Procesadores de Lenguajes I:

Una introducción a la fase de análisis

J. M. Benedí Ruíz
V. Gisbert Giner
L. Moreno Boronat
E. Vivancos Rubio

Capítulo 5

Análisis semántico

En este capítulo se establecen las condiciones básicas que permiten el desarrollo del análisis semántico y de la generación de código de un compilador. Para ello se presenta una técnica general de especificación, asociada a la definición sintáctica de un lenguaje de programación: las gramáticas atribuidas. Esta técnica de especificación permite establecer comprobaciones semánticas, actualizar y consultar la tabla de símbolos y generar código intermedio. También se estudian los métodos de evaluación e implementación, asociados al análisis sintáctico de estas gramáticas atribuidas. Finalmente, se aplican estas técnicas para especificar el análisis semántico y, en particular, el estudio del análisis y control de tipos.

5.1. Introducción al problema. Especificación semántica

5.1.1. Definiciones básicas

Sea G=(N,Σ,P,S) una gramática independiente del contexto, asociemos a cada $X_i \in (N \cup \sum)$ un conjunto de identificadores a_{ij} para $j \in [1..\ n_i]$ a los que llamaremos ***atributos*** asociados al símbolo X_i. En algún caso el conjunto de atributos asociados a un símbolo podrá ser vacío. Habitualmente nos referiremos al atributo a_{ij} de X_i como $X_i.a_j$

Ejemplo 5.1.- Sea G=({S, A, B}, {c, d}, {S → AB, A → c B, B → d}, S)

Si asociamos al no terminal A los identificadores a1 y a2, podremos hablar de los atributos a1 y a2 de A y hacer referencia a ellos mediante las expresiones A.a1 y A.a2. Algo similar podríamos hacer con B, c y d.

El número de atributos de un símbolo puede ser cualquiera y es independiente del número que tengan los otros símbolos de la gramática, y en principio, pueden representar cualquier tipo de información.

El conjunto de atributos de un símbolo puede considerarse como un registro asociado a él donde cada atributo sería un campo de este registro.

Ejemplo 5.2.- Sea G=({S,E,T,F}, {num, +, *, (,) }, P, S) donde P =

S → E

E → E + T | T

T → T * F | F

$F \rightarrow (E) \mid num$

Asociemos a E, T, F y num el identificador 'val'. Podremos hablar del atributo 'val' asociado a E, refiriéndonos a éste como E.val , análogamente nos referiremos a los atributos T.val , F.val y num.val.

Los nombres de los atributos asociados a diferentes símbolos pueden coincidir como en el ejemplo anterior.

Llamaremos ***regla semántica*** a cualquier asignación donde la parte izquierda sea un atributo $X.a_j$ y cuya parte derecha sea una función cuyos argumentos podrán ser, a su vez, otros atributos, por ejemplo,

Dada la producción:

$X_0 \rightarrow X_1 X_2 X_3 \ldots X_n \qquad (1)$

$$X_{i_0}.a_{i_0 j_0} := f(X_{i_1}.a_{i_1 j_1}, X_{i_2}.a_{i_2 j_2}, \ldots, X_{i_p}.a_{i_p j_p}, z_1, z_2, \ldots z_q)$$

es una regla semántica, donde los argumentos z_i no corresponden a atributos de la gramática y **p** y **q** pueden valer cero. Es decir, puede que no aparezca ningún atributo como argumento o ningún argumento que no sea atributo.

Por ejemplo A.a1 := c.c + B.b podría ser una regla semántica asociada a la producción A $\rightarrow$ c B del Ejemplo 5.1.

Entenderemos por ***Gramática de atributos o atribuida*** a una G.I.C. a cuyos símbolos se les han asociado atributos y a cuyas producciones se les han asociado reglas semánticas. Los atributos que ocurran en una regla semántica sólo podrán pertenecer al conjunto de atributos de los símbolos que aparezcan en la producción asociada a esa regla semántica.

Es decir: Sea $X_0 \rightarrow X_1 X_2 X_3 \ldots X_n$ una producción de la gramática. Entonces en cualquier regla semántica asociada a esta producción sólo podrán ocurrir los atributos asociados a los símbolos X_i con $i \in [0..n]$.

Por ejemplo la regla semántica:

A.a1:= d.c + B.b

no puede estar asociada a la producción B $\rightarrow$ d porque el símbolo A asociado al atributo A.a1 no ocurre en esta producción.

Un mismo símbolo, normalmente un no terminal, puede ocurrir en una misma producción varias veces, en ese caso, con la notación introducida, un atributo de este símbolo puede quedar indefinido. Con el fin de evitar este problema los símbolos que se repitan en una producción deberán ser indexados y en consecuencia, la referencia a los atributos correspondientes. Por

ejemplo si queremos asociar a la producción $E \rightarrow E + T$ una regla semántica que calcule el atributo 'val' de la primera ocurrencia de E a partir de los atributos 'val' de los restantes símbolos podemos expresarlo como:

$E \rightarrow E_1 + T$ { E.val := E_1.val + T.val }

Ejemplo 5.3.- El siguiente ejemplo de gramática de atributos permite calcular las expresiones aritméticas generadas por la gramática.

$S \rightarrow E$	{ S.val := E.val }
$E \rightarrow E_1 + T$	{ E.val := E_1.val + T.val }
$E \rightarrow T$	{ E.val := T.val }
$T \rightarrow T_1 * F$	{ T.val := T_1.val * F.val }
$T \rightarrow F$	{ T.val := F.val }
$F \rightarrow (E)$	{ F.val := E.val }
$F \rightarrow num$	{ F.val := num.val }

Obsérvese que cada atributo una vez evaluado representa el valor de la expresión que se deriva a partir del no terminal correspondiente a cada atributo. El atributo num.val representa el valor numérico asociado al terminal num.

- Diremos que el atributo $a_{i j_0}$, asociado al símbolo X_i, es un ***atributo sintetizado*** si para cualquier regla semántica que lo evalúe se cumple, que X_i ,su símbolo asociado, aparece en la cabecera de la producción a la cual pertenece cada una de las reglas semánticas.

  Por lo tanto, las reglas que evalúen un atributo sintetizado tendrán, en el supuesto de (1), la forma siguiente:

  $$X_0.a_{0j_0} := f(X_{i_1}.a_{i_1 j_1}, X_{i_2}.a_{i_2 j_2}, \ldots, X_{i_p}.a_{i_p j_p}, z_1, z_2, \ldots z_q)$$

  Por ejemplo los atributos 'val' del ejemplo de las expresiones aritméticas son todos atributos sintetizados.

  No lo sería el atributo A.a de

  $B \rightarrow A$ { A.a := B.b }

- Diremos que el atributo $a_{i_0 j_0}$ es un ***atributo heredado*** si para cualquier regla semántica que lo evalúe se cumple, que su símbolo asociado aparece en la parte derecha de la producción a la cual pertenecen cada una de las reglas semánticas.

  Por lo tanto, las reglas que evalúen un atributo heredado tendrán, en el supuesto de (1), la forma siguiente:

$$X_{i_0}.a_{i_0 j_0} := f(X_{i_1}.a_{i_1 j_1}, X_{i_2}.a_{i_2 j_2}, \ldots, X_{i_p}.a_{i_p j_p}, z_1, z_2, \ldots z_q)$$

con i_0 perteneciente al conjunto {1, 2 … n }

Ejemplo 5.4.- La gramática de atributos del siguiente ejemplo pasa a forma decimal números binarios y números hexadecimales. El lenguaje generado por la gramática es (b|h) (0|1|2 ….d|e|f)*

S → b C	{ C.base:= 2 ; S.val := C.val }
| h C	{ C.base := 16 ; S.val := C.val }
C → C_1 D	{ C_1.base := C.base ; C.val := C_1.val * C.base + D.val }
C → D	{ C.val := D.val }
D → 0	{ D.val := 0 }
| 1	{ D.val := 1 }
| 2	{ D.val := 2 }
……..	
| f	{ D.val := 15 }

En esta gramática atribuida, el atributo “base” del no terminal C es heredado y los atributos “val” de C, S y D son sintetizados. Obsérvese el papel del atributo “base”, permite “informar”, desde la raíz del árbol sintáctico hasta las hojas, del carácter binario o hexadecimal de la cadena y utilizar su valor para el cálculo del valor decimal.

- ***Reglas con efectos de borde***. Es posible incluir reglas semánticas que no evalúen atributos o que además de evaluar algún atributo realice alguna otra acción semántica con efectos colaterales, globales, ...

En el Ejemplo 5.3 la acción semántica de la primera producción podría sustituirse por:

S → E { print (E.val)}

Esta acción semántica no evalúa nada pero podría permitir disponer de ese valor en algún dispositivo de salida.

Desde el punto de vista formal podemos asignar un atributo “virtual”, que consideraremos de tipo sintetizado, al símbolo de la izquierda de la producción a la cual están asignadas las reglas semánticas que no evalúen atributos. En el ejemplo anterior:

S → E { S.virtual := print (E.val)}

Ejemplo 5.5.- Ocurrencia de un identificador en la declaración.

S → S D	{ }
| D	{ }
D → T : id L	{ L.tipo := T.tipo ; AñadirTipo(id.lexema , T.tipo) }

$L \rightarrow$, id L_1	{ L_1.tipo := L.tipo ; AñadirTipo(id.lexema , L.tipo) }
|	ε	{ }
$T \rightarrow$ integer	{ T.tipo := Número_Entero }
$T \rightarrow$ real	{ T.tipo := Número_Real }

Las acciones semánticas de esta gramática atribuida permiten asignar a cada identificador su tipo. Suponemos para ello, que el procedimiento AñadirTipo accede, a través de su primer argumento, al registro correspondiente en la Tabla de Símbolos y rellena el campo "tipo" con el valor de su segundo argumento.

5.2. Evaluación de atributos

Sea G una gramática de atributos. Dado un árbol de derivación sintáctica para una cadena generada por esta gramática llamaremos ***árbol de derivación anotado*** al árbol resultante de sustituir cada nodo del árbol de derivación sintáctica por todos los atributos asociados al símbolo de ese nodo. Si un nodo no tiene ningún atributo asociado dejaremos como etiqueta del nodo la misma que la del árbol sintáctico.

Ejemplo 5.6.- Construyamos el árbol anotado para la frase integer : a, b, c de la gramática del Ejemplo 5.5. En primer lugar asociemos a la acción semántica AñadirTipo(_ , _) un atributo sintetizado virtual, por ejemplo L.v. A continuación construimos el árbol de análisis sintáctico para esta frase. (Fig. 5.1).

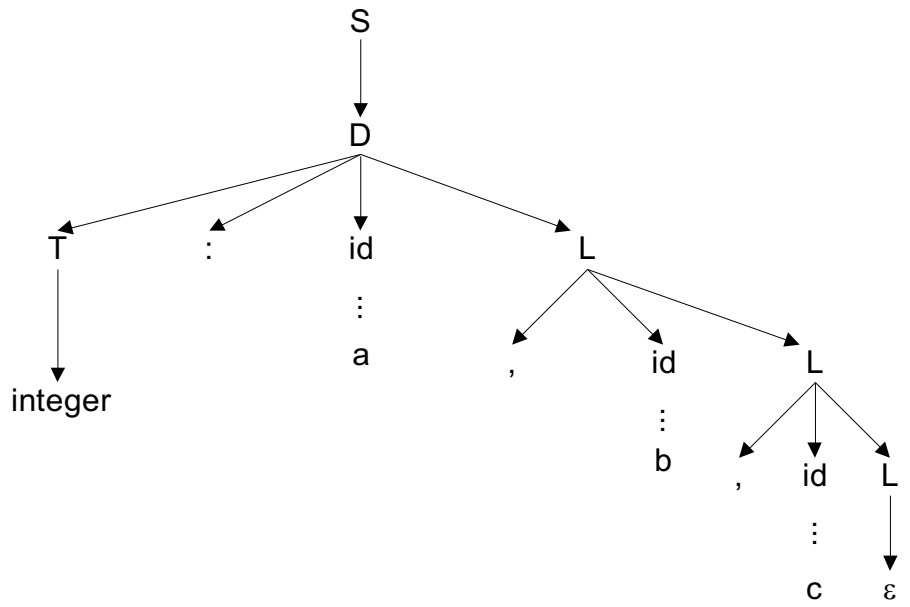


Figura 5.1. Árbol de análisis sintáctico.

De acuerdo a la definición, el árbol anotado sería el de la Fig. 5.2.

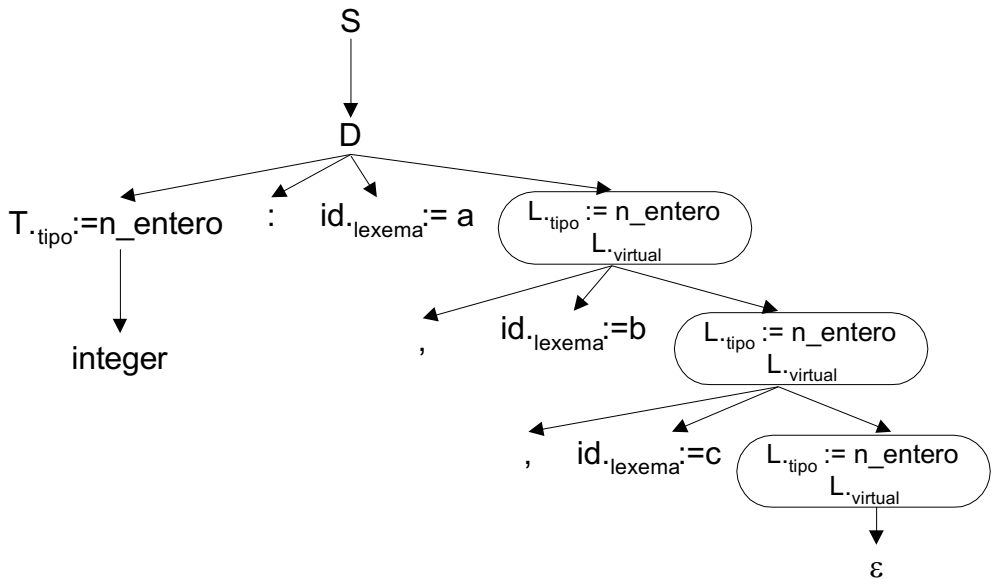


Figura 5.2. Árbol de derivación anotado.

La dependencia funcional que establecen las reglas semánticas obligan a seguir un orden determinado de evaluación de éstas. Por ejemplo, en el árbol anotado anterior, el atributo 'tipo' del no terminal L no se puede calcular si antes no conocemos el valor del atributo 'tipo' de T.

Sea G una gramática de atributos, dado un árbol de derivación sintáctica para una cadena generada por esta gramática llamaremos ***grafo de dependencias*** al grafo construido mediante las siguientes reglas:

1º Por cada nodo del árbol sintáctico y por cada atributo asociado al símbolo de este nodo construiremos un nodo en el grafo de dependencias.

2º Dados dos nodos X.a y Y.b del grafo estableceremos un arco del primero al segundo si existe una regla semántica de la forma:

$$Y.b := f(X_{i_1}.a_{i_1 j_1}, X_{i_2}.a_{i_2 j_2}, \ldots, X.a, \ldots, X_{i_p}.a_{i_p j_p}, z_1, z_2, \ldots z_q)$$

y los símbolos X e Y pertenecen a un mismo subárbol, de profundidad uno, del árbol de análisis sintáctico. En otras palabras, la regla semántica usa el valor de X.a para calcular el valor de Y.b

Ejemplo 5.7.- El grafo de dependencias del Ejemplo 5.6 sería el de la Fig. 5.3.

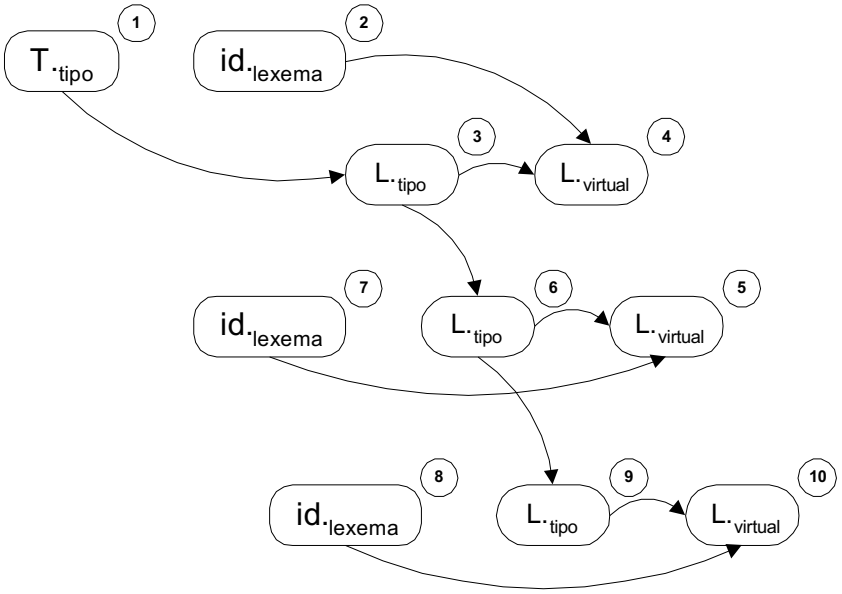


Figura 5.3. Grafo de dependencias.

Dado un grafo de dependencias asociado a un árbol de derivación, supondremos numerados los nodos de este grafo, diremos que una secuencia $n_1, n_2, \ldots, n_p$ es un ***orden topológico de evaluación del grafo*** si se cumple que para cualquier par de nodos n_i y n_j, si $i < j$ entonces no hay ningún arco de n_j a n_i .

En el ejemplo anterior el orden 1, 2, 3, 4, 7, 6, 5, 8, 9, 10 sería un orden topológico de evaluación del grafo.

Es necesario determinar un orden topológico, ya sea explícitamente o implícitamente, si se desean evaluar los atributos de un árbol anotado. Puesto que no es posible evaluar un atributo sin conocer el valor de los argumentos de la función que lo calcula. Esta dependencia queda de manifiesto al construir el grafo de dependencias.

Los tipos básicos de determinación de ordenes topológicos pueden ser: generales, por reglas y simples.

1. **Generales**. Para cada árbol de análisis se obtiene el grafo de dependencias y luego se busca un orden topológico mediante un algorítmo genérico. (Las hojas de cálculo utilizan ese método, cuando se actualizan según el orden natural).

   El mecanismo de determinación utilizado en los ejemplos anteriores podría ser de tipo general, puesto que el orden topológico se ha calculado específicamente para cada ejemplo.

2. **Por reglas**. Se determinan un conjunto de reglas fijas que se utilizaran siempre para determinar el orden de evaluación para cualquier árbol sintáctico.

Ejemplo 5.8.-

$S \rightarrow AB$	{A.h := B.n , print(A.s)}
$A \rightarrow A_1\ a$	{A.s := A_1.s + 2*(A.h - A_1.n) , A_1.h := A.h , A.n := A_1.n + 1}
$A \rightarrow \varepsilon$	{A.s := 0 , A.n := 0 }
$B \rightarrow B_1\ b$	{ B.n := B_1.n + 1}
$B \rightarrow \varepsilon$	{B.n := 0 }

Independientemente del árbol sintáctico que se construya en el proceso de análisis, el siguiente orden es topológico: evaluar los atributos de B desde las hojas, evaluar los atributos A.h desde la raíz, evaluar los atributos A.n desde las hojas y finalmente evaluar A.s desde las hojas.

- **Simples**. En estos métodos se elige un orden de evaluación sin tener en cuenta las reglas semánticas. Normalmente el orden viene determinado por el método de análisis (ascendente o descendente). A las gramáticas atribuidas cuyos atributos no son susceptibles de evaluarse de esta forma, para cualquier frase, no se les pueden aplicar este tipo de métodos de evaluación. Los apartados siguientes están dedicados a estos últimos métodos.

5.3. Gramáticas S - atribuidas

5.3.1. Definición

Diremos que una gramática de atributos es ***S - atribuida*** si todos los atributos de todos sus símbolos son atributos sintetizados. Téngase en cuenta que las acciones semánticas que no evalúan ningún atributo se consideran como atributos sintetizados virtuales.

Ejemplo 5.9.- La siguiente gramática S - atribuida calcula el valor decimal de un número binario entero y/o decimal. El atributo 'n' cuenta el número de cifras de la parte decimal. (Cuenta este número en cualquier caso pero sólo se utiliza en la parte decimal)

$A \rightarrow L\ .\ L_1$	{ print(L.val + L_1.val * $2^{-L_1.n}$ }
| L	{ print(L.val)}
$L \rightarrow L_1\ B$	{ L.val := L_1.val * 2 + B.val ; L.n := L_1.n + 1 }
| B	{ L.val := B.val ; L.n := 1 }
$B \rightarrow 0$	{ B.val := 0 }
| 1	{ B.val := 1 }

5.3.2. Evaluación de los atributos

1. Ascendente asociado a un analizador LR

Supongamos inicialmente que el número máximo de atributos sintetizados evaluables, no virtuales, es uno por símbolo de la gramática.

Asociemos a la pila del analizador una pila paralela, en la cual almacenaremos los valores de los atributos asociados al símbolo de la pila del analizador.

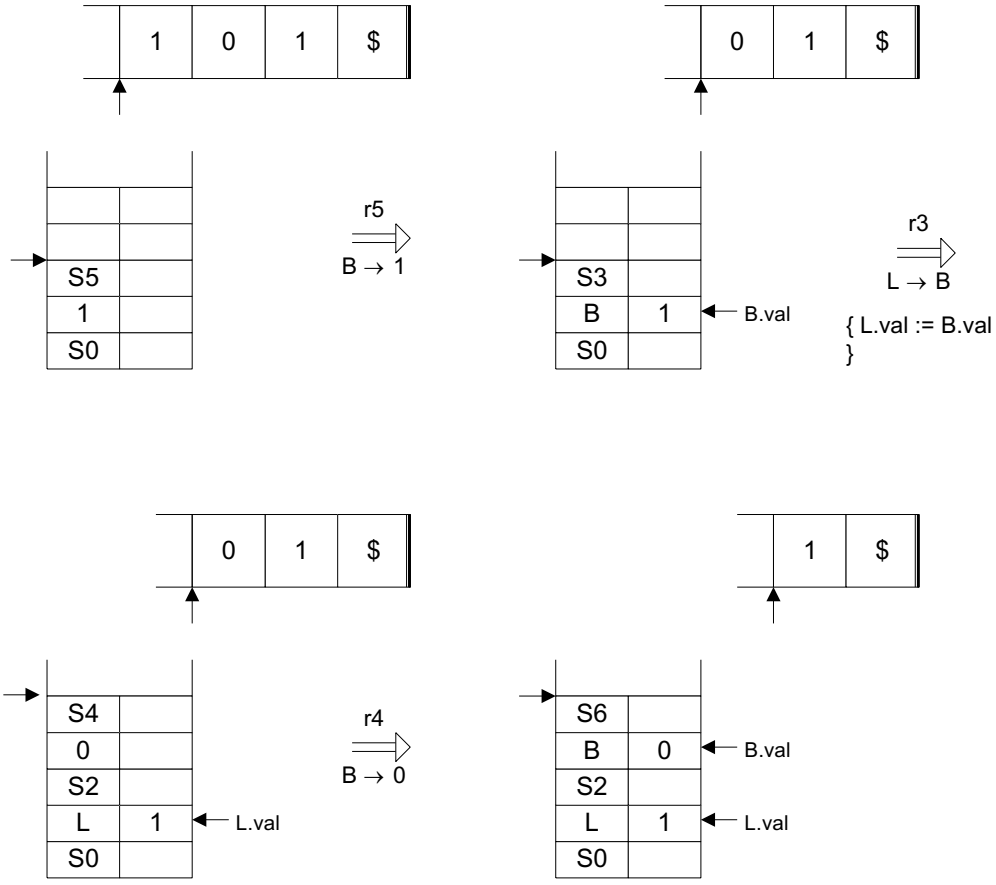
Cada vez que el analizador realice una reducción se evaluarán las acciones semánticas asociadas a esa producción. El resultado se almacenara en la cumbre de la pila paralela junto al no terminal apilado tras la reducción.

Cada vez que el analizador apile un símbolo correspondiente a la cadena de entrada (acción desplazar), el analizador léxico habrá leído un nuevo token, si este símbolo tiene un atributo sintetizado -entrada TDS, valor, …- su valor se almacenará en la pila paralela, análogamente al caso anterior.

Ejemplo 5.10.- Suprimamos la producción A → L . L en el Ejemplo 5.9 y no consideremos el atributo 'n' ni las acciones semánticas que lo evalúan. La tabla de análisis SLR(1) para esta gramática vendría dada por la Fig. 5.4 y la traza para la frase 101 por la Fig. 5.5.

	0	**1**	**$**		**A**	**L**	**B**
S0	dS4	dS5			S1	S2	S3
S1			ACEPTAR				
S2	dS4	dS5	r1				S6
S3	r3	r3	r3				
S4	r4	r4	r4				
S5	r5	r5	r5				
S6	r2	r2	r2				

Figura 5.4. Tabla de análisis SLR(1)


1 0 1 $
S5
1
S0
r5
B → 1
0 1 $
S3
B 1
S0
B.val
r3
L → B
{ L.val := B.val
}
0 1 $
S4
0
S2
L 1
S0
L.val
r4
B → 0
1 $
S6
B 0
S2
L 1
S0
B.val
L.val

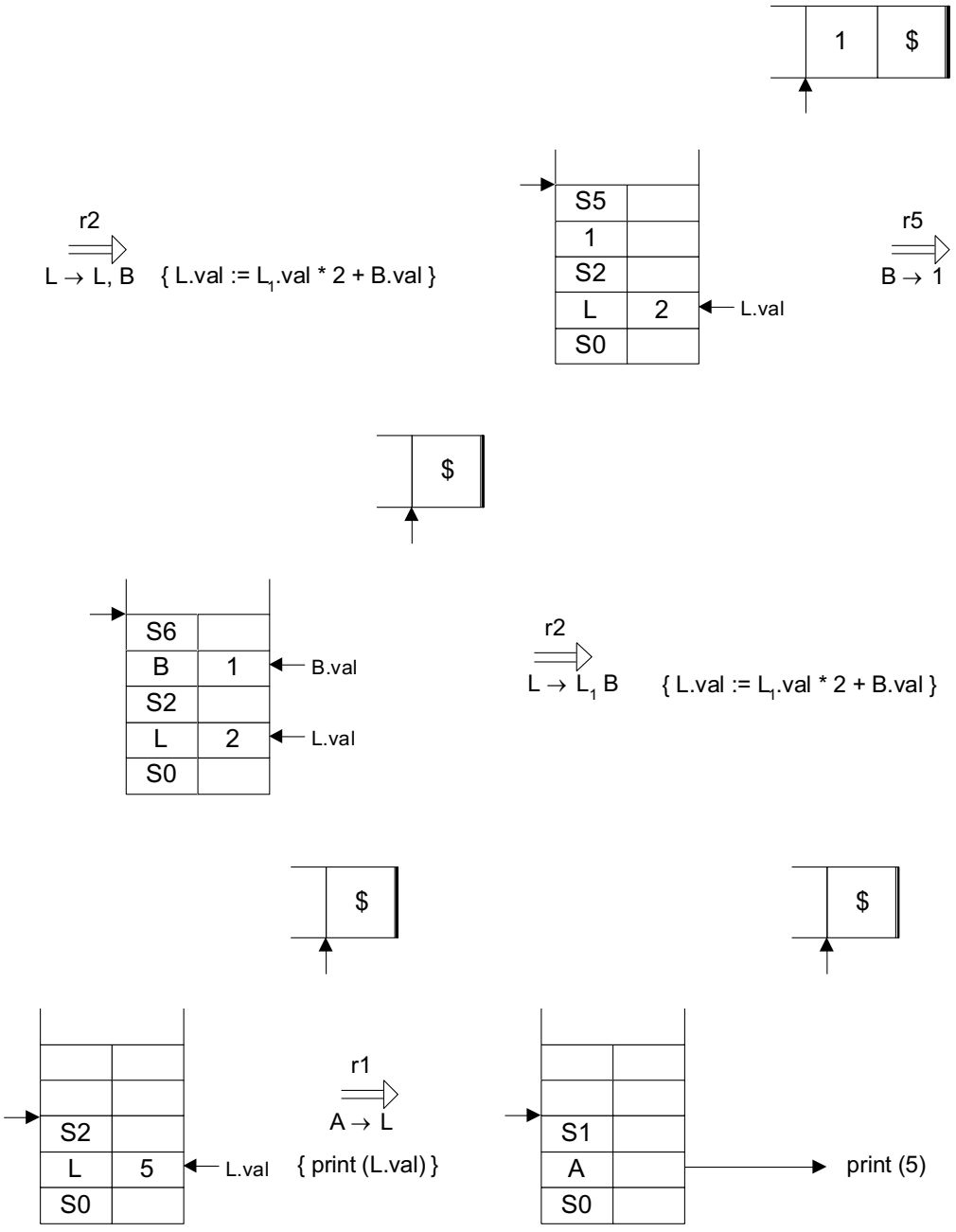


Figura 5.5. Traza para la frase 101

Teorema. Con este método cualquier analizador LR evaluará correctamente los atributos de cualquier gramática S - atribuida.

El anterior esquema de evaluación ascendente de un atributo sintetizado es fácilmente generalizable para gramáticas S-atribuidas con más de un atributo para algún símbolo. Se pueden usar múltiples pilas paralelas o una pila en la que los valores a almacenar sean registros (estos registros almacenarían en campos diferentes los diferentes atributos)

2. Descendente asociado a un A.D.R.

Se verá más adelante dentro de las gramáticas L - atribuidas)

3. Descendente asociado a un analizador mediante tabla LL(1)

La construcción de un evaluador para este caso exige la construcción previa del árbol de análisis.

Ejemplo 5.11. Para la siguiente gramática S-atribuida, en la cual las acciones semánticas calculan el número decimal correspondiente a un número binario,

$B \rightarrow 0\ B_1$ { B.v := B_1.v , B.n := B_1.n + 1 }

$B \rightarrow 1\ B_1$ { B.v := 2^B_1.n + B_1.v , B.n := B_1.n + 1 }

$B \rightarrow \varepsilon$ { B.v := 0 , B.n := 0 }

la traza del análisis mediante tabla LL(1) de la cadena 101 sería:

Pila	Entrada	Acción
B$	101$	$B \rightarrow 1B$
1B$	101$	pop
B$	01$	$B \rightarrow 0B$
0B$	01$	pop
B$	1$	$B \rightarrow 1B$
1B$	1$	pop
B$	$	$B \rightarrow \varepsilon$
$	$	Aceptar

Obsérvese que no es posible evaluar los atributos simultáneamente al proceso de análisis, ya que el valor inicial para los atributos, B.v := 0 y B.n := 0, no se conoce hasta el último paso del análisis. En ese momento ya no hay ningún símbolo en la pila.

Los atributos se pueden evaluar si primero se construye el árbol de análisis sintáctico. Una vez construido el árbol asóciese a cada nodo las funciones que evalúan cada uno de los atributos del símbolo asignado a ese nodo. Estas funciones pueden ser asignarse de una manera determinista, se conocen los hijos de cada nodo y por lo tanto la producción aplicada.

Después se debe calcular las funciones así definidas de abajo a arriba, en el árbol de análisis. Puede ser aplicado cualquier orden siempre que no se evalúe un nodo antes de evaluar sus hijos.

5.4. Gramáticas L - atribuidas

5.4.1. Definición

Una gramática de atributos es ***L-atribuida*** si para cualquier atributo heredado y para cualquier regla semántica que lo evalúe, los argumentos de ésta sólo pueden depender de atributos de

símbolos situados a la izquierda del símbolo del atributo que se evalúe. En otros términos sea $X_{i.h}$ un atributo heredado del símbolo X_i, sea

$$X_{i\cdot h} := f(X_{i_1}.a_{i_1 j_1}, X_{i_2}.a_{i_2 j_2}, \ldots, X_{i_p}.a_{i_p j_p}, z_1, z_2, \ldots z_q)$$

una regla que lo calcule, y $X_0 \rightarrow X_1\ X_2\ \ldots..\ X_i\ \ldots.\ X_k$ la producción donde ocurre la regla, entonces los atributos que aparecen como argumentos de la función sólo podrán corresponder a los símbolos $X_0, X_1, X_2, \ldots X_{i-1}$

Consecuentemente toda gramática que sólo tenga atributos sintetizados será una gramática L-atribuida.

Ejemplo 5.12.- La siguiente gramática es L-atribuida.

$S \rightarrow B$	{ B.v := 0, print(B.t) }
$B \rightarrow 0\ B_1$	{ B_1.v := B.v * 2 , B.t := B_1.t }
$B \rightarrow 1\ B_1$	{ B_1.v := B.v * 2 + 1 , B.t := B_1.t }
$B \rightarrow \varepsilon$	{ B.t := B.v }

En cambio, la siguiente gramática no sería L-atribuida, puesto que el atributo L.tipo en la producción D → id L : T depende del atributo T.tipo, correspondiente a un símbolo situado a su derecha.

$S \rightarrow S\ D$	{ }
| D	{ }
$D \rightarrow$ id L : T	{ L.tipo := T.tipo ; AñadirTipo(id.lexema , T.tipo) }
$L \rightarrow$, id L_1	{ L_1.tipo := L.tipo ; AñadirTipo(id.lexema , L.tipo) }
| ε	{ }
T → integer	{ T.tipo := Número_Entero }
T → real	{ T.tipo := Número_Real }

5.4.2. Evaluación de los atributos

Los atributos de una gramática L-atribuida pueden evaluarse realizando un recorrido de primero en profundidad y de izquierda a derecha del árbol de análisis sintáctico. El siguiente algoritmo define con más precisión este método de evaluación.

Sea G una gramática L-atribuida entonces para cualquier cadena del lenguaje generado por G,

1º Constrúyase el árbol sintáctico anotado

2º Recórrase el árbol de arriba a abajo y de izquierda a derecha.

3º En cada nodo evalúense los atributos heredados en el primer acceso a dicho nodo.

4º En cada nodo evalúense los atributos sintetizados cuando se vuelva a él, después de recorrer todo el subárbol dependiente de este nodo.

El algoritmo anterior evalúa correctamente todos los atributos de una gramática si esta es una gramática L-atribuida. (Bajo la hipótesis de que las reglas semánticas están bien definidas). Por tanto, la evaluación de los atributos de estas gramáticas caen dentro de los grupos de "reglas fijas" y "métodos simples" de evaluación mencionados en el punto dos.

Según el anterior algoritmo podemos establecer con precisión en que momento se evalúa cada regla semántica con relación a cada uno de los símbolos que aparecen en cada una de las producciones.

Por otra parte el formalismo visto hasta ahora resulta insuficiente para algunos casos -acciones semánticas con efecto colateral- en los que se precisa indicar en que momento hay que evaluar la regla semántica.

Esta situación es abordable en este contexto porque en las gramáticas L-atribuidas hay un orden de evaluación subyacente.

Un ***Esquema de traducción dirigido por la sintaxis (E.T.D.S.)*** es una gramática de atributos excepto que las acciones semánticas pueden aparecer al principio, entre y al final de los símbolos de la parte derecha de las producciones de la gramática.

Un E.T.D.S. lleva implícito un orden de evaluación análogo al descrito anteriormente. Dada cualquier cadena del lenguaje:

1º Constrúyase el árbol sintáctico anotado

2º Recórrase el árbol de arriba a abajo y de izquierda a derecha.

3º En cada nodo ejecútense las acciones semánticas que precedan inmediatamente al símbolo correspondiente a este nodo.

4º En cada nodo ejecútense las acciones semánticas que aparecen a la derecha del último símbolo de la producción y evalúense los atributos sintetizados cuando se vuelva a este nodo, después de recorrer todo el subárbol dependiente de éste.

En un E.T.D.S. el orden en aparecen las regla semánticas y los símbolos determinan el orden de evaluación.

Ejemplo 5.13.- El ETDS siguiente transforma una expresión escrita en notación infija a notación postfija (sin paréntesis).

$E \rightarrow T\ R$	$L \rightarrow \bullet\ F$ { print("•") } L
$R \rightarrow +\ T$ { print("+") } R	$L \rightarrow \varepsilon$
$R \rightarrow \varepsilon$	$F \rightarrow num$ {print(num.val)}
$T \rightarrow F\ L$	$F \rightarrow (\ E\ )$

Si se aplica el algoritmo anterior a la expresión 2 • (5 + 4) obtendremos la expresión 2 5 4 + •

5.4.3. Gramáticas L - atribuidas y E.T.D.S.

Si las acciones semánticas que evalúan los atributos heredados de una gramática L-atribuida se han colocado en el E.T.D.S. inmediatamente antes del símbolo correspondiente al atributo que se evalúa, éstos se evaluarán correctamente. Las acciones semánticas que evalúan los atributos sintetizados deberán colocarse después del último símbolo de la producción si se quiere tener la seguridad de que se evalúen correctamente. .

Por lo tanto, toda gramática L - atribuida tiene un E.T.D.S. asociado, cuya construcción está dada implícitamente en el párrafo anterior. En cambio, dado un E.T.D.S. no siempre existe un gramática L-atribuida que realice las mismas acciones semánticas y en el mismo orden, es suficiente considerar un ETDS con una acción semántica entre los símbolos de la parte derecha de una producción que no evalúe ningún atributo. En el Algoritmo 5.1 se muestra como realizar una evaluación descendente de los atributos de un E.T.D.S. asociado a un analizador descendente recursivo (A.D.R.).

ENTRADA: El código de un analizador descendente recursivo (ADR)
Un E.T.D.S. L-atribuido
SALIDA: El código de un ADR que realiza todas las acciones semánticas especificadas en el E.T.D.S.
MÉTODO:

- Para cada procedimiento A (asociado al no terminal A) inclúyanse tantos parámetros formales por valor como atributos heredados tenga el no terminal, y tantos parámetros formales por referencia como atributos sintetizados.
- Para cada atributo de un terminal, y antes de leer el siguiente terminal, almacénese el valor de éste en una variable local.
- Para cada llamada a un procedimiento B (con el no terminal B asociado):

1. Los parámetros actuales correspondientes a los atributos heredados corresponderán al resultado de la evaluación de las reglas semánticas que evalúen estos atributos heredados
2. Los parámetros actuales correspondientes a los atributos sintetizados serán variables locales.

- El código para la evaluación de cada regla semántica se copiará en el lugar que corresponda de acuerdo al E.T.D.S.

Algoritmo 5.1. Construcción de un traductor descendente recursivo

Ejemplo 5.14.- El código Pascal para la gramática del Ejemplo 5.12 podría ser:

```
procedure S;
var v, t:integer;
begin
  v := 0;
  B(v, t);
```

```
  print(t)
end;

procedure B(v: integer; var t:integer);
var t1:integer;
begin
  case sim of
  '0' : begin
          emparejar(sim, 0);
          v := v * 2;
          B(v , t1);
          t := t1;
        end;
  '1' : begin
          emparejar(sim, 1);
          v := v * 2 + 1;
          B(v , t1);
          t := t1;
        end;
  '$' : begin
          t := v;
        end;
  end;
end;
```

5.4.4 Evaluación de un ETDS en un analizador LR.

Como se vió en el apartado 5.3.2 en lo que se refiere a un analizador del tipo LR, las acciones semánticas de un ETDS que aparezcan al final de la regla se pueden evaluar en el momento en que se produzca la reducción mediante esa regla. El problema surge con las acciones a mitad regla y con la evaluación de los atributos heredados.

Considérese el siguiente ETDS:

S → A { accion_1}
A → **a** {acción_2} **b**
A → **a** {acción_3} **c**

Si se construye el autómata LR(0), se observa fácilmente que hasta que no se carga en la pila el terminal **b** o el **c**, no se sabe si la acción que hay que realizar es la {acción_2} o la {acción_3}, por otro lado estas acciones semánticas hay ejecutarlas antes desplazar a la pila el terminal lo que impide decidir cuál de las dos acciones hay que ejecutar. En consecuencia las acciones a mitad regla, y la evaluación de los atributos heredados, que necesariamente son evaluados en acciones a mitad regla, no pueden evaluarse, directamente, en un analizador de tipo LR. Veamos a continuación como podemos, en algunos casos, subsanar estas deficiencias.

Evaluación de acciones a mitad regla que no calculen atributos heredados.

En un análisis del tipo LR el analizador realiza dos tipos de acciones o desplaza un símbolo a la pila o reduce con una determinada producción. En este tipo de análisis solamente podemos asociar una acción a una producción, puesto que como hemos visto antes, un desplazamiento de

un símbolo puede producirse en muchos contextos. Ahora la reducción sólo se produce cuando toda la parte derecha de la producción está en la pila, por lo tanto la acción a mitad regla debería haberse realizado ya.

Una solución a este problema consiste en introducir nuevas reglas en la gramática, una por cada acción a mitad regla distinta. El procedimiento es el siguiente sustituimos cada acción semántica a mitad regla por un nuevo terminal, y añadimos una regla en la que este no terminal se rescriba a cadena vacía y añadimos la acción semántica al final de la nueva regla. Si aplicamos este método al ejemplo 5.13 obtendríamos lo siguiente:

$E \rightarrow T\ R$	$L \rightarrow \bullet\ F\ \mathbf{B}\ L$
$R \rightarrow +\ T\ \mathbf{A}\ R$	$L \rightarrow \varepsilon$
$R \rightarrow \varepsilon$	$F \rightarrow num$ {print(num.val)}
$T \rightarrow F\ L$	$F \rightarrow \mathbf{(}\ E\ \mathbf{)}$
$A \rightarrow \varepsilon$ { print("+") }	$\mathbf{B} \rightarrow \varepsilon$ { print("•") }

Como podemos observar esta nueva gramática es equivalente a la anterior y es una gramática S-atribuida, lo que implica que puede ser evaluada mediante el análisis ascendente de tipo LR. Ahora las acciones a mitad regla se evalúan en el momento de las reducciones asociadas a las nuevas reglas.

Este método se puede aplicar siempre, ahora es posible que la nueva gramática no sea LR(1) aunque si que lo fuera la gramática inicial. Sólo está garantizado que si la gramática inicial es LL(1) la nueva gramática es LR(1).

Esta técnica es la que utiliza yacc y bison de forma implícita para implementar las acciones a mitad regla. Es decir, en yacc y bison se pueden escribir acciones semántica a mitad regla y no hace falta hacer las transformaciones anteriores, será el compilador de yacc y bison el que realizará estas transformaciones de forma transparente para el usuario.

Nótese que esta técnica permite también asignar un valor/atributo al no terminal introducido. Por ejemplo, en el siguiente ETDS suponemos que se utiliza una variable global "n" y que ésta variable se puede modificar en algunas de las producciones del ETDS, supongamos, también, que deseamos utilizar en la acción **acc2**, el valor que "n" tenía en el momento de ejecutarse la acción semántica **acc1**.

A → B {acc1} C {acc2}
B → …
C → …

Una solución podría ser, introducir el atributo sintetizado M.n y la acción semántica M.n := n en la producción M → ε, con lo que el ETDS quedaría:

A → B M C {acc2(M.n)}
M → ε {acc1, M.n:= n }
B → …
C → …

Evaluación de atributos heredados simultáneamente a una análisis de tipo LR.

La solución anterior no resuelve el problema de la evaluación de los atributos heredados. En el siguiente ejemplo el ETDS comprueba que el número, binario, octal o hexadecimal, que se genera sea coherente con el identificador de la base. Para dicho fin se utiliza un atributo heredado C.base.

S → **b** { C.base := 2 } C
  | **o** { C.base := 8 } C
  | **h** { C.base := 16 } C
C → { C_1.base := C.base } C_1 D { if C.base <= D.val then MenError}
C → D { if C.base <= D.val then MenError}
D → **0** | **1** | **2** | ... | **f** { D.val := 0 / 1 / 2 / .../ 15}

En este caso las acciones semánticas como { C.base := 2 } presentan un doble problema, por un lado son acciones semánticas a mitad regla y por otro lado evalúan un atributo heredado.

El primer problema puede ser resuelto utilizando la técnica del apartado anterior, es decir:

S → **b** M C
M → ε { C.base := 2 }

Pero, la acción semántica en la nueva regla es incorrecta, en ella no se puede evaluar el atributo heredado C.base puesto que el símbolo C no pertenece a la regla M → ε. Podemos convertir el atributo heredado de C en un atributo sintetizado de M.

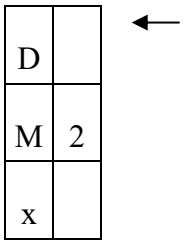
S → **b** M C
M → ε { M.base := 2 }

Ahora, el valor del atributo heredado necesario para evaluar la acción semántica de la regla:

C → D { if C.base <= D.val then MenError}

queda sin resolver.

Ahora, si observamos cuales serán las configuraciones de la pila de análisis en el caso LR, omitiendo los estados, cuando se va a producir la reducción C → D obtenemos:



donde x ∈ {b, o, h}

Si suponemos que la pila donde se almacenan los valores de los atributos en un analizador de tipo LR se ha implementado como un array **val**[ ] y que el valor del índice que indica el tope de la pila es top. El atributo heredado se encontrará siempre en **val**[**top** – 1] luego bajo estas hipótesis podíamos sustituir:

$C \rightarrow D$ { if C.val <= D.val then MenError}

por

$C \rightarrow D$ { if **val**[**top** – 1] <= D.val then MenError}

La otra producción donde se presenta un problema similar es:

$C \rightarrow \{ C_1.base := C.base \} C_1 D$ { if C.base <= D.val then MenError}

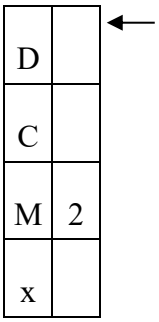
En este caso la situación es ligeramente distinta. Primero resolvamos el problema de la acción semántica:

{ if C.base <= D.val then MenError}

Si analizamos la situación de la pila en el momento de la reducción:

$C \rightarrow C D$

obtenemos que está será:



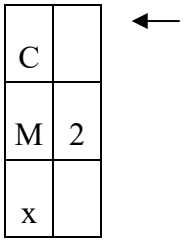
donde x ∈ {b, o, h}

razonando análogamente al caso anterior podemos escribir:

$C \rightarrow \{ C_1.base := C.base \}\ C_1\ D\ \{ \text{if } \mathbf{val}[top - 2] <= D.val \text{ then MenError}\}$

puesto que es el atributo de M el que contiene el valor de C.base.

Después de la reducción la pila quedaría:



donde x ∈ {b, o, h}

Con lo que el valor que debería heredar C, acción { C1.base := C.base }, sigue disponible en el atributo de M, con lo que esta acción semántica es superflua. Podemos escribir definitivamente:

$C \rightarrow C\ D\ \{ \text{if } \mathbf{val}[top - 2] <= D.val \text{ then MenError}\}$

quedando el ETDS como sigue:

$S \rightarrow \mathbf{b}\ M\ C$
$\quad | \ \mathbf{o}\ P\ C$
$\quad | \ \mathbf{h}\ Q\ C$
$M \rightarrow \varepsilon \qquad \{ M.base := 2 \}$
$P \rightarrow \varepsilon \qquad \{ P.base := 8 \}$
$Q \rightarrow \varepsilon \qquad \{ Q.base := 16 \}$
$C \rightarrow C\ D\ \{ \text{if } \mathbf{val}[top - 2] <= D.val \text{ then MenError}\}$
$C \rightarrow D \quad \{ \text{if } \mathbf{val}[top - 1] <= D.val \text{ then MenError}\}$
$D \rightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid ... \mid \mathbf{f} \quad \{ D.val := 0 / 1 / 2 / .../ 15\}$

En este ejemplo hemos visto cómo acceder a un atributo heredado y como propagarlo. Veamos otro ejemplo donde hay que recalcular el valor del atributo heredado antes de propagarlo. Analicemos el ETDS del ejemplo 5.12

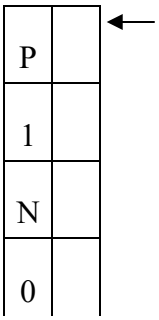
$S \rightarrow$		$\{ B.v := 0\}$	B	$\{ print(B.t) \}$
$B \rightarrow$	**0**	$\{B_1.v := B.v * 2 \}$	B_1	$\{ B.t := B_1.t \}$
$B \rightarrow$	**1**	$\{B_1.v := B.v * 2 + 1\}$	B_1	$\{ B.t := B_1.t \}$
$B \rightarrow$	ε	$\{ B.t := B.v \}$		

En el tenemos el atributo sintetizado t y el atributo heredado v. Si en un primer paso eliminamos las acciones semánticas a mitad regla, como lo hemos hecho antes, e introducimos atributos sintetizados para los nuevos no terminales obtenemos:

(1) $S \rightarrow$ M B { print(B.t) }
(2) $M \rightarrow \varepsilon$ { M.v := 0}
(3) $B \rightarrow 0$ N B_1 { B.t := B_1.t }
(4) $N \rightarrow \varepsilon$ { N.v := B.v * 2 }
(5) $B \rightarrow 1$ P B_1 { B.t := B_1.t }
(6) $P \rightarrow \varepsilon$ { P.v := B.v * 2 + 1}
(7) $B \rightarrow \varepsilon$ { B.t := B.v }

Ahora nos queda recuperar el valor del atributo heredado. Para resolver este problema procedemos igual que en el caso anterior, es decir, observamos las pilas del analizador LR en el momento en el que se produce la reducción con la regla 7, 4 ó 6.

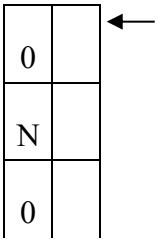
En el caso de la regla 7, en el momento de producirse la reducción por esta regla, la pila será similar a:



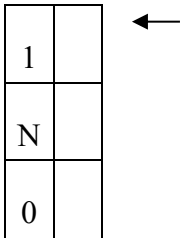
Suponiendo que el atributo v del no terminal P ya se ha calculado en el momento de la reducción por la regla 6, y que este contiene el valor del atributo heredado de B, tendremos:

(7) B → ε { B.t := **val**[top] }

En el caso de las reglas 4 ó 6 la pila será similar a:



ó



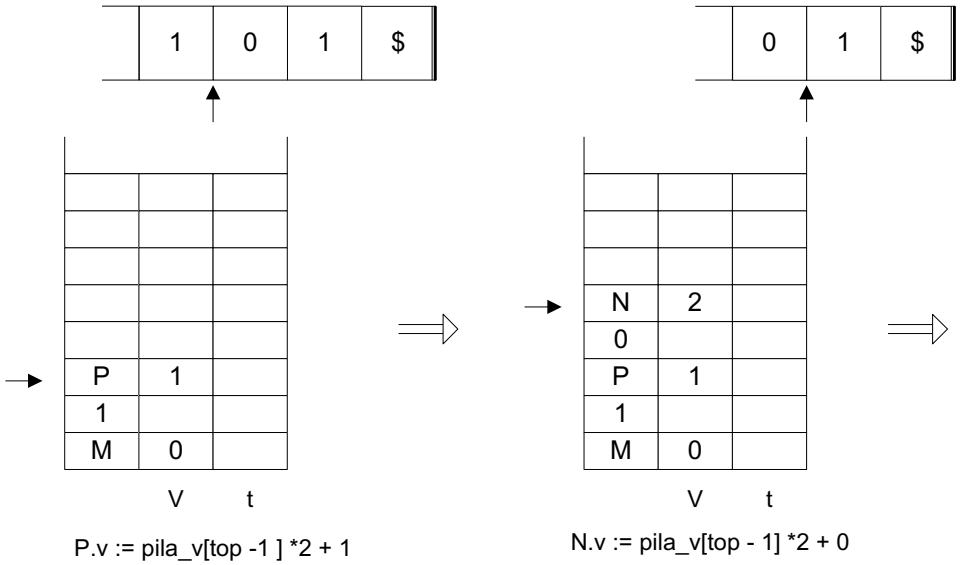
Análogamente al caso anterior, suponemos que N.v contiene el valor del atributo B.v lo que nos induce a escribir:

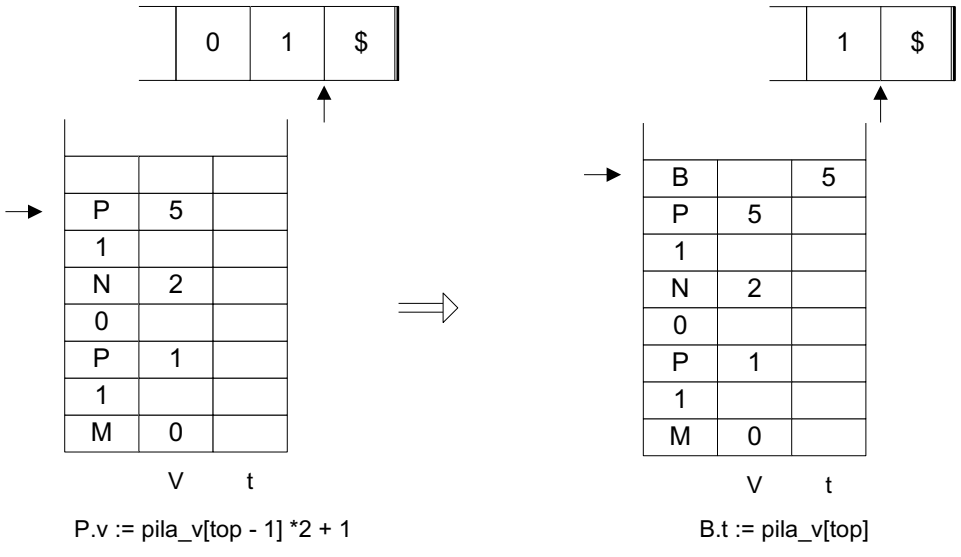
(4) N → ε { N.v := **val**[top-1] * 2 }
(6) P → ε { P.v := **val**[top-1] * 2 + 1}

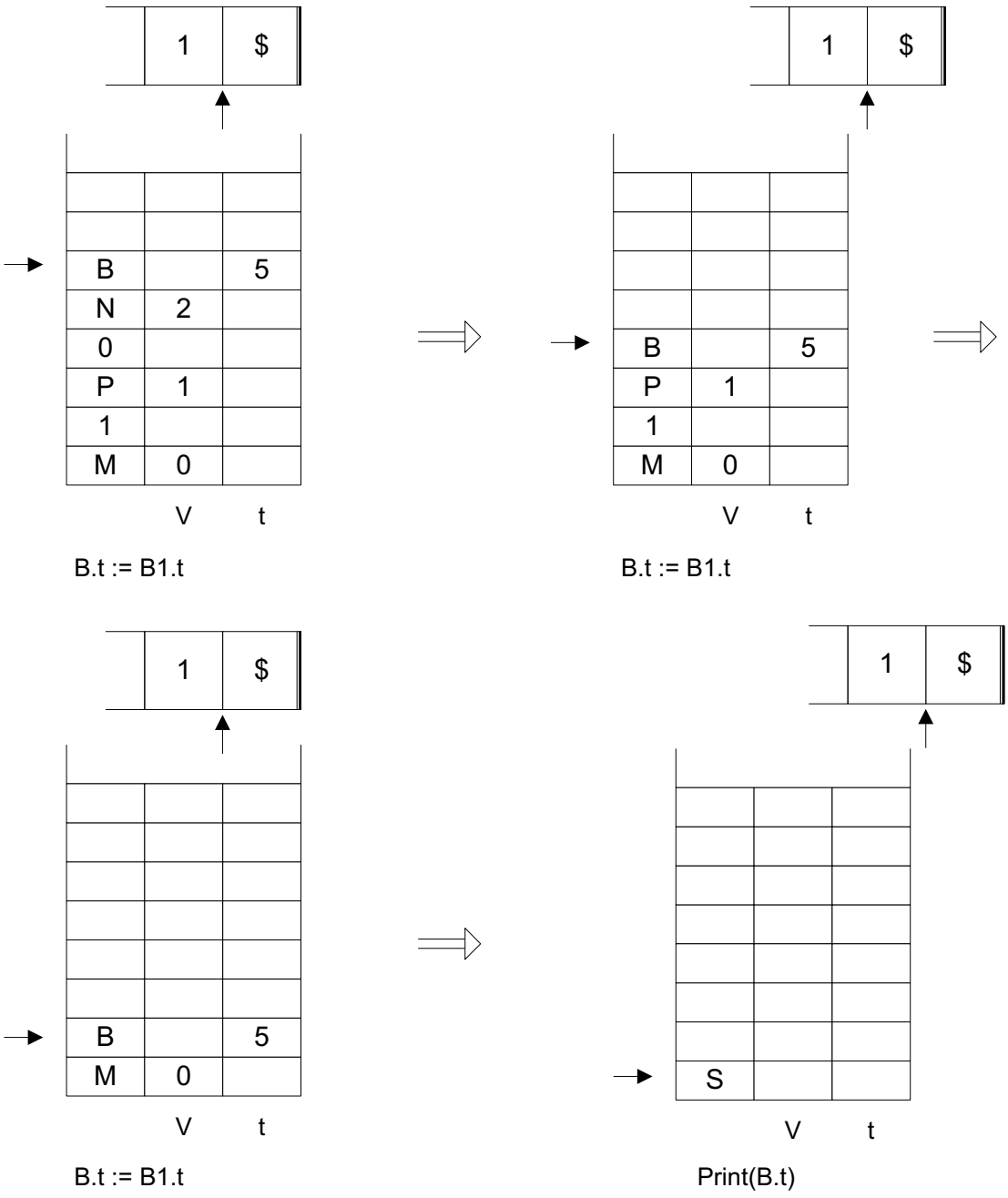
con lo que el nuevo ETDS, que podrá ser evaluado por el analizador LR, será:

(1) S → M B { print(B.t) }
(2) M → ε { M.v := 0}
(3) B → 0 N B_1 { B.t := B_1.t }
(4) N → ε { N.v := **val**[top-1] * 2 }
(5) B → 1 P B_1 { B.t := B_1.t }
(6) P → ε { P.v := **val**[top-1] * 2 + 1}
(7) B → ε { B.t := **val**[top] }

Observemos a continuación cuál será la traza para la cadena b101 realizada por un analizador LR con una pila para cada uno de los atributos.



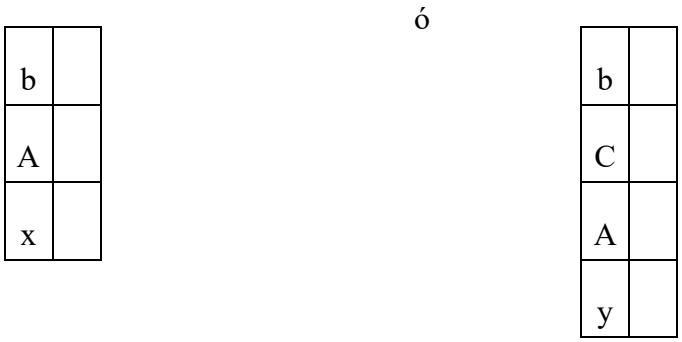




Esta técnica se puede aplicar siempre que podamos predecir cuál será la posición del atributo heredado en la pila. Pero esto no siempre es posible como lo demuestra el siguiente ejemplo:

(1) S → x A { B.h := A.s } B
(2) S → y A C { B.h := A.s } B
(3) A → **a** { A.s := 1 }
(4) B → **b** { f(B.h) }

En el momento que se va producir la reducción por la regla 4 y por lo tanto a ejecutarse la acción semántica f(B.h) la pila puede tener una de estas dos configuraciones:



En el primer caso el atributo A.s estará en val[top-1] y en el segundo caso en val[top-2]. Esto impide modificar adecuadamente la regla 4, puesto que no podríamos sustituir B.h por un acceso a la pila correcto para todos los casos.

5.5. Comprobación de Tipos: Introducción

En el **Análisis Semántico** se debe verificar que el programa fuente cumple con las restricciones semánticas especificadas para el lenguaje fuente. Estas comprobaciones semánticas pueden ser de dos tipos: ***estáticas***, realizadas en tiempo de compilación, y ***dinámicas***, incorporadas al código generado para que se ejecuten en tiempo de ejecución del programa objeto. Un ejemplo de restricciones semánticas puede verse en la Fig. 5.6.

Entre las comprobaciones semánticas de tipo estático se pueden destacar: las ***comprobaciones de tipo***, que se estudiarán a continuación con mayor detenimiento, las ***comprobaciones del control de flujo***, para verificar que las instrucciones que provocan rotura del control de flujo están bien definidas. Las ***comprobaciones de unicidad***, en la definición de los objetos, etc.

Entre las comprobaciones dinámicas, las más relevantes son:

- Las que dependen del estado de la máquina en el momento de la ejecución del programa, como los fallos en la apertura de ficheros, errores de dispositivos, etc.
- Las que dependen del propio programa, como los errores numéricos, divisiones por cero, desbordamiento numérico, etc.
- Las que controlan los errores de direccionamiento de memoria, como el desbordamiento de la pila o del montículo.
- Las que controlan el acceso a variables indexadas.

En este capítulo, se trata principalmente la comprobación de tipos. La mayoría de las otras comprobaciones estáticas son rutinarias y se pueden resolver aplicando técnicas similares. En cuanto a las comprobaciones dinámicas solo mencionar que será necesario generar código para realizarlas en tiempo de ejecución.

* Conversión implícita de números enteros en reales, tanto en expresiones como en asignaciones.

- El tipo índice de un array debe ser **integer**. Además, se debe realizar una comprobación dinámica de que el índice está dentro del rango definido, y una comprobación estática en los propios límites (límite inferior ≤ límite superior).
- Los nombres de las funciones pueden aparecer en la parte derecha de una asignación, pero no como valor izquierdo, excepto en el cuerpo de su propia función.
- Los nombres de los procedimientos son instrucciones, por tanto, no pueden aparecer ni en expresiones ni en asignaciones.
- Se permite la recursividad.
- Con respecto al alcance de las variables, se siguen las restricciones semánticas propias de un Lenguaje con Estructura de Bloques (LEB).
- Los argumentos de las instrucciones de entrada/salida, solo pueden ser: **integer** o **real**.
- En otros casos, las restricciones semánticas, por defecto, serán las propias del PASCAL estándar.

Figura 5.6. Ejemplo de restricciones semánticas.

Para realizar la comprobación de tipos de un cierto lenguaje de programación, es necesario conocer la definición de tipos asociada al mismo; es decir su sistema de tipos.

5.5.1. Sistemas de tipos

Un ***sistema de tipos*** son todas aquellas reglas que permiten asignar tipos a las distintas partes del programa y verificar su corrección. En concreto formarán el sistema de tipos: las definiciones y reglas que permiten comprobar cuál es el dominio asignado a una variable, y en que contextos puede ser usada.

El tipo de una construcción del lenguaje (variable, función, etc.) se define mediante una ***expresión de tipo***. La definición de expresión de tipo dependerá del sistema de tipos del lenguaje de programación para el que se defina. Por ejemplo, para un lenguaje de programación smilar a Pascal, podemos definir una expresión de tipo como:

- Un tipo básico (**tentero**, **treal**, **tlógico**, etc.) es una expresión de tipo. El error de tipos se indica con el tipo básico: **terror**. La ausencia de tipo se indica con el tipo básico: **tvacio**.
- El nombre de una expresión de tipo (básico o no) es una expresión de tipo.
- Un constructor de tipos aplicado a una expresión de tipo es una expresión de tipo:
    - Si **T** una expresión de tipo, **puntero(T)** es una expresión de tipo.

- Si T una expresión de tipo, **vector(I, T)** es una expresión de tipo, donde I es un rango de enteros (que denotamos por n_1 .. n_2).
- Si D (dominio) y R (rango) son expresiones de tipo, **D → R** es una expresión de tipo.
- Si P y Q son expresiones de tipo, **P x Q** es una expresión de tipo.
- Si N_1 … N_k son nombres de campos (de un registro) y T_1 … T_k son expresiones de tipo, **registro((N_1 x T_1) x … x (N_k x T_k))** es una expresión de tipo.

5.5.2. Construcción de un comprobador de tipos

La especificación de un comprobador de tipos se puede realizar usando el formalismo de las gramáticas atribuidas visto anteriormente. Como ejemplo, mostramos a continuación un comprobador de tipos para un sencillo lenguaje de alto nivel estructurado, similar a Pascal.

P	→	D ; I	
D	→	D ; D	
	→	**ε**	
	→	id : T	InsertarTds (id.nom,"variable",T.tipo);
	→	function id PF : T ; D ; I	InsertarTds (id.nom,"función", PF.tipo → T.tipo);

T	→	integer	T.tipo := tentero;
	→	real	T.tipo := treal;
	→	boolean	T.tipo := tlógico;
	→	^ T_1	T.tipo := puntero (T_1.tipo)
	→	record C end	T.tipo := registro (C.tipo)
	→	array [L]of T_1	T.tipo := vector (L.tipo,T_1.tipo);
PF	→	(id : T	InsertarTds (id.nom,"parámetro",T.tipo); PF.tipo:=T.tipo;
		{ id_1 : T }	{ InsertarTds(id_1.nom,"parámetro",T.tipo);
		)	PF.tipo x T.tipo; }
	→	**ε**	PF.tipo := tvacio;
C	→	id : T	C.tipo := id.nom x T.tipo;
		{ ; id : T }	{ C.tipo := ({ C.tipo) x (id.nom x T.tipo) }
L	→	cte_1 .. cte_2	Si (cte_1 t ≠ cte_2.t ≠ tinteger) ∨ (cte_1.v > cte_2.v) entonces MenError(.); L.tipo := (cte_1.v, cte_2.v);
		{, cte_1 .. cte_2 }	{ Si (cte_1 t ≠ cte_2.t ≠ tinteger) ∨ (cte_1.v > cte_2.v) entonces MenError() L.tipo := L.tipo ⊕ (cte_1.v, cte_2.v)}

E	→ cte	E.tipo := cte.t;
	→ id	Si not ObtenerTds(id.nom, E.tipo) entonces E.tipo := terror;
	→ E_1 **mod** E_2	Si E_1.tipo = E_2.tipo = tentero entonces E.tipo := tentero sino E.tipo:= terror;
	→ id [LE]	Si ObtenerTds(id.nom, vector(I,tipo)) ∧ (NumDim(I) = LE.ndim) ∧ (LE.tipo ≠ terror) entonces E.tipo := tipo sino E.tipo := terror;
	→ id PA	Si ObtenerTds(id.nom, "función", D → R) ∧ (D = PA.tipo) entonces E.tipo := R sino E.tipo := terror:
	→ id ^	Si ObtenerTds(id.nom, puntero(tipo) entonces E.tipo := tipo sino E.tipo := terror;
	→ id_1 . id_2	Si ObtenerTds(id_1.nom, registro (tipo)) ∧ (BuscarCampo(tipo, id_2.nom, tip-cam) entonces E.tipo := tip-cam sino E,tipo := terror;

LE	→ E	Si E.tipo = tentero entonces LE.tipo := E.tipo sino LE.tipo := terror; LE.ndim := 1;
	{ , E_1 }	{ Si E_1.tipo = tentero entonces LE.tipo:=E_1.tipo sino LE.tipo:=terror; LE.ndim := LE.ndim + 1}
PA	→ (E { , E_1 })	PA.tipo, E.tipo; { PA.tipo := PA.tipo x E_1.tipo); }
	→ **ε**	PA.tipo := tvacio;

I	→ id := E	Si not ObtenerTds(id.nom, id.tipo) ∨ id.tipo≠E.tipo entonces MenError(.);
	→ while E do I	Si E.tipo≠tlógico entonces MenError(.);
	→ I ; I	

Bibliografía recomendada

Para complementar este capítulo, se recomienda consultar la bibliografía tomada como base para su desarrollo:

[Aho 72] Aho, Alfred V.; Ullman, Jeffrey D. *The Theory of Parsing, Translation, and Compiling. Vol. 1: Parsing*. Prentice-Hall, 1972.

[Aho 90] Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D. *Compiladores: Principios, técnicas y herramientas*. Addison-Wesley Iberoamericana, 1990.

Ejercicios

*__5.1.-__ Dada la siguiente gramática,

S → **(** A **)**

A → A **,** D

A → D

D → **a**

D → **b**

D → **(** A **)**

- Dar una gramática S-atribuida que calcule el número de 'a's y de 'b's que tiene una frase cualquiera del lenguaje generado por esta gramática.

- A la vista del autómata construido en el apartado a, obtener la traza de análisis ascendente y evaluación de los atributos de la gramática S-atribuida anterior, para la frase: (a,(b))

*__5.2.-__ Construir una gramática S-atribuida que obtenga el número binario correspondiente al complemento a dos de otro número, basándose en la siguiente gramática independiente del contexto:

S → C

C → **0** C | **1** C | **0** | **1**

Las reglas semánticas deben definirse únicamente en función de los operadores aritméticos decimales básicos y de la concatenación "•".

*__5.3.-__ La siguiente gramática corresponde a la especificación sintáctica de algunas listas válidas de PROLOG:

Lista → **[ ]**

Lista → **[** T **]**

T→ T **,** T

T→ **id**

T→ Lista

a) Diseñar una gramática atribuida capaz de procesar una lista y averiguar su longitud y extensión. Se define longitud de una lista o término, como el número de elementos que la componen. Análogamente, se define extensión de una lista o término, como la longitud máxima de cualquiera de sus elementos. Como ejemplo, la lista *[[a1 , a2] , a3, [a4 , [a5 , a6] , [a7] , a8]]* tiene longitud 3 y extensión 4.

[*]**5.4.-** Dada la siguiente gramática, que genera ecuaciones de una incógnita (e.g.:3•x+5=7•x•x+3•x+1):

S → E = E

E → E + T | T

T → T • F | F

F → **cte** | **(** E **)** | **x**

a) Modificar la gramática para permitir potencias (xˆ(x+2)). La precedencia y asociatividad del operador potencia debe ser la habitual y quedar determinada por la propia gramática.

b) Sobre la gramática original diseñar un ETDS que permita reconocer como ecuaciones válidas únicamente las ecuaciones lineales.

c) Sobre la gramática original construir un ETDS que permita resolver las ecuaciones lineales en tiempo de compilación. Al finalizar la evaluación, el valor de *x* que satisface la ecuación estará contenido en un atributo.

[*]**5.5.-** Dada la siguiente gramática:

S → **a** S A

S → **ε**

A → B **b**

B → A **c**

B → **ε**

Construir un Esquema de Traducción Dirigida por la Sintaxis tal que dada la gramática de antes, devuelva la cantidad de: "**a**", "**b**" y "**c**" que se produce en el análisis de una cadena, respectivamente.

[*]**5.6.-** En química inorgánica se utiliza un lenguaje formal para expresar las fórmulas de los compuestos. Así, por ejemplo, la molécula del agua se escribe H2O, indicando que está compuesta por dos átomos de hidrógeno y uno de oxígeno. Algo mas complicadas son las fórmulas del bicarbonato sódico: NaHCO3, del sulfato férrico: Fe2(SO4)3, o del ferrocianuro férrrico Fe2(Fe(CN)3)3 en la que los paréntesis se utilizan para indicar que un grupo de átomos forma un ión, y los subíndices exteriores indican el número de iones que entran a formar parte de la molécula.

Cada átomo está caracterizado, además de por su símbolo químico, por una valencia[*], cuyo valor es un número entero: carbono: -4, oxigeno: -2, hierro: +3. Se dispone de una función que consulta una tabla y devuelve la valencia de un elemento: Valencia (elemento).

Dada la gramática:

Formula → F

F → F X N | X N

X → **id** | **(** F **)**

N → **num** | ε

Construir un ETDS que compruebe que una fórmula está bien formulada desde el punto de vista de las valencias con las que actúan los átomos (equilibrio iónico). Esto significa que la suma total de las cargas positivas y negativas de todos los átomos que integran la molécula debe ser cero. Así por ejemplo, la molécula del sulfato férrico Fe2(SO4)3, está en equilibrio iónico porque: Valencia (Fe) = 3 ; Valencia (S) = 6 ; Valencia (O) = -2

3*2 + (6+ (-2)*4)*3 = 6 + 18 – 24 = 0

* Se trata de una simplificación para facilitar la resolución del problema.

***5.7.-** Dado el siguiente fragmento de gramática:

S → **id :=** E

| S **;** S

| **for (** S **;** E **;** S **)** S

E → E ≺ E

| E ⊕ E

| **id**

construye un ETDS que realice la comprobación de tipos, teniendo en cuenta que los operadores ≺ y ⊕ son operadores sobrecargados.

≺ es un operador de orden que puede tener operandos enteros y booleanos.

⊕ es la suma de enteros, el 'o' lógico o la concatenación de cadenas de caracteres, según los operandos sean enteros, booleanos o cadenas respectivamente.

La tabla de símbolos se supone debidamente iniciada con los tipos de cada identificador.

Además, en la instrucción ' **for (** S_1 **;** E **;** S_2 **)** S_3 ' las ocurrencias tanto del terminal S_1 como del S_2, deben contener alguna instrucción de asignación. Al mismo tiempo, el no terminal E debe reescribirse de forma que contenga algún identificador que ocurra en la parte izquierda de alguna asignación que aparezca en S_1 o en S_2.

***5.8.-** Defínase una gramática de atributos que calcule expresiones en una calculadora con una memoria. Por ejemplo una expresión de esta calculadora podría ser:

2 + 5 = M_in 7 * 3 / M_out = (en este caso el resultado será 3)

***5.9**.- Construir una gramática L-atribuida para la gramática siguiente que realice las acciones semánticas detalladas posteriormente:

S → D ; I
D → Lista_tipos Lista_nombres | Lista_tipos (D) | D D
Lista_tipos → Tipo Lista_tipos | Tipo
Tipo → entero | real | complejo
Lista_nombres → id | id Lista_nombres
I → id := E
E → id
E → E + E | (E , E)
E → num_entero | num_real

a) Desarrollar la declaración de tipos, introduciendo en la tabla de símbolos cada identificador que aparezca en Lista_nombres tantas veces como tipos distintos tenga asociados en Lista_tipos. El significado de D → Lista_tipos (D) es que a todos los nombres mencionados en la declaración D dentro del paréntesis se les dan los tipos que aparecen en Lista_tipos, independientemente de cuantos niveles de anidamiento existan.

b) Realizar la comprobación de tipos.

5.10.- Dada la siguiente gramática que genera declaraciones para un único identificador:

S → declaración **id** LOpciones
LOpciones → LOpciones Opciones | ε
Opciones → Modo | Escala | Precision | Base
Modo → **real** | **complejo**
Escala → **fija** | **flotante**
Precisión → **simple** | **doble**
Base → **binaria** | **decimal**

Obtener un ETDS que asigne a cada identificador su lista de opciones y que no permita opciones repetidas.

5.11.- Constrúyanse los árboles anotados para las frases 3*5+6 y b1011 correspondientes, respectivamente, a las gramáticas atribuidas de los Ejemplos 5.3 y 5.4.

5.12.- Obténganse los grafos de dependencias correspondientes a los árboles anotados del problema 5.11.

5.13.- Determinar otros ordenes topológicos para este ejemplo. Calcular para cada uno de los grafos del problema 5.12 algún orden topológico.

5.14.- Determinar ordenes topológicos para los grafos de dependencias asociados a las cadenas: aabbb y ab. Evaluar los atributos correspondientes.

5.15.- Calcular la traza de la cadena 101.101 correspondiente a la gramática S - atribuida del Ejemplo 5.9.

Soluciones a ejercicios seleccionados

5.1.-

a)

S → (A)	{ S.num_a := A.a ; S.num_b := A.b }
A → A_1 , D	{ A.a := A_1.a + D.a ; A.b := A_1.b + D.b }
A → D	{ A.a := D.a ; A.b := D.b }
D → a	{ D.a := 1 ; D.b := 0 }
D → b	{ D.a := 0 ; D.b := 1 }
D → (A)	{ D.a := A.a ; D.b := A.b }

5.2.-

El complemento a dos de un número binario, se puede obtener sumando uno al número binario resultante de cambiar los ceros por unos y los unos por ceros en el número original.

Usaremos el atributo *.c* para ir calculando el complemento a dos del número. El operador de concatenación '•', nos servirá para ir concatenando una a una las cifras que forman el complemento a dos. El atributo sintetizado *.acar* tomará el valor TRUE cuando haya acarreo tras convertir un dígito, y FALSE en caso contrario.

El ETDS resultante quedaría:

S → C	{ S.solución := C.c }
C → **0** C_1	{ si C_1.acar ent C.c := '0' • C_1.c ; C.acar := TRUE sino C.c := '1' • C_1.c ; C.acar := FALSE }
C → **1** C_1	{ si C_1.acar ent C.c := '1' • C_1.c ; C.acar := FALSE sino C.c := '0' • C_1.c ; C.acar := FALSE }
C → **0**	{ C.c := 0 ; C.acar := TRUE }
C → **1**	{ C.c := 1 ; C.acar := FALSE }

5.3.-

El atributo *.l* asociado a los no-terminales *Lista* y *T*, representa la lóngitud. Se emplea el atributo *.e* para representar la extensión de las listas.

Lista → **[ ]**	{ Lista.l := 0 ; Lista.e := 0 }
Lista → **[** T **]**	{ Lista.l := T.l ; Lista.e := T.e }
T → T_1 **,** T_2	{ T.l := T_1.l + T_2.l ; T.e := máximo (T_1.e , T_2.e) }
T → **id**	{ T.l := 1 ; T.e := 1 }
T → Lista	{ T.l := 1 ; T.e := Lista.l }

5.4.-

a) Suponiendo que el nuevo operador '^' es asociativo por la derecha, y tiene mayor prioridad que '+' y '•', añadimos un nuevo no-terminal Q, y la gramática quedaría:

S → E = E

E → E + T

E → T

T → T • Q

T → Q

Q →F ^ Q

Q → F

F → cte

F → (E)

F → x

b) Asociaremos a los no-terminales de la gramática un atributo que llamaremos 'grado', que contendrá el grado de la expresión que representa el no-terminal. Finalmente, al atributo S.tipo le asignaremos la expresión de tipo 'vacio' si la ecuación es lineal, y TipoError en caso contrario. El ETDS quedaría[1]:

[1] En este ETDS no se está considerando la posibilidad de que algún coeficiente sea cero. Así, 0X se considera de grado 1. Para considerar esta posibilidad, sería necesario introducir un atributo que controle los casos en los que **cte** vale 0.

S → E1 = E2	{ S.tipo := si (E_1.grado <= 1) and (E_2.grado <=1) ent Vacio sino TipoError }
E → E_1 + T	{ E.grado := máximo (E_1.grado, T.grado) }
E → T	{ E. grado := T.grado }
T → T_1 • F	{ T.grado := T_1.grado + F.grado }
T → F	{ T.grado := F.grado }
F → **cte**	{ F.grado := 0 }
F → **(** E **)**	{ F.grado := E.grado }
F → **x**	{ F.grado := 1 }

c) Para resolver la ecuación lineal necesitaremos conocer el coeficiente de orden 1 (el de la X), y el coeficiente independiente o de orden 0. Estos valores se almacenarán en unos atributos que llamaremos 'x' e 'ind' respectivamente. Vamos a reconocer dos expresiones, una a cada lado de la igualdad (producción 1). Iremos efectuando todas las operaciones de simplificación posibles a cada lado de la ecuación. Al finalizar esto, en la producción 1, tendremos a cada lado de la igualdad, la forma general: $aX + b = cX + d$, donde la solución de la ecuación vendrá dada por:

$$X = \frac{d - b}{a - c}$$

Notar que la producción T → T_1 • F, representa un producto cuya forma general podría ser $(aX + b) \bullet (cX + d)$, y por lo tanto el resultado del producto será $acX^2 + adX + bcX + bd$. No obstante, alguno de los dos no-terminales (T_1 o F) deberá tener su coeficiente de grado 1 nulo, puesto que la ecuación debe ser lineal, de ahí que el término acX^2 no se considere.

Supondremos que el valor de la cte. nos lo proporciona el analizador léxico en el atributo cte.lexval. El ETDS quedará:

S → E_1 = E_2	{ S.solución:= (E_2.ind - E_1.ind) / (E_1.x - E_2.x) }
E → E_1 + T	{ E.x := E_1.x + T.x ; E.ind:= E_1.ind + T.ind }
E → T	{ E.x := T.x ; E.ind := T.ind }
T → T_1 • F	{ T.x := (T_1.x • F.ind) + (T_1.ind • F.x) ; T.ind:= T_1.ind • F.ind }
T → F	{ T.x := F.x ; T.ind := F.ind }
F → **cte**	{ F.x :=0 ; F.ind:= cte.lexval }
F → **(** E **)**	{ F.x := E.x ; F.ind := E.ind }
F → **x**	{ F.x:= 1 ; F.ind:= 0 }

5.5.-

S → **a** S_1 A {S.**a** := S_1.**a** +1; S.**b** := S_1.**b** + A.**b**; S.**c** := S_1.**c** + A.**c**}

$S \rightarrow \boldsymbol{\varepsilon}$	{S**.a** := 0;	S**.b** := 0;	S**.c** := 0}
$A \rightarrow B\ \mathbf{b}$	{A**.b** := B**.b** +1;	A**.c** := B**.c**}	
$B \rightarrow A\ \mathbf{c}$	{B**.b** := A.b;	B**.c** := A**.c** +1}	
$B \rightarrow \boldsymbol{\varepsilon}$	{B**.b** := 0;	B**.c** := 0}	

$S \Rightarrow^1 \mathbf{a}\ S\ A \Rightarrow^1 \mathbf{a\ a}\ S\ A\ A \Rightarrow^2 \mathbf{a\ a}\ A\ A \Rightarrow^3 \mathbf{a\ a}\ B\ \mathbf{b}\ A \Rightarrow^4 \mathbf{a\ a}\ A\ \mathbf{c\ b}\ A \Rightarrow^3 \mathbf{a\ a}\ B\ \mathbf{b\ c\ b}\ A$
$\Rightarrow^5 \mathbf{a\ a\ b\ c\ b}\ A \Rightarrow^3 \mathbf{a\ a\ b\ c\ b}\ B\ \mathbf{b} \Rightarrow^5 \mathbf{a\ a\ b\ c\ b\ b}$

5.6.-

Fórmula → F	{ if (F.v <> 0) then MemError( ) }
F → F I N	{ F.v := F$_{\|}$.v + I.v * N.Val }
| IN	{ F.v := I.v * N.val }
I → id	{ I.v := Valencia(id.nom) }
| (F)	{ I.v := F.v }
N → número	{ N.val := número.valor }
| ε	{ N.val := 1 }

5.7.-

S → **id :=** E	S.ident := {id.nom} Si BuscaTipo(id.ind) <> E.tipo ent MemError();
| S_1 **,** S_2	S.ident := S_1.ident ∪ S_2.ident ;
| **for (** S_1 **;** E **;** S_2 **)** S	Si S_1.ident=∅ or S_2.ident=∅ ent MemError() sino si E.ident ∩ (S_1.ident ∪ S_2.ident) = ∅ ent MemError() sino si E.tipo <> Tlogico ent MemError() ; S.ident := ∅ ;
E → E_1 ≺ E_2	Si E_1.tipo <> E_2.tipo or (not E_1.tipo in [Tentero, Tlogico]) ent MemError() ; E.tipo := Terror sino E.tipo := E_1.tipo E.ident := E_1.ident ∪ E_2.ident ;
| E_1 ⊕ E_2	Si E_1.tipo <> E_2.tipo or (not E_1.tipo in [Tentero, Tlogico, Tcadena]) ent MemError() ; E.tipo := Terror sino E.tipo := E_1.tipo ; E.ident := E_1.ident ∪ E_2.ident ;
| **id**	E.tipo := BuscaTipo (id.ind) ; E.ident := {id.nom}

5.8.-

El siguiente E.T.D.S. genera expresiones del tipo que se pide pero sólo con el operador '+' y evalúa dichas expresiones.

```
S →E =              {writeln(E.v) ; Pon_mem.v := E.v}
  Pon_mem S
S → ε
E → Fm + E1         {E.v := Fm.v + E1.v}
E → Fm              {E.v := Fm.v}
Fm → F Pon_mem      {Fm.v := F.v ; Pon_mem.v := F.v}
F → ( E )           {F.v := E.v}
F → m_out           {F.v := memoria}
F → num             {F.v := num.v}
Pon_mem → m_in      {memoria := Pon_mem.v}
Pon_mem → ε
```

Como ‘memoria’ se utiliza la variable, **memoria**, global al ETDS y cuyo valor inicial debe ser cero.