambigüedad de las gramáticas da lugar a lo que se conoce como *conflictos shift/reduce* y *conflictos reduce/reduce*.

6.13.1 Conflictos shift/reduce (s/r)

Para analizar la entrada, el analizador sintáctico va cargando en la pila los tokens que le pasa el analizador léxico y cuando los elementos que ocupan su parte superior se ajustan a alguna de las reglas de la gramática realiza una reducción, sustituyendo estos símbolos por el que aparece en la parte izquierda de la regla aplicada.

Cuando en un momento determinado del análisis, el analizador no sabe si debe cargar (shift) o reducir (reduce), se dice que se ha producido un *conflicto s/r*.

Por ejemplo, un caso típico es el de las sentencias *if-then* e *if-then-else*. Si tenemos las reglas:

```
sentencia_if :  IF expr THEN sentencias
             |  IF expr THEN sentencias ELSE sentencias
             ;
```

cuando se lee el token *ELSE*, el analizador puede hacer dos cosas: reducir utilizando la primera regla, o cargar el *ELSE* para reducir más adelante utilizando la segunda regla. Entonces se ha producido un conflicto s/r.

Cuando aparece uno de estos conflictos, el analizador elige siempre cargar (a menos que se le indique otra cosa), porque la mayor parte de las veces ésta es la opción más adecuada. En el ejemplo visto, cada *ELSE* que aparezca en la entrada será relacionado con el *IF* ya leído más cercano a él (que será el más interno si hay varios anidados) y esto es lo que hacen la mayoría de los lenguajes de programación que utilizan esta sentencia. Por supuesto sería mejor que la gramática no fuera ambigua, pero eso en este caso es bastante difícil de conseguir.

Pero no siempre que se produce un conflicto s/r se debe cargar. Por ejemplo, si estamos analizando la expresión *3*5+1*, al llegar al signo '+' se produce un conflicto s/r. Si esta vez eligiéramos cargar iríamos en contra de las leyes de la aritmética, que dicen que el producto tiene una precedencia superior a la de la suma.

Así pues, en las expresiones aritméticas, para decidir si se debe cargar o reducir habrá que tener en cuenta las precedencias y asociatividades de los operadores, que se definen en la sección de declaraciones de la forma vista.

La mayoría de las gramáticas tienen conflictos s/r que una vez estudiados son inofensivos y que podrían ser difíciles de eliminar. Si al ejecutar YACCOV con nuestra gramática ya sabemos cuantos conflictos s/r tiene, podemos suprimir el mensaje de aviso que saca siempre al final de su ejecución informando del número de conflictos producidos. Para ello se utiliza la declaración *%expect*. Si en la sección de declaraciones aparece *%expect N* (siendo *N* el número de conflictos s/r que esperamos que se produzcan) y en la gramática se producen *N* conflictos s/r, el mensaje no saldrá.

%expect anula los avisos de conflictos s/r, pero no los de conflictos reduce/reduce. Siempre que se produzca algún conflicto de este tipo se notificará mediante un mensaje de aviso ya que estos conflictos generalmente no son tan "inofensivos" como los s/r y pueden producir problemas serios.

La técnica general para utilizar *%expect* sigue los siguientes pasos:

a) Ejecutar YACCOV sin *%expect* y utilizando la opción '-v', para saber cuántos conflictos s/r hay en la gramática y dónde y por qué se producen.

b) Estudiar cada uno de los conflictos para asegurarse de que la resolución que se toma por defecto es la correcta. Si no es así, habrá que reescribir la gramática.

c) Añadir *%expect N* a la sección de declaraciones.

Una vez hecho esto, YACCOV no volverá a avisarnos de que existen los conflictos que ya conocemos, pero sí lo hará si aparece alguno nuevo.

6.13.2 Conflictos reduce/reduce (r/r)

Se produce un *conflicto r/r* cuando en un momento determinado del análisis se puede reducir utilizando dos o más reglas distintas.

Por ejemplo, si tenemos:

```
%token NUMERO
%token IDENTIFICADOR
%token GOTO
%%
programa :  sentencia                  [1]
         | programa sentencia          [2]
         ;
sentencia :  sent_goto                 [3]
          | expresion                  [4]
          ;
sent_goto :  GOTO etiqueta             [5]
          ;
etiqueta : IDENTIFICADOR               [6]
         ;
expresion :  expresion '+' expresion   [7]
          | NUMERO                     [8]
          | IDENTIFICADOR              [9]
          ;
```

El analizador, cuando se encuentra un *IDENTIFICADOR*, tiene que elegir entre reducir a *expresion* utilizando la regla [9] o a *etiqueta* mediante la [6].

Por defecto, YACCOV elige reducir utilizando la regla que aparece primero en la descripción de la gramática, pero esto es muy arriesgado.

Generalmente, los conflictos r/r se producen por errores en el diseño del lenguaje o en la escritura de la gramática. Si el error se produjo en el diseño del lenguaje, habrá que revisarlo y diseñarlo de nuevo. Si el conflicto se produjo por una mala escritura de la gramática, existen algunos métodos para eliminarlo. Uno de ellos consiste en utilizar directamente las partes derechas de las reglas que produjeron el conflicto en todos los sitios donde aparecía la parte izquierda de éstas. Siguiendo con el ejemplo anterior, las reglas que producían el conflicto eran la [6] y la [9]. Si suprimimos la regla [6] y reescribimos la [5] como sigue:

```
sent_goto :  GOTO IDENTIFICADOR
          ;
```

habremos eliminado el conflicto r/r.

Pero no siempre es tan fácil hacerlo, no existe un método general que elimine los conflictos r/r de las gramáticas. Cada conflicto deberá ser estudiado detenidamente y, a ser posible, eliminado.

6.13.3 Precedencia

No sólo los símbolos terminales tienen precedencia, también las reglas de producción pueden tenerla. Cada regla toma la precedencia y asociatividad del último símbolo terminal especificado en su parte derecha (si existe este símbolo y si tiene precedencia y asociatividad declaradas) a menos que se indique explícitamente otra cosa. Para indicar explícitamente la precedencia y asociatividad de una regla se utiliza la declaración *%prec*, que es especialmente útil cuando un mismo terminal tiene varios significados dependiendo del contexto en que aparezca. Por ejemplo, el signo menos puede ser utilizado como operador binario para realizar restas y como operador unario para el cambio de signo. La precedencia es distinta según se trate de un caso o de otro.

Las declaraciones de precedencia y asociatividad (*%left*, *%right* y *%nonassoc*) sólo pueden ser usadas una vez para cada token, por lo tanto un token sólo tiene una precedencia declarada de esta forma. Para declarar la otra habrá que utilizar *%prec*. Esta declaración indica cuál es la precedencia de una regla particular especificando un símbolo terminal cuya precedencia sea la misma que queremos que tenga esa regla. No es necesario que ese terminal aparezca en otra parte de la regla. La sintaxis de esta declaración es :

```
%prec TERMINAL
```

y se escribe después de los componentes de la parte derecha de la regla y antes de la acción (si la hay). Lo que hace es asignar a la regla la precedencia de *TERMINAL* en lugar de la que tendría dependiendo del último símbolo terminal que apareciera en su parte derecha.

La definición de una gramática que describa expresiones aritméticas incluyendo los dos usos de '-' puede ser:

```
%token NUMERO
%left '+' '-'
%left '*' '/'
%left MENOSUNARIO
%%
expr :  expr '+' expr    { $$ = $1 + $3;  }
      | expr '-' expr    { $$ = $1 - $3;  }
      | expr '*' expr    { $$ = $1 * $3;  }
      | expr '/' expr    { $$ = $1 / $3;  }
      | '-' expr %prec MENOSUNARIO  { $$ = - $2;  }
      | NUMERO
      ;
```

El símbolo terminal *MENOSUNARIO* no se utiliza para nada en la descripción de la gramática, su único fin es representar la precedencia más alta para poder asignársela a la regla que maneja el menos unario.

Para resolver conflictos s/r lo que se hace es comparar la precedencia de la regla que está siendo considerada con la del token que va a ser leído a continuación. Si es mayor la del token se elige cargar y si es mayor la de la regla se reduce. Si tienen la misma precedencia, la elección se basa en la asociatividad: si ese nivel de precedencia es asociativo de izquierda a derecha (*%left*) se elige reducir, si lo es de derecha a izquierda (*%right*) se carga y si no es asociativo se producirá un error. Si la regla o el token que se va a leer no tienen precedencia, por defecto se elige cargar.

Si al ejecutar YACCOV se utiliza la opción '-v', además del que contiene a *yyparse* se generará otro fichero de salida en el que se indicará cómo se resolvió cada conflicto.

6.14 Funciones que acompañan a *yyparse*

YACCOV no produce un programa C completo sino una función que no podrá ser ejecutada si no se le añaden algunas más. Básicamente habrá que añadir una función principal (*main*), una función analizador léxico (*yylex*) y una función de manejo de errores (*yyerror*). Normalmente se añaden más, pero sólo estas tres son imprescindibles.

6.14.1 La función principal *main*

Esta función, en su forma más simple consistirá en una llamada a *yyparse*:

```
main()
{
   yyparse;
}
```

pero esta simplísima *main* sólo podrá ser utilizada en programas tan simples como ella. Generalmente, además de llamar a *yyparse* se ocupará de inicializar, abrir ficheros, procesar los argumentos de la línea de comandos, etc. antes de la llamada a *yyparse*, y de cerrar ficheros, comunicar si el análisis ha resultado correcto o no, etc. después ella.

Se puede llamar a *yyparse* directamente desde *main* o bien desde alguna otra función, que a su vez sea llamada por *main*.

6.14.2 La función analizador léxico *yylex*

Dentro de un compilador, el analizador sintáctico se encarga de agrupar tokens en sentencias, pero no sabe tratar el fichero de entrada para obtener de él los tokens, de eso se encarga el analizador léxico. Lo que YACCOV genera es un analizador sintáctico (*yyparse*), que, como tal, no sabe leer un fichero y descomponerlo en tokens, así que necesita un analizador léxico que lo haga por él. *yyparse* supone que éste existe y que es una función de nombre *yylex* a la que llama cada vez que necesita un nuevo token, por lo tanto el programador debe suministrar una función con ese nombre que le pase los tokens del fichero de entrada a *yyparse*.

Cada vez que *yyparse* la llama, *yylex* le devuelve el código del token que acaba de leer o 0 si llegó al final del fichero de entrada. A todos los tokens definidos en la gramática se les asigna un código numérico mediante una *#define*, que aparecerá al principio del fichero producido por YACCOV, así que *yylex* puede referirse a ellos utilizando su nombre en lugar del número que les corresponde. Los literales de un solo carácter, aunque no tengan la *#define* correspondiente, pueden ser usados de la misma forma, ya que el número que YACCOV les asigna es el que les corresponde según el código ASCII, excepto el carácter nulo, ya que su código numérico es 0 y ese es el valor que se utiliza para indicar el final de la entrada (YACCOV también toma como *fin de fichero* cualquier valor negativo).

yylex al devolver el control a *yyparse*, le dice cuál es el token que acaba de leer, pero no qué valor semántico tiene. Por ejemplo, si tenemos una gramática que reconoce expresiones aritméticas, la sentencia de tres tokens: *31 + 13* hará que *yylex* le pase a *yyparse* la secuencia *NUMERO '+' NUMERO*. Esta es toda la información que el analizador sintáctico necesita para saber si la sentencia es sintácticamente correcta o no. Los valores semánticos se utilizan en la ejecución de las acciones, pero *yyparse* no se preocupa de averiguarlos ni de ver si son correctos o no. Supone que el del último token leído está almacenado en la variable *yylval*. *yylex* debe encargarse de leerlo y depositarlo en esa variable.

En resumen, lo que tiene que hacer *yylex* cada vez que *yyparse* lo llama es:

- Leer el token que viene a continuación en el fichero de entrada.
- Ver qué clase de token es.
- Leer su valor semántico (si lo tiene) y almacenarlo en *yylval*.
- Devolver el token leído.

Si el analizador léxico no va incluído en la sección de *código C adicional*, no estará en el fichero de salida de YACCOV y por lo tanto no podrá utilizar las *#define* generadas por los tokens, pero si al ejecutar YACCOV se especifica la opción '-d', éstas, además de ir en el de *yyparse*, serán escritas en un fichero de cabecera, que podrá ser incluido en el fichero de *yylex*.

Se puede tomar la salida de la utilidad LEX (generador de analizadores léxicos incorporado al sistema operativo UNIX) o de programas similares, compatibles con ella, como analizador léxico para *yyparse* sin necesidad de hacer ningún cambio en ella.

6.14.3 La función *yyerror*

La principal función de esta rutina es informar al programador de que se ha producido un error, imprimiendo un mensaje, que *yyparse* le pasará como argumento al llamarla (la llamará cada vez que encuentre un error). En su forma más simple será:

```
void yyerror(s)
char *s;
{
   printf("%s\n", s);
}
```

aunque generalmente será algo más complicada.

6.15 Ficheros generados por *yaccov*

YACCOV puede generar hasta tres ficheros: el de salida, que contiene la función *yyparse*, el que contiene información acerca del funcionamiento del analizador generado, que se produce al utilizar la opción -v al ejecutar YACCOV, y el de cabecera, que contiene las *#define* de los tokens y que se produce si se utiliza la opción -d al ejecutar YACCOV. A continuación estudiaremos cada uno de ellos.

6.15.1 El fichero de salida

El fichero que se produce como salida de YACCOV contiene un analizador LR, que como tal consta de las siguientes partes:

- *Programa analizador*. Es la estructura que se utiliza para construir el fichero de salida y será el contenido del fichero *SIMPLE.PRS* a menos que aparezca la declaración *%semantic_parser* en la sección declaraciones (en este caso se utilizará el fichero *SEM.PRS*) o que se utilice la opción *-p* indicando el nombre del fichero que contiene la estructura que se quiere utilizar. En él se insertan las acciones de las reglas de la gramática en donde aparece el símbolo *$*.
- *Tabla de análisis*. En cierto modo se podría decir que la tabla de análisis es *yytabla*, que consta de dos partes, una a la que se accede por medio de *yypacc* (*accion*) y otra a la que se llega mediante *yypgoto* (*goto*). Pero en realidad, todas las tablas que aparecen al principio del fichero de salida forman parte de la tabla de análisis, ya que todas ellas se utilizan de una forma u otra para acceder a *yytabla*.
- *Pila de analizador*. Todo analizador LR, utiliza una pila en donde va almacenando los símbolos de la entrada que lee y los estados por los que pasa. La pila de *yyparse* consta en realidad de tres "subpilas" paralelas, una de ellas de tipo YYSTYPE (definido en la sección de declaraciones) almacena los símbolos de la gramática (es *yyvsa*), otra, de tipo *short*, almacena los números de los estados por los que va pasando el analizador (*yyssa*) y la otra almacena por cada símbolo la información necesaria para localizarlo con las construcciones "@N" (*yylsa*).

La entrada será lo que *yylex* le vaya pasando y la salida el análisis sintáctico de la entrada.

Al final del analizador generado se incluye todo lo que aparecía en la sección de código C adicional en el fichero de entrada a YACCOV.

6.15.2 Funcionamiento de *yyparse*

La función de *yyparse* es ir agrupando los tokens que le pasa el analizador léxico *yylex* de acuerdo con las reglas de producción de la gramática y ejecutar las acciones correspondientes a las reglas que utiliza. Si la entrada es válida, la secuencia de tokens completa se reduce a un solo símbolo (que será el símbolo inicial de la gramática). Finaliza su ejecución cuando encuentre el final de la entrada o un error sintáctico del que no se pueda recuperar. Devuelve el valor 0 si no hubo problemas en el análisis y 1 si se produjo algún error.

Dentro de las acciones que acompañan a las reglas gramaticales se pueden utilizar dos macros que hacen que el análisis finalice inmediatamente. Son:

YYACCEPT .- Devuelve el control y el valor 0 (se ha completado el análisis con éxito).

YYABORT .- Devuelve el control y el valor 1 (se ha producido un error).

El valor semántico del último token estará almacenado en la variable *yylval* (la función *yylex* debe encargarse de dejarlo allí). Esta es una variable de tipo YYSTYPE, que es el que utilizan los elementos de la pila de *yyparse*. Si todos los valores semánticos de los símbolos de la gramática son del mismo tipo, *yylval* también lo será, pero si se utilizan múltiples tipos de datos, el tipo de *yylval* será una unión construida a partir de la declaración *%union*. En este caso, al almacenar el valor de un token deberá usarse el campo adecuado de la unión. Si se utilizan construcciones "@N" en las acciones, la función *yyparse* espera encontrar la localización del token que acaba de ser analizado en la variable *yylloc*. *yylex* debe encargarse de poner en ella los valores correspondientes. Esta variable es una estructura que contiene los campos *first_line*, *first_column*, *last_line* y *last_column*, que permiten localizar a los distintos símbolos durante la ejecución de *yyparse*. El uso de estas construcciones hace que el analizador sea notablemente lento.

yyparse va cargando en la pila los tokens que le pasa *yylex*, hasta llegar a una situación en la que se pueda producir una reducción. Entonces consulta el token *lookahead* y las tablas para saber si tiene que reducir y si es así realiza la reducción correspondiente y después ejecuta la acción semántica que lleva asociada la regla por la que se redujo.

6.15.3 Recuperación de errores

Cuando un compilador analiza un programa fuente, no debe finalizar su ejecución cada vez que se encuentra un error. Lo ideal sería que informara de todos los errores contenidos en él en una sola compilación.

Normalmente cuando encuentra un error, *yyparse* llama a la rutina *yyerror* y finaliza su ejecución. Pero existen algunos mecanismos que pueden hacer que el analizador se recupere de los errores y sea capaz de seguir adelante con el proceso.

En un analizador interactivo simple en el que cada entrada sea una línea, puede ser suficiente con hacer que *yyparse* devuelva 1 si se produce un error, ignorar el resto de la línea de entrada y después llamar a *yyparse* de nuevo para que analice la siguiente. Pero para un compilador esto no es la solución más adecuada porque se pierde el contexto. Para ayudar a *yyparse* o recuperarse de los errores se utiliza el token especial *error* y las macros *yyclearin* e *yyerrok*.

El token *error* es un símbolo terminal especial reservado para el manejo de errores. No es necesario declararlo, YACCOV lo hace automáticamente. Para explicar su utilización consideremos:

```
sent :    /* vacío */      [1]
      |  sent '\n'         [2]
      |  sent exp '\n'     [3]
      |  sent error '\n'   [4]
      ;
```

La regla [4] dice que puede aparecer un error seguido de una sentencia (*sent*) y antes de un carácter nueva línea. De esta forma, si se produce un error en medio de una expresión (*exp*), se puede hacer una reducción al token *error* y después reducir utilizando la regla [4]. Estas reducciones no son exactamente iguales a las que se producen normalmente en el analizador, hay que hacer que la situación se ajuste a la regla. En primer lugar habrá que eliminar estados y símbolos de la pila, retrocediendo hasta un lugar en el que el token *error* sea aceptable cargar este token en la pila. Después se salta todo lo que aparece en la entrada hasta llegar algo que se ajuste a lo que viene tras *error* en la regla ('\n', en este caso), que será cargado en la pila para posteriormente reducir por la regla que contiene *error*.

Es aconsejable poner algún token detrás de *error* en la regla, para que el analizador pueda saber exactamente qué parte de la entrada tiene que saltar para volver a analizar normalmente. También es útil, cuando se produce un error, tener en cuenta las parejas de paréntesis, llaves, corchetes, etc., porque podría suceder que el de apertura fuera reconocido antes de producirse el error y el de cierre no, lo que podría producir un mensaje de un error que no existe en realidad. Si en la gramática aparece:

```
sent :   '(' exp ')'
      |  '(' error ')'
      ;
```

si se produjera un error dentro de un paréntesis, se saltaría todo el texto de entrada que viniera a continuación hasta llegar al próximo ')'.

Estas estrategias de recuperación de errores se basan en suposiciones, el programador las utilizará en aquellos sitios en que "supone" que se pueden producir errores. Si luego las cosas no sucedieran de esta forma, un error podría llevar a otro. En estos casos, para evitar que se produzcan demasiados mensajes de error, que podrían confundir más que aclarar las cosas, el analizador no produce ningún mensaje más hasta que haya conseguido cargar con éxito tres tokens seguidos.

La macro *yyerrok* informa al analizador de que el error ha sido completamente superado y hace que vuelvan a salir mensajes de error, aunque aún no se hayan cargado tres tokens después del último que se produjo. Tanto ésta como *yyclearin* son sentencias C válidas que pueden ser utilizadas en las acciones de las reglas de producción. Por ejemplo, en el fichero de entrada a YACCOV podría aparecer:

```
sent :   '(' exp ')'
      |  '(' error ')'    { yyerrok;  }
      ;
```

Después de la recuperación de un error, el *lookahead* que teníamos en el momento en que éste se produjo, es reanalizado. Si ya no es válido, se puede utilizar la macro *yyclearin*, que se ocupa de que sea desechado.

6.15.4 El fichero de cabecera

Es el fichero que se produce con la opción *-d* o *-D*. Contiene todas las definiciones de los tipos y macros que utiliza *yyparse* además de las de los tokens de la gramática.

Por ejemplo, si ejecutáramos YACCOV utilizando la opción *-d* con el siguiente fichero de entrada:

```
%{
#define YYSTYPE double
#include <math.h>
%}
```

```
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG
%right '^'
/* gramática */
%%
input:   /* vacio */
       | input linea
       ;
linea:    '\n'
       | exp '\n'   {  printf("\t%.10g\n", $1);  }
       | error '\n' {  yyerrok;                  }
       ;
exp:     NUM                {  $$ = $1;            }
       | exp '+' exp        {  $$ = $1 + $3;       }
       | exp '-' exp        {  $$ = $1 - $3;       }
       | exp '*' exp        {  $$ = $1 * $3;       }
       | exp '/' exp        {  $$ = $1 / $3;       }
       | '-' exp %prec NEG  {  $$ = -$2;           }
       | exp '^' exp        {  $$ = pow ($1, $3);  }
       | '(' exp ')'        {  $$ = $2;            }
       ;
%%

/* código c adicional  */
/*  analizador léxico  */
#include <ctype.h>
yylex ()
{
  int c;
  while ((c = getchar()) == ' ' || c == '\t')  ;
  if (c == '.' || isdigit (c))
  {
    ungetc(c, stdin);
    scanf ("%lf", &yylval);
    return NUM;
  }
  if (c == EOF)  return 0;
  return c;
}
/*  principal  */
main()
{
  yyparse();
}
/*  subrutina de error  */
#include <stdio.h>
yyerror (s)
   char  *s;
{
  printf ("%s\n", s);
}
```

obtendríamos el fichero de cabecera:

```
#ifndef YYLTYPE
typedef
  struct yyltype
    {
      int timestamp;
      int first_line;
      int first_column;
      int last_line;
      int last_column;
      char *text;
   }
  yyltype;
#define YYLTYPE yyltype
#endif
#define YYACCEPT   return(0)
#define YYABORT    return(1)
#define YYERROR    goto yyerrlab
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define NUM   258
#define NEG   259
```

En primer lugar define el tipo YYLTYPE que es el que utilizan las construcciones @*N*. Después define las tres macros YYACCEPT, YYABORT e YYERROR, el tipo de la pila de símbolos (YYSTYPE) si aún no había sido definido (por defecto su tipo es *int*) y por último las *#define* correspondientes a los tokens que no son literales de un solo carácter (en este caso *NUM* y *NEG*), asignándoles la posición que ocupan en la tabla *yytraduce*, es decir, su código externo.

6.15.5 Depuración en tiempo de ejecución

Los analizadores sintácticos generados por *yacc* se pueden depurar en tiempo de ejecución por medio de la declaración de la macro *YYDEBUG* o por medio del uso de la opción *-t* en las opciones de llamada a *yacc* (ver apartado 6.9.1).

```
%{
#define YYDEBUG 1
%}
```

También se puede utilizar la variable entera *yydebug* con valor 1 antes de que el código llame a `yyparse()`.

Estas opciones de depuración imprimirán las acciones que se van realizando. Puede que en la mayor parte de las veces genere una información excesivamente amplia. Sin embargo en muchos casos es el único camino para detectar errores recalcitrantes.

6.15.6 El fichero de conflictos y estados

Al ejecutar YACCOV con la opción *-v* o *-V* se produce un fichero que contiene información acerca de los conflictos producidos y de los estados del analizador generado. Para el ejemplo visto en el apartado anterior sería:

```
CONFLICTOS RESUELTOS:
En el estado, 10 entre la regla 11 y el token '-'. Resuelto como reducir.
En el estado, 10 entre la regla 11 y el token '+'. Resuelto como reducir.
En el estado, 10 entre la regla 11 y el token '*'. Resuelto como reducir.
En el estado, 10 entre la regla 11 y el token '/'. Resuelto como reducir.
En el estado, 10 entre la regla 11 y el token '^'. Resuelto como cargar.
En el estado, 19 entre la regla 8 y el token '-'. Resuelto como reducir.
En el estado, 19 entre la regla 8 y el token '+'. Resuelto como reducir.
En el estado, 19 entre la regla 8 y el token '*'. Resuelto como cargar.
En el estado, 19 entre la regla 8 y el token '/'. Resuelto como cargar.
En el estado, 19 entre la regla 8 y el token '^'. Resuelto como cargar.
En el estado, 20 entre la regla 7 y el token '-'. Resuelto como reducir.
En el estado, 20 entre la regla 7 y el token '+'. Resuelto como reducir.
En el estado, 20 entre la regla 7 y el token '*'. Resuelto como cargar.
En el estado, 20 entre la regla 7 y el token '/'. Resuelto como cargar.
En el estado, 20 entre la regla 7 y el token '^'. Resuelto como cargar.
En el estado, 21 entre la regla 9 y el token '-'. Resuelto como reducir.
En el estado, 21 entre la regla 9 y el token '+'. Resuelto como reducir.
En el estado, 21 entre la regla 9 y el token '*'. Resuelto como reducir.
En el estado, 21 entre la regla 9 y el token '/'. Resuelto como reducir.
En el estado, 21 entre la regla 9 y el token '^'. Resuelto como cargar.
En el estado, 22 entre la regla 10 y el token '-'. Resuelto como reducir.
En el estado, 22 entre la regla 10 y el token '+'. Resuelto como reducir.
```

```
En el estado, 22 entre la regla 10 y el token '*'. Resuelto como reducir.
En el estado, 22 entre la regla 10 y el token '/'. Resuelto como reducir.
En el estado, 22 entre la regla 10 y el token '^'. Resuelto como cargar.
En el estado, 23 entre la regla 12 y el token '-'. Resuelto como reducir.
En el estado, 23 entre la regla 12 y el token '+'. Resuelto como reducir.
En el estado, 23 entre la regla 12 y el token '*'. Resuelto como reducir.
En el estado, 23 entre la regla 12 y el token '/'. Resuelto como reducir.
En el estado, 23 entre la regla 12 y el token '^'. Resuelto como cargar.



TIPOS DE TOKENS:
numero -1 -> $
numero 10 -> '\n'
numero 40 -> '('
numero 41 -> ')'
numero 42 -> '*'
numero 43 -> '+'
numero 45 -> '-'
numero 47 -> '/'
numero 94 -> '^'
numero 256 -> error
numero 258 -> NUM
numero 259 -> NEG



----------------------------   ESTADO 0   ----------------------------
    input        cargar y pasar al estado  1
    $default     reducir por  1  (input)


----------------------------   ESTADO 1   ----------------------------
    input  ->  input . linea   (2)
    $            cargar y pasar al estado  24
    error        cargar y pasar al estado  2
    NUM          cargar y pasar al estado  3
    '-'          cargar y pasar al estado  4
    '\n'         cargar y pasar al estado  5
    '('          cargar y pasar al estado  6
    linea        cargar y pasar al estado  7
    exp          cargar y pasar al estado  8


----------------------------   ESTADO 2   ----------------------------
    linea  ->  error . '\n'   (5)
    '\n'         cargar y pasar al estado  9


----------------------------   ESTADO 3   ----------------------------
    exp  ->  NUM .   (6)
    $default     reducir por  6  (exp)


----------------------------   ESTADO 4   ----------------------------
    exp  ->  '-' . exp   (11)
    NUM          cargar y pasar al estado  3
    '-'          cargar y pasar al estado  4
    '('          cargar y pasar al estado  6
    exp          cargar y pasar al estado  10


----------------------------   ESTADO 5   ----------------------------
    linea  ->  '\n' .   (3)
    $default     reducir por  3  (linea)
```

```
---------------------------   ESTADO 6   ---------------------------
    exp  ->  '(' . exp ')'   (13)
    NUM          cargar y pasar al estado  3
    '-'          cargar y pasar al estado  4
    '('          cargar y pasar al estado  6
    exp          cargar y pasar al estado  11


---------------------------   ESTADO 7   ---------------------------
    input  ->  input linea .   (2)
    $default     reducir por  2  (input)


---------------------------   ESTADO 8   ---------------------------
    linea  ->  exp . '\n'   (4)
    exp  ->  exp . '+' exp   (7)
    exp  ->  exp . '-' exp   (8)
    exp  ->  exp . '*' exp   (9)
    exp  ->  exp . '/' exp   (10)
    exp  ->  exp . '^' exp   (12)
    '-'          cargar y pasar al estado  12
    '+'          cargar y pasar al estado  13
    '*'          cargar y pasar al estado  14
    '/'          cargar y pasar al estado  15
    '^'          cargar y pasar al estado  16
    '\n'         cargar y pasar al estado  17


---------------------------   ESTADO 9   ---------------------------
    linea  ->  error '\n' .   (5)
    $default     reducir por  5  (linea)


---------------------------   ESTADO 10   ---------------------------
    exp  ->  exp . '+' exp   (7)
    exp  ->  exp . '-' exp   (8)
    exp  ->  exp . '*' exp   (9)
    exp  ->  exp . '/' exp   (10)
    exp  ->  '-' exp .   (11)
    exp  ->  exp . '^' exp   (12)
    '^'          cargar y pasar al estado  16
    $default     reducir por  11  (exp)


---------------------------   ESTADO 11   ---------------------------
    exp  ->  exp . '+' exp   (7)
    exp  ->  exp . '-' exp   (8)
    exp  ->  exp . '*' exp   (9)
    exp  ->  exp . '/' exp   (10)
    exp  ->  exp . '^' exp   (12)
    exp  ->  '(' exp . ')'   (13)
    '-'          cargar y pasar al estado  12
    '+'          cargar y pasar al estado  13
    '*'          cargar y pasar al estado  14
    '/'          cargar y pasar al estado  15
    '^'          cargar y pasar al estado  16
    ')'          cargar y pasar al estado  18


---------------------------   ESTADO 12   ---------------------------
    exp  ->  exp '-' . exp   (8)
    NUM          cargar y pasar al estado  3
    '-'          cargar y pasar al estado  4
    '('          cargar y pasar al estado  6
    exp          cargar y pasar al estado  19
```

```
---------------------------   ESTADO 13   ---------------------------
    exp  ->  exp '+' . exp   (7)
    NUM          cargar y pasar al estado  3
    '-'          cargar y pasar al estado  4
    '('          cargar y pasar al estado  6
    exp          cargar y pasar al estado  20


---------------------------   ESTADO 14   ---------------------------
    exp  ->  exp '*' . exp   (9)
    NUM          cargar y pasar al estado  3
    '-'          cargar y pasar al estado  4
    '('          cargar y pasar al estado  6
    exp          cargar y pasar al estado  21


---------------------------   ESTADO 15   ---------------------------
    exp  ->  exp '/' . exp   (10)
    NUM          cargar y pasar al estado  3
    '-'          cargar y pasar al estado  4
    '('          cargar y pasar al estado  6
    exp          cargar y pasar al estado  22


---------------------------   ESTADO 16   ---------------------------
    exp  ->  exp '^' . exp   (12)
    NUM          cargar y pasar al estado  3
    '-'          cargar y pasar al estado  4
    '('          cargar y pasar al estado  6
    exp          cargar y pasar al estado  23


---------------------------   ESTADO 17   ---------------------------
    linea  ->  exp '\n' .   (4)
    $default     reducir por  4  (linea)


---------------------------   ESTADO 18   ---------------------------
    exp  ->  '(' exp ')' .   (13)
    $default     reducir por  13  (exp)


---------------------------   ESTADO 19   ---------------------------
    exp  ->  exp . '+' exp   (7)
    exp  ->  exp . '-' exp   (8)
    exp  ->  exp '-' exp .   (8)
    exp  ->  exp . '*' exp   (9)
    exp  ->  exp . '/' exp   (10)
    exp  ->  exp . '^' exp   (12)
    '*'          cargar y pasar al estado  14
    '/'          cargar y pasar al estado  15
    '^'          cargar y pasar al estado  16
    $default     reducir por  8  (exp)


---------------------------   ESTADO 20   ---------------------------
    exp  ->  exp . '+' exp   (7)
    exp  ->  exp '+' exp .   (7)
    exp  ->  exp . '-' exp   (8)
    exp  ->  exp . '*' exp   (9)
    exp  ->  exp . '/' exp   (10)
    exp  ->  exp . '^' exp   (12)
```

```
    '*'          cargar y pasar al estado  14
    '/'          cargar y pasar al estado  15
    '^'          cargar y pasar al estado  16

    $default     reducir por  7  (exp)


---------------------------   ESTADO 21   ---------------------------

    exp  ->  exp . '+' exp   (7)
    exp  ->  exp . '-' exp   (8)
    exp  ->  exp . '*' exp   (9)
    exp  ->  exp '*' exp .   (9)
    exp  ->  exp . '/' exp   (10)
    exp  ->  exp . '^' exp   (12)

    '^'          cargar y pasar al estado  16

    $default     reducir por  9  (exp)


---------------------------   ESTADO 22   ---------------------------

    exp  ->  exp . '+' exp   (7)
    exp  ->  exp . '-' exp   (8)
    exp  ->  exp . '*' exp   (9)
    exp  ->  exp . '/' exp   (10)
    exp  ->  exp '/' exp .   (10)
    exp  ->  exp . '^' exp   (12)

    '^'          cargar y pasar al estado  16

    $default     reducir por  10  (exp)


---------------------------   ESTADO 23   ---------------------------

    exp  ->  exp . '+' exp   (7)
    exp  ->  exp . '-' exp   (8)
    exp  ->  exp . '*' exp   (9)
    exp  ->  exp . '/' exp   (10)
    exp  ->  exp . '^' exp   (12)
    exp  ->  exp '^' exp .   (12)

    '^'          cargar y pasar al estado  16

    $default     reducir por  12  (exp)


---------------------------   ESTADO 24   ---------------------------

    $            cargar y pasar al estado  25


---------------------------   ESTADO 25   ---------------------------

    NO HAY ACCIONES
```

En primer lugar saca la lista de conflictos shift/reduce que aparecieron en la gramática y fueron resueltos utilizando las precedencias y asociatividades de los tokens y las reglas y la de los que no pudieron resolverse, indicando el número del estado en el que aparecieron. A continuación aparece una lista se relacionan los códigos externos de los tokens con sus nombres. El token *$* lo define YACCOV automáticamente y se utiliza para representar el final de la entrada.

Después saca información acerca de todos los estados del analizador generado. Dentro del apartado dedicado a cada estado aparecen en primer lugar los items que lo forman (cada estado es un conjunto de items) indicando entre paréntesis el número de la regla de la que proceden, seguido de las acciones a tomar para cada *lookahead* posible en ese estado y por último la reducción por defecto si la hay. Si en un estado se produjeron conflictos y no fueron resueltos, se elegirá cargar si el conflicto es shift/reduce o reducir por una de todas las reglas posibles si es reduce/reduce. Todas las demás reducciones (que nunca se llevarán a cabo porque se escogió o bien cargar o reducir utilizando otra regla) aparecerán entre corchetes. En este caso no hay ninguna porque no quedó ningún conflicto sin resolver.

6.16 Diferencias entre YACCOV y PCYACC[3]

Existen algunas diferencias entre YACCOV y PCYACC. Entre ellas se podría destacar:

1. PCYACC no admite las declaraciones *%expect*, *%semantic_parser* y *%pure_parser* en la sección de declaraciones.
2. En la línea de comandos que invoca a PCYACC las opciones tienen que aparecer siempre antes del nombre del fichero de entrada, mientras que al llamar a YACCOV el orden de estos elementos no tiene importancia.
3. En el fichero producido por PCYACC todo lo que aparece en la sección de código C adicional se coloca antes de la función *yyparse*, mientras que en el producido por YACCOV va detrás, exactamente al final del fichero.

6.17 Desarrollo de una minicalculadora con *yacc*

Vamos a desarrollar una pequeña aplicación utilizando YACCOV. Será un programa que analice y ejecute expresiones aritméticas, está basado en el HOC de *Kernighan y Pike* [KERN84, capítulo 8]. Se va ha desarrollar en varias etapas.

6.17.1 Calculadora elemental

Lo primero que hay que hacer es definir la gramática, que tiene que ser libre de contexto. Se analizará por separado cada línea y se ejecutará antes de pasar a la siguiente, y cada línea estará compuesta por expresiones relacionadas entre sí seguidas de un carácter nueva línea, por lo tanto podemos escribir las siguientes reglas de producción:

```
input ::=   /* vacio */
          | input linea
          ;
linea ::=    '\n'
          | exp '\n'
          | error '\n'
          ;
exp  ::=   NUM
        | exp '+' exp
        | exp '-' exp
        | exp '*' exp
        | exp '/' exp
        | '-' exp
        | exp '^' exp   /* para exponentes */
        | '(' exp ')'
        ;
```

Ahora habrá que añadir la sección de declaraciones en donde se define el tipo de la pila de valores semánticos como double y los tokens que aparecen en la gramática indicando su precedencia y asociatividad. Será:

```
%{
#define YYSTYPE double
#include <math.h>
%}
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG  /* Para darle precedencia al menos unario. */
%right '^'
```

Añadiéndole a cada regla las acciones que deben ejecutarse cada vez que se reduce utilizándolas:

```
%%
input:   /* vacio */
       | input linea
       ;
linea:    '\n'
       | exp '\n'   {  printf("\t%.10g\n", $1);  }
       | error '\n' {  yyerrok;                  }
       ;
```

3 - PCYACC es una marca registrada por ABRAXAS Inc.

```
exp:     NUM                  {   $$ = $1;             }
       | exp '+' exp          {   $$ = $1 + $3;        }
       | exp '-' exp          {   $$ = $1 - $3;        }
       | exp '*' exp          {   $$ = $1 * $3;        }
       | exp '/' exp          {   $$ = $1 / $3;        }
       | '-' exp %prec NEG    {   $$ = -$2;            }
       | exp '^' exp          {   $$ = pow ($1, $3);   }
       | '(' exp ')'          {   $$ = $2;             }
       ;
%%
```

Con *%prec NEG* se indica que la regla que lo lleva debe tener la precedencia que se le ha asignado al token *NEG* en la sección de declaraciones, que es mayor que la de sumas, restas, productos y divisiones.

Por último habrá que añadir el código C adicional, que en este caso consistirá en un sencillísimo analizador léxico que reconoce números y operadores aritméticos, una rutina de errores que saca un mensaje de error en el caso de que se produzca alguno, y una función *main* que se ocupa de llamar a *yyparse*:

```
/*  analizador léxico  */
#include <ctype.h>
yylex ()
{
  int c;
  while ((c = getchar()) == ' ' || c == '\t')  ;
  if (c == '.' || isdigit (c))
  {
    ungetc(c, stdin);
    scanf ("%lf", &yylval);
    return NUM;
  }
  if (c == EOF)  return 0;
  return c;
}


/*  principal  */
main()
{
  yyparse();
}


/*  subrutina de error  */
#include <stdio.h>
yyerror (s)
   char  *s;
{
  printf ("%s\n", s);
}
```

El fichero que se obtiene como resultado es el que aparece a continuación, que una vez compilado podrá ser utilizado para calcular expresiones aritméticas. Es:

```
/*  analizador generado a partir de calculad.y  */
#define NUM 258
#define NEG 259
#line 1 "calculad.y"
#define YYSTYPE double
#include <math.h>
#ifndef YYLTYPE
typedef
  struct yyltype
    {
      int timestamp;
      int first_line;
      int first_column;
      int last_line;
```

```
      int last_column;
      char *text;
   }
  yyltype;
#define YYLTYPE yyltype
#endif
#define YYACCEPT return(0)
#define YYABORT return(1)
#define YYERROR goto yyerrlab
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#include <stdio.h>
#ifndef __STDC__
#define const
#endif


#define YYFINAL   25
#define YYFLAG    -32767
#define YYNTBASE  13
#define YYTRADUCE(x) ((unsigned)(x) <= 259 ? yytraduce[x] : 16)
static const char yytraduce[] = {     0,
   2,   2,   2,   2,   2,   2,   2,   2,   2,  10,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,  11,
  12,   6,   5,   2,   4,   2,   7,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   9,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   1,   2,   3,   8
};
static const short yyrlinea[] = {     0,
  14,  15,  18,  19,  20,  23,  24,  25,  26,  27,
  28,  29,  30
};
static const char * const yytnom[] = {     0,
"error","$ilegal.","NUM","'-'","'+'","'*'","'/'","NEG","'^'","'\\n'",
"'('","')'","input"
};
static const short yyr1[] = {     0,
  13,  13,  14,  14,  14,  15,  15,  15,  15,  15,
  15,  15,  15
};
static const short yyr2[] = {     0,
   0,   2,   1,   2,   2,   1,   3,   3,   3,   3,
   2,   3,   3
};
```

```
static const short yydefacc[] = {   1,
   0,   0,   6,   0,   3,   0,   2,   0,   5,  11,
   0,   0,   0,   0,   0,   0,   4,  13,   8,   7,
   9,  10,  12,   0,   0
};
static const short yydefgoto[] = {     1,
     7,     8
};
static const short yypacc[] = {-32767,
     3,    -9,-32767,    18,-32767,    18,-32767,    21,-32767,    10,
    11,    18,    18,    18,    18,    18,-32767,-32767,    26,    26,
    10,    10,    10,     5,-32767
};
static const short yypgoto[] = {-32767,
-32767,    -4
};

#define YYLAST  35

static const short yytabla[] = {  10,
   9,  11,  24,   2,  25,   3,   4,  19,  20,  21,
  22,  23,   5,   6,  12,  13,  14,  15,  16,  16,
   3,   4,  18,   0,  12,  13,  14,  15,   6,  16,
  17,  14,  15,   0,  16
};
static const short yycheq[] = {   4,
  10,   6,   0,   1,   0,   3,   4,  12,  13,  14,
  15,  16,  10,  11,   4,   5,   6,   7,   9,   9,
   3,   4,  12,  -1,   4,   5,   6,   7,  11,   9,
  10,   6,   7,  -1,   9
};
#define YYPURE 1
#line 2 "simple.prs"
/* Estructura para el analizador generado por YACCOV */
#define yyerrok    (yyerrstatus = 0)
#define yyclearin  (yychar = YYEMPTY)
#define YYEMPTY    -2
#define YYEOF      0
#define YYFAIL     goto yyerrlab;
#define YYTERROR   1
#ifndef YYIMPURE
extern int yylex( void) ;
#define YYLEX  yylex()
#endif
#ifndef YYPURE
extern int yylex( YYLTYPE *, YYLTYPE *) ;
#define YYLEX  yylex(&yylval, &yylloc)
#endif

#ifndef YYIMPURE
int  yychar;                    /*  contiene el token lookahead         */
YYSTYPE yylval;                 /*  contiene el valor semántico del     */
                                /*  token lookahead.                    */
YYLTYPE yylloc;                 /*  contiene la localización del token  */
                                /*  lookahead.                          */
int yynerr;                     /*  número de errores hasta ahora.      */
#endif  /* YYIMPURE */

/*  YYMAXPROF indica el tamaño inicial de las pilas del analizador.    */
#ifndef YYMAXPROF
#define YYMAXPROF 200
#endif

/*  YYMAXLIMIT es el tamaño máximo que pueden alcanzar las pilas. */
```

```
#ifndef YYMAXLIMIT
#define YYMAXLIMIT 10000
#endif

#line 167 "simple.prs"
int yyparse()
{
  register int yystate;
  register int yyn;
  register short *yyssp;
  register YYSTYPE *yyvsp;
  YYLTYPE *yylsp;
  int yyerrstatus;  /* número de tokens a cargar antes de volver a */
                    /* emitir mensajes de error.                   */
  int yychar1;      /* contiene el código interno del lookahead.   */

  short yyssa[YYMAXPROF];       /*  pila de estados                    */
  YYSTYPE yyvsa[YYMAXPROF];     /*  pila de valores semánticos         */
  YYLTYPE yylsa[YYMAXPROF];     /*  pila de localización               */

  short *yyss = yyssa;
  YYSTYPE *yyvs = yyvsa;
  YYLTYPE *yyls = yylsa;

  int yymaxprof = YYMAXPROF;

#ifndef YYPURE
  int yychar;
  YYSTYPE yylval;
  YYLTYPE yylloc;
#endif

  YYSTYPE yyval;                /*  devuelve valores semánticos de las */
                                /*  acciones.                          */

  int yylen;

#ifdef YYDEBUG
    fprintf(stderr, "Empieza el análisis\n");
#endif

  yystate = 0;
  yyerrstatus = 0;
  yynerr = 0;
  yychar = YYEMPTY;/* Hace que se lea un token.  */

  /* Inicializar los punteros a las pilas. */

  yyssp = yyss - 1;
  yyvsp = yyvs;
  yylsp = yyls;

/* Cargar un nuevo  estado, que se encuentra en yystate. Siempre que se */
/* llega aquí las pilas de valores y localización acaban de recibir una */
/* carga por lo que al cargar un estado, quedan niveladas.              */

yynewstate:

  *++yyssp = yystate;

  if (yyssp >= yyss + yymaxprof - 1)
    {
      /* Darle al usuario la oportunidad de relocalizar la pila. */

      YYSTYPE *yyvs1 = yyvs;
      YYLTYPE *yyls1 = yyls;
      short *yyss1 = yyss;

      int size = yyssp - yyss + 1;

#ifdef yyoverflow
      yyoverflow("overflow",
        &yyss1, size * sizeof (*yyssp),
        &yyvs1, size * sizeof (*yyvsp),
        &yyls1, size * sizeof (*yylsp),
        &yymaxprof);
```

```
      yyss = yyss1; yyvs = yyvs1; yyls = yyls1;
#else
      if (yymaxprof >= YYMAXLIMIT)
         yyerror("overflow");
      yymaxprof *= 2;
      if (yymaxprof > YYMAXLIMIT)
         yymaxprof = YYMAXLIMIT;
      yyss = (short *) malloc (yymaxprof * sizeof (*yyssp));
      memcpy ((char *)yyss, (char *)yyss1, size * sizeof (*yyssp));
      yyvs = (YYSTYPE *) malloc (yymaxprof * sizeof (*yyvsp));
      memcpy ((char *)yyvs, (char *)yyvs1, size * sizeof (*yyvsp));
#ifdef YYLSP_NEEDED
      yyls = (YYLTYPE *) malloc (yymaxprof * sizeof (*yylsp));
      memcpy ((char *)yyls, (char *)yyls1, size * sizeof (*yylsp));
#endif
#endif /* yyoverflow */

      yyssp = yyss + size - 1;
      yyvsp = yyvs + size - 1;
#ifdef YYLSP_NEEDED
      yylsp = yyls + size - 1;
#endif

#ifdef YYDEBUG
      fprintf(stderr, "Se ha aumentado la pila hasta %d\n", yymaxprof);
#endif

      if (yyssp >= yyss + yymaxprof - 1)
         YYABORT;
    }

#ifdef YYDEBUG
    fprintf(stderr, "Estado %d\n", yystate);
#endif

/* Ejecutar las acciones correspondientes para el estado actual. Leer */
/* un lookahead si es necesario.                                      */

yyresume:

  /* Antes de nada, intentar decidir qué hacer sin utilizar el lookahead */

  yyn = yypacc[yystate];
  if (yyn == YYFLAG)
    goto yydefault;

  /* No se puede => leer un lookahead si aún no lo tenemos.  */

  /* yychar es YYEMPTY, YYEOF o un token válido en forma externa. */

  if (yychar == YYEMPTY)
    {
#ifdef YYDEBUG
      fprintf(stderr, "Leer un token: ");
#endif
      yychar = YYLEX;
    }

  /* Poner en yychar1 el código interno del token para utilizarlo como */
  /* índice en las tablas.                                             */

  if (yychar <= 0)      /* Fin de la entrada. */
    {
      yychar1 = 0;
      yychar = YYEOF;

#ifdef YYDEBUG
       fprintf(stderr, "Final de la entrada.\n");
#endif
    }
  else
    {
      yychar1 = YYTRADUCE(yychar);

#ifdef YYDEBUG
     fprintf(stderr, "El siguiente token es %d (%s)\n", yychar,
         yytnom[yychar1]);
#endif
    }

  yyn += yychar1;
  if (yyn < 0 || yyn > YYLAST || yycheq[yyn] != yychar1)
    goto yydefault;
```

```
  yyn = yytabla[yyn];
  /* yyn indica qué hacer para este token en este estado. */
  /*    Negativo => reducir por la regla -yyn             */
  /*    Positivo => cargar y pasar al estado yyn          */
  /*    0, o mínimo número negativo => error.             */
  if (yyn < 0)
    {
      if (yyn == YYFLAG)
         goto yyerrlab;
      yyn = -yyn;
      goto yyreduce;
    }
  else if (yyn == 0)
    goto yyerrlab;
  if (yyn == YYFINAL)
    YYACCEPT;
  /* Cargar el token lookahead.  */
#ifdef YYDEBUG
    fprintf(stderr, "Cargar el token %d (%s), ", yychar, yytnom[yychar1]);
#endif

  /* Deshacerse del token que ha sido cargado a menos que sea eof. */
  if (yychar != YYEOF)
    yychar = YYEMPTY;
  *++yyvsp = yylval;
#ifdef YYLSP_NEEDED
  *++yylsp = yylloc;
#endif
  /* Contar los tokens que han sido cargados desde el último error. */
  if (yyerrstatus) yyerrstatus--;
  yystate = yyn;
  goto yynewstate;

/* Ejecutar la acción por defecto para este estado. */
yydefault:
  yyn = yydefacc[yystate];
  if (yyn == 0)
    goto yyerrlab;

/* Reducir por la regla yyn.  */
yyreduce:
  yylen = yyr2[yyn];
  yyval = yyvsp[1-yylen];
#ifdef YYDEBUG
      if (yylen == 1)
         fprintf (stderr, "Reducir 1 valor via linea %d, ",
             yyrlinea[yyn]);
      else
         fprintf (stderr, "Reducir %d valores via linea %d, ",
            yylen, yyrlinea[yyn]);
#endif


   /* Copiar aquí el fichero de acciones sustituyendo a '$'. */
  switch (yyn) {
case 4:
#line 19 "calculad.y"
{  printf("\t%.10g\n", yyvsp[-1]);      break;}
case 5:
#line 20 "calculad.y"
{  yyerrok;                      break;}
case 6:
#line 23 "calculad.y"
{  yyval = yyvsp[0];                break;}
case 7:
```

```
#line 24 "calculad.y"
{  yyval = yyvsp[-2] + yyvsp[0];           break;}
case 8:
#line 25 "calculad.y"
{  yyval = yyvsp[-2] - yyvsp[0];           break;}
case 9:
#line 26 "calculad.y"
{  yyval = yyvsp[-2] * yyvsp[0];           break;}
case 10:
#line 27 "calculad.y"
{  yyval = yyvsp[-2] / yyvsp[0];           break;}
case 11:
#line 28 "calculad.y"
{  yyval = -yyvsp[0];              break;}
case 12:
#line 29 "calculad.y"
{  yyval = pow (yyvsp[-2], yyvsp[0]);      break;}
case 13:
#line 30 "calculad.y"
{  yyval = yyvsp[-1];               break;}
}
#line 308 "simple.prs"
  yyvsp -= yylen;
  yyssp -= yylen;
#ifdef YYLSP_NEEDED
  yylsp -= yylen;
#endif
#ifdef YYDEBUG
      short *ssp1 = yyss - 1;
      fprintf (stderr, "pila de estados actual");
      while (ssp1 != yyssp)
         fprintf (stderr, " %d", *++ssp1);
      fprintf (stderr, "\n");
#endif
  *++yyvsp = yyval;
#ifdef YYLSP_NEEDED
  yylsp++;
  if (yylen == 0)
    {
      yylsp->first_line = yylloc.first_line;
      yylsp->first_column = yylloc.first_column;
      yylsp->last_line = (yylsp-1)->last_line;
      yylsp->last_column = (yylsp-1)->last_column;
      yylsp->text = 0;
    }
  else
    {
      yylsp->last_line = (yylsp+yylen-1)->last_line;
      yylsp->last_column = (yylsp+yylen-1)->last_column;
    }
#endif
  /* Ahora "cargar" el resultado de la reducción. Determinar a qué estado */
  /* se pasa a continuación.                                              */
  yyn = yyr1[yyn];
  yystate = yypgoto[yyn - YYNTBASE] + *yyssp;
  if (yystate >= 0 && yystate <= YYLAST && yycheq[yystate] == *yyssp)
    yystate = yytabla[yystate];
  else
    yystate = yydefgoto[yyn - YYNTBASE];
  goto yynewstate;
yyerrlab:   /* Detectar errores */
  if (! yyerrstatus)
    /* Informar de que se ha producido un error si no nos estamos recu- */
    /* perando de otro.                                                 */
    {
      ++yynerr;
      yyerror("error de análisis");
    }
```

```
  if (yyerrstatus == 3)
    {
      /* Si hemos intentado volver a utilizar el lookahead después del */
      /* error y no hemos podido, saltarlo.                            */
      if (yychar == YYEOF)
         YYABORT;
#ifdef YYDEBUG
     fprintf(stderr, "Desechar token %d (%s).\n", yychar, yytname[yychar1]);
#endif
      yychar = YYEMPTY;
    }
  /* Si no, intentar volver a utilizar el lookahead después de cargar el */
  /* token error.                                                        */
  yyerrstatus = 3;
  goto yyerrhandle;

yyerrpop:
  if (yyssp == yyss) YYABORT;
  yyvsp--;
  yystate = *--yyssp;
#ifdef YYLSP_NEEDED
  yylsp--;
#endif
#ifdef YYDEBUG
      short *ssp1 = yyss - 1;
      fprintf (stderr, "Error: pila de estados actual");
      while (ssp1 != yyssp)
         fprintf (stderr, " %d", *++ssp1);
      fprintf (stderr, "\n");
    }
#endif
yyerrhandle:
  yyn = yypacc[yystate];
  yyn += YYTERROR;
  yyn = yytabla[yyn];
  if (yyn < 0)
    {
      if (yyn == YYFLAG)
         goto yyerrpop;
      yyn = -yyn;
      goto yyreduce;
    }
  else if (yyn == 0)
    goto yyerrpop;
  if (yyn == YYFINAL)
    YYACCEPT;
#ifdef YYDEBUG
    fprintf(stderr, "Cargar el token error, ");
#endif
  *++yyvsp = yylval;
#ifdef YYLSP_NEEDED
  *++yylsp = yylloc;
#endif
  yystate = yyn;
  goto yynewstate;
}

#line 33 "calculad.y"


/* código c adicional  */
/*  analizador léxico  */
#include <ctype.h>
```

```
yylex ()
{
  int c;
  while ((c = getchar()) == ' ' || c == '\t')  ;
  if (c == '.' || isdigit (c))
  {
    ungetc(c, stdin);
    scanf ("%lf", &yylval);
    return NUM;
  }
  if (c == EOF)  return 0;
  return c;
}
/*  principal  */
main()
{
  yyparse();
}
/*  subrutina de error  */
#include <stdio.h>
yyerror (s)
   char  *s;
{
  printf ("%s\n", s);
}
```

6.17.2 Calculadora elemental con tratamiento de errores

Se mejora la calculadora anterior para dar tratamiento de errores, indicándose el número de línea y el nombre del programa.

```
%{
#define YYSTYPE double  /* tipo de datos de la pila de yaccov */
#include <math.h>
#include <stdio.h>
void yyerror(char*);
void aviso (char *s, char *t);
char *nombreprograma; /* nombre de programa para mensajes de error */
int nlinea=1;         /* número de línea para mensajes de error */
%}
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG  /* Para darle precedencia al menos unario. */
%right '^'
%%
input:   /* vacio */
       | input linea
       ;
linea:    '\n'
       | exp '\n'   {  printf("\t%.10g\n", $1);  }
       | error '\n' {  yyerrok;                  }
       ;
exp:     NUM               {  $$ = $1;            }
       | exp '+' exp       {  $$ = $1 + $3;       }
       | exp '-' exp       {  $$ = $1 - $3;       }
       | exp '*' exp       {  $$ = $1 * $3;       }
       | exp '/' exp       {  $$ = $1 / $3;       }
       | '-' exp %prec NEG {  $$ = -$2;           }
       | exp '^' exp       {  $$ = pow ($1, $3);  }
       | '(' exp ')'       {  $$ = $2;            }
       ;
%%
```

```
/*  analizador léxico  */
#include <ctype.h>
int yylex (void)
{
  int c;
  while ((c = getchar()) == ' ' || c == '\t')  ;
  if (c == '.' || isdigit (c))
  {
    ungetc(c, stdin);
    scanf ("%lf", &yylval);
    return NUM;
  }
  if (c == EOF)  return 0;
  return c;
}

/*  principal  */
void main(int argc, char *argv[])
{
  nombreprograma=argv[0];
  printf("\nCalculadora 1: Escriba una expresión aritmética y pulse <intro>");
  printf("\nOperadores permitidos: +,-,*,/,^,- unario, y ()");
  printf("\nNuevo tratamiento de errores");
  printf("\nPulse <control>-C (EOF) para salir\n");
  yyparse();
}

/*  subrutinas de error: se dejan preparadas para futuras versiones  */

void yyerror (char *s)
{
  aviso(s, (char *) 0 );
}
void aviso (char *s, char *t)
{
 fprintf(stderr, "%s: %s", nombreprograma, s);
 if (t) fprintf(stderr, " %s", t);
 fprintf(stderr, " cerca de la línea %d\n", nlinea);
}
```

6.17.3 Calculadora con variables

Se realiza una nueva revisión para incluir: 26 variables de la a la z, tratamiento de errores de división por cero, operador asignación (que permite asignaciones múltiples en una línea, por ejemplo x=y=z=0). También se ilustra el uso de *%union* y de *%token* con tipo.

```
%{
/* Calcula2 */
double mem[26];      /* Memoria para las variables 'a'..'z' */
#include <math.h>
#include <stdio.h>
void yyerror(char*);
void error_en_float(void);
%}
%union {                /* tipo de la pila */
        double valor;   /* valor actual */
        int indice;     /* índice de mem[] */
       }
%token <valor> NUM
%token <indice> VAR
```

```
%type  <valor> exp
%right '='
%left '-' '+'
%left '*' '/'
%left NEG  /* Para darle precedencia al menos unario. */
%right '^'
%%
input:   /* vacio */
       | input linea
       ;
linea:    '\n'
       | exp '\n'   {  printf("\t%.10g\n", $1);  }
       | error '\n' {  yyerrok;                  }
       ;
exp:     NUM                 {  $$ = $1;            }
       | VAR                 {  $$ = mem[$1];       }
       | VAR '=' exp         {  $$ = mem[$1] = $3;  }
       | exp '+' exp         {  $$ = $1 + $3;       }
       | exp '-' exp         {  $$ = $1 - $3;       }
       | exp '*' exp         {  $$ = $1 * $3;       }
       | exp '/' exp         {
                                if ($3==0.0) yyerror("División por cero");
                                $$ = $1 / $3;       }
       | '-' exp %prec NEG  {  $$ = -$2;           }
       | exp '^' exp         {  $$ = pow ($1, $3);  }
       | '(' exp ')'         {  $$ = $2;            }
       ;
%%
/*  analizador léxico  */
#include <ctype.h>
int yylex (void)
{
  int c;
  while ((c = getchar()) == ' ' || c == '\t'); /* Salta espacios en blanco */
  if (c == '.' || isdigit (c))
  {
    ungetc(c, stdin);
    scanf ("%lf", &yylval.valor);
    return NUM;
  }
  if (islower(c))
  {
   yylval.indice=c-'a';
   return VAR;
  }
  if (c == EOF)  return 0;
  return c;
}

/*  principal  */
#include <signal.h>
#include <setjmp.h>
jmp_buf inicio;
void main(void)
{
  printf("\nCalculadora 2: Escriba una expresión aritmética y pulse <intro>");
  printf("\nOperadores permitidos: +,-,*,/,^,- unario, y ()");
  printf("\nNuevas opciones:");
  printf("\n\tPermite uso de variables desde 'a' a 'z'");
```

```
  printf("\n\tComprueba la división por cero");
  printf("\n\tRecupera errores de punto flotante en ejecución");
  printf("\nPulse <control>-C (EOF) para salir\n");
  setjmp(inicio);
  signal(SIGFPE, error_en_float);
  yyparse();
}

/*  subrutina de error  */

void yyerror (char *s)
{
  printf ("%s\n", s);
  longjmp(inicio,0);
}
void error_en_float(void)
{
 yyerror("Error trabajando con números reales");
}
```

6.17.4 Calculadora con funciones

Se puede ampliar la calculadora del ejemplo anterior añadiéndole funciones predefinidas como seno, coseno, etc. También se puede conseguir sin mucho trabajo que permita declarar variables y almacenar valores en ellas. Esto significa que el analizador léxico *yylex* tendrá que reconocer identificadores y por lo tanto será necesaria una tabla de símbolos donde se almacenen tanto las variables como las funciones predefinidas.

La sección de declaraciones en este caso será:

```
%{
#include <math.h>
#include <ctype.h>
#include <stdio.h>
#include "calc.h"
%}
%union
{
   double val;    /* Devolver números */
   ptabla *tptr;  /* Punteros a la tabla de símbolos */
}
%token <val>  NUM
%token <tptr> VAR FUNC
%type  <val>  exp
%right '='
%left  '-' '+'
%left  '*' '/'
%left  NEG       /* Para el menos unario */
%right '^'       /* Exponentes           */
%%
```

En este caso los valores semánticos de los símbolos pueden ser de dos tipos: *val* y *ptabla*, por lo que habrá que indicar cual es el que le corresponde a cada uno. El fichero incluido "calc.h" comtiene la definición del tipo de la tabla de símbolos, que será una lista encadenada de registros. El contenido de este fichero será:

```
typedef
  struct ptabla
  {
     char *nombre;       /* nombre del símbolo */
     int tipo;           /* tipo del símbolo   */
     union
     {
        double var;             /* valor de una VAR  */
        double (*pfuncion)();   /* valor de una FUNC */
     } valor;
```

```
    struct ptabla *sig;  /* puntero al siguiente elemento */
  }
  ptabla;
extern ptabla *stabla;
ptabla *ponsimb();
ptabla *buscasimb();
```

Las dos funciones que se definen son las que tratan la tabla de símbolos. La definición de la gramática es muy similar a la del ejemplo anterior. Sólo hay que añadir tres reglas que hacen que la calculadora acepte variables y funciones. Será:

```
%%
input:   /* vacio */
       | input linea
       ;
linea:     '\n'
       | exp '\n'   {  printf("\t%.10g\n", $1);  }
       | error '\n' {  yyerrok;                  }
       ;
exp:     NUM                {  $$ = $1;            }
       | VAR                {  $$ = $1->valor.var; }
       | VAR '=' exp        {  $$ = $3;
                               $1->valor.var = $3; }
       | FUNC '(' exp ')'   {  $$=(*($1->valor.pfuncion))($3);  }
       | exp '+' exp        {  $$ = $1 + $3;       }
       | exp '-' exp        {  $$ = $1 - $3;       }
       | exp '*' exp        {  $$ = $1 * $3;       }
       | exp '/' exp        {  $$ = $1 / $3;       }
       | '-' exp %prec NEG  {  $$ = -$2;           }
       | exp '^' exp        {  $$ = pow ($1, $3);  }
       | '(' exp ')'        {  $$ = $2;            }
       ;
%%
```

Por la definición de la gramática se puede ver que la nueva calculadora admite asignaciones múltiples y funciones anidadas.

En el código C adicional, además de las funciones *main*, *yylex* e *yyerror* aparecerán otras dos: *ponsimb* y *buscasimb* que son las que manejan la tabla de símbolos. La función *yylex* tiene que ser modificada para que pueda reconocer identificadores. Resultará:

```
/*  analizador léxico  */
yylex ()
{
  int c;

  /* Saltar los espacios en blanco. */
  while ((c = getchar()) == ' ' || c == '\t')  ;
  if (c == EOF)  return 0;
  if (c == '.' || isdigit (c))  /* Se ha encontrado un número */
  {
    ungetc(c, stdin);
    scanf ("%lf", &yylval);
    return NUM;
  }
  if (isalpha(c)   /* Se ha encontrado un identificador. */
  {
     ptabla *s;
     static char *buf = 0;
     static int long = 0;
     int i;
```

```
     if (long == 0)
     {
        long = 40;
        buf = (char *) malloc (long + 1);
     }
     i = 0;
     do
     {   /* Si el buffer está lleno, hacerlo mayor. */
        if (i == long)
        {
           long *= 2;
           buf = (char *) realoc(buff, long + 1);
        }
        /* Añadir este carácter al buffer. */
        buf[i++] = c;
        /* Leer otro carácter */
        c = getchar;
     }
     while (c != EOF && isalnum (c));
     ungetc(c,stdin);
     buf[i] = '\0';
     s = buscasimb(buf)
     if (s == 0)
         s = ponsimb(buf, VAR);
     yylval.tptr = s;
     return s->tipo;
  }
  /* Cualquier otro carácter es un token. */
  return c;
}


/*  Función principal  */
main()
{
  initabla();
  yyparse();
}


/*  Subrutina de error  */
yyerror (s)
char  *s;
{
  printf ("%s\n", s);
}


struct ini
{
   char *fnombre;
   double (*func)();
};
struct ini farit[] =
{
   "sen",  sen,
   "cos",  cos,
   "atan", atan,
   "ln",   log,
   "exp",  exp,
   "sqrt", sqrt,
   0, 0
};

ptabla *stabla = (ptabla *)0;

/* Pone en la tabla las funciones aritméticas */
```

```
initabla()
{
   int i;
   ptabla *ptr;

   for (i=0; farit[i].fnombre != 0; i++)
   {
      ptr = ponsimb(farit[i].fnombre, FUNC);
      ptr->valor.pfuncion = farit[i].func;
   }
}


ptabla *ponsimb(snombre, stipo)
char *snombre;
int stipo;
{
   ptabla *ptr;

   ptr = (ptabla *) malloc (sizeof(ptabla));
   ptr->nombre = (char *) malloc (strlen(snombre)+1);
   strcpy(ptr->nombre, snombre);
   ptr->tipo = stipo;
   ptr->valor.var = 0;
   ptr->sig = (struct ptabla *) stabla;
   stabla = ptr;
   return ptr;
}


ptabla *buscasimb(snombre)
char *snombre;
{
   ptabla *ptr;

   for (ptr=stabla; ptr!=(ptabla *) 0; ptr=(ptabla *)ptr->sig)
      if (strcmp(ptr->nombre, snombre) == 0)
        return ptr;
   return 0;
}
```

6.17.5 Calculadora HOC

Se puede seguir ampliando la calculadora de los ejemplos anteriores añadiéndole más operadores (como AND, OR, >, <, etc), sentencias como **if**, **while**, funciones y procedimientos, obteniéndose el lenguaje HOC definido por *Kernighan* y *Pike* [KERN84, capítulo 8]. La entrada al *yacc* es la que viene a continuación:

```
%{
#include "hoc.h"
#define code2(c1,c2) code(c1); code(c2)
#define code3(c1,c2,c3) code(c1); code(c2); code(c3)
%}
%union
{
  Symbol *sym;  /* puntero a la tabla de simbolos */
  Inst   *inst  /* instrucción de máquina         */
  int    narg   /* número de argumentos           */
}
%token <sym>  NUMBER STRING PRINT VAR BLTIN UNDEF WHILE IF ELSE
%token <sym>  FUNCTION PROCEDURE RETURN FUNC PROC READ
%token <narg> ARG
%type  <inst> expr stmt asgn prlist stmtlist
%type  <inst> cond while if begin end
%type  <sym>  procname
%type  <narg> arglist
%right '='
%left  OR
%left  AND
%left  GT GE LT LE EQ NE
%left  '+' '-'
%left  '*' '/'
%left  UNARYMINUS NOT
%right '^'
```

```
%%
list:   /* vacío */
      | list '\n'
      | list defn '\n'
      | list asgn '\n'  { code2(pop, STOP); return 1;  }
      | list stmt '\n'  { code(STOP); return 1;        }
      | list expr '\n'  { code2(print, STOP); return 1;}
      | list error '\n' { yyerrok;                     }
      ;
asgn:   VAR '=' expr    { code3(varpush,(Inst)$1,assign); $$=$3;  }
      | ARG '=' expr    { defnonly("$"); code2(argassign,(Inst)$1); $$=$3; }
      ;
stmt:   expr           { code(pop);      }
      | RETURN         { defnonly("return"); code(procret); }
      | RETURN expr    { defnonly("return"); $$=$2; code (funcret);  }
      | PROCEDURE begin '(' arglist ')'
                       { $$ = $2; code3(call, (Inst)$1, (Inst)$4);    }
      | PRINT  prlist  {$$ = $2;  }
      | while cond stmt end
                       { ($1)[1] = (Inst)$3;   /* cuerpo del ciclo */
                         ($1)[2] = (Inst)$4; } /* fin si la condición no se cumple */
      | if cond stmt end          /* if sin else */
          { ($1)[1] = (Inst)$3;   /* parte then       */
            ($1)[3] = (Inst)$4; } /* fin si la condición no se cumple */
      | if cond stmt end ELSE stmt end  /*if con else */
          { ($1)[1] = (Inst)$3;   /* parte then       */
            ($1)[2] = (Inst)$6;   /* parte else       */
            ($1)[3] = (Inst)$7; } /* fin si la condición no se cumple */
      | '{' stmtlist '}'          { $$ = $2;  }
      ;
cond:   '(' expr ')'          { code(STOP); $$ = $2;   }
      ;
while:  WHILE                 { $$ = code3(whilecode,STOP,STOP);  }
      ;
if:     IF                    { $$ = code(ifcode); code3(STOP,STOP,STOP); }
      ;
begin:  /* vacío */           { $$ = progp;   }
      ;
end:    /* vacío */           {  code(STOP); $$ = progp;  }
      ;
stmtlist:  /* vacío */        { $$ = progp;   }
      | stmtlist '\n'
      | stmtlist stmt
      ;
expr:   NUMBER      { $$ = code2(constpush, (Inst)$1); }
      | VAR         { $$ = code3(varpush, (Inst)$1, eval);  }
      | ARG         { defnonly("$"); $$ = code2(arg, (Inst)$1);  }
      | asgn
      | FUNCTION  begin '(' arglist ')'
                    { $$ = $2; code3(call, (Inst)$1, (Inst)$4);  }
      | READ '(' VAR ')'   { $$ = code2(varread, (Inst)$3);  }
      | BLTIN '(' expr ')' { $$=$3; code2(bltin, (Inst)$1->u.ptr); }
      | '(' expr ')'       { $$ = $2;  }
      | expr '+' expr      { code(add);  }
      | expr '-' expr      { code(sub);  }
      | expr '*' expr      { code(mul);  }
      | expr '/' expr      { code(div);  }
      | expr '^' expr      { code(power);  }
      | '-' expr    %prec UNARYMINUS   { $$=$2; code(negate);  }
      | expr GT expr      { code(gt);  }
      | expr GE expr      { code(ge);  }
      | expr LT expr      { code(lt);  }
      | expr LE expr      { code(le);  }
      | expr EQ expr      { code(eq);  }
      | expr NE expr      { code(ne);  }
      | expr AND expr     { code(and);  }
      | expr OR expr      { code(or);  }
      | NOT expr          { $$ = $2; code(not);  }
      ;
```

```
prlist: expr              { code(prexpr);  }
      | STRING            { $$ = code2(prstr, (Inst)$1);  }
      | prlist ',' expr   { code(prexpr);  }
      | prlist ',' STRING { code2(prstr, (Inst)$3);  }
      ;
defn:   FUNC procname     { $2->type=FUNCTION; indef=1;  }
          '(' ')' stmt    { code(procret); define($2); indef=0;  }
      | PROC procname     { $2->type=PROCEDURE; indef=1;  }
          '(' ')' stmt    { code(procret); define($2); indef=0;  }
      ;
procname:  VAR
      | FUNCTION
      | PROCEDURE
      ;
arglist:   /* vacío */    { $$ = 0;  }
      | expr              { $$ = 1;  }
      | arglist ',' expr  { $$ = $1 + 1;  }
      ;
%%    /*  fin de la gramática  */
#include <stdio.h>
#include <ctype.h>
#include <signal.h>
#include <setjmp.h>
char *progname;
int lineno = 1;
jmp_buf  begin;
int   indef;
char  *infile;          /* nombre de archivo de entrada */
FILE  *fin              /* apuntador a archivo de entrada */
char  **gargv;          /* lista global de argumentos */
int   gargc;
int c;                    /* global, para uso de warning() */
yylex()
{
   while ((c=getc(fin)) == ' '  || c== '\t') ;
   if (c == EOF)
        return 0;
   if (c == '.' || isdigit(c))     /* número */
   {
      double d;
      ungetc(c, fin);
      fscanf(fin, "%lf", &d);
      yylval.sym = install("", NUMBER, d);
      return NUMBER;
   }
   if (isalpha(c))
   {
       Symbol *s;
       char sbuf[100], *p =sbuf;
       do
       {
           if (p >= sbuf + sizeof(sbuf) - 1)
           {
              *p = '\0';
              execerror("nombre demasiado largo", sbuf);
           }
           *p++ =c;
       } while ((c=getc(fin) != EOF && isalnum(c));
       ungetc(c, fin);
       *p = '\0';
       if ((s=lookup(sbuf)) == 0)
           s = install(sbuf, UNDEF, 0.0);
       yylval.sym =s;
       return s->type == UNDEF ? VAR : s->type;
   }
   if (c == '$')       /* ¿argumento? */
   {
      int n = 0;
```

```
        while (isdigit(c=getc(fin)))
            n = 10 * n + c - '0';
        ungetc(c,fin);
        if (n == 0)
            excerror("strange $...", (char *)0);
        yylval.narg = n;
        return ARG;
    }
    if (c == '"')          /* cadena entre comillas */
    {
        char sbuf[100], *p, *emalloc();
        for (p = sbuf; (c=getc(fin)) !='"'; p++)
        {
            if (c == '\n' || c==EOF)
                execerror("comilla no encontrada","");
            if (p >= sbuf + sizeof(sbuf) -1
            {
                *p = '\0';
                execerror("cadena demasiado larga", sbuf);
            }
            *p = backslash(c);
        }
        *p = 0;
        yylval.sym = (Symbol *)emalloc(strlen(sbuf)+1);
        strcpy(yylval.sym,sbuf);
        return STRING;
    }
    switch (c)
    {
     case '>' :       return follow('=', GE, GT);
     case '<' :       return follow('=', LE, LT);
     case '=' :       return follow('=', EQ, '=');
     case '!' :       return follow('=', NE, NOT);
     case '|' :       return follow('|', OR, '|');
     case '&' :       return follow('&', AND,'&');
     case '\n':       lineno++; return '\n';
     default:         return c;
    }
}


backslash(c)  /* tomar el siguiente caracter con las \ interpretadas*/
int c;
{
    char *index(); /* ' strchr()' en algunos sistemas */
    static char transtab[]="b\bf\fn\nr\rt\t";
    if (c!='\\') return c;
    c = getc(fin);
    if (islower(c) && index(transtab,c))
        return index (transtab,c)[1];
    return c;
}


follow(expect,ifyes, ifno) /* búsqueda hacia adelante para >=, etc */
{
    int c=getc(fin);
    if (c == expect)
         return ifyes;
    ungetc(c,fin);
    return ifno;
}


defnonly(s) /* advertencia si hay definición ilegal */
char *s;
{
    if (!indef) execerror(s,"definición ilegal");
}
```

```
yyerror(s)/* comunicar errores de tiempo de compilación */
char *s;
{
   warning(s, (char *)0);
}


execerror(s,t)  /* recuperación de errores de tiempo de ejecución */
char *s, *t;
{
   warning(s,t);
   fseek(fin,OL,2); /* sacar el resto del archivo */
   longjmp(begin,0);
}


fpecatch()  /* detectar errores por punto flotante */
{
    execerror("error de punto flotante", (char *)0);
}


main(argc,argv)
char *argv[];
{
   int i,fpecatch();
   progname = argv[0];
   if (argc == 1)   /* simular una lista de argumentos */
   {
      static char *stdinonly[] = {"-"};
      gargv = stdinonly;
      gargc = 1;
   }
   else
   {
      gargv = argv +1;
      gargc = argc - 1;
   }
   init();
   while (moreinput())
      run();
   return 0;
}


moreinput()
{
   if (gargc-- <= 0)
      return 0;
   if (fin && fin != stdin)
      fclose(fin);
   infile = *gargv++;
   lineno = 1;
   if (strcmp(infile, "-")) == 0)
   {
      fin = stdin;
      infile = 0;
   }
   else if ((fin=fopen(infile, "r")) ==NULL)
   {
        fprintf(stderr,"%s: can't open %s\n",progname,infile);
        return moreinput();
   }
   return 1;
}


run()     /* ejecutar hasta el fin de archivo (EOF) */
{
     setjmp(begin);
     signal(SIGFPE, fpecatch);
```

```
     for (initcode(); yyparse(); initcode())
        execute(progbase);
}


warning(s,t)  /* imprimir mensaje de advertencia */
char *s, *t;
{
    fprintf(stderr, "%s: %s", progname, s);
    if (t)  fprintf(stderr, " %s",t);
    if (infile)  fprintf(stderr, " in %s", infile);
    fprintf(stderr, " en la línea %d\n", lineno);
    while (c != '\n' && c != EOF)
      c = getc(fin);   /* sacar el resto del renglón de entrada  +/
    if (c == '\n')  lineno++;
}
```

6.18 Compilador de MUSIM/1 a ENSAMPOCO/1

Se construye un compilador de MUSIM/1 a ENSAMPOCO/1, ambos definidos en [CUEV94b].

```
%{
/* cmusim1.y */
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <mem.h>
int nlinea=1;
FILE *yyin,*yyout;
void yyerror(char*);
int yyparse(void);
void genera(char*);
void genera2(char*, int);
%}
%union
 { char identificador;
   int constante;
 }
%token  <constante> NUM
%token <identificador> VAR
%type <constante> bloque sentencia lectura escritura asignacion
%right '='
%left '-' '+'
%left '*' '/' '%'
%left NEG  /* Para darle precedencia al menos unario. */
%right '^'
%start programa
%%
programa     : 'M' '{' bloque finBloque
           ;
bloque       : { genera(".CODE"); }  instrucciones
           ;

finBloque    : '}'    {genera("END");}
           ;
instrucciones      : /* vacío */
          | instrucciones sentencia
          | instrucciones sentencia ';'
          ;
sentencia   : lectura
          | escritura
          | asignacion
          ;
```

```
lectura     : 'R' VAR         { genera2("INPUT", $2);  }
          ;
escritura   : 'W' VAR         { genera2("OUTPUT", $2); }
            ;
asignacion  : VAR             { genera2("PUSHA", $1); }
              '=' exp         { genera("STORE");      }
            ;
exp      : NUM                {  genera2("PUSHC", $1); }
         | VAR                 {  genera2("PUSHA", $1);
                                 genera("LOAD");       }
         | exp '+' exp         {  genera("ADD");        }
         | exp '-' exp         {  genera("NEG");
                                 genera("ADD");        }
         | exp '*' exp         {  genera("MUL");        }
         | exp '/' exp         {  genera("DIV");        }
         | exp '%' exp         {  genera("MOD");        }
         | '-' exp %prec NEG   {  genera("NEG");        }
         | exp '^' exp         {  genera("EXP");        }
         | '(' exp ')'
         ;
%%
/*  analizador léxico  */
#include <ctype.h>
int yylex (void)
{
  int c;
  while ((c = getc(yyin)) != EOF)
   {
    if (c == ' '| c == '\t') continue; /* Elimina blancos y tabuladores */
    if (c == '\n')
      {++nlinea;
       continue;  /* Elimina los caracteres fin de línea */
      }
    if (isdigit (c))
     {
       yylval.constante = c;
       return NUM;
     }
    if (islower(c))
     {
      yylval.identificador = c;
      return VAR;
     }
    return c;
   }
  return 0;
}

/*  principal  */
void main(int argc, char *argv[])
{
  if ((argc<3) ||  (argc>4))
   {
    printf("\nCompilador MUSIM/1-ENSAMPOCO/1 desarrollado con YACCOV");
    printf("\nForma de uso:");
    printf("\ncmusim1 fich_fuente fich_destino\n");
    exit(-1);
   }
```

```
  if ((yyin=fopen(argv[1],"r")) == NULL)
   {
    printf("\nNo se puede abrir el fichero -> %s\n", argv[1]);
    exit(-2);
   }
  if ((yyout=fopen(argv[2],"w")) == NULL)
   {
    printf("\nNo se puede abrir el fichero -> %s\n", argv[2]);
    exit(-3);
   }
  printf("\nTraduciendo ...\n");
  yyparse();
  fcloseall();
  exit(0);
}

/*  subrutina de error  */
void yyerror (char *s)
{
  fprintf (stdout,"Error en línea %d : %s\n", nlinea, s);
}
void genera(char *s)
{
 fprintf(yyout,"%s\n",s);
}
void genera2(char *s, int x)
{
 fprintf(yyout,"%s %c\n", s, x);
}
```

6.19 YACCOV como entrada al yacc

El fichero de entrada a YACCOV, se puede considerar en cierto modo como la entrada a un compilador, ya que YACCOV es en cierto modo un compilador, por lo tanto, este fichero puede ser descrito utilizando una gramática de entrada a YACOOV. Sería:

```
/* Gramática que describe la entrada a YACCOV. */

/* Sección de declaraciones. */
%token IDENTIFICADOR   /* Representa identificadores y literales. */
%token IDENTIFICADOR_C /* Representa un identificador seguido por ':'. */
%token NUMERO
%token LEFT, RIGHT, NONASSOC, TOKEN, TYPE, START, UNION, EXPECT, PREC
%token SEMANTICO, REENTRANTE
%token DOSPORC         /*   '%%'   */
%token LLAVEI          /*   '%{'   */
%token LLAVED          /*   '%}'   */
%start spec

%%

/* Gramática */
spec  :   defs DOSPORC reglas final
       ;
final :   /* vacío, el segundo '%%' es opcional  */
       |  DOSPORC   { en esta acción saltar el resto del fichero de entrada }
       ;
```

```
defs  :   /* vacío */
       |  defs def
       |  defs ',' def
       ;
def   :   START IDENTIFICADOR  { Marcar IDENTIFICADOR como inicial }
       |  UNION  { Copiar la union al fichero de salida }
       |  EXPECT { Copiar número de conflictos esperados }
       |  SEMANTICO { Hacer que el analizador generado use "sem.prs" }
       |  REENTRANTE { Hacer que el analizador generado sea reentrante }
       |  LLAVEI { Copiar al fichero de salida } LLAVED
       |  def1 tipo lista
       ;
def1  :   TOKEN
       |  LEFT
       |  RIGHT
       |  NONASSOC
       |  TYPE
       ;
tipo  :   /* vacío */
       |  '<' IDENTIFICADOR '>'
       ;
lista :   var
       |  nlist var
       |  nlist ',' var
       ;
var   :   IDENTIFICADOR     /* Con %type no pueden ser literales */
       |  IDENTIFICADOR NUMERO  /* %type no puede usar esta opción */
       ;
reglas:   IDENTIFICADOR_C cuerpo prec
       |  reglas regla
       ;
regla :   IDENTIFICADOR_C cuerpo prec
       |  '|' cuerpo prec
       ;
cuerpo:   /* vacío */
       |  cuerpo IDENTIFICADOR
       |  cuerpo accion
       ;
accion:   '{'  { Copiar la acción en el lugar correspondiente }
       ;
prec  :   /* vacío */
       |  PREC IDENTIFICADOR
       |  PREC IDENTIFICADOR accion
       |  prec ';'
       ;
```

Se definen dos tipos de identificadores para distinguir los que van seguidos por ':', lo que indicaría el inicio de una nueva regla, de los que forman parte de la parte derecha de las reglas. Por comodidad se ha incluído la declaración *%type* entre todas las que definen símbolos en la sección de declaraciones, pero sería más apropiado hacerle una definición especial, dado que tiene que llevar un nombre de tipo detrás obligatoriamente y que no puede llevar números detrás de los nombres de los símbolos porque lo que define son no terminales, no tokens.

7 EJERCICIOS PROPUESTOS

Ejercicio 7.1

Sea la gramática cuyo símbolo inicial es <A> y las reglas de producción son las siguientes:

<A> ::= x | (<B>)
<B> ::= <A> <B>
<C> ::= { + <A> }

a) Construir un autómata de pila que reconozca el lenguaje generado por dicha gramática.
b) ¿El autómata es determinista?
c) ¿La gramática es ambigua?
d) ¿Es LL(1)?
e) ¿Cuál es el lenguaje reconocido por la gramática?

Ejercicio 7.2

Sea la gramática: S → SS+|SS*|a

a) ¿Es ambigua? En el caso de que la respuesta sea afirmativa demostrarlo. En el caso de que sea negativa justificarlo lo mejor posible.
b) Indicar de alguna forma el lenguaje reconocido por la grmática
c) ¿Es LL(1)? Si no lo es indicar las razones y escribir si es posible una gramática equivalente LL(1)
d) Construir un autómata que reconozca el lenguaje.
e) ¿Es posible construir un autómata determinista para reconocer este lenguaje? En caso afirmativo construirlo.
f) Escribir una entrada al *yacc* que dada una cadena de lenguaje generado por la gramática dada nos indique si pertenece o no al lenguaje.

Ejercicio 7.3

Sea la gramática S → xSyS | ySxS | λ.

a) ¿Es ambigua? ¿Por qué?
b) ¿Qué lenguaje genera la gramática?
c) La cadena *xyxy* pertenece al lenguaje. Determinar su árbol sintáctico.

Ejercicio 7.4

Sea la gramática:

S → x | (S R
R → , S R |)

a) Determinar los conjuntos de símbolos INICIALES, SIGUIENTES y DIRECTORES.
b) A partir de los conjuntos anteriores comprobar si es LL(1).

Ejercicio 7.5

Sea la gramática:

A → d A | d B | f
B → g

a) Determinar los conjuntos de símbolos INICIALES, SIGUIENTES y DIRECTORES.
b) A partir de los conjuntos anteriores comprobar si es LL(1).

Ejercicio 7.6

Sea la gramática:

S → X d
X → C
X → B a
C → λ
B → d

a) Determinar los conjuntos de símbolos INICIALES, SIGUIENTES y DIRECTORES.

b) A partir de los conjuntos anteriores comprobar si es LL(1).

Ejercicio 7.7

Sea la gramática:

A → B x y | x
B → C D
C → A | C
D → d

a) ¿Es recursiva a izquierdas de forma directa o indirecta?
b) En caso afirmativo determinar una gramática equivalente que no sea recursiva a izquierdas.
c) Si es posible obtener una gramática equivalente LL(1) y comprobarlo.

Ejercicio 7.8

Escribir un intérprete de MUSIM/1 con lex y yacc.

Ejercicio 7.9

Construir las tablas de precedencia y asociatividad de los operadores en los lenguajes FORTRAN, Pascal y C.

Ejercicio 7.10

Construir una gramática LL(1) para las expresiones de los lenguajes FORTRAN, Pascal y C.

Ejercicio 7.11

Diseñar una gramática LL(1) para el lenguaje Pascal estándar completo.

Ejercicio 7.12

Construir una entrada al lex y yacc para reconocer el lenguaje C ANSI completo.

8 PRACTICAS DE LABORATORIO

8.1 Compilador de MUSIM/95 a ENSAMPIRE

El compilador debe incluir la siguiente documentación:

- **Gramática LL(1) de MUSIM/95**. En un fichero de texto denominado MUSIM95.TXT y en un fichero para el analizador de gramáticas LL(1): MUSIM95.GRM
- **Módulos en C++ del compilador debidamente comentados**. Como mínimo deberán presentarse los módulos: LEXICO.H, LEXICO.CPP, SINTACTI.H, SINTACTI.CPP, SEMANTI.H, SEMANTI.CPP, TABLASIM.H, TABLASIM.CPP, GENERA.H y GENERA.CPP. Es obligatorio utilizar clases para construir el compilador. Fichero proyecto (.PRJ) o make (.MAK) para obtener el ejecutable. Fichero ejecutable denominado MUSIM95.EXE.
- **Validación del compilador mediante baterias de test**. Se deberan incluir los ficheros de prueba, agrupados en directorios denominados CTEST_A, CTEST_B, CTEST_D, y CTEST_E.
- **Documentación del compilador**: Manual del programador (PROGRA.TXT), manual del usuario (USUARIO.TXT), y manual de referencia técnica (TECNICO.TXT).

8.2 Traductor de ENSAMPIRE a ENSAMBLADOR 80x86

El traductor debe incluir la siguiente documentación:

- **Módulos en C++ del traductor debidamente comentados**. Es obligatorio utilizar clases para construir el traductor. Fichero proyecto (.PRJ) o make (.MAK) para obtener el ejecutable. Fichero ejecutable denominado ENSAMPI.EXE.
- **Validación del traductor mediante baterias de test**. Se deberan incluir los ficheros de prueba, agrupados en directorios denominados TTEST_A, TTEST_L, y TTEST_D.

- **Documentación del traductor**: manual del usuario y manual de referencia técnica. Ambos se incluirán en los ficheros del compilador (USUARIO.TXT y TECNICO.TXT).

8.3 Intérprete de ENSAMPIRE

El intérprete debe incluir la siguiente documentación:

- **Módulos en C++ del intérprete debidamente comentados**. Es obligatorio utilizar clases para construir el intérprete. Fichero proyecto (.PRJ) o make (.MAK) para obtener el ejecutable. Fichero ejecutable denominado ENSAMPI.EXE.
- **Validación del intérprete mediante baterias de test**. Se deberan incluir los ficheros de prueba, agrupados en directorios denominados ITEST_A, ITEST_B, e ITEST_D.
- **Documentación del intérprete**: manual del usuario y manual de referencia técnica. Ambos se incluirán en los ficheros del compilador (USUARIO.TXT y TECNICO.TXT).

8.4 Traductor de Eiffel/95 a C++

En esta práctica se debe construir un traductor de un subconjunto del lenguaje Eiffel a C++ utilizando las herramientas LEX y YACC. Es necesario realizar el proceso de validación (incluir baterias de test). Se deben incluir como mínimo los siguientes ficheros: EIFFEL95.L, EIFFEL95.Y, YYLEX.C, EIFFEL95.C, EIFFEL95.EXE. Además se deben incluir los ficheros de las baterias de test.

8.4.1 Subconjunto del lenguaje Eiffel denominado Eiffel/95

Tipos de datos: Sólo puede trabajar con los siguientes tipos.

simples: INTEGER y CHAR

clases

- **Constantes:** caracter, cadena y entero
- **Operadores:** los que soporta el lenguaje y puedan operar con los tipos anteriores.
- **Rutinas no predefinidas**: Procedimientos y Funciones que puedan incluir variables locales, precondiciones, postcondiciones y rutinas externas.
- **Instrucciones**: llamadas a procedimientos, asignaciones, condicionales y bucles.
- **Expresiones**: constantes, entidades y llamadas de función.
- **Herencia**: simple y múltiple. Se debe contemplar el caso de herencia repetida indirecta pero no la directa.

La generación de código debe efectuarse simultáneamente a los demás procesos de análisis. Dado que en Eiffel la definición de clases no ha de guardar un orden con el fin de que sea coherente el análisis semántico, a diferencia de C++, es necesario generar una información previa que le indique al compilador de C++ la presencia de clases y miembros de manera completa.

El análisis semántico debe efectuarse dando tres pasadas al código fuente:

- **Paso 1**: se determinan las clases integrantes del módulo fuente completo y los identificadores miembros que representan atributos exportables.
- **Paso 2**: se determinan y analizan los identificadores de atributos de las clases. Se estudia la herencia entre clases así como el renombrado de atributos heredados.
- **Paso 3**: se analizan las variables locales de las rutinas miembro, asignaciones, expresiones aritméticas, llamadas a funciones miembro o no etc.

Se debe usar el **lex** de UNIX ya que en ninguno de los analizadores léxicos para MS-DOS funciona cprrectamente la función **yywrap** imprescindible para efectuar la relectura del fichero fuente. Después se exporta la función *yylex()* a MS-DOS con FTP y se trabaja normalmente con cualquier *yacc*.

Sería conveniente crear dos ficheros cabecera: uno que contenga los nombres de las clases (en forma de prototipos de clases) contenidas en el fichero fuente; otro que contenga la descripción detallada de las clases, en formato de prototipo de C++ para clases.

Se debe crear otro fichero de desglose y desarrollo de las rutinas de clases con posibles constructores y destructores así como otros ficheros auxiliares.

8.4.2 Eiffel versus C++ en la generación de código

8.4.2.1 Declaración de clases

Se debe contemplar solamente el caso de clases sin tipos genéricos.

Eiffel

```
class Clase1
...
end
```

C++

```
class Clase1 { ...
}
...
```

8.4.2.2 Declaración de funciones

Eiffel

```
class Clase1
...
 funcion(x:INTEGER):INTEGER is
 do
  ...
  Result:=x+y*z
 end;
end
```

C++

```
class Clase1
{ ...
  int funcion (..., int x);
}
int Clase1::funcion(..., int x)
{
 int result_funcion=0;
 ...
 result_funcion=x+y*z;
 ...
 return result_funcion;
}
...
```

8.4.2.3 Tratamiento de precondiciones

Eiffel

```
class Clase1
...
 proc(x:INTEGER) is
 require
    x > 0
 do
  ...
 end;
end
```

C++

```
class Clase1
{ ...
  void proc(..., int x);
}
void Clase1::proc(..., int x)
{
   Require_proc(..., x);
   ...
}
void Clase1::Require_proc(..., int x)
{
 ...
 if (!(x > 0)) longjmp(jmp, ...);
}
...
```

Las precondiciones solamente afectan a los atributos de la clase a la que pertenece la rutina y a los parámetros de esta. *jmp* debería señalar el inicio de la sección de manejo de excepciones de la rutina que la invocó.

8.4.2.4 Tratamiento de postcondiciones

Eiffel

```
class Clase1
...
 proc(x:INTEGER) is
 do
  ...
 ensure
   x > 0;
 end;
end
```

C++

```
#define Ensure_proc(...)\
{\
    if (!(x > 0)) lonjmp(jmp,...);\
}
class Clase1
{ ...
  void proc(..., int x);
}
void Clase1::proc(..., int x)
{
   Ensure_proc(...);
   ...
}
...
```

Se construye una macro ya que pueden estar imlicados aparte de los parámetros y atributos de clase, las variables locales de la rutina

8.4.2.5 Herencia

La herencia se debe gestinar de forma análoga a como lo hace el lenguaje Eiffel.

Eiffel

```
class A
export var1, funA
feature
  var1:INTEGER;
  funA (x:INTEGER; y:INTEGER):INTEGER is
  do
   ...
  end;
end

class B
export var2, procB, funA
inherit
   A redefine funA
feature
   var2:INTEGAR;

   procB is
     require
      ...
     do
       ...
     ensure
       ...
     end

   funA (x:INTEGER; y:INTEGER):INTEGER is
      require
      do
        ...
      ensure
        ...
      end;
end
```

C++

```
class A
{
public:
int var1;
virtual int funA (...);
};

class B: public A
{
public:
int var2;

void procB (...);
int funA (...);
}

int a::funA (...)
{
 ...
 return ...
}

void B::procB (...)
{
 ...
}

int B::funA (...)
{
  ...
  return ...
}

...
```

Otros ejemplos

Eiffel

```
-- Herencia repetida indirecta
class A
inherit
C
feature
 mostrar is
  do
   print(" Clase A ")
  end
end
```

```
class B
inherit
C
feature
 mostrar is
  do
   print(" Clase B ")
  end
end

class C
feature
 escribir is
  external
      print (macs:STRING; y:INTEGER) name "printf" language "C";
  do
   print(" Clase C ")
  end
end

class D
inherit
 A
 rename mostrar as mostrar_A
 end
 B
 rename mostrar as mostrar_B
 end
feature
 principal is
 do
  crear;
 end
 crear is
 do
 print(" Clase D ");
 mostrar_A;
 mostrar_b;
 end
end
```

C++

```
// Herencia repetida indirecta
#include <stdio.h>
...
class A: public C {
public:
 void mostrar();
};

class B: public C {
public:
 void mostrar();
};

class C {
public:
  void escribir();
};
```

```
class D:public A, B {
public:
 void crear();
 void mostrar_A();
 void mostrar_B();
};

void A::mostrar()
 {
   printf("clase A \n");
 }

void B::mostrar()
 {
   printf("clase B \n");
 }

void C::escribir()
 {
   printf("clase C \n");
 }

void D::crear()
 {
   printf("clase D \n");
   mostrar_A();
   mostrar_B();
 }

void D::mostrar_A()
  {
    A::mostrar();
  }

void D::mostrar_B()
  {
    B::mostrar();
  }

void main()
 {
  D princi;
  princi.crear();
 }
```

Eiffel

```
-- Este programa determina el cambio necesario para devolver una cantidad
-- de dinero en monedas de 100 pesetas, 25 pesetas y de una peseta.
--
class cambio
export coge_pesetas, conv_a_centenas, conv_a_venticinco, conv_a_duros
feature
 ptas:INTEGER;
 principal is
  external
      print (macs:STRING; y:INTEGER) name "printf" language "C";
      scan (macs:STRING; y:INTEGER) name "scanf" language "C";
      puts (masc:STRING) name "puts" language "C"
  local
```

```
    cantidad,c,d,e:INTEGER;
  do
   puts ("Dame las pesetas a cambiar:");
   scan("%d", cantidad);
   coge_pesetas(cantidad);
   c:=conv_a_centenas;
   d:=conv_a_venticinco(c);
   e:=conv_a_duros(d);
   print ("%d  pesetas",e)
  end
 coge_pesetas (p:INTEGER) is
 do
  ptas=p;
  print("%d  ptas, en monedas: \n",ptas);
 end
 conv_a_centenas:INTEGER is
  do
   ptas:= ptas DIV 100;
   print ("%d  monedas de cien,",ptas);
   Result:= ptas mod 100;
  end
 conv_a_veinticinco (p:INTEGER):INTEGER is
  do
   ptas:= p DIV 25;
   print ("%d  monedas de venticinco,",ptas);
   Result:= p mod 25;
  end

 conv_a_duros (p:INTEGER):INTEGER is
  do
   ptas:=p div 5;
   print ("%d monedas de duro, y",ptas);
   Result:= p mod 5;
  end
end
```

C++

```
// Este programa determina el cambio necesario para devolver una cantidad
// de dinero en monedas de 100 pesetas, 25 pesetas y de una peseta.
//
#include <iostream.h>
#include <stdio.h>

class cambio {
            int ptas;
          public:
            cambio();    //constructor
            ~cambio();   //destructor
            void coge_pesetas(int p);
            int conv_a_centenas();
            int conv_a_veinticinco(int p);
            int conv_a_duros(int p);
           };
cambio::cambio()
 { // se puede usar para inicializar variables
 }
cambio::~cambio()
 { // se puede usar para liberar objetos
 }
```

```
void cambio::coge_pesetas(int p)
{
 ptas=p;
 printf("%d ptas, en monedas: \n",ptas);
}
int cambio::conv_a_centenas()
{
 printf("%d monedas de cien,",ptas/100);
 return(ptas%100);
}
int cambio::conv_a_veinticinco(int p)
{
 printf("%d monedas de veinticinco,",p/25);
 return(p%25);
}
int cambio::conv_a_duros(int p)
{
 printf("%d monedas de duro, y ",p/5);
 return(p%5);
}
void main(void)
{
 cambio vuelta;
 int cantidad, c,d,e;
 puts("Deme las pesetas a cambiar:  ");
 scanf("%d",&cantidad);
 vuelta.coge_pesetas(cantidad);
 c=vuelta.conv_a_centenas();
 d=vuelta.conv_a_veinticinco(c);
 e=vuelta.conv_a_duros(d);
 printf("%d  pesetas",e);
}
```

ANEXO I Lenguaje MUSIM/95

I.1 Especificación léxica

Es indiferente el uso de mayúsculas o minúsculas (al igual que el lenguaje Pascal). Los componentes léxicos del lenguaje se especifican a continuación:

- **Identificadores**: comienzan por una letra y pueden estar seguidos por letras o dígitos. Puede definirse un máximo de caracteres para un identificador no menor de 15.
- **Constantes**: pueden ser enteras, reales sin exponente, de tipo caracter y de tipo cadena. Las constantes de tipo simple se escriben entre comillas simples por ejemplo 'a'. Las de tipo cadena se escriben entre comillas dobles. Por ejemplo "La casa verde". Existe unas constantes de tipo caracter predefinidas '\n', '\0' y '$'. Las dos últimas usadas para delimitar cadenas.
- **Operadores aritméticos**: +, -, *, /, %, ^ y - unario.
- **Operadores de comparación**: <, >, #
- **Operadores lógicos o booleanos**: &, |, !
- **Operadores de manejo de punteros**: & y *
- **Símbolo de la sentencia de asignación**: es el = (igual)
- **Paréntesis, corchetes, punto, llaves, comas, dos puntos y punto y coma.**

  Tipos de datos: INT, FLOAT, CHAR, VOID
- **Palabras reservadas**: MAIN, STRUCT, UNION, RETURN, IF, THEN, ELSE, WHILE, FOR, TO, DOWNTO, DO, REPEAT, UNTIL, CASE, OF, END, GOTO, y FILE

- **Procedimientos estándar de entrada/salida**: READ(parámetros), WRITE(parámetros), OPEN(ident_fich, cadena de caracteres o puntero a cadena, modo), CLOSE(ident_fich), READ(ident_fich, parámetros) y WRITE(ident_fich, parámetros).
- **Procedimientos y funciones estándar de gestión de memoria dinámica heap**: función MALLOC y procedimiento FREE (similares a los del lenguaje C).
- **Comentarios**: Comienzan por \ y acaban en fin de línea

I.2 Especificación sintáctica

Continuando con las especificaciones dadas hasta el MUSIM/7 [CUEV94b, pág 78] se añaden:

I.2.1 Estructuras de control de flujo siguiendo la sintaxis de Pascal estándar

```
FOR <expr> [TO|DOWNTO] <expr> DO <sent>
IF <expr> THEN <sent> [ ELSE <sent> ]
WHILE <expr> DO <sent>
REPEAT <sent> UNTIL <expr>
CASE <expr> OF {<valor>: <sent>} END
GOTO cte_entera
```

I.2.2 Declaración de funciones prototipo delante de MAIN y construcción de funciones definidas por el usuario detrás de MAIN

Se añaden funciones definidas por el usuario al estilo C pero con la obligatoriedad de tener función prototipo declarada antes de MAIN, y que la construcción de funciones se realiza después de MAIN. Las funciones tienen que soportar recursividad, paso por valor y por dirección, y variables locales. Se incluye el tipo void y la palabra reservada RETURN. En la evaluación de expresiones las funciones tienen la máxima precedencia como las variables y constantes.

I.2.3 Ficheros secuenciales de texto

Se incorporan las funciones estándar de manejo de ficheros y la palabra reservada FILE para declarar los descriptores de fichero. Las funciones son: OPEN(ident_fich, cadena de caracteres o puntero a cadena, modo), CLOSE(ident_fich), READ(ident_fich, parámetros) y WRITE(ident_fich, parámetros).

I.2.4 Gestión de memoria dinámica heap y punteros al estilo C

Se introducen punteros a tipos simples y estructuras, con sintaxis igual a C. También se introducen los operadores de manejo de punteros & y *. Las funciones de gestión de memoria son MALLOC y FREE con sintaxis igual al lenguaje C.

I.3 Especificación semántica

El lenguaje tiene comprobación estricta de tipos al estilo del Pascal estándar, tal y como se describe en [CUEV94b, pág 65].

ANEXO II Lenguaje Eiffel

El lenguaje Eiffel ha sido desarrollado por Bertrand Meyer con el propósito de obtener un lenguaje orientado a objetos puro. Las ideas más notables de Eiffel proceden de Simula 67, ALGOL y lenguajes derivados, Alphard, CLU, Ada y el lenguaje de especificación Z [MEYE88, MEYE92, LEIV93, KATR94].

Las implementaciones existentes de lenguaje Eiffel son traductores a lenguaje C. Por otra parte la característica de gestión de memoria automática, implementada con recolectores de basura (garbage collection) suponen un límite a su uso como lenguaje en tiempo real.

Sin embargo las aportaciones de Eiffel como lenguaje pueden marcar las líneas maestras del diseño de los lenguajes orientados a objetos del futuro.

II.1 ELEMENTOS BÁSICOS DE PROGRAMACIÓN EN EIFFEL

Los constituyentes básicos del lenguaje orientado a objetos Eiffel son: objetos, referencias, clases y entidades.

II.1.1 Objetos

Se puede hablar de objetos *internos* y *externos*. Por un lado, los externos, se refieren a la realidad física; en un sistema gráfico: puntos, líneas, ángulos, superficies, etc.; en un sistema de gestión de nóminas: empleados, cheques, niveles salariales, etc. Por otro lado, nuestro software no tratará directamente con estos objetos, sino con representaciones apropiadas para el ordenador: son los objetos internos.

La solución orientada a objetos usa objetos en ambos sentidos: en la etapa de diseño la cuestión es describir las clases de objetos externos cuyas funciones el sistema intentará modelar; en la etapa de implementación, los lenguajes orientados a objetos, aseguran que los sistemas se escriban como colecciones de descripciones de objetos internos, en lugar de procedimientos.

A un nivel elemental, los objetos internos son sencillamente como los *registros* que maneja un programa en Pascal o *estructura* en C.

Al igual que un registro, un objeto es una estructura ocupando cierto espacio de la memoria durante la ejecución del sistema, y consta de cierto número de constituyentes, llamados campos.

II.1.2 Referencias

Es conveniente, a menudo, introducir objetos internos que, más allá de simples campos, incluyan campos que hagan referencia a otros objetos. Una técnica consiste en permitir que los objetos contengan a otros objetos, sin embargo esto no permite compartir objetos (sharing). El hecho de compartir es necesario cuando campos de diferentes objetos hacen referencia al mismo objeto (como oposición a objetos distintos pero idénticos).

Para permitir objetos compartidos, los campos deben poder contener referencias a otros objetos. Una referencia puede tener un objeto por valor; dos o más referencias pueden compartir el mismo objeto como valor. Esta técnica es muy útil para representar estructuras de datos complejas (listas, árboles, colas, etc.). En Eiffel, todos los campos de los objetos son simples (integer, boolean, carácter, real) o referencias. Así pues, un objeto puede hacer referencia a un objeto pero no contener un objeto.

En caso de que una referencia no tenga asociado ningún objeto, se dirá que esta está vacía (void). Notesé que void no es un valor especial para referencias, sino uno de sus dos posibles estados. Una referencia está, o asociada con un objeto, o vacía.

Creación dinámica

Como oposición a la programación en lenguajes estáticos u orientados a bloques, la programación en un lenguaje orientado a objeto consiste en la creación dinámica de un número de objetos, de acuerdo con un patrón que es imposible predecir en tiempo de compilación.

En Eiffel, la ejecución de un sistema comienza con un objeto inicial, llamado root (raíz) del sistema. A partir de él, en ejecución, el sistema añadirá nuevos objetos.

Durante la evolución del sistema, puede ocurrir que algunos objetos se hagan, directa o indirectamente, inaccesibles desde el root. En implementaciones con lenguajes orientados a objetos, donde los sistemas pueden crear gran número de objetos y abandonar su uso en cualquier momento, es un problema fundamental reclamar el espacio consumido por estos.

II.1.3 clases

Hemos visto hasta ahora el comportamiento en tiempo de ejecución en un sistema. Veamos ahora la forma del texto de los programas que generan tal comportamiento.

Clases con atributos

Para describir objetos correctamente, se deberá hacer a través de una aproximación al tipo abstracto de datos. En lugar de describir objetos individuales, aquellos que sean comunes a un objeto (libros, autores, puntos, sólidos, etc.) deberán concentrarse en una clase. Las clases son la base de la programación orientada a objetos.

Una definición simple de una clase podría ser:

```
class LIBRO
feature
   titulo: STRING;
   fecha_publicacion: INTEGER;
   numero_paginas: INTEGER
end -- class LIBRO
```

Esta clase define la estructura de un conjunto de objetos (potencialmente infinitos). Tales objetos se llamarán *instancias* de la clase.

Es importante tener en mente la distinción entre objetos y clases: *los objetos* son elementos en tiempo de ejecución que se crearán durante la ejecución del sistema; *las clases* son descripciones puramente estáticas de un conjunto de posibles objetos. En tiempo de ejecución sólo tenemos objetos; en el programa sólo vemos clases.

La cláusula **feature...** introduce los componentes de una clase. Los componentes dados en este ejemplo son todos atributos; un atributo es un componente de una clase que corresponderá a un campo de cada objeto de la clase.

Tipos y referencias

Eiffel es un lenguaje tipificado estáticamente: cada atributo debe declararse con un único tipo.

Los atributos de la clase LIBRO se declararon todos de tipos predefinidos: INTEGER y STRING. El identificador STRING es reconocido como palabra reservada y es en la clase STRING donde se describen los strings de caracteres. Veamos ahora como se pueden declarar atributos que representen referencias.

```
class PERSONA
feature
   nombre,apellido,pais: STRING;
   anyo_nacimiento, anyo_muerte: INTEGER;
   casado: BOOLEAN
end -- class PERSONA
```

Podemos ahora incluir en la definición de la clase LIBRO una referencia a su autor:

```
class LIBRO
feature
   titulo: STRING;
   fecha_publicacion, numero_paginas: INTEGER;
   autor: PERSONA
end -- class LIBRO
```

Existen dos clases de tipos:

- Los cuatro tipos simples, llamados INTEGER, BOOLEAN, CHARACTER y REAL
- cualquier otro tipo debe definirse por una declaración de clase y se llamará *un tipo clase*.

Dado que no se permiten objetos dentro de objetos, los correspondientes campos objeto contienen referencias.

Usando las clases (clientes y proveedores)

Definición:

Una Clase A se dice que es cliente de una clase B, y B una clase proveedora de una clase A, siempre que A contenga una declaración del tipo e:B.

En esta definición la entidad es el identificador e y equivale en programación orientada a objeto a lo conocido como variable en la programación tradicional.

De este modo, la clase LIBRO del último ejemplo es un cliente de la clase PERSONA, pues en la definición de LIBRO aparece:

```
autor:PERSONA
```

Una clase puede ser su propio cliente:

```
class CLIENTE_DEL_BANCO
feature
   numero_cuenta, nombre, DNI: STRING;
   aval: CLIENTE_DEL_BANCO
```

Creación de objetos

Dado que, como ya se dijo, no se permiten objetos dentro de objetos, y por tanto, serán referencias, hasta que se haga algo al respecto, estas referencias permanecerán vacías. Para cambiar esta situación se debe crear un nuevo objeto del tipo clase apropiado y asociarlo con la referencia. Esto llevará a la referencia del estado vacío al estado llamado creado. Una referencia está asociada con un objeto si y sólo si está en estado creado.

Supongamos que la clase X es cliente de la clase LIBRO. Por ejemplo X contiene un atributo b de tipo LIBRO. Entonces todas las instancias de X incluirán un campo correspondiente a b.

Estos campos contendrán referencias que pueden estar durante la ejecución o bien vacias o haciendo referencia a algún objeto de tipo LIBRO.

Para pasar al estado *created* desde el estado *void* se utilizará la operación *create* de la siguiente manera:

```
b.Create
```

Disociando una referencia de un objeto

A veces, puede ser necesario disociar una referencia para ponerla al estado *void*. Esto se consigue poniendo:

```
b.Forget
```

La instrucción *Forget* debe realizarse sebre una referencia no sobre el objeto al que la referencia está asociada.

Una operación *Forget* no afecta al objeto en sí mismo, que podrá seguir siendo accesible a través de otras referencias. Si la referencia estaba vacía antes de la operación *Forget,* esta no tiene ningún efecto. En particular, *Forget* no tiene nada que ver con las instrucciones usadas para devolver espacio al sistema operativo, tales como *dispose* en PASCAL o *free* en C. En Eiffel, como en otros lenguajes orientados a objeto serios, la gestión de memoria es automática.

Estados de una referencia

Para conocer el estado de una referencia asociada con b, se puede usar el test

```
b.Void
```

que devuelve *true* si y sólo si la referencia está vacía.

No se deberán confundir los conceptos de objeto, referencia y entidad:

Objeto es una realidad en tiempo de ejecución: un objeto es una instanciación de una determinada clase, creada en tiempo de ejecución y constituido por ciertos campos.

Referencia es también una realidad en tiempo de ejecución: una referencia es un valor que puede estar vacío o denotando un objeto.

entidad no tiene existencia en tiempo de ejecución: una entidad es simplemente un nombre (identificador) que aparece en el texto de una clase, representando una o más referencias que pueden existir en tiempo de ejecución.

Inicialización

Una creación de un objeto como la supuesta con b.Create debe poner algún valor en los campos del objeto. Los campos se inicializan de acuerdo con el tipo de sus atributos correspondientes, tal y como sigue:

TIPO	VALOR INICIAL
INTEGER	0
BOOLEAN	false
CHARACTER	carácter nulo
REAL	0.0
TIPO CLASE	referencia vacía

Acceso a campos

Si dentro de la clase LIBRO queremos hacer referencia al atributo *titulo*, basta con usar el mismo identificador:

```
titulo := "Construccion de software  Orientado a Objeto"
```

pero si queremos acceder al atributo *apellido* de la entidad autor de tipo clase PERSONA de la que LIBRO es cliente debemos hacer:

```
autor.apellido := "MEYER"
```

Rutinas

Las clases definidas hasta ahora sólo tenían atributos, en otras palabras eran equivalentes a registros pero esto no es suficiente. Es necesario describir una implementación de un tipo abstracto de datos la cual debe incluir no solamente la representación de las instancias de tipos, sino también operaciones sobre esas instancias. Las rutinas son las implementaciones de las operaciones sobre las instancias de una clase.

Hay dos tipos de rutinas:

- **Procedimientos**: ejecutan una acción, es decir, pueden cambiar el estado de un objeto.
- **Funciones**: calculan algún valor deducido del estado del objeto.

La noción de estado usada aquí es simple: cada instancia de una clase tiene cierto número de campos, correspondientes a los atributos de la clase. Los valores de estos campos en cualquier punto durante la ejecución, determina el estado del objeto. Una llamada a un procedimiento puede cambiar este estado, es decir, el valor de uno o más campos. Una llamada a una función retorna un valor calculado del estado (es decir, de los atributos).

Cuando se introduce una nueva clase, a menudo es una buena idea mostrar primero como será usada dicha clase por los clientes, y sólo entonces proceder a su implementación. La clase escrita a continuación, llamada PUNTO, puede usarse de la siguiente manera. Supongamos que una clase cliente contiene las siguientes declaraciones:

```
p1, p2: PUNTO;
r, s: REAL
```

Entonces el cliente puede crear instancias de PUNTO:

```
p1.Create; ...; p2.Create; ...
```

Entre las características disponibles en la clase PUNTO están sus coordenadas; un ejemplo de instrucciones de almacenado de sus valores es:

```
r := p1.x;
s := p1.y
```

Aquí := es el símbolo de asignación. Las características x e y dan las coordenadas; el cliente no conoce si estas características están implementadas como atributos o funciones sin argumentos, y no necesita saberlo.

Las operaciones que pueden hacerse sobre puntos incluyen traslación y escalado:

```
p2.traslada(-3.5, s); ...; p2.escala(3.0)
```

Este ejemplo muestra la sintaxis general para la ejecución de operaciones, usando la notación de punto:

```
entidad.operación(argumentos)
```

que significa: "*aplica operación, con argumentos, al objeto asociado con entidad*". Puede no haber argumentos. Notesé una vez más la característica distintiva de la programación orientada a objetos: cada operación se aplica a un objeto específico, que aparece a la izquierda del punto (representado por entidad). En programación clásica, la notación correspondiente será la más simétrica

operación(entidad,argumentos)

es decir, la entidad será uno más de entre los argumentos de operación.

Aquí la primera operación del ejemplo ejecuta sobre p2 una traslación de -3.5 horizontalmente y el valor de s verticalmente; el segundo escala p2 por un factor de tres desde el origen. Ambas operaciones son procedimientos, que pueden modificar el objeto asociado con la referencia a la cual se aplican. Finalmente se puede aplicar a un punto la función distancia, que retorna su distancia a otro punto, tal como

```
r := p1.distancia(p2)
```

En contraste a procedimientos, las funciones normalmente no modifican el objeto asociado con la referencia a la cual se aplican, pero devuelven algún valor acerca de ese objeto.

Cada operación se aplica a cierto objeto, accedido a través de una referencia denotada por una entidad. Para que tenga significado la operación, el objeto debe existir. Así, si entidad.operacion(argumentos) ha de ejecutarse correctamente, es esencial que la referencia asociada con entidad no esté vacía void. Esto no se refiere sólo a la ejecución de rutinas, sino también al acceso a atributos, como en entidad.atributo. La principal causa de errores en tiempo de ejecución en la programación Eiffel es el intento de acceder a una referencia vacía.

Hemos visto cómo la clase PUNTO puede ser usada por clientes. Veamos ahora cómo se escribe en sí misma.

```
class PUNTO export
    x, y, traslada, escala, distancia
feature
    x, y : REAL;
    escala (factor:REAL) is
      -- escala en un ratio de factor
    do
       x := factor*x;
       y := factor*y
    end; -- escala
    traslada (a, b:REAL) is
        -- mueve a horizonta, b vertical
    do
      x := x+a;
      y := y+b
    end; -- traslada
    distancia (otro: PUNTO): REAL is
        -- distancia a otro
    do
      Result:=sqrt((x - otro.x)^2+(y - otro.y)^2)
    end -- distancia
end -- class PUNTO
```

Esta clase muestra muchas de las propiedades básicas de las clases y sus características.

La cláusula **export** lista las características que están disponibles a los clientes. Esta cláusula se necesita para forzar una distinción clara entre características *internas*, usadas sólo por la clase actual y las *públicas*, ofrecidas al mundo exterior. Esto potencia el principio de ocultación de la información.

En una clausula B la aparición de una secuencia como:

```
a1 : A;
...
a1.f ...
```

sólo será válida si f aparece en la cláusula **export** de la class A. Esto es aplicable tanto a atributos como a rutinas.

las funciones pueden no tener argumentos formales. Por ejemplo, en la clase PUNTO podría definirse la función:

```
distancia_al _origen: REAL is
     -- calcula la distancia al punto (0,0)
do
   Result := sqrt(x^2 + y^2)
end   -- distancia_al _origen
```

La declaración de una función sin argumentos es muy similar a la declaración de un atributo.

En la clase PUNTO podríamos tener un par de característivas más como son las coordenadas polares: ro y theta y desde el punto de vista de una clase cliente, la naturaleza de estas dos nuevas características debe ser irrelevante: podrían ser funciones sin argumentos, calculadas a partir de los atributos x e y

```
ro is
   -- módulo
do
   Result := sqrt(x^2 + y^2)
end -- ro
```

```
theta is
    -- ángulo
do
   Result := arctan(x/y)
end -- theta
```

o atributos, en cuyo caso, las coordenadas cartesianas x e y podrían ser las funciones:

```
x is
   -- abscisa
do
  Result := ro * sin(theta)
end -- x
```

```
y is
   -- ordenada
do
  Result := ro * cos (theta)
end -- y
```

Variables locales

La implementación de distancia_al_origen podría haber usado una función más general distancia

```
distacia_al_origen:REAL is
-- distancia al origen
local
   origen : PUNTO
do
  origen.Create;
  Result:=distancia(origen)
end -- distancia_al_origen
```

Esta versión introduce el concepto de variable local: la cláusula opcional local, al comienzo del cuerpo de la rutina, contiene cualquier entidad local que pueda ser útil para la ejecución de la rutina. Una variable local sólo es accesible dentro de la rutina a la cual pertenece.

II.1.4 Entidades

Una entidad es una de las siguientes cosas:

1. Un atributo de clase.
2. Una variable local de una rutina.
3. Un argumento formal de una rutina.
4. Un identificador denotando el resultado de una función, como la expresada por la entidad predefinida *Result*.

Rutinas Predefinidas

Además de las rutinas definidas en una clase, algunas rutinas están predefinidas y se aplican a todas las clases. Esto significa que los nombres correspondientes son palabras reservadas. Hay cinco rutinas predefinidas:

```
Create
Forget
Void
Clone
Equal
```

Create, Forget y Void ya han sido mencionadas, clone duplica objetos y equal compara objetos. Veamos más detalladamente las dos últimas.

CLONE

La llamada

```
a.Clone(b)
```

crea un objeto idéntico al objeto apuntado por b y hace que a apunte a dicho objeto, o hace que a sea Void si lo es b. Las entidades a y b deben estar declaradas como del mismo tipo de clase. La ejecución de Clone es la segunda forma de crear objetos; la primera es ejecutar Create, y no hay otras.

EQUAL

Si a y b son dos entidades del mismo tipo, entonces a.Equal(b) es TRUE si y sólo si la referencia asociada con a y b son, o ambas Void, o se refieren a objetos que son campo a campo iguales. Nótese que este es un test de igualdad entre objetos, y no entre referencias como ocurre con a = b

Se debe tener presente que:

```
a=b => a.Equal(b)
a.Equal(b) <> a=b
```

Create explícito (no por defecto)

En algunos casos podemos necesitar una inicialización más flexible de los objetos en el momento de su creación, por ejemplo asignando valores iniciales determinados a los atributos del nuevo objeto. Se pueden definir tales inicializaciones específicas definiendo un procedimiento Create para la clase. Un procedimiento con el nombre Create se reconoce como especial y será usado para la inicialización del objeto. Se puede, por ejemplo, añadir tal procedimiento a la clase PUNTO

```
class PUNTO exports
    x, y, traslada, escala, distancia
feature
    x, y : REAL;
    Create (a, b:REAL) is
        -- inicializa el punto con las coordenadas a y b
    do
      x:=a; y:=b
    end; --Create
      -- resto de features ...
end  -- class PUNTO
```

Si se declara así, los clientes lo invocarán con los actuales argumentos, tal como

```
p: PUNTO;
...
p.Create(34.6,-65.1)
```

Referencia y valor semántico

Una asignación de la forma a:=b es una asignación de valores si a y b son de tipos simples (integer, real, character, boolean); es una asignación de referencias, no una asignación de objetos, si son de tipo class. De modo similar, un test de igualdad o desigualdad a=b o a/=b es una comparación de valores para entidades de tipos simples; es una comparación de referencias a objetos (no una comparación de objetos) para entidades de tipo class.

II.2 DE CLASES A SISTEMAS

Hasta ahora no hemos hablado de qué cosas deben ocurrir al comienzo de ejecución de un programa Eiffel. Cada objeto es una instancia de una clase. Al comienzo el objeto debe asociarse con una entidad, por ejemplo un atributo a y debe ejecutarse una instrucción de la forma

```
a.Create
```

a tiene que estar declarada en alguna clase C y create debe ser ejecutada por alguna rutina r de C. Por tanto a representa un campo de la instancia actual de C, esto significa que *alguien* debe haber creado una instancia de C y aplicado r sobre ella. Este *alguien* volverá a ser una rutina relativa a algún objeto. Así pues, ¿ cómo crear un objeto inicial por primera vez ?

Sistemas

La ausencia de la noción de programa principal en Eiffel es una importante característica del diseño del lenguaje. Es un error, si se está comprometido con los principios de reusabilidad y extensibilidad, organizar el diseño de un sistema alrededor de su función principal. Por contra Eiffel enfatiza el diseño de componentes de software reutilizables, construidos como implementaciones de tipos abstractos de datos-clases.

El proceso de unir clases con vistas a un resultado de ejecución se llama **ensamblado** y es el último paso en la fase de diseño de software. El resultado de tal ensamblado de clases autónomas se llama sistema.

Ensamblando sistemas

Un sistema se caracteriza por una raíz, que es el nombre de la clase. La ejecución del sistema consiste en crear una instancia de dicha clase y ejecutar su procedimiento Create. Este procedimiento, normalmente, creará otros objetos y ejecutará otros procedimientos.

El Fichero de Descripción del Sistema (SDF)

Para que el proceso de ensamblaje quede claro, veamos cómo se ensambla y ejecuta realmente un sistema utilizando Eiffel/s

El proceso comienza con la orden

ecc <nombre del programa>

<nombre del programa> es un nombre sin extensión pues es el nombre del programa ejecutable.

Esta orden necesita un SDF, el equivalente en Eiffel/s es un fichero con extensión PDL y nombre el de la clase raíz. Cuando no existe ecc genera uno. Para crear uno nuevo simplemente se deben rellenar los datos que aparecen y que son los siguientes

nombre de la clase raiz: <nombre>
nombre del procedimiento crear: <nombre_cr>

<nombre> debe coincidir con el nombre del programa ya que los ficheros en eiffel contienen únicamente una clase.

<nombre_cr> puede ser el nombre *create* u otro nombre y dede ser el nombre de un procedimiento que se encuentre en la clase raiz. Se diferencia este procedimiento del resto debido a que en la clase raiz se tiene que poner lo siguiente

```
    creation
  <nombre_cr>
```

A continuación se presenta el contenido de un fichero .PDL

```
program listrace

root
        listrace : "make"

cluster
  "./"
  find
    "linked_list" in "lnk_list.e",
    "link_node"   in "lnk_node.e",
    "link_iter"   in "lnk_iter.e"
  end

  "$EIFFEL_S/library/basic"
    find
      "environment"   in "environ.e",
      "exception"     in "except.e",
      "comparable"    in "compar.e",
      "integer_ref"   in "int_ref.e",
      "character_ref" in "char_ref.e",
      "boolean_ref"   in "bool_ref.e",
      "file_system"   in "file_sys.e",
      "system_time"   in "sys_time.e",
      "character"     in "characte.e"
  end
  "$EIFFEL_S/library/contain"
    find
      "dictionary"         in "dictnary.e",
      "hash_table"         in "h_table.e",
      "traversable"        in "travers.e",
      "sorted_table"       in "s_table.e",
      "sorted_list"        in "s_list.e",
      "twoway_iter"        in "tw_iter.e",
      "sort_table"         in "srt_tbl.e",
      "collection"         in "colctn.e",
      "sorted_collection"  in "srt_cln.e"
      "traversable"        in "travers.e",
      "twoway_traversable" in "tw_trvrs.e",
      "list_catalog"       in "l_catlg.e",
      "sorted_catalog"     in "s_catlg.e",
      "short_table"        in "sh_table.e",
      "short_sorted_list"  in "shs_list.e",
      "short_sorted_table" in "shs_tbl.e",
      "simple_table"       in "smp_tbl.e",
```

```
        "priority_queue"     in "prqueue.e",
        "key_priority_queue" in "kprqueue.e"
    end
    "$EIFFEL_S/library/math"
    end
end -- program listrace
```

en este caso el nombre de la clase raíz es *listrace* y el nombre del procedimiento crear *make*

Listrace también es el nombre del programa ejecutable y cuando se llama se crea automáticamente una instancia de la clase raíz. El objeto así creado es el encargado de llamar al procedimiento crear.

II.3 CLASES VERSUS OBJETOS

La cuestión es sencilla: un objeto es una instancia de un tipo. El concepto de clase es estático: es un elemento reconocible en el texto del programa. Por contra el objeto es puramente dinámico, que no pertenece al texto del programa sino a la memoria del ordenador en el momento de la ejecución.

Formas de declaración

La forma de una característica (feature) determina si es un atributo, función o procedimiento. Así tendremos:

```
f(a1:A; b1:B;...) is ...  -> Procedimiento
f(...):Tipo is ...        -> Función
f:Tipo;           -> Atributo
```

Atributos vs. Funciones

El hecho de agrupar rutinas y atributos en la misma categoría es intencionada. Es aplicación directa del **principio de referencia uniforme.** Este principio indica que un cliente de un módulo deberá tener disponible un servicio promovido por el módulo de una forma uniforme, sin importarle si este servicio está implementado sobre almacenamiento o sobre computación. Por ejemplo, un atributo almacenado en un objeto o una función que calcula un valor. De esta forma la notación usada para acceder a un atributo es el misma que para acceder a una rutina

Resultado de funciones

Otro de los aspectos sintácticos del Eiffel, independiente del método de diseño orientado a objeto, es la convención de computación del resultado de una función, usando la variable predefinida Result. Esta entidad predefinida sirve para denotar sin ambigüedad el resultado de la función en curso. Así Result está sujeta a las reglas de inicialización por defecto de todas las entidades. Por ejemplo, una función que devuelva el resultado de un tipo clase puede tener un cuerpo de la forma:

```
    if <condición> then
  Result.Create(<argumentos>)
end
```

si <condición> es falsa el valor devuelto será será una referencia a *Void* ya que es el valor por defecto de del tipo clase.

II.4 GENERICIDAD

Eiffel incorpora técnicas dirigidas a potenciar la flexibilidad y reusabilidad de los módulos. Dando parámetros de clases que representen tipos arbitrarios evitamos la escritura de muchas clases idénticas en diseño pero para tipos diferentes.

Parametrización de clases

Genericidad es la capacidad de parametrizar clases. La necesidad de esta facilidad es evidente en el caso de clases que representan estructuras de datos generales: arrays, listas, árboles, etc.

Si nos referimos al tradicional ejemplo de implementación de la clase PILA, resulta evidente que si no disponemos de genericidad, tendremos que repetir la escritura de clases para cada tipo de pila: pila de enteros, pila de reales etc. La representación de los datos y algoritmos no se ven afectados por el tipo de la pila. Así con el uso de la genericidad es posible escribir una pila de elementos de un tipo arbitrario T como la siguiente:

```
class PILA[T] export
    nb_elem, vacia, llena, push, pop, top
feature
    nb_elem : INTEGER;
    vacia:BOOLEAN is do ... end;
    llena:BOOLEAN is do ... end;
    push (x:T) is do ... end;
```

```
        pop is do ... end;
        top:T is do ... end;
end -- class[T]
```

Los nombres dados entre corchetes tras el nombre de la clase se llaman parámetros formales de clase. Puede haber más de uno; en tal caso se separan con comas.

Un cliente puede usar una clase genérica para definir entidades. En tal caso la declaración debe proveer tipos llamados parámetros genéricos actuales, correspondiendo en número a los parámetros genéricos formales. Por ejemplo:

```
sp: PILA[PUNTO];
```

Un parámetro genérico actual es uno de los siguientes:

1. Un tipo simple (INTEGER, REAL, CHARACTER, BOOLEAN).
2. Un tipo clase.
3. Un parámetro genérico formal de la clase cliente.

Chequeo de tipos

De hecho, la genericidad sólo tiene significado en un lenguaje tipeado, donde cada entidad se declara como de un determinado tipo, de ese modo se puede determinar si una operación es correcta desde el punto de vista de tipos, bien mirando el texto del programa, o chequeando en tiempo de ejecución. Eiffel está tipeado estáticamente, donde todo el control de tipos lo realiza el compilador.

Operaciones sobre entidades de tipos genéricos

Consideremos la clase C[U,V] y una entidad cuyo tipo es uno de los parámetros genéricos formales, por ejemplo x de tipo U. Cuando la clase se usa por un cliente para declarar entidades, U puede representar en última instancia cualquier tipo *simple* o *clase.* Así, cualquier operación sobre x debe ser aplicable a todos los tipos. Esto permite sólo cuatro tipos de operaciones:

1. Uso de x en la parte izquierda de una asignación, x:=y, donde y debe ser también de tipo U.
2. Uso de x en la parte derecha de una asignación, y:=x, donde y es también de tipo U.
3. Uso de x como argumento actual en una llamada a rutina f(..., x, ...), correspondiente a un argumento formal declarado de tipo U (lo que implica que f ha de ser una rutina declarada en la misma clase que x).
4. Uso de x en una expresión booleana de la forma x=y o x/=y donde y es también de tipo U.

En particular, operaciones tales como x.Create, x.Forget o cualquier otra aplicación de característica sobre x, son ilegales, dado que no está garantizado que el parámetro genérico actual provisto por U será un tipo clase.

II.5 METODOS SISTEMÁTICOS PARA LA CONSTRUCCIÓN DE SOFTWARE

En este momento podemos escribir módulos de software que implementan, posiblemente, estructuras de datos parametrizadas. Este es un paso significativo en la batalla de construcción de mejores arquitecturas de software. Pero las técnicas vistas hasta aquí no son suficientes para implementar una comprensiva visión de calidad. Los factores de calidad más pregonados en el paradigma de orientación a objetos *reusabilidad, extensibilidad, compatibilidad* no deben lograrse a expensas de la corrección y de la robustez. Lo concerniente a la corrección estaba, por supuesto, reflejado en el énfasis puesto en un chequeo estricto de tipos. Pero se necesita algo más.

En realidad, las clases que hemos visto hasta ahora no son más que conjuntos de atributos y rutinas, que pueden servir para representar las funciones de una especificación de tipo abstracto de datos. Pero un tipo abstracto de datos es más que una lista de operaciones disponibles: es necesario resaltar el papel importante que juegan las propiedades semánticas expresadas por axiomas y precondiciones. Estas son fundamentales para capturar las propiedades fundamentales de las instancias de los tipos. Deberemos dejar disponible este concepto en el método de diseño orientado a objetos si queremos que nuestro software sea no sólo flexible y reusable, sino también correcto.

Las aserciones y conceptos afines aportan algunas respuestas. El mecanismo que se expone provee al programador de herramientas esenciales para expresar y validar argumentos correctos. La noción clave aquí será el concepto de programación por contrato: las relación entre una clase y su cliente se ve como un contrato formal que expresa para cada parte derechos y obligaciones.

Revisando estos conceptos nos encontraremos con un problema clave de la ingeniería del software: cómo tratar los errores en tiempo de ejecución. Se presentará un método disciplinado de tratamiento de excepciones.

El Concepto de Aserción

Una aserción es una propiedad de alguno de los valores de las entidades del programa. Por ejemplo, una aserción puede expresar que ciertos enteros tienen valor positivo o que cierta referencia está vacía.

Sintácticamente, las aserciones no son más que expresiones booleanas, con algunas extensiones. La primera extensión es el uso del punto y coma:

```
n>0; not x.Void
```

El significado del punto y como es equivalente al del **and.** Facilita la identificación de los componentes individuales de la aserción. Estos componentes pueden estar etiquetados:

```
Positivo: n>0; No_Vacio: not x.Void
```

Las aserciones son una poderosa herramienta de depuración. Son, además, la base de un disciplinado manejo de excepciones, posibilitando al sistema el intento de recuperación tras el fallo. Así pues, las aserciones se estudian como una técnica de construcción de sistemas de software correctos y robustos y de documentación de por qué lo son.

II.6 PRECONDICIONES Y POSTCONDICIONES

El primer uso de las aserciones es la especificación semántica de las rutinas. Una rutina no es sólo una porción de código; como implementación de alguna función de una especificación de un tipo abstracto de datos, debería efectuar alguna tarea útil. Es esencial expresar esta tarea de manera precisa, en dos aspectos: uno, como ayuda al diseño y otro, como ayuda a la comprensión de su texto.

La tarea efectuada por una rutina puede especificarse por dos aserciones asociadas con la rutina: una precondición y una postcondición. La precondición expresa las propiedades que se deben cumplir cuando se llama a la rutina; la postcondición describe las propiedades que garantiza la rutina cuando retorna la rutina.

Por ejemplo, en una clase PILA tendremos las siguientes aserciones con las precondiciones, representadas como **require** y las postcondiones, representadas como **ensure**

```
class PILA1[T] export
        nb_elem, vacia, llena, push, pop, top
feature
        nb_elem: INTEGER;
        vacia: BOOLEAN is do ... end;
        llena: BOOLEAN is do ... end;
        push(x:T) is
            require
        not llena
            do ...
            ensure
              not vacia;
              top = x;
              nb_elem = old nb_elem + 1;
            end; -- push
        pop is
            require
              not vacia
            do ...
            ensure
              not llena;
              nb_elem = old nb_elem -1
            end; _ pop
        top: T  is
            require
              not vacia
            do ...
            end; -- top
        end -- class PILA[T]
```

Una clase importante de la librería básica de Eiffel es ARRAY que debe verse como una clase más. Un esquema de su definición es el siguiente:

```
class ARRAY[T] export
    lower, upper, size, entry, enter
feature
  lower:INTEGER;
  upper:INTEGER
  size:INTEGER;

  Create(minb, maxb:INTEGER) is
      -- crea un array con limites minb y maxb
      -- Si minb > maxb el array será vacio
    do ...
    end;

  entry(i:INTEGER):T is
      -- Devuelve el elemento de indice i
    require
      lower <= i;
      i <= upper
    do ...
  end;  -- entry

  enter(i:INTEGER; value:T) is
      -- Asigna value a la posición de índice i del array
    require
      lower <= i;
      i <= upper
    do ...
    ensure
        entry(i) = value
  end;  -- enter
invariant
   size=upper-lower+1;
   size >= 0
end;   -- clase ARRAY
```

Veamos otro ejemplo de la clase PILA haciendo uso de la clase ARRAY

```
class PILA2[T] export
      annadir, extraer, tope, vacia, llena, num_elementos
feature
      implementation: ARRAY[T];
      max_tama: INTEGER;
      num_elementos: INTEGER;
      Create (n:INTEGER) is
          -- Asignar a la pila un máximo de n elementos
          -- o ninguno si n > 0
        do
          if n > 0 then max_tam:= n end;
          implementacion.create (1, max_tam)
       end; -- Create

      vacia:BOOLEAN is
           -- ¿Está la pila vacia ?
        do
          Result:=(num_elementos = 0)
        end; --vacia
```

```
llena:BOOLEAN is
        -- ¿Está la pila llena ?
  do
     Result:=(num_elementos = max_tam)
  end; --llena

extraer is
     -- Quitar elemento del tope
  require
     not vacia      -- num_elementos > 0
  do
     num_elementos:= num_elementos-1
  ensure
    not llena
    numelementos = old num_elementos -1
  end; --extraer

tope is
        -- Devuelve el tope de la pila
    require
       not vacia    -- num_elementos > 0
    do
      Result:= implementation.entry(num_elementos)
    end; -- tope

annadir(x:T)  is
         -- Añadir x sobre el tope de la pila
    require
       not llena         -- num_elementos < max_tam
    do
       num_elementos:= num_elementos + 1;
       implementation.enter(num_elementos, x);
    ensure
      not vacia;
      tope = x;
      num_elementos = old num_elementos + 1
    end; -- añadir
end -- class PILA2[T]
```

Ambas cláusulas, **require** y **ensure,** son opcionales.

La palabra **old** seguida de un atributo, denota el valor que el correspondiente campo del objeto tenía a la entrada de la rutina.

Invariantes de clases y exactitud de clases

Las precondiciones y postcondiciones describen las propiedades de rutinas individuales. Pero es necesario expresar propiedades globales de las instancias de una clase, que deberán ser preservadas por todas las rutinas.

Una invariante de clase es una lista de aserciones, que expresa una coacción de consistencia general que se aplica a todas las instancias de la clase. Estas aserciones se refieren exclusivamente a atributos.

Sintácticamente, una invariante de clase es una aserción que aparece en la cláusula **invariant** de la clase, tal como

```
class PILA[T] export ... feature
                implementacion: ARRAY[T];
                max_tama: INTEGER;
                nb_elem: INTEGER;
               ... otras características ...
        invariant
               0<=nb_elem; nb_elem <= max_tama;
               vacia = (nb_elem = 0)
end -- class PILA[T]
```

Una invariante para una clase C es un conjunto de aserciones que ha de ser satisfecha por cada instancia de C en todo momento *estable*. Momentos estables se definen como aquellos en los cuales las instancias están:

- En la creación de la instancia, es decir, después de la ejecución de una llamada de la forma *a.Create,* donde a es de tipo C.
- Antes y después de cualquier llamada remota a.r(...) de una rutina de la clase.

Un invariante puede ser temporalmente violado durante la ejecución de una llamada remota, siempre que la rutina lo restaure antes de salir. Como consecuencia un invariante de clase sólo se aplica a rutinas exportables: a las rutinas secretas, que no son ejecutadas directamente por clientes y sólo sirven como herramientas auxiliares, no se les exige el mantenimiento del invariante.

Otras construcciones que contienen aserciones

Las aserciones pueden usarse no sólo en precondiciones, postcondiciones e invariantes de clases, como vimos antes, sino también en bucles y en la instrucción especial check.

Variante e invariante de bucle

Un bucle puede contener un invariante de bucle y un variante de bucle. Estas nociones, aunque importantes, son independientes de los conceptos de orientación a objetos.

La forma general de un bucle Eiffel es

```
from instrucciones_de_inicialización
invariant invariante
variant variante
until condición_de_salida
loop instrucciones_de_bucle
end
```

Las cláusulas **invariant** y **variant** son opcionales. La cláusula **from** es necesaria (aunque puede estar vacía); marca la inicialización del bucle, que se considera formando parte del bucle.

Como ejemplo simple, que no incluye todavía variante o invariante, se muestra una función que calcula el máximo común divisor (mcd) de dos enteros positivos a y b por el algoritmo de Euclides, usando un bucle:

```
mcd(a, b: INTEGER): INTEGER is
        -- máximo común divisor de a y b
        require a>0; b>0
        do
            from  x:=a; y:=b
            until  x=y
            loop
               if  x>y  then  x:=x-y  else  y:=y-x  end
            end;
            Result:=x
        ensure  -- Result es el máximo común divisor de a y b
end;  -- mcd
```

Notesé que este bucle se corresponde (en Pascal, C etc.) a un bucle while, en el cual el cuerpo del bucle se ejecuta cero o más veces.

Los variantes e invariantes se utilizan para expresar y verificar la corrección de un bucle con respecto a su especificación. Un modo de chequear que la función verifica su postcondición, es decir, que calcula el máximo común divisor de a y b, es darse cuenta de que la siguiente expresión es cierta tras la inicialización del bucle y que se preserva en cada iteración:

x>0; y>0
-- (x,y) tienen el mismo máximo común divisor que (a,b)

De modo más general, un invariante correcto para un bucle es una aserción que se satisface tras la inicialización del bucle y que se preserva en cada iteración del cuerpo del bucle.

La presencia de un invariante correcto no es suficiente para hacer correcto el bucle: se debe estar seguro de que el bucle termina. Esto se obtiene exhibiendo un adecuado variante de bucle. Un variante es una expresión entera cuyo valor es no negativo tras la inicialización del bucle, y que decrece en al menos una unidad en cada ejecución del cuerpo del bucle, pero que nunca es negativo. Claramente, la presencia de variante, asegura la terminación del bucle dado que el proceso no podrá decrecer infinitamente hacia una expresión entera no negativa.

En el bucle expuesto anteriormente, max(x, y) es un variante apropiado. Ahora podemos escribir el bucle con todas sus cláusulas:

```
from  x:=a; y:=b
invariant  x>0;  y>0
variant  max(x, y)
until  x=y
loop
   if  x>y  then  x:=x-y  else  y:=y-x  end
end;
```

La instrucción Check

Otra instrucción más que puede utilizarse en conjunción con las aserciones es la siguiente

```
check
    aserción 1;
    aserción 2;
    ...
    aserción n;
end
```

y significa: *en este punto del código, se espera que siempre se satisfagan las siguientes aserciones*.

La instrucción check es una manera de asegurarse de que ciertas propiedades se satisfacen.

USANDO ASERCIONES

¿ Por qué usar aserciones?. Hay cuatro aplicaciones principales:

- Ayuda a la escritura de software correcto.
- Ayuda a la documentación.
- Herramienta de depuración.
- Soporte de software tolerante a fallos.

• Las aserciones como herramienta de escritura de software correcto

Indicar los requerimientos exactos de cada rutina, y las propiedades globales de clases y bucles, ayudan al programador a desarrollar software que sea correcto.

La idea clave ha de ser enfatizada de nuevo, esta idea es el principio de **programación por contrato.** Usar características de cierto módulo es contratar servicios. No se puede esperar de un contrato que cubra todos los casos posibles; los buenos contratos son aquellos que expresan claramente los derechos y obligaciones de cada parte, y los límites de estos derechos y obligaciones.

• Las aserciones como documentación

Este uso es esencial en la producción de elementos de software reusables y, de modo más general, en la organización de interfaces de módulos en grandes sistemas de software. Las precondiciones, postcondiciones e invariantes de clases proveen a los clientes potenciales de un módulo información crucial sobre los servicios ofertados por el módulo, expresados de manera clara y concisa.

Además de una ayuda conceptual al diseño de programas, las aserciones tienen un efecto en su ejecución. Es posible chequear aserciones en tiempo de ejecución; esto provee dos mecanismos: depuración, y tolerancia a fallos y recuperación de fallos.

• Detectando violaciones

¿ Qué ocurre cuando uno de los modos de chequeo se encuentra con una violación de aserción en ejecución?. Si no se ha especificado ninguna acción para el caso dado, se mostrará mensaje de error y se detendrá el sistema.

Excepciones

En algunos casos no es aceptable saber que un fallo degerará en un mensaje y en la terminación sin más del sistema. Se puede tomar el control. Tales recuperaciones de fallos en ejecución son el fin de los **mecanismos de excepción.**

Las excepciones son una técnica potencialmente peligrosa. A menudo las excepciones se usan simplemente como una forma de instrucción goto que hace posible saltar fuera de la rutina cuando se encuentra alguna condición diferente del caso estándar.

Permanece un rol de mecanismos de excepciones, que sin embargo refleja dos situaciones:

• Cuando un programador quiere tener en cuenta la posibilidad de que permanezcan errores en el software, e incluir mecanismos que manejen cualquier fallo en ejecución, bien saliendo limpiamente, bien intentando la recuperación.

• Cuando un fallo se debe a alguna condición anormal detectada en el hardware o en el sistema operativo.

La violación de aserciones se corresponde con la primera situación.

Tratemos ahora de diferenciar excepción, fallo y error.

Una **excepción** es la aparición de una condición anormal durante la ejecución de un elemento de software.

Un **fallo** es la imposibilidad de un elemento de software para satisfacer su propósito.

Un **error** es la presencia en el software de algunos elementos que no satisfacen su especificación.

Nótese que los fallos causan excepciones, y se deben generalmente a errores. En Eiffel, una excepción ocurrirá incluso si aparece una violación de una aserción de una rutina; se asume que se han seleccionado las correspondientes opciones de compilación para habilitar el control de todas o parte de las excepciones.

Sólo dos cosas razonables se pueden hacer cuando se detecta una excepción:

• Limpiar el entorno, terminar y avisar del fallo al usuario. La limpieza consiste en llevar al entorno a un estado estable. Esto puede llamarse respuesta de pánico organizado a una excepción. Es esencial en tal caso avisar del fallo al usuario.

• Intentar cambiar la condición que causó la excepción y reintentar la operación que falló. Esta operación puede llamarse reintento (*retry*).

Una solución que no es aceptable es aquella en que habiendo detectado un fallo, la rutina devuelva el control al cliente sin ninguna notificación especial. Esto desbarataría el propósito de cualquier intento de corrección y robustez.

El mecanismo de excepciones en Eiffel se basa en estas consideraciones. Una rutina puede o no fallar. No fallará si cumple su contrato.

Se necesitan dos extensiones sintácticas para describir los detalles de este mecanismo. La primera es la cláusula opcional **rescue** (cláusula de rescate) que puede aparecer en una rutina tras el cuerpo, precondición y postcondición, si las hay.

```
rutina is
    require ...
    local ...
    do
      cuerpo
    ensure ...
    rescue
       cláusula_de_rescate
end
```

La cláusula_de_rescate es una secuencia de instrucciones. Siempre que se provoque una excepción durante la ejecución de cuerpo, se detendrá esta ejecución y en su lugar se ejecutará la cláusula_de_rescate.

La otra nueva construcción es la instrucción **retry.** Esta instrucción sólo puede ser ejecutada como parte de una cláusula de rescate de un rutina aunque también puede estar en una rutina que sea llamada por una de las instrucciones de esta cláusula. Su ejecución consiste en comenzar el cuerpo de la rutina desde el principio.

Siempre que la cláusula de rescate se ejecute hasta el final, sin ejecutarse la instrucción retry, fallará la ejecución de la rutina; se avisará del fallo al usuario a través de una excepción. Una rutina sin cláusula de rescate se considerará poseedora de una cláusula vacía y, por tanto, todas las excepciones causarán el fallo inmediato de la rutina. Sin embargo se puede incluir una cláusula de rescate a nivel de clase, justo antes del invariante, y será usado por cualquier rutina de la clase que no tenga cláusula propia.

He aquí la definición en Eiffel para excepciones. Una excepción puede ocurrir durante la ejecución de una rutina como resultado de cualquiera de las siguientes situaciones:

1. Se detecta la violación de la precondición de r a la entrada.
2. Se detecta la violación de la postcondición de r cuando r termina.
3. Se detecta la violación de la invariante de clase a la entrada o terminación.

4. Se detecta otra violación de aserción (check violado, no mantenimiento de invariante de bucle en una iteración, variante de bucle no decrementada), durante la ejecución de una rutina.
5. Fallo de una rutina llamada por r.
6. r intenta una aplicación de característica remota a.f en un estado donde a es una referencia vacía (Void).
7. Una operación ejecutada por r resulta en una condición anormal detectada por el hardware o por el sistema operativo.

Regla de rescate: La cláusula de rescate para una rutina debe ser correcta con respecto a la precondición verdadera **true** y, excepto para cualquier rama acabada en **retry,** a la postcondición dada por la invariante de clase.

El reintento en un software tolerante a fallos

Un ejemplo de reintento es una implementación de la técnica de *programación de la versión n-ésima* sugerida por algunos autores como una herramienta para tolerancia a fallos. La idea es ofrecer dos o más implementaciones para una tarea dada, desarrolladas por diferentes equipos en entornos que sean tan diferentes como sea posible, esperando que los errores, de producirse, serán diferentes. Veamos de qué manera tan simple el mecanismo de **retry** soluciona este problema:

```
haz_tarea is
   require ...
   local
       num_intentos: INTEGER;
   do
       if num_intentos = 0 then
           implementación_1;
       elseif num_intentos = 1 then
           implementación_2
       end
   ensure ...
   rescue
       num_intentos:=num_intentos + 1;
       if num_intentos<2 then
          "quizás algunas instrucciones para retornar el estado estable"
       retry
  end
end -- haz_tarea
```

La generalización de más de una alternativa de implementación es inmediata.

II.7 MÁS ASPECTOS DE EIFFEL

Se recomiendan ciertas convenciones simples de estilo, aunque no forman parte propiamente del lenguaje.

Comentarios de cabecera

La convención más importante se refiere a los comentarios en las rutinas. El comentario de cabecera, indentado como se muestra a continuación, se mostrará en cada rutina; por ejemplo:

```
distancia_al_origen: REAL is
      -- Distancia al punto (0,0)
   local
       origen: PUNTO
   do
       origen.Create;
       Result:=distancia(origen)
end -- distancia_al_origen
```

Tal comentario de cabecera debería ser informativo, claro y conciso. En general, la brevedad es una cualidad esencial de los comentarios en los programas.

Disposición

La disposición recomendada en el texto del Eiffel resulta de la forma general de la sintaxis del Eiffel. Como ejemplo, dependiendo del tamaño de sus constituyentes a, b y c, una instrucción condicional se escribiría de las siguientes formas:

```
if  c  then  a  else  b  end
```

o anidado de diversas formas. Por ejemplo:

```
if
   c
then
   a
else
   b
end
```

Identificadores

Eiffel usa el conjunto de caracteres ASCII estándar. Losidentificadores son secuencias de caracteres, todos los cualesdeben ser letras, dígitos o caracteres de subrayado ('_'); el primer carácter de un identificador debe ser una letra.

Tamaño de las letras en los identificadores

Se admiten letras mayúsculas y minúsculas en los identificadores; así Hi, hi, HI y hI representarán al mismo identificador.

Se recomiendan ciertas convenciones estándar para aumentar la legibilidad de los programas. Son las siguientes:

• Los nombres de tipos, tales como los tipos simples (INTEGER, etc.), nombres de clases (PUNTO) y parámetros formales genéricos de clases (tal como T en las clase PILA[T]) se escribirán en letras mayúsculas.

• Los nombres de características predefinidas (Create, Void, Clone, Forget), entidades y expresiones predefinidas (Result y Current) y constantes simbólicas definidas por el programador (tales como Pi), comenzarán por letra mayúscula.

• El resto de identificadores se escribirán en letra minúscula: atributos con constantes, argumentos formales de rutinas, variables locales.

Elección de nombres

Eiffel tiene un conjunto de palabras reservadas o identificadores predefinidos en la definición del lenguaje, que no pueden usarse en otros elementos del lenguaje.

Lista de palabras reservadas:

```
and
as
BOOLEAN
check
class
CHARACTER
Clone
Create
Current
debug
deferred
do
else
elsif
end
ensure
export
Equal
external
false
feature
Forget
from
if
inherit
INTEGER
invariant
is
language
like
local
loop
```

```
mod
name
not
old
once
or
REAL
redefine
rename
require
rescue
Result
retry
STRING
then
true
until
variant
Void
```

Rutinas externas

Cualquier método de diseño de software que enfatice la reusabilidad debe reconocer la necesidad de acceder a código escrito en otros lenguajes. Es duro convencer a usuarios potenciales de que la reusabilidad comienza ahora y de que todo el software existente debe descartarse.

En general la apertura al resto del mundo es un requerimiento importante en las nuevas herramientas de software. Esto podría llamarse el *principio de la modestia:* los autores de nuevas herramientas deberían estar seguros de que todavía pueden acceder a las herramientas disponibles previamente.

Una rutina Eiffel que necesite una o más rutinas escritas en otros lenguajes debe listarlas en una cláusula external. Por ejemplo, una función en Eiffel para calcular la raíz cuadrada podría estar en una librería de funciones escrita en C. La función en Eiffel puede declararse de la siguiente forma:

```
raiz_cuadrada(x: REAL): REAL is
        -- raíz cuadrada de x
require
       x>=0
external
       sqrt(x: REAL):REAL name "sqrt" language "C";
       abs(x: REAL): REAL name "abs" language "C";
do
       Result:=sqrt(x)
ensure
       abs(Result^2-x)<=10^(-8)
end -- sqrt
```

Para cada rutina externa, la cláusula external lista los tipos de sus argumentos y, si la rutina es una función, el tipo de su resultado. También se especifica el nombre del lenguaje en el cual está implementada la rutina externa. En nuestro caso, este lenguaje siempre será C.

Un modo extremo, aunque no absurdo, de utilizar Eiffel, implicaría el uso exclusivo de rutinas externas escritas en otro lenguaje, dejando al Eiffel como una pura herramienta de empaquetamiento, usando el poderoso mecanismo de encapsulación del diseño orientado a objetos: clases, información oculta, la relación cliente y la herencia.

Paso de argumentos

El problema del paso de argumentos es qué pasa a a1,...,an en una llamada a rutina

```
p(a1, a2, ..., an)
```

correspondiente a la rutina

```
p(x1:T1, x2:T2, ..., xn: Tn) is ...
```

donde la rutina puede ser tanto una función como un procedimiento, y la llamada podría estar calificada, tal como b.p(...). Los a_i se llaman argumentos actuales, y los x_i se llaman argumentos formales.

El aspecto más importante del paso de argumentos es la cuestión de qué operaciones se permiten hacer sobre los argumentos formales, y qué efecto tienen sobre los correspondientes argumentos actuales.

La solución en Eiffel es simple. Dentro del cuerpo de la rutina, los argumentos formales están *protegidos:* no se permite ninguna modificación directa sobre ellos. Una modificación directa sobre x es alguna de las siguientes operaciones:

• Una asignación con x como destino, de la forma x:=...

• Si x es de tipo clase, cualquier operación que pueda modificar la referencia asociada con x: x.Create(...), x.Forget, x.Clone(y) para algún y.

El conjunto de operaciones que puede ejecutar una rutina sobre sus argumentos está así altamente restringida.

A causa de esta convención, la única manera de que una rutina devuelva un valor a quien la llama es a través de su resultado, si la ruina es una función. Esto implica que, como máximo, se puede retornar un único valor. El efecto de múltiples resultados se puede solucionar devolviendo como resultado una cierta clase con más de un atributo.

Instrucciones

Eiffel es un lenguaje procedural en el cual la computación se expresa a través de comandos o instrucciones. Las instrucciones incluyen:

• llamadas a procedimientos

• asignaciones

• condicionales

• bucles

• chequeos (check)

• depuraciones (debug)

Llamadas a procedimientos

Una llamada a procedimiento puede ser local o remota.

Una llamada local es una llamada a rutina aplicada a la instancia en curso, y aparece bajo la forma:

p	(sin argumentos), o
p(x,y,...)	(con argumentos)

Una llamada remota se aplica a un objeto representado por una expresión: asumiendo que a es una expresión de tipo C, donde C es una clase, y q es uno de los procedimientos en la clase C, entonces una llamada remota es de la forma a.q. Repetidamente, q puede estar seguida de una lista de argumentos actuales; a puede ser una función local con argumentos, como en a(m).q(n).

La exportabilidad controla la aplicación de llamadas remotas. Decimos que una característica f declarada en una clase C está disponible a una clase A si f está listada en la cláusula de exportables de B.

Regla de llamadas remotas: Una llamada remota de la forma b.q1.q2...qn que aparezca en una clase C es correcta sólo si las siguientes condiciones se satisfacen:

1. La característica que aparece a continuación del primer punto, q1, debe estar disponible para C, o debe ser una de las características predefinidas Create, Clone, Forget, Equal, Void.
2. En llamadas multipuntuadas, cada característica posterior al segundo punto, es decir, cada qi para i>1, debe estar disponible para C; la última característica puede ser también Equal o Void. No puede ser Create, Clone o Forget.

Asignaciones

La instrucción de asignación se escribe

x := e

donde x es una entidad y e una expresión. Recordemos que una entidad es:

• Un atributo de la clase

• Una variable local de una rutina.

• Un argumento formal

• La entidad predefinida Result.

Como ya vimos, un argumento formal no puede ser el destinatario de una asignación.

Condicional

La base de una instrucción condicional se escribe:

```
if expresión_booleana then
    instrucción; instrucción; ...
else
    instrucción; instrucción; ...
end
```

La parte else puede omitirse.

Cuando se consideran más de dos casos, se puede utilizar la siguiente estructura condicional:

```
if  c1  then
     instrucción; instrucción; ...
elsif  c2  then
     instrucción; instrucción; ...
elsif  c3  then
     instrucción; instrucción; ...
...
else
   instrucción; instrucción; ...
end
```

Bucles

La estructura de los bucles es la siguiente:

```
from  instrucciones_de_inicialización
invariant  invariante
variant  variante
until  condición_de_salida
loop  instrucciones_de_bucle
end
```

Check

Estructura

```
check
   aserción; aserción; ...; aserción
end
```

Debug

Se escribe

```
debug
   instrucción; instrucción; ...; instrucción
end
```

Esta instrucción sirve para introducir acciones especiales que se ejecutarán sólo en modo de depuración.

Retry

Puede ejecutarse formando parte de una cláusula de rescate para recomenzar una rutina que fue interrumpida por una excepción. Cualquier intento de ejecutar esta instrucción en otro momento que no sea durante el proceso de una cláusula de rescate, generará una excepción.

Expresiones

Las expresiones incluyen las siguientes variedades:

- constantes
- entidades
- llamadas de función
- Current
- expresiones con operadores

Constantes

Hay dos constantes booleanas; *true* y *false*.

Las constantes enteras siguen la forma usual y pueden estar precedidas por un signo. Por ejemplo:

453 -678 +66623

Las constantes reales usan el punto decimal. Bien la parte entera o la decimal pueden estar ausentes. Se puede poner signo. Una potencia de 10 se especifica por e seguido por el valor del exponente. Por ejemplo:

52.4 -54.44 +54.55 .983 -897. 999.e12

Las constantes de carácter consisten en simple carácter escrito entre comillas, tal como 'A'. Describen caracteres sencillos.

Llamadas a funciones

Siguen la misma sintaxis que las llamadas a procedimientos. Pueden ser locales o remotas; se permite la notación multipunto.

Por ejemplo:

```
b.f
b.g(x,y,...)
b.h(u,v).i.j(x,y,...)
```

La expresión "Current"

La palabra reservada Current denota la instancia en curso de la clase y puede usarse como una expresión. Nótese que Current es en sí mismo una expresión, no una entidad; así, una asignación a Current, de la forma

```
Current := valor
```

es sintácticamente incorrecto.

Ejemplos de uso de Current incluyen:

• Creación de una copia de la instancia en curso, como en x.Clone(Current).

• Testear si una referencia se aplica a la instancia en curso, como en x=Current.

Pasar a la instancia en curso como argumento a una rutina, como en x.f(Current).

Expresiones con operadores

Los operadores están disponibles para construir expresiones compuestas.

Los operadores unarios son + y -, aplicables a expresiones reales y enteras, y **not**, aplicable a expresiones booleanas.

Los operadores binarios, que engloban a dos operandos, son la operación de exponenciación ^ (a^2 es el cuadrado de a) y los operadores relacionales:

```
= /= < > <= >=
```

donde /= es el operador *no igual*. Los operadores relacionales generan resultados booleanos. Sus operandos pueden ser de tipo entero o real; se permite también la comparación de caracteres, usando el orden del conjunto de caracteres ASCII.

Las expresiones múltiples envuelven a uno o más operandos, combinados con operadores. Operandos numéricos pueden combinarse usando los siguientes operadores:

```
+ - * /
```

Los operandos booleanos pueden combinarse con los operadores **and** y **or**.

PRECEDENCIA DE LOS OPRADORES

La tabla siguiente muestra precedencia de los operadores de mayor a menor

NIVEL	Operadores
10	. (punto de notación para características cualificadas
9	**old**
8	^ (potenciación)
7	**not** + - (unario)
6	* / **mod div**
5	+ - (binario)
4	= /= < > <= >=
3	**and**
2	**or**
1	; (menor precedencia *and* en aserciones)

Strings

La clase STRING describe las cadenas de caracteres. Esta clase se trata de una manera especial: el identificador STRING se reconoce como una palabra reservada, y constantes de cadena forman parte de la sintaxis predefinida del lenguaje.

Una constante de cadena se escribe encerrada entre comillas dobles, como en

```
"ABCdefg~*_01"
```

II.8 INTRODUCCIÓN A LA HERENCIA

Todo nuevo software se expande de otros desarrollos anteriores; la mejor manera de crearlo parece ser por imitación, refinamiento y combinación.

Las clases, como vimos hasta ahora, no son suficientes. Proveen una buena técnica de descomposición modular. También poseen muchas de las cualidades que se esperan de componentes de software reutilizables: son módulos coherentes, homogéneos, su interfaz puede separarse claramente de su implementación de acuerdo con el principio de ocultación de información y pueden ser especificadas claramente gracias a las aserciones. Pero se necesita más para completar los principios de reusabilidad y extensibilidad.

Para la **reusabilidad,** cualquier aproximación comprensiva debe afrontar el problema de la repetición y variación. Para evitar reescribir lo mismo una y otra vez, gastando tiempo, introduciendo inconsistencias y arriesgando errores, debemos encontrar técnicas que capturen las características comunes más significativas que existen en grupos de estructura similar - todas las implementaciones de listas, editores de texto, manejadores de ficheros, etc. - mientras se responde por todas las diferencias que permanecen en los casos individuales.

Para la **extensibilidad,** el sistema de tipos descrito hasta ahora tiene la ventaja de garantizar la consistencia de tipos en tiempo de compilación, pero prohibe la combinación de elementos de formas diversas, incluso en casos legítimos. Por ejemplo, no podemos todavía definir un array que contenga objetos geométricos de tipos diferentes: PUNTO, VECTOR, SEGMENTO, etc.

El progreso, bien en la reusabilidad, bien en la extensibilidad, demandan que aprovechemos las fuertes relaciones conceptuales existentes entre clases: una clase puede ser la extensión, especialización o combinación de otras. Necesitamos soporte del método y del lenguaje para representar y usar estas relaciones. La herencia nos proporciona todo esto.

Polígonos y rectángulos

Supongamos que queremos construir una librería gráfica de forma que las clases describirán abstracciones geométricas: puntos, segmentos, vectores, elipses, polígonos en general, triángulos, rectángulos, cuadrados, ...

Una clase que describa los polígonos sería de la forma:

```
class POLIGONO export
    vertices, traslada, gira, perimetro
feature
  ...
  vertices: LINKED_LIST[PUNTO]; -- el uso de lista es sólo una posible
                  implementación
  traslada(a, b: REAL) is
```

```
        -- mueve en a horizontal, b vertical
      do ... end;
    gira(centro; PUNTO; angulo: REAL) is
        -- gira en ángulo alrededor de centro
      do ... end;
    visualiza is
        --visualiza el polígono en la pantalla
      do ... end;
    perimetro: REAL is
        -- longitud del perímetro
      do ... end;
  invariant
    vertices.nb_elements>=3
        -- un polígono tiene al menos tres vértices
  end -- clase POLIGONO
```

Se ha elegido para representar los vértices del polígono la estructura de lista, descrita en la clase LINKED_LIST[T] de la librería básica Eiffel. Esta clase exporta entre otras, las siguientes características.

```
    start is
   -- Mueve el cursor al primer elemento de la lista
 do ... end

    value:T is
   -- Devuelve el valor del elemento que ocupa
   -- la posición actual
do ... end

    islast:BOOLEAN is
   -- ¿El apuntador señala al último elemento ?
do ... end

    first:T;
   -- valor del primer elemento
   -- es un atributo

    forth is
   -- avanza el cursor al siguiente elemento
 do ... end;

    i_th(i:INTEGER):T is
   -- valor del i-ésimo elemento
 do ... end;
```

Esto nos permite definir la rutina perímetro de la clase POLIGONO del siguiente modo:

```
    perimetro:REAL is
  -- longitud del perímetro
local
  actual, anterior:PUNTO
do
  from
     vertices.start;
     actual:=vertices.value
  until
     vertices.islast
  loop
     anterior:=actual;
     vertices.forth;
     actual:=vertices.value;
     result:=result+actual.distancia(anterior)
```

```
end;
result:=result+actual.distancia(vertices.first)
  end  --perimetro
```

Supongamos que queremos una clase que represente rectángulos. Un rectángulo es un caso especial de polígono; muchas de las características son las mismas. Por ejemplo, un rectángulo se trasladará, girará o visualizará de la misma manera que un polígono. Por otro lado, un rectángulo tiene características particulares (como una diagonal), propiedades especiales (el numero de vértices es cuatro, sus ángulos son rectos), y versiones especiales de algunas operaciones (por ejemplo, hay una manera mejor de calcular el perímetro que en un polígono en general).

Podemos obtener ventaja de estos aspectos comunes definiendo la clase RECTANGULO como heredera de la clase POLIGONO. Esto significa que todas las características de POLIGONO llamado **padre** de RECTANGULO son aplicables de la misma manera a la clase heredera. Este efecto se logra dándole a RECTANGULO un **cláusula de herencia,** como vemos:

```
class RECTANGULO export
        vertices, traslada, gira, perimetro diagonal, cara1, cara2, ...
inherit
      POLIGONO
feature
     ... características específicas de rectángulos ...
end
```

La cláusula **feature** de la clase heredera normalmente no repite las características del padre: están disponibles automáticamente a causa de la cláusula de herencia. Sólo se especifican las características específicas de la clase heredera.

Puede haber una excepción: algunas características del padre pueden estar redefinidas para tener una implementación diferente. Como ejemplo aquí está perimetro, que tiene una implementación más adecuada para los rectángulos. Un heredero que redefina una característica del padre debe anunciarlo en la cláusula de herencia:

```
class RECTANGULO export
     ...
inherit
     POLIGONO redefine perimetro
feature
     ...
end
```

Si la subcláusula **redefine** no estuviese presente, entonces una nueva declaración de perimetro en las características de RECTANGULO produciría un error.

La clase rectángulo podría quedar como sigue:

```
 class RECTANGULO export
      vertices, traslacion, rotacion, perimetro, diagonal, lado1, lado2,
...
 inherit
      POLIGONO redefine perimetro
 feature
      lado1, lado2:REAL;         -- longitudes de los lados
diagonal:REAL;            -- longitud de la diagonal
create(centro:PUNTO; s1,s2,angulo:REAL) is
   -- crea rectangulo centrado en centro con logitudes
   -- de lados s1 y s2 y orientación ángulo
 do ... end; -- create
perimetro:REAL is
   -- longitud del perímetro
   -- redefinición de la versión de POLIGONO
 do
   result:= 2*(lado1+lado2
 end;  -- perímatro
```

```
invariant
vertices.num_elementos:=4;
vertices.i_th(1).distancia(vertices.i_th(2)) = lado1;
vertices.i_th(2).distancia(vertices.i_th(3)) = lado2;
vertices.i_th(3).distancia(vertices.i_th(4)) = lado1;
vertices.i_th(4).distancia(vertices.i_th(1)) = lado2;
    -- asertos expresando que los ángulos son rectos
    end    -- clase RECTANGULO
```

El proceso de herencia es transitivo: si una clase heredase de RECTANGULO (por ejemplo la clase cuadrado) también heredaría todas las características de POLIGONO.

Definición: una clase que hereda directa o indirectamente de A se dice que es **descendiente** de A. Una clase que es descendiente de sí misma. Los descendientes de A distintos de A mismo, se llaman sus **descendientes propios**.

Definición: si B es un descendiente (descendiente propio) de A, A se dice que es un antepasado (antepasado propio) de B.

El caso de Create

Hay una excepción a la regla que dice que todas las características del padre están disponibles automáticamente para sus descendientes.

Regla para *Create:* El procedimiento Create no se hereda nunca directamente. Es la única característica con esta propiedad.

Consistencia de tipos

La herencia es consistente con el sistema de tipos de Eiffel.

Regla de aplicación de Características: en una aplicación de característica x.f, donde el tipo de x se obtiene de la clase A, la característica f debe estar definida en uno de los antepasados de A (recordemos que esto incluye a A también).

Supongamos las siguientes declaraciones:

```
p: POLIGONO;
r: RECTANGULO;
```

entonces las siguientes expresiones son correctas:

```
p.perímetro
p.vertices, p.traslacion(...), p.rotacion(...)
r.diagonal, r.lado1, r.lado2
r.vertices, r.traslacion(...), r.rotacion(...)
r.perimatro
```

mientras que no son correctas:

```
p.lado1, p.lado2, p.diagonal
```

Polimorfismo

Las propiedades vistas hasta ahora se encaminaban hacia la reusabilidad: construcción de módulos como extensiones de otros ya existentes. Hay otro aspecto igualmente importante, relacionado más directamente con la extensibilidad: el concepto de **polimorfismo,** y su complemento natural: el **enlace dinámico**.

Polimorfismo significa la posibilidad de manifestar diversas formas. En programación orientada a objetos se refiere a la posibilidad que tiene una entidad de referirse, en ejecución, a instancias de varias clases. En un entorno tipeado como Eiffel esto está restringido por la herencia: por ejemplo, es posible permitir que una entidad de tipo POLIGONO se refiera a un objeto RECTANGULO; pero una entidad declarada de tipo RECTANGULO no puede referirse a un objeto POLIGONO.

Suponiendo las anteriores declaraciones:

```
p:POLIGONO;  r:RECTANGULO;
```

una asignación como p:=r es totalmente correcta.

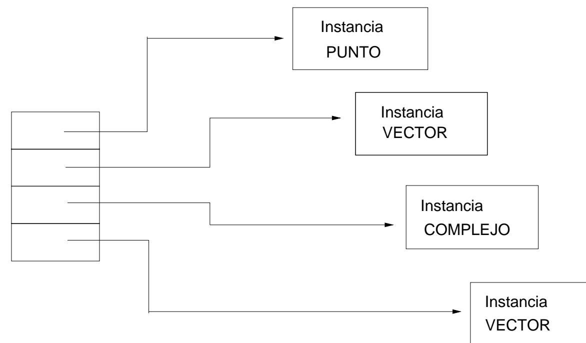
No olvidemos que entidades de tipos clase denotan referencias a objetos, no objetos en sí mismos. La asignación anterior no es más que una mera asignación de referencias.

El polimorfismo es la clave para hacer el sistema de tipos más flexible: Por ejemplo, esto nos permite dar una respuesta al problema planteado cuando pretendíamos obtener un array de referencias a objetos de tipos diferentes.

Suponiendo que varias clases como: PUNTO, VECTOR, COMPLEJO heredan de una clase común DOS_COORDENADAS, una declaración de la forma

a:ARRAY[DOS_COORDENADAS]

permitirá situaciones como la siguiente:



Límites del polimorfismo

Un polimorfismo sin restricciones sería incompatible con la noción de tipo. En el sistema de tipos Eiffel, el polimorfismo está controlado por la herencia.

Regla de compatibilidad de tipos: La asignación x := y, donde el tipo de x está dado por la clase A y el tipo de y está dado por la clase B, sólo es legal si B es un descendiente de A. La misma regla se aplica si x es un argumento formal de rutina e y es el argumento actual correspondiente en una llamada a la rutina.

Tipo estático, tipo dinámico

A veces es útil usar los términos *tipo estático* y *tipo dinámico* de una referencia. El tipo con el cual una entidad está declarada se llama tipo estático de la referencia correspondiente. Si, en ejecución la referencia se asocia con un objeto de cierto tipo, este tipo se convierte en el tipo dinámico de la referencia. La regla de compatibilidad de tipos dice que el tipo dinámico es siempre un descendiente del tipo estático.

Enlace dinámico

Las operaciones definidas para todas las variedades de polígonos no han de ser implementadas de manera idéntica para todas esas variedades. Por ejemplo, perimetro tiene diferentes versiones para polígonos generales y para rectángulos; llamémoslos $perimetro_{POL}$ y $perimetro_{RECT}$. Esto plantea inmediatamente una cuestión fundamental; ¿qué ocurre a una entidad polimórfica cuando se le aplica una rutina con más de una versión?

En un fragmento de código como

```
p.Create(...); x:=p.perimetro
```

está claro que se aplicará $perimetro_{POL}$. Esto está tan claro como en

```
r.Create(...); x:=r.perimetro
```

donde se aplicará $perimetro_{RECT}$. Pero, ¿qué ocurre si la entidad polimórfica p, declarada estáticamente como un polígono, se refiere dinámicamente a un rectángulo?. Supongamos que se ha ejecutado

```
r.Create(...);
p:=r;
x:=p.perimetro
```

La regla conocida como **enlace dinámico** (dinamic binding) implica que **la forma dinámica** del objeto determina qué versión de la operación se aplica. Aquí se aplicará $perimetro_{RECT}$.

Esta capacidad de las operaciones de adaptarse automáticamente a los objetos a los que están aplicadas es una de las propiedades más importantes de los sistemas orientados a objetos.

Redefinición y aserciones

El enlace dinámico es una herramienta poderosa, pero obviamente acarrea ciertos riesgos. Si un cliente de POLIGONO llama a p.perimetro, este espera obtener el valor del perímetro de p. Pero ahora, a causa del enlace dinámico, el cliente puede llamar a otra rutina, redefinida en algún descendiente. En RECTANGULO, la redefinición era para mejorar la eficiencia; pero en principio se podría redefinir perimetro para calcular, por ejemplo, el área.

Esto es contrario al espíritu de la redefinición. La redefinición debería cambiar la implementación de una rutina, no su semántica. Afortunadamente, en Eiffel, tenemos un camino para restringir la semántica de una rutina: las aserciones. La regla básica para controlar el poder de la redefinición y del enlace dinámico es simple: La precondición y la postcondición de una rutina se aplica a cualquier redefinición en una clase descendiente. Todavía más, la invariante de una clase también se aplica a descendientes.

Clases diferidas

Procedemos ahora a estudiar más técnicas asociadas con la herencia.

Como vimos, las rutinas pueden estar redefinidas en clases descendientes. En algunos casos se puede desear forzar una redefinición. Esta es la razón para una herramienta de diseño muy importante: **clases diferidas**.

Moviendo figuras arbitrarias

Para comprender la necesidad de rutinas y clases diferidas, consideremos la jerarquía FIGURA.

La noción más general es la de FIGURA. Aplicando el mecanismo de polimorfismo y enlace dinámico, se puede escribir código como el siguiente:

```
f:FIGURA;  c:CIRCULO;  r:RECTANGULO;
...
c.Create(...);  r.Create(...);
...
   -- "dejar al usuario escoger un icono"
if  icono_elegido=icono_circulo then
       f:=c
elsif  icono_elegido=icono_rectangulo then
       f:=r
elsif ...
       ...
end;
f.traslada(a,b);
...
```

La hipótesis es que cada una de las clases, CIRCULO y RECTANGULO, tiene su propia versión de traslada; la versión de RECTANGULO viene de POLIGONO y la de CIRCULO es fácil de escribir.

Con el mecanismo de redefinición, esto trabajaría perfectamente. Pero aquí está el problema: ¡aquí no hay nada que redefinir! FIGURA es una noción muy general que cubre todo tipo de figuras bidimensionales. No hay ninguna forma de escribir una versión de propósito general de traslada sin más información acerca de las figuras consideradas.

Así pues, aquí hay una situación donde el código anterior debería ejecutarse correctamente gracias al enlace dinámico, pero es estáticamente incorrecto dado que traslada no es una característica válida de FIGURA. El mecanismo de chequeo de tipos detectará como una operación inválida f.traslada.

Se podría, por supuesto, introducir un procedimiento traslada en el nivel de FIGURA, que no hiciese nada. Pero este es un camino a seguir peligroso; traslada(a, b) tiene una semántica intuitiva bien definida, y *no hacer nada* no es una implementación propia de esta.

Lo que se necesita es una manera de especificar traslada en el nivel de FIGURA. Esto se consigue declarando la rutina como **deferred.** Por ejemplo, en FIGURA, se podría declarar

```
traslada (a, b: REAL) is
      -- mueve en a horizontal, b vertical
   deferred
   end -- traslada
```

Esto significa que la rutina se conoce en la clase en la que se define, pero está implementada en las descendientes.

FIGURA se llamará a sí misma una clase diferida.; esto define un grupo de implementaciones de tipos de datos relacionados, en lugar de uno sólo. La definición precisa es la siguiente:

Definición: Una clase diferida es una clase C que contiene una rutina diferida. Esta rutina puede estar introducida en C como diferida, o heredada de una clase madre y no redefinida efectivamente en C.

Una clase diferida debe comenzar con las dos palabras claves **deferred class** en lugar de simplemente class, para recordar al lector que una o más rutinas están diferidas.

Un descendiente de una clase diferida será una clase efectiva si provee definiciones efectivas para todas las rutinas diferidas en sus ancestros, y no introduce ninguna en sí misma.

Qué hacer con las clases diferidas

La cuestión fundamental es preguntar ¿qué ocurre si una rutina diferida se aplica a una instancia de una clase diferida?. El Eiffel responde drásticamente: no existe nada parecido a una instancia de una clase diferida.

Regla de no-instanciación de clase diferida: Create no se puede aplicar a una entidad cuyo tipo está dado por una clase diferida.

Así, no se pueden crear objetos del tipo FIGURA, pero se pueden declarar entidades polimórficas de este tipo, como f arriba, que juega un papel esencial en la programación orientada a objetos.

Especificando la semántica de las rutinas diferidas

Las rutinas diferidas se hacen todavía más útiles por el mecanismo de aserciones. Sintácticamente **deferred** ocupa el lugar del bloque **do** *instrucciones.* Esto deja espacio para los otros componentes: cabecera de la rutina, precondición y post-condición.

Sólo con las aserciones las clases diferidas obtienen su total poder. Recuérdese que precondiciones y postcondiciones se aplican a todas las redefiniciones de una rutina. Esto es especialmente importante para una rutina diferida: su precondición y postcondición, si están presentes, se aplican a todas las definiciones efectivas de la rutina en descendientes. A causa de esta propiedad, una definición diferida puede, de hecho, ser altamente informativo a pesar de no prescribir ninguna implementación específica.

De vuelta a los tipos abstractos de datos

Cargadas con aserciones, las clases diferidas se aproximan mucho a la representación de tipos abstractos de datos. Un típico ejemplo se da con la clase PILA

```
deferred class PILA[T] export
      nb_elem, vacia, llena, top, push, pop, cambia_top, borra_todo
feature
      nb_elem : INTEGER is
                  -- número de elementos insertados
          deferred
          end; --nb_elem
      vacia : BOOLEAN is
                  -- está vacia ?
          do
             Result:=(nb_elem=0)
          ensure
             Result=(nb_elem=0)
          end; -- vacia
      llena : BOOLEAN is
                  -- está llena?
            deferred
            end; -- llena
      top : T is
                  -- último elemento introducido
            require
                  not vacia
            deferred
            end; -- top
      push (x : T)  is
                  -- introduce x en la pila
            require
                not llena
            deferred
            ensure
               not vacia; top = x; nb_elem = nb_elem + 1
            end; -- push
      pop is
```

```
          -- elimina el último elemento
         require
           not vacia
         deferred
         ensure
           not llena; nb_elem = old nb_elem - 1
         end; -- pop
    cambia_top (x : T) is
            -- reemplazar el último elemento por x
         require
           not vacia
         do
           pop; push(x)
         ensure
           not vacia; top = x; nb_elem = old nb_elem
         end; -- cambia_top
    borra_todo  is
            -- elimina todos los elementos
         deferred
         ensure
             vacia
         end; -- borra_todo
invariant
    nb_elem >= 0
end -- class PILA
```

Se pueden usar las clases diferidas para **capturar comportamientos comunes** en un conjunto de problemas. Sólo la parte común se describe en la clase diferida, las variaciones se dejen a los descendientes.

Herencia múltiple

En los ejemplos vistos anteriormente, el heredero sólo tenía un padre. Esto se conoce como **herencia simple.** La **herencia múltiple** es también muy útil.

El matrimonio de conveniencia

Una de las aplicaciones importantes de la herencia múltiple es proveer una implementación de una abstracción definida por una clase diferida, usando facilidades provistas por una clase efectiva. Por ejemplo, las pilas pueden estar implementadas por arrays. Dado que ambos, arrays y pilas, están descritos por clases, la mejor manera de implementar la clase PILA_FIJA, describiendo pilas implementadas como arrays, es definirla como heredera de ambas, PILA y ARRAY. La forma general es:

```
class PILA_FIJA[T] export
        -- las mismas exportaciones que en PILA
inherit
     PILA[T];
     ARRAY[T]
feature
     ... implementación de las rutinas diferidas de PILA
     ... en términos de operaciones de ARRAY
end -- class PILA_FIJA
```

PILA_FIJA ofrece la misma funcionalidad que PILA. Da versiones efectivas de las rutinas diferidas en PILA, implementadas aquí en términos de las operaciones de array:

```
 llena: BOOLEAN is
 -- ¿ está la pila llena ?
do
  Result:= (num_elementos= tamaño)
end -- llena
```

En el ejemplo anterior, lo que aportan los padres de la clase heredera es, por parte de la clase PILA *funcionalidad* y por parte de la clase ARRAY la *implementación*.

Formas de herencia múltiple

Los matrimonios de conveniencia no son las únicas aplicaciones interesantes de la herencia múltiple. A menudo la relación entre padres es más equilibrada; cada uno aporta cierta funcionalidad y alguna implementación. En muchos casos, ninguna es una clase diferida. Un ejemplo típico es una clase que describe ventanas en un sistema de multi-ventanas, que soporta un anidamiento jerárquico: una ventana puede contener subventanas. Esta noción puede ser descrita mediante la clase VENTANA que hereda de tres padres:

• El primer padre podría ser OBJETO_PANTALLA quizás un descendiente de figura. Encerrando las propiedades geométricas de las ventanas, con características como altura, anchura, procedimientos para mover los objetos por la pantalla, etc.

• El segundo, TEXTO, conteniendo las propiedades textuales de las ventanas, vistas como sucesiones de caracteres o líneas, con primitivas para manipular el contenido textual.

• El último podría ser la clase ARBOL, que colabora aportando la estructura jerárquica: subventanas, ventanas padres, procedimientos para añadir y eliminar subventanas, etc.

Colisión de nombres y renombrado

Los herederos de más de una clase tiene acceso a todas las características de sus padres, sin ningún tipo de cualificación. Esto trae el problema de la colisión de nombres. ¿Qué ocurre en

```
class C export ... inherit
  A;
  B
feature ... end
```

si ambas A y B tienen una característica con el mismo nombre, por ejemplo f ?

La regla en Eiffel es simple: tales colisiones están prohibidas.

Es un error culpar a los padres de la colisión de nombres en herencia: el problema está en el heredero. Allí se deberá implementar la solución. La colisión de nombres puede solucionarse incluyendo una o más subcláusulas de renombrado **rename** en la cláusula de herencia. Por ejemplo

```
class C export ... inherit
    A rename f as A_f;
    B
feature
    ...
end
```

Acceso al Create de los padres

El renombrado tiene otra aplicación para el caso de la regla de creación de objetos. Como ya se dijo, Create nunca se hereda; sin embargo puede ser útil reusar el algoritmo usado en el Create del padre. Esto se soluciona renombrándolo

```
inherit
   C rename Create as C_Create
```

II.9 MÁS ACERCA DE LA HERENCIA

Herencia y aserciones

Invariantes

La regla para invariantes es simple:

Regla de los invariantes paternos: Los invariantes de todos los padres de una clase se aplican a dicha clase.

Se considera que los invariantes de los padres se añaden al invariante de la clase heredera (entiéndase añadir como **and**).

Precondiciones y Postcondiciones

El caso de pre y postcondiciones es más delicado.

Decimos que una aserción es **más fuerte** que otra si implica a esta última y es distinta; por ejemplo, x>=5 es más fuerte que x>=0. Si A es más fuerte que B, B es **más débil** que A. Las notaciones prer y posr denotan la precondición y postcondición de un rutina r. Entonces:

Regla de redefinición de aserciones: Sea r una rutina en una clase A y s una redefinición de r en un descendiente de A, o una definición efectiva de r si r era diferida. Entonces pres debe ser más débil o igual a prer, y poss debe ser más fuerte o igual que posr.

El sistema de tipos de Eiffel

Regla de compatibilidad de tipos; la noción de conformancia

Definición (tipos en Eiffel): en Eiffel un tipo es alguna de las siguientes cosas:

1. Un tipo simple (INTEGER, BOOLEAN, REAL, CHARACTER).

2. Un parámetro genérico formal de la clase en curso.

3. Un tipo clase, es decir, un nombre de clase posiblemente seguido por sus parámetros genéricos actuales.

4. **like** *tirante,* o declaración por asociación. (No se verá)

La regla completa de compatibilidad es la siguiente:

Regla de compatibilidad de tipos: Una asignación x:=y o una llamada r(...,y,...), donde el argumento formal correspondiente a y es x, son correctos si y sólo si uno de los siguientes preserva los tipos respectivos X e Y de x e y:

1. X e Y son idénticos.

2. X es REAL, e Y es INTEGER.

3. Y conforma con X (definido a continuación).

Definición de conformancia: un tipo Y se dice que conforma con otro tipo X si y sólo si se cumple uno de los puntos siguientes:

1. X e Y son idénticos.

2. X e Y son tipos clase, X no tiene parámetros genéricos e Y lista X en su cláusula de herencia.

3. X e Y son tipos clase, X es de la forma P[U1, U2, ..., Un] y la cláusula de herencia de Y lista P[V1, V2, ..., Vn] como padre, donde cada Vi conforma con el correspondiente Ui.

4. Y es de la forma like tirante, y el tipo de tirante conforma con X.

5. Hay un tipo Z tal que Y conforma con Z y Z conforma con X.

Regla de redefinición: Un atributo, un resultado de función o un argumento formal de rutina declarado en una clase puede ser redeclarado con un nuevo tipo en una clase descendiente, cuidando de que el nuevo tipo conforme con el original.

El atributo, función o rutina se considera para ser redefinido. A menos que originalmente estuviese diferido, debe estar listado en la subcláusula redefine de la clase descendiente.

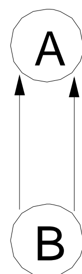
Herencia repetida

Compartiendo antepasados

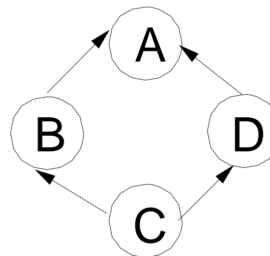
Uno de los problemas delicados provocados por la presencia de herencia múltiple es qué ocurre cuando una clase es antepasada de otra por más de una vía.

En Eiffel no hay ninguna restricción a la herencia repetida.

Herencia repetida



Directa



Indirecta

Cuando una clase aparece más de una vez en la lista de clases heredadas existe herencia repetida directa.

```
class B export ... inherit
 A rename ... redefine ...;
 A rename ... redefine ...;
...
```

Regla de herencia repetida: en la herencia repetida, cualquier característica del antepasado común se considera compartida si no ha sido renombrada a lo largo de los caminos de herencia. Cualquier característica que haya sido renombrada al menos una vez a lo largo de cualquiera de las sendas de la herencia, se considerará replicado.

Herencia repetida y genericidad

Sea la declaración de clases siguiente

```
class A[T] feature
  f:T; ...;
end

class B inherit
   A[INTEGER]; A[REAL]
end
```

En la clase B la característica f es heredada sin embargo, su tipo es ambiguo pues puede devolver un entero o un real. En Eiffel este caso no se acepta.

Regla de herencia repetida y genericidad: El tipo de cualquier característica que pudiera compartirse bajo la regla de herencia repetida, y el tipo de sus argumentos si es una rutina, no pueden ser un parámetro genérico de la clase de la cual la característica se hereda.

Consideremos el siguiente ejemplo dado por la clase CONTRYBUYENTE, que tiene los siguientes atributos:

```
edad: INTEGER;
direccion: STRING;
cuenta_bancaria: CUENTA
id_contrib: INTEGER;
```

y rutinas como:

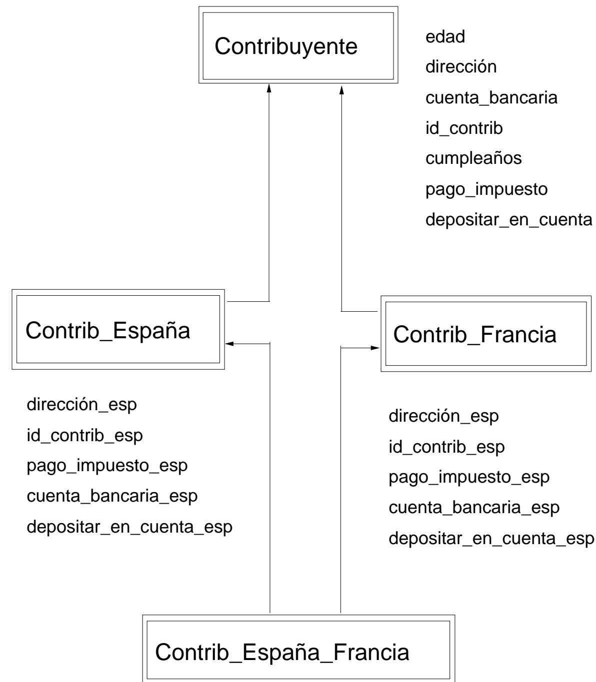
```
cumpleannos is do edad:= edad + 1 end;
pago_impuesto is ...;
depositar_en_cuenta (cant: INTEGER); is ...
...
```

Esta clase puede tener como herederos las clases:

```
CONTRIB_ESPANNA
CONTRIB_FRANCIA
```

Ahora bien, puede ser necesario también considerar las personas que pagan impuestos tanto en Francia como en España (quizás porque viven en cada pais durante ciertas temporadas). La forma natural de expresar esta situación es usar herencia multiple en la definición de una clase como CONTRIB_ESPANNA_FRANCIA.

Las características heredadas dos veces del ascendiente común CONTRIBUYENTE, como *direccion, id_contrib*, ... se deben renombrar en la clase hija.




```
  class CONTRIB_ESPANNA_FRANCIA export
...
  inherit
     CONTRIB_ESPANNA
   rename
    direccion as direccion_esp;
    id_contrib as id_contrib_esp;
    pago_impuesto as pago_impuesto_esp;
    cuenta_bancaria as cuenta_bancaria_esp;
    depositar_en_cuenta as depositar_en_cuenta_esp;
    ...

 CONTRIB_FRANCIA
   rename
    direccion as direccion_fr;
    id_contrib as id_contrib_fr;
    pago_impuesto as pago_impuesto_fr;
    cuenta_bancaria as cuenta_bancaria_fr;
    depositar_en_cuenta as depositar_en_cuenta_esp;
    ...
```

II.10 GRAMATICA DE EIFFEL

Class_declaracion	=	Class_header [Formal_generics] [Exports] [Parents] [Features] [Class_invariant] **end** ["--" **class** Class_name]
Class_header	=	[Deferred_mark] **class** Class_name
Defferred_mark	=	**deferred**

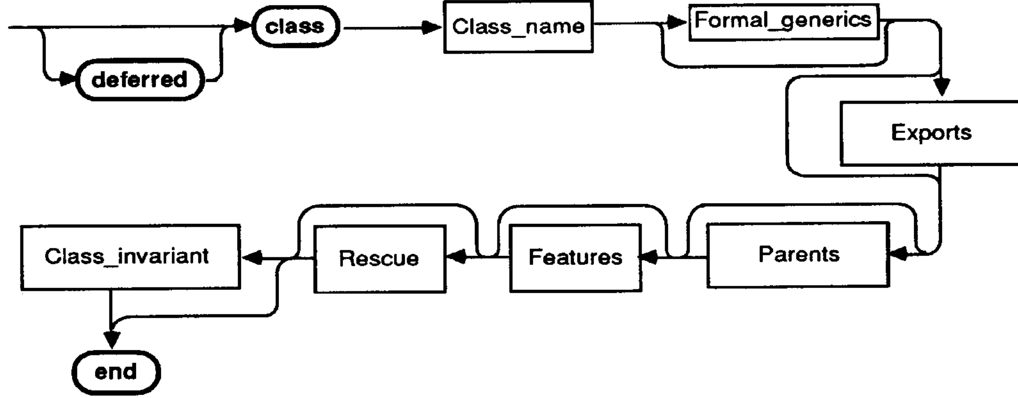
Class_name	=	Identifier
Formal_generics	=	"[" Formal_generics_list "]"
Formal_generics_list	=	{Formal_generic ","...}
Formal_generics	=	Identifier
Export	=	**export** Export_list
Export_list	=	{Export_item ","...}
Export_item	=	Feature_name [Export_restriction]
Feature_name	=	Identifier
Export_restriction	=	"{" Class_list "}"
Class_list	=	{Class_name "," ...}
Parents	=	**inherit** Praent_list
Parent_list	=	{Parent ";" ...}
Parent	=	Class_type [Rename_clause] [Redefine_clause]
Class_type	=	Class_name[Actual_generics]
Actual_generics	=	"[" Type_list "]"
Type_list	=	{Type "," ...}
Type	=	INTEGER | BOOLEAN | CHARACTER | REAL | Class_type | Formal_generic | Association
Association	=	**like** Anchor
Anchor	=	Feature_name | *current*
Rename_clause	=	**rename** Rename_list
Rename_list	=	{Rename_pair "," ...}
Rename_pair	=	Feature_name **as** Feature_name
Redefine_clause	=	**redefine** Feature_list
Feature_list	=	{Feature_name "," ...}
Features	=	**feature** {Feature_declaration ";" ...}
Feature_declaracion	=	Feature_name [Formal_arguments] [Type_mark] [Feature_value_mark]
Formal_arguments	=	Entity_declaration_list
Entity_declaration_list	=	{Entity_declaration_group ";" ...}
Entity_declaration_group	=	{Identifier ";" ...}$^+$ Type_mark
Type_mark	=	":" Type
Feature_value_mark	=	**is** Feature_value
Feature_value	=	Constant | Routine
Constant	=	Integer_constant | Characterconstant | Boolean_constant | Real_constant | String_constant
Integer_constant	=	[Sign] Integer
Sign	=	'+' |'-'
Character_constant	=	"'" Character "'"
Boolean_constant	=	**true** | **false**
Real_constant	=	[Sign] Real
String_constant	=	'"' String '"'
Routine	=	[Precondition] [Externals] [Local_variables] Body [Postcondition] [Rescue] **end** ["--" Feature_name]
Precondition	=	**require** Assertion

Assertion = {Assertion_clause ";" ...}
Assertion_clause = [Tang_mark] Unlabeled_assertion_clause
Tag_mark = Tag ":"
Tag = Identifier
Unlabeled_assertion_clause = Boolean_expresion | Comment
Boolean_expresion = Expresion
Comment = "--" String
Externals = **external** Externals_list
External_list = {External_declaration ";" ...}
External_declaration = Feature_name [Formal_arguments] [Type_mark] [External_name] Language
Language = **language** String_constant
External_name = **name** String_constant
Local_variables = Entity_declaration_list
Body = Full_body | Deferred_body
Deferred_body = **deferred**
Full_body = Normal_body | Once_body
Normal_body = **do** | Compound
Once_body = **once** Compound
Compound = {Instruction ";" ...}
Instruction = Call | Assignment | Conditional | Loop | Check | Retry | Debug
Call = Qualified_call | Unqualified_call
Unqualified_call = Feature_name [Actuals]
Actuals = "(" Expresion_list ")"
Expresion_list = {Expresion_separator ...}
Separator = "," | ";"
Assignment = Entity ":=" Expresion
Entity = Identifier | *Result*
Expresion = {Unqualified_expresion "." ...}
Unqualified_expresion = Constant | Entity | Unqualified_call | ***Current*** | Old_value | Nochange | Operator_expression
Old_value = **old** Expresion
Nochange = **nochange**
Operator_expresion Unary_expresion | Binary_expression | Multiary_expression | Parenthesized
Unary_expression = Unary_expression
Unary = **not** | "+" | "-"
Binary_expression = Expression Binary Expression
Binary = ^ | = | /= | < | > | <= | >=
Multiary_expression = {Expression Multiary ...}$^+$
Multiary = "+" | "-" | "*" | "/" | **and** | **and then** | **or** | **or else**
Parenthesized = "(" Expression ")"
Conditional = **if** Then_part_list [Else_part] **end**
Then_part_list = {Then_part **elsif** ...}$^+$
Then_part = Boolen_expression **then** Compound
Else_part = **else** Compound

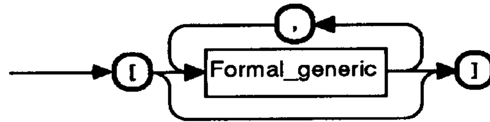
Loop	=	Initialization [Loop_invariant] [Loop_variant] Exit_clause Loop_body **end**
Initialization	=	**from** Compound
Loop_invariant	=	**invariant** Assertion
Loop_variant	=	**variant** Integer_expression
Integer_expression	=	Expression
Exit_clause	=	**until** Boolean_expression
Loop_body	=	**loop** Compound
Check	=	**check** Assertion **end**
Retry	=	**retry**
Debug	=	**debug** Compound **end**
Postcondition	=	**ensure** Assertion
Rescue	=	**rescue** Compound
Class_invariant	=	**invariant** Assertion

II.11 DIAGRAMAS SINTACTICOS DE EIFFEL

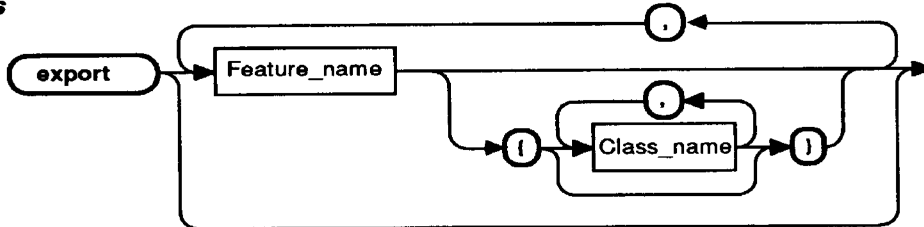
Class_declaration



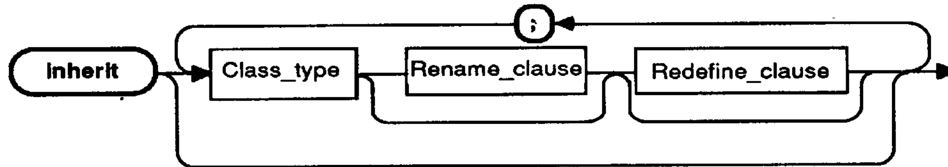
Formal_generics



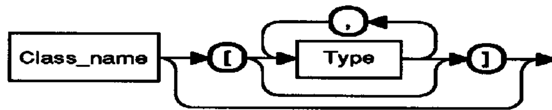
Exports



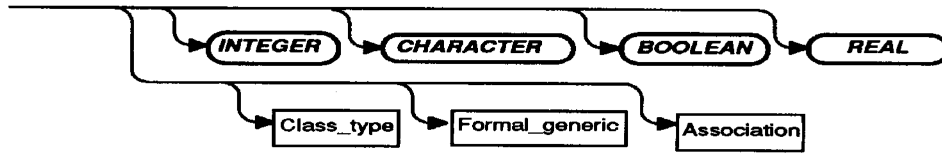
Parents



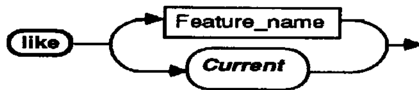
Class_type



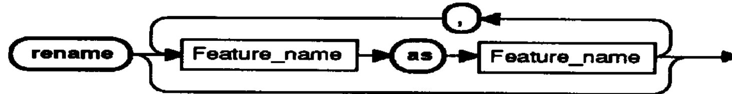
Type



Association



Rename_clause

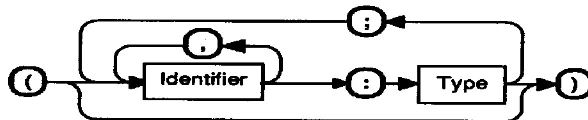


Redefine_clause

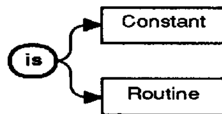
Features

Feature_declaration

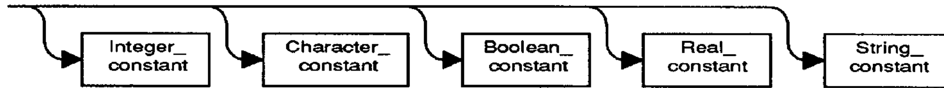
Formal_arguments



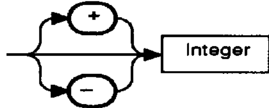
Feature_value_mark



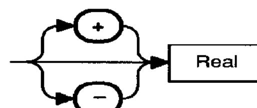
Constant



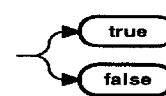
Integer_constant



Real_constant

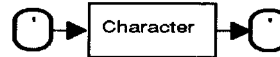


Boolean_constant

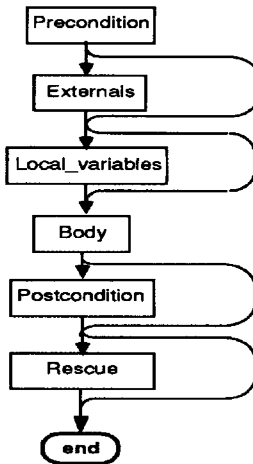


String_constant

Character_constant



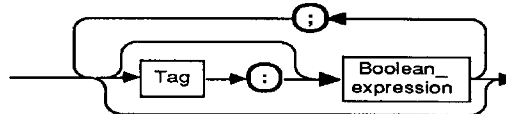
Routine



Precondition

Postcondition

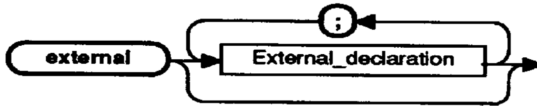
Assertion



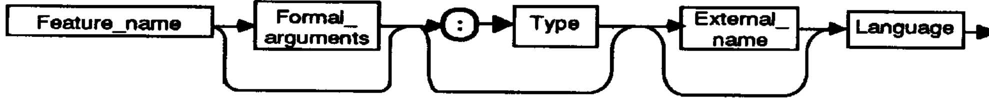
Boolean_expression,
Integer_expression

Rescue

Externals

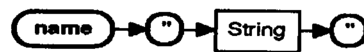


External_declaration

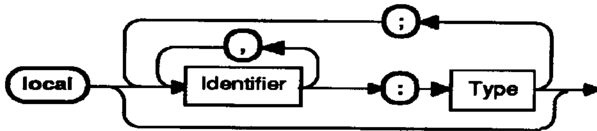


Language

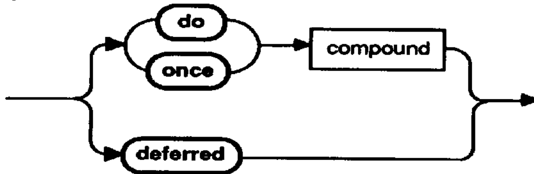
External_name



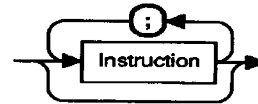
Local_variables



Body

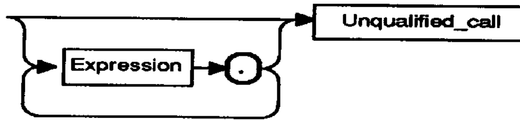


Compound

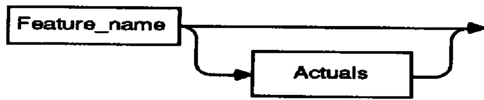


Instruction

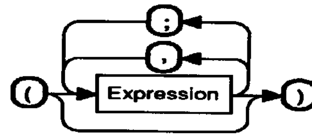
Call



Unqualified_call

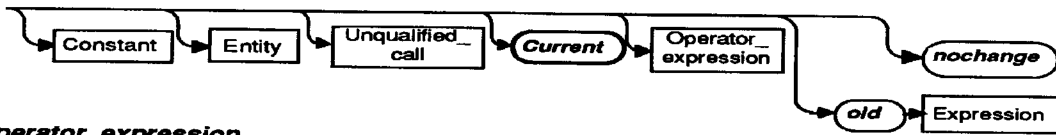


Actuals

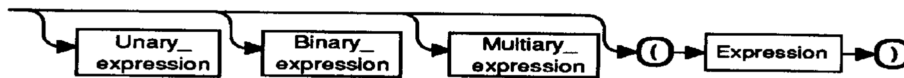


Expression

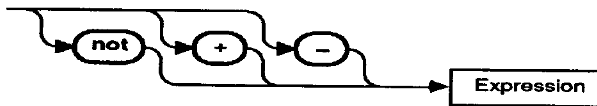
Unqualified_expression



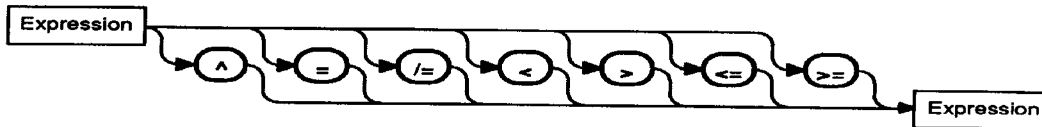
Operator_expression



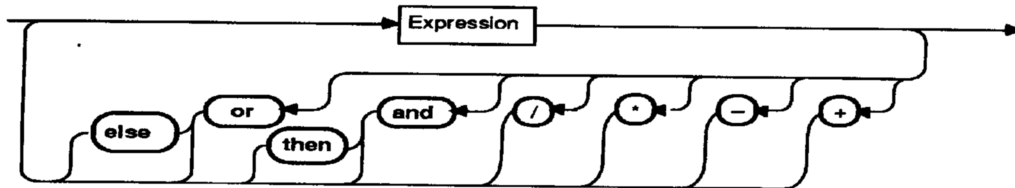
Unary_expression



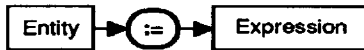
Binary_expression



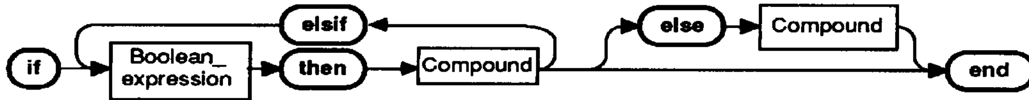
Multiary_expression



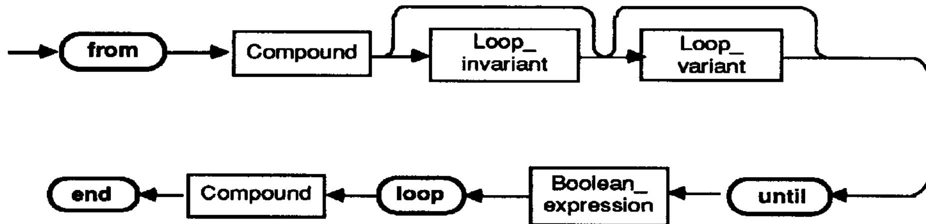
Assignment



Conditional

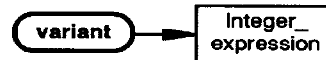


Loop



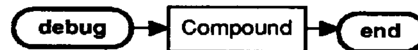
Class_invariant, Loop_invariant

Loop_variant



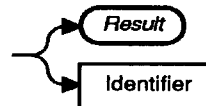
Check

Debug



Class_name, Formal_generic,
Feature_name, Tag

Entity



BIBLIOGRAFIA

AHO86 Aho A.V. , R. Sethi and J.D. Ullman. *Compilers: Principles, techniques, and tools*. Addison-Wesley, 1986. Versión castellana: *Compiladores: Principios, técnicas y herramientas*. Addison-Wesley Iberoamericana, 1990.

ALVA91 Alvarez Rojo, A. R. *Traductor de Pascal a C*. Proyecto Fin de Carrera. Escuela Universitaria de Informática de Oviedo, Universidad de Oviedo, 1991.

BAUE74 Bauer F.L., Eickel J. *Compiler construction: An Advanced Course*. Lecture Notes in Computer Science 21. Springer-Verlag, 1974.

BENN90 Bennett, J. P. *Introduction to compiling techniques. A first course using ANSI C, LEX and YACC*. McGraw-Hill, 1990.

CABA91 Cabal Montes E. y Cueva Lovelle, J.M. *Generador de Analizadores Sintácticos: YACCOV*. Cuaderno Didáctico nº45, Dto. de Matemáticas, Universidad de Oviedo, 1991.

CUEV91 Cueva Lovelle, J.M. *Lenguajes, Gramáticas y Autómatas*. Cuaderno Didáctico nº36, Dto. de Matemáticas, Universidad de Oviedo, 1991.

CUEV93 Cueva Lovelle, J.M. *Análisis léxico en procesadores de lenguaje*. Cuaderno Didáctico nº48, Dto. de Matemáticas, Universidad de Oviedo, 2ª Edición 1993.

CUEV93b Cueva Lovelle, J. M. y Mª P. A. García Fuente. *Programación en FORTRAN*. Cuaderno Didáctico nº 67, Dto. de Matemáticas, Universidad de Oviedo, 1993.

CUEV94 Cueva Lovelle J. M., Mª P. A. García Fuente, B. López Pérez, Mª C. Luengo Díez, y M. Alonso Requejo. *Introducción a la programación estructurada y orientada a objetos con Pascal*. Cuaderno Didáctico nº 69, Dto. de Matemáticas, Universidad de Oviedo, 1994.

CUEV94b Cueva Lovelle, J.M. *Conceptos básicos de Traductores, Compiladores e Intérpretes*. Cuaderno Didáctico nº9, Dto. de Matemáticas, Universidad de Oviedo, 4ª Edición 1994.

ELLI90 Ellis M.A. y Stroustrup B. *The annotated C++ reference manual. ANSI base document*. Addison-Wesley, 1990. Versión en Castellano: *C++ manual de referencia con anotaciones*. Addison-Wesley/ Díaz de Santos, 1994.

GARM91 Garmón Salvador Mª M., J.M. Cueva Lovelle, y Salgueiro Vázquez J.C. *Diseño y construcción de un compilador de C (Versión 2.0)*. Cuaderno Didáctico nº46 Dto. de Matemáticas. Universidad de Oviedo, 1991.

HOLU90 Holub A.I. *Compiler design in C*. Prentice-Hall, 1990.

HOPC79 Hopcroft J. E. y Ullman J.D. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979.

JOHN75 Johnson S.C. Yacc-yet another compiler compiler. *UNIX Programmer's Manual 2.* AT&T Bell Laboratories, 1975.

KATR94 Katrib Mora, M. *Programación orientada a objetos en C++*. Infosys, México, 1994.

KATR94b Katrib Mora, M. *Programación orientada a objetos a través de C++ y Eiffel*. Quinta Escuela Internacional de invierno en temas selectos de computación, Zacatecas, México, 1994.

KERN84 Kernighan B.W. y Pike R. *The UNIX programming environment*. Prentice Hall, 1984. Versión en castellano: *El entorno de programación UNIX*, Prentice-Hall Hispanoamericana, 1987.

KERN88 Kernighan B.W. y D.M. Ritchie. *The C programming language*. *Second Edition* Prentice-Hall, 1988. Versión en castellano: *El lenguaje de programación C. Segunda edición*. Prentice-Hall, 1991.

LEIV93 Leiva Vázquez J.A. y Cueva Lovelle J.M. *Construcción de un traductor de lenguaje Eiffel a C++*. Cuaderno didáctico nº 76. Departamento de Matemáticas. Universidad de Oviedo (1993).

LESK75 Lesk M.E. y Schmidth E. Lex-a lexical analyzer generator. *UNIX Programmer's Manual 2.* AT&T Bell Laboratories, 1975.

LEVI92 Levine J. R., T. Mason, Brown D. *lex & yacc*. *UNIX programming tools*. O'Reilly & Associates (1992).

MART93 Martínez García, J.M. y Cueva Lovelle, J.M. *Generador de analizadores léxicos: GALEX*. Cuaderno Didáctico Nº 66. Dto. de Matemáticas, Universidad de Oviedo, 1993.

MEYE88 Meyer B. *Object-oriented Software Construction*. Prentice-Hall 1988.

MEYE92 Meyer B. *Eiffel. The language*. Prentice-Hall 1992.

PYST88 Pyster A. B. *Compiler design and construction (with C, Pascal and UNIX tools)*. Second Edition. Ed. Van Nostrand Reinhold, 1988.

SANC86 Sanchís Llorca, F.J. y C. Galán Pascual *Compiladores: Teoría y construcción*. Ed. Paraninfo, 1986.

SANC89 Sanchez Dueñas, G. y J.A. Valverde Andreu. *Compiladores e intérpretes. Un enfoque pragmático*. Ed. Diaz de Santos, 2ª edición, 1989.

SCHR85 Schreiner T. A. and Friedman H.G. Jr. *Introduction to compiler construction with UNIX*. Prentice-Hall, 1985.

STRO86 Stroustrup B. *The C++ programming language*. Addison-Wesley 1986.

STRO91 Stroustrup B. *The C++ programming language*. Second Edition. Addison-Wesley 1991. Versión en castellano: *El lenguaje de programación C++*, segunda edición, Addison-Wesley/Díaz de Santos, 1993.

STRO94 Stroustrup B. *The Design and Evolution of C++*. Addison-Wesley 1994.

TREM85 Tremblay J. P. and P.G. Sorenson. *The theory and practice of compiler writing*. Ed. McGraw-Hill, 1985.

WAIT85 Waite M. W. and Goos G. *Compiler construction*. Springer- Verlag, second edition 1985.

WATT91 Watt D.A. *Programming Language Syntax and Semantics*. Prentice-Hall, 1991.

WIRT76 Wirth, N. *Algorithms+data structures= programs*, Prentice-Hall, 1976. Versión Castellana *Algoritmos+estructuras de datos= programas*. Ed. del Castillo, 1980.

Index

Accion semántica, 60, 89
Acciones, 75, 80
Algoritmo de decisión
  Lewis, 33
Ambigüedad, 84
Análisis ascendente
  con retroceso, 58
Análisis sintáctico
  descendente, 84
Analizador
  léxico, 85, 88
  sintáctico, 74
Analizador sintáctico
  LALR, 58
  LR, 58, 74
  LR canónico, 58, 66
  shift/reduce, 57, 58
  SLR, 58, 62, 66
Analizadores predictivos, 22
Analizadores sintácticos
  Recursivo descendentes, 53
Analizadores sintácticos descendentes, 22
Arbol sintáctico, 1
Asociatividad, 78, 85, 86
Autómata de pila, 52

Backtracking
  retroceso, vuelta atrás, 22
BNF, 74

Chomsky, 15
Código C adicional, 80, 88, 89, 97
Conflicto, 84
  reduce/reduce, 78, 86
  shift/reduce, 74, 78, 84, 85, 87
Conflicto s/r, 96
Conjunto de items, 63
  LR(1), 65, 66, 69
Construcción $$, 81, 82
Construcción $N, 81, 82
Construcciones "@N", 89

Declaración %expect, 78, 85, 97
Declaración %left, 78, 87
Declaración %nonassoc, 78, 87
Declaración %prec, 86
Declaración %pure_parser, 78
Declaración %right, 78, 87
Declaración %semantic_parser, 79, 89, 97
Declaración %start, 78
Declaración %token, 77, 80
Declaración %type, 78, 80
Declaración %union, 77, 80, 81, 89
Declaración pure_parser, 97
Declaraciones C, 77
Declaraciones YACCOV, 77
Derivación, 84
  más a la derecha, 59
  más a la izquierda, 59
Descripción de la gramática, 79

Expresiones aritméticas, 1, 46

Fichero de entrada, 76
FIRST
  Conjunto de símbolos INICIALES, 29
FOLLOW
  Conjunto de símbolos SEGUIDORES, 31
Forma de frase, 59
Forma de frase derecha, 59
Forma de frase izquierda
  más a la izquierda, 59
Frase, 59
Función accion, 59
Función acción, 61, 69
Función cierre, 62, 65
Función goto, 59, 61, 62, 63, 65
Función main, 75, 87
Función yyerror, 75, 80, 88
Función yylex, 75, 76, 80, 88
Función yyparse, 74, 75, 76, 77, 81, 85, 87

Generador de analizadores sintácticos, 74
Gramática
  ambigüa, 84
  LALR(1), 69
  libre de contexto, 74, 82
  LR, 62
  LR(1), 66
  SLR, 62
  SLR(1), 66
Gramática LL(1), 32
Gramáticas libres de contexto
  Gramáticas tipo 2, 1
Gramáticas LL(1) simples, 30
gramáticas LL(k), 62
Gramáticas LL(k), 28

if-then-else, 46
INICIALES
  FIRST, 29
Item
  LR(0), 62
  LR(1), 65

Kernel, 70
Knuth
  Condiciones de, 32

LEX, 88
lookahead, 90, 96
Lookahead, 58, 62, 65, 89

Núcleo, 68, 69

Precedencia, 78, 84, 85, 86
Prefijo viable, 63, 64
Prefijos viables, 59

Recursividad, 82
  a la derecha, 83, 84
  a la izquierda, 82, 84
  directa, 84
  indirecta, 84
Recursividad a izquierdas, 25, 44
Reducción, 57, 84
Regla de producción, 75, 79, 80, 82, 86
Relación binaria, 16

S-gramáticas, 28

Sección de declaraciones, 76, 85, 89
Seccion de declarciones, 97
SELECT
  Conjunto de símbolos directores, 32
Semántica, 80
Símbolo, 76
  a leer, 60
  no terminal, 76, 79, 80, 81
  terminal, 76, 79, 81
Símbolos directores, 32
Símbolos seguidores, 31
Sintaxis, 80

Tabla de análisis, 59, 60, 61, 70
  LALR, 67, 69, 70
  LR canónica, 65, 67
  SLR, 62, 64, 67
Tabla goto, 69
Tipo YYSTYPE, 89
Token, 76, 85, 87, 88
Token error, 76, 90
Transformación de gramáticas
  LL(1), 43
Tratamiento de errores sintácticos, 55

UNIX, 74, 88

Valor semántico, 76, 77, 80, 82, 88, 89
Variable, 76
Variable yylloc, 78
Variable yylval, 78, 88

Warshall, 21
Wirth, 53

yacc, 74
  BISON, 74
  PCYACC, 74
  YACCOV, 74