TEMA 5

ANÁLISIS SINTÁCTICO ASCENDENTE

Bibliografía:

- Aho, A.V., Sethi, R., Ullman, J.D. (1990), *Compiladores: principios, técnicas y herramientas*, Tema 4, páginas: 200-286.
- Louden, K.C. (1997), *Compiler Construction: Principles and Practice*, Tema 5, páginas: 197-256.
- Vivancos, E. (2000), *Compiladores I: una introducción a la fase de análisis*, Tema 3, páginas: 51-85. Especialmente para ejercicios.

Contenido:

1 Análisis ascendente: el autómata *desplaza/reduce*.

2 Tipos de conflictos: *desplaza/reduce* y *reduce/reduce*.

3 Gramáticas LR:

- 3.1 Modo de operación del análisis ascendente.
- 3.2 Mango de una forma secuencial derecha.
- 3.3 Elemento LR(0) y Autómata finito de elementos LR(0).

4 Análisis LR(0):

- 4.1 Algoritmo de análisis LR(0).
- 4.2 Limitaciones del análisis LR(0).

5 Análisis SLR(1):

5.1 Algoritmo de análisis SLR(1).

5.2 Resolución de conflictos por precedencia de operadores.

5.3 Limitaciones del análisis SLR(1).

6 Análisis LR(1) y LALR(1):

6.1 Elemento LR(1).

6.2 Autómata finito de elementos LR(1).

6.3 Algoritmo de análisis LR(1).

6.4 Agrupación de estados: análisis LALR(1).

7 Análisis sintáctico por precedencia de operadores.

8 Recuperación de errores en los analizadores ascendentes.

5.1 ANÁLISIS ASCENDENTE: EL AUTÓMATA *DESPLAZA/REDUCE*

Análisis ascendente: se construye el árbol de análisis sintáctico de la cadena de entrada desde las hojas hasta la raíz. En las hojas tenemos la cadena a analizar (los símbolos terminales) que se intentan reducir al axioma, que se encontrará en la raíz, si la cadena es correcta sintácticamente.

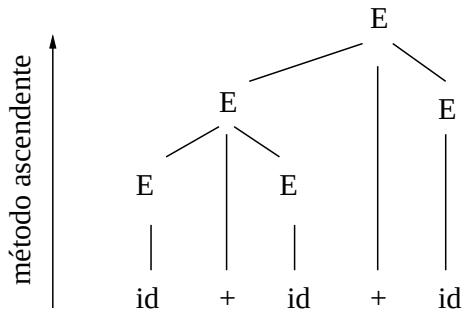
¿Como construir el árbol?

Se trata de **desplazarse** en la entrada hasta encontrar una subcadena de símbolos que represente la parte derecha de una producción, en ese momento sustituimos esa subcadena por el no-terminal de la parte izquierda correspondiente de la producción, la **reducimos**.

Ejemplo: Supongamos la siguiente gramática que permite generar expresiones aritméticas donde aparece el operador suma y potencia y combinar números e identificadores.

$$E \rightarrow E + E$$
$$E \rightarrow E \hat{} E$$
$$E \rightarrow \textbf{id}$$
$$E \rightarrow \textbf{num}$$

Para la entrada `id + id + id`



Nos restringimos al caso de métodos deterministas (no hay vuelta atrás) analizando un sólo símbolo de preanálisis sabemos exactamente en todo momento que acción realizar: bien desplazarnos en la entrada o bien aplicar una reducción. En el caso de una reducción debemos saber de forma única que producción aplicar.

Los métodos ascendentes se caracterizan porque analizan la cadena de componentes léxicos de izquierda a derecha, obtienen la derivación más a la derecha y el árbol de derivación se construye desde la raíz hasta las hojas.

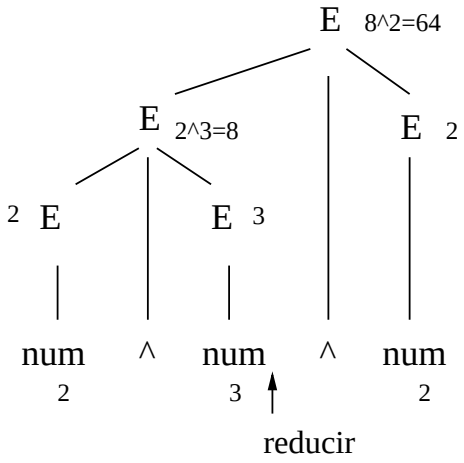
5.2 TIPOS DE CONFLICTOS

- ¿Qué acción realizar: desplazar o reduccir?
- ¿Qué producción elegir al reducir cuando hay varias posibles?

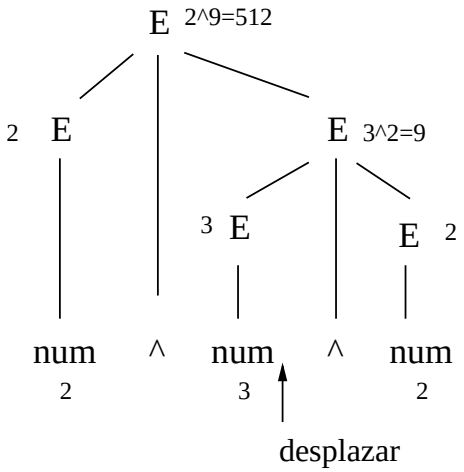
Conflicto desplaza-reduce:

$$E \rightarrow E + E \mid E \hat{} E \mid \textbf{id} \mid \textbf{num}$$

Para la entrada `2 ^ 3 ^ 2`



(a) Árbol incorrecto



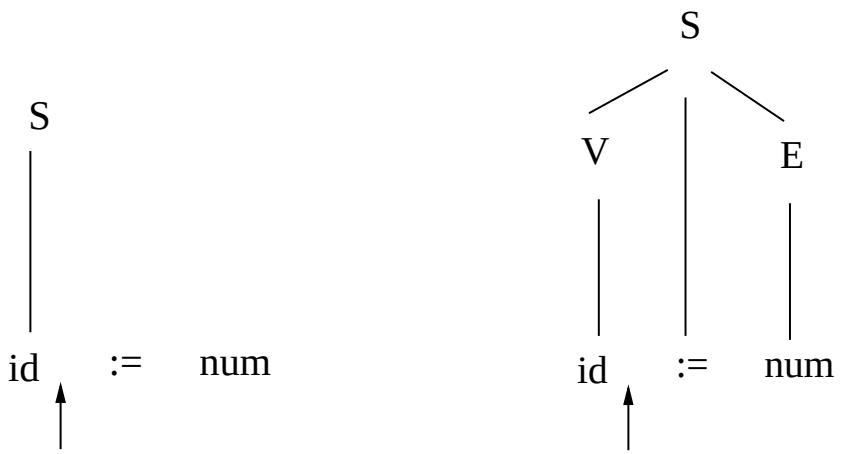
(b) Árbol correcto

Conflicto reduce-reduce:

Supongamos una sencilla gramática que permite generar sentencias que son bien llamadas a procedimientos o asignaciones.

$$S \rightarrow \textbf{id} \mid V{:=}E$$
$$V \rightarrow \textbf{id}$$
$$E \rightarrow \textbf{id} \mid \textbf{num}$$

Para la entrada `suma := 3`, equivalente a los componentes léxicos: `id := num`, tenemos las posibilidades



(a) Árbol incorrecto (b) Árbol correcto

5.3 GRAMÁTICAS LR

Los métodos de análisis sintáctico LR permiten reconocer la mayoría de las construcciones de los lenguajes de programación. Son métodos muy potentes, con mayor poder de reconocimiento que los métodos LL (Las gramáticas LL son un subconjunto de las gramáticas LR).

Las gramáticas LR(k) se caracterizan porque:

- L: Procesamos la cadena de entrada de izquierda a derecha (*from left-to-right*)
- R: proporcionan la derivación más a la derecha de la cadena de entrada en orden inverso (*rigth-most derivation*)
- k: se examinan k-símbolos de la entrada por anticipado para tomar la decisión sobre la acción a realizar.

Ventajas: es un método potente que permite reconocer la mayoría de las construcciones de los lenguajes de programación con k=0,1. Es un método sin retroceso y por tanto determinista.

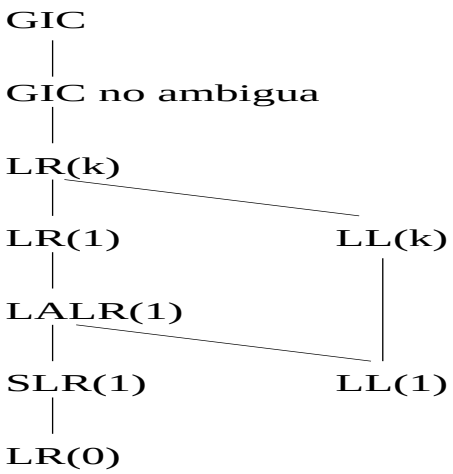
Desventajas: los métodos son difíciles de comprender, de implementar a

mano y de realizar una traza. Necesidad de usar generados automáticos de analizadores sintácticos LR. Por ejemplo: Bison (para Linux) y Yacc (`yet another compiler compiler`) para Unix.

Dentro del análisis sintáctico LR se distinguen cuatro técnicas:

- **El método LR(0)**. Es el más fácil de implementar, pero el que tiene menos poder de reconocimiento. No usa la información del símbolo de preanálisis para decidir la acción a realizar.
- **El método SLR(1)** (del inglés *Simple LR)*. Usa ya un símbolo de preanálisis.
- **El método LR(1)**. Es el más poderoso y costoso. El tamaño del automáta a pila para el reconocimiento se incrementa considerablemente.
- **El método LALR(1)** (del inglés *Look-Ahead LR*, con símbolo de anticipación). Es una versión simplificada del LR(1), que combina el coste en tamaño (eficiencia) de los métodos SLR con la potencia del LR(1).

Jerarquía de las gramáticas:



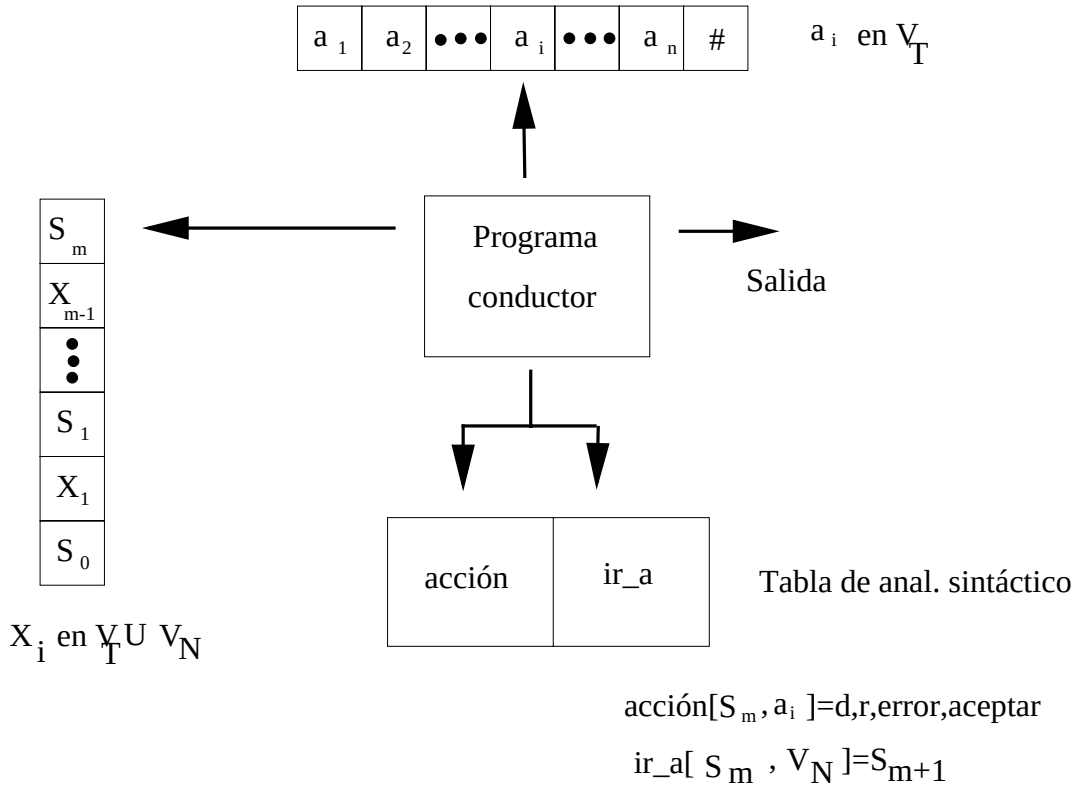
La diferencia que hace a los métodos LR más poderosos que los LL es que los primeros disponen de la información de la entrada y los estados

por lo que ha ido pasando el analizador (la cadena que ha ido reconociendo hasta ahora), mientras que los métodos LL sólo disponen de la información de la entrada.

- para que una gramática sea LR(k) tenemos que ser capaces de reconocer la presencia del lado derecho de una producción habiendo visto todo lo que deriva del lado derecho y usando k-símbolos por anticipado.
- para que una gramática sea LL(k) tenemos que ser capaces de reconocer el uso de una producción viendo sólo los k-símbolos que derivan su lado derecho.

Modo de operación del análisis ascendente:

De forma esquemática un analizador sintáctico consta de:

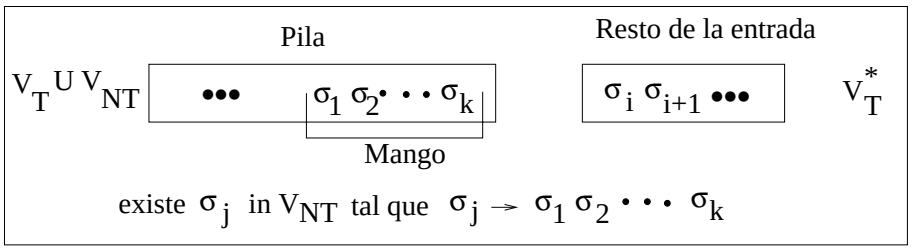


- La **entrada** formada por la serie de componentes léxicos a reconocer.

- Un **programa conductor** que lee *tokens* de la cadena de entrada de uno en uno y utiliza una pila para almacenar los símbolos que va reconociendo y los estados por los que pasa el analizador.
- Una **pila**. El contenido de la pila es de la forma $s_0X_1s_1X_2s_2 \ldots s_mX_m$, donde cada X_i son símbolos de la gramática que se van reconociendo, $X_i \in (V_T \cup V_{NT})$, y los s_i son los estados por los que pasa el analizador. Los símbolos terminales se introducen en la pila mediante desplazamientos de la cadena de entrada a la pila, los no-terminales se apilan como resultado de hacer una reducción.
- La **tabla de análisis sintáctico** formada por dos partes. La primera parte es de la forma $accion(s_m, a_i)$ indica que acción se debe de realizar si observamos en la entrada el *token* a_i y el analizador está en el estado s_m. Las acciones posibles son: desplazar, reducir, error, aceptar. La segunda parte es de la forma $ir\_a(s_m, V_{NT})$ indica el estado al que tenemos que ir después de hacer una reducción.
- **Salida**: la derivación más a la derecha de la cadena de entrada en orden inverso (de abajo hacia arriba).

Definición de mango (handle):

Mango o asidero: conjunto de símbolos terminales y no-terminales en la cima de la pila que forman la parte derecha de una producción y que se pueden reducir sacandolos de la pila y sustituyendolos por el no-terminal de la parte izquierda de la producción.



Resumiendo lo visto hasta ahora necesitamos:

- Un mecanismo que nos determine el tipo de acción a realizar: desplazar o reducir.
- En el caso de que tengamos que reducir nos debe proporcionar la subcadena de símbolos a reducir (el mango) y qué producción a utilizar.

Este mecanismo nos lo proporciona el *Autómata Finito Determinista de elementos LR(0)*.

5.3.1 Definición de Elemento LR(0) y Construcción del Autómata Finito de elementos LR(0)

Un **elemento de análisis sintáctico LR(0)** de una gramática G, es una producción de G con un punto en alguna posición del lado derecho.

$$[A \rightarrow \beta_1 \bullet \beta_2]$$

siendo $A \rightarrow \beta_1\beta_2$ una producción. Este punto separa la parte que se ha analizado hasta ese momento, y por tanto está en la cima de la pila, de la parte que se espera aún analizar, que es el resto de la cadena de entrada. Por ejemplo para la producción $A \rightarrow XYZ$ tenemos los posibles siguientes items LR(0):

$A \rightarrow \bullet XYZ$	se espera en la entrada una cadena derivable de XYZ
$A \rightarrow X \bullet YZ$	se ha reconocido una cadena derivable de X y se espera en la entrada una cadena derivable de YZ
$A \rightarrow XY \bullet Z$	-
$A \rightarrow XYZ\bullet$	**Item completo**. Indica que ya se ha reconocido por completo una cadena derivable de A. Hemos encontrado el mango.

Autómata finito de elementos LR(0)

Los *estados* son los propios items LR(0).
Las *transiciones* son de la forma:
Si $X \in V_T$ tenemos una transición

$$[A \to \alpha \bullet X\beta] \xrightarrow{X} [A \to \alpha X \bullet \beta]$$

Si $X \in V_{NT}$ tenemos una transición

$$[A \to \alpha \bullet X\beta] \xrightarrow{X} [A \to \alpha X \bullet \beta]$$

y además

$$[A \to \alpha \bullet X\beta] \xrightarrow{\epsilon} [X \to \bullet\gamma] \quad \forall X \to \gamma$$

En el primer caso, cuando $X \in V_T$ significa que hemos reconocido X en la entrada y estamos haciendo un desplazamiento del terminal X desde la entrada a la cima de la pila. En el segundo la interpretación es más complicada. Significa que hemos reconocido un no-terminal como consecuencia de haber hecho una reducción, por tanto tenemos que haber reconocido ese no terminal previamente, por tanto tenemos que añadir transiciones a estados que indiquen que vamos a reconocer ese no-terminal. Además, como no consumimos ningún *token* de la entrada son ϵ-transiciones.

5.4 ANÁLISIS LR(0)

El análisis LR(0) es el más sencillo de los métodos ascendentes. No usa información de la entrada, el símbolo de preanálisis, para decidir la acción a realizar.

Algoritmo de análisis LR(0)

Sea S el estado de la cima de la pila. Entonces:

- Si el estado S contiene cualquier item de la forma $A \to \alpha \bullet X\beta$ con X terminal $X \in V_T$, entonces la acción es **desplazar** X a la pila. El nuevo estado al que pasa el analizador, y que se coloca en la cima de la pila, es aquel que contenga un item de la forma $A \to \alpha X \bullet \beta$.

- Si el estado S contiene un item de la forma $A \to \alpha\bullet$, entonces la acción es **reducir** aplicando la producción $A \to \alpha$. Sacamos $2||\alpha||$ elementos de la pila y el estado que queda al descubierto debe contener un item de la forma $B \to \gamma \bullet A\sigma$, metemos el no-terminal A en la pila y como nuevo estado, aquel que contenga el item de la forma $B \to \gamma A \bullet \sigma$.

- Si no se puede realizar ninguna de las acciones anteriores entonces error.

Se dice que una **gramática es LR(0)** si la aplicación de las reglas anteriores no son ambiguas, es decir, que si un estado contiene un item de la forma $A \to \alpha\bullet$, entonces no puede contener cualquier otro item.
Si se produjera eso aparecerían conflictos. En efecto:

- Si tuviera además un item de la forma $A \to \alpha \bullet X\beta$ con X terminal tendríamos un conflicto *desplaza-reduce*.

- Si tuviera además un item de la forma $B \to \beta\bullet$, se produciría un conflicto *reduce-reduce*.

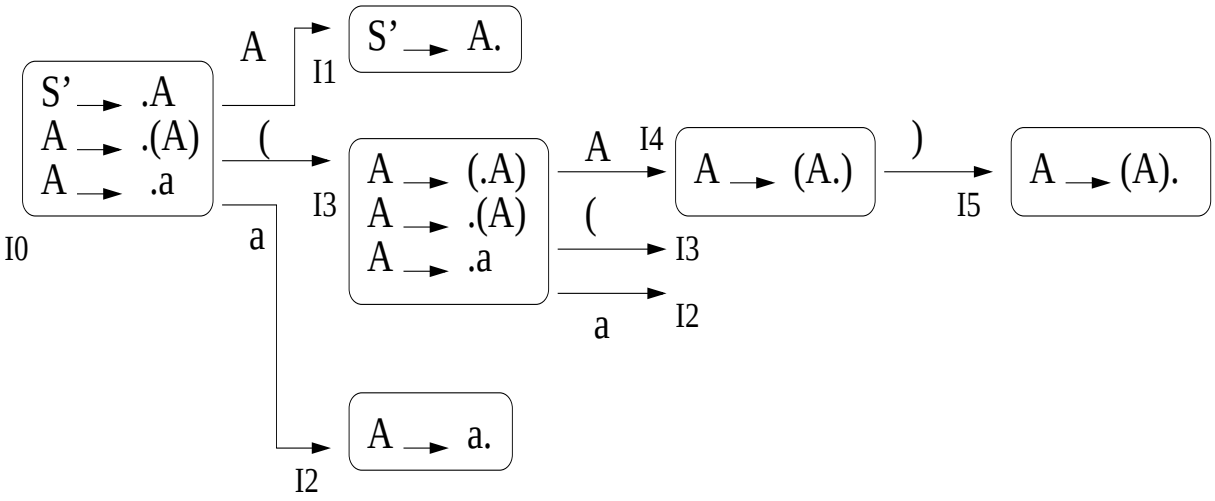
Por tanto, una gramática es LR(0) sii cada estado es bien un estado de desplazamiento (contiene sólo items que indican desplazar) o bien un estado de reducción (contiene un único item completo).

Ejemplo:

Consideremos la siguiente gramática para generar paréntesis anidados:

$$A \to (A)|a$$

El AFD de elementos LR(0) es:



A partir del autómata se construye la tabla de análisis sintáctico:

		Entrada			Ir_a
Estado	Acción	(	a	)	A
0	desplazar	3	2		1
1	reducir $S' \to A$ (acep.)				
2	reducir $A \to a$				
3	desplazar	3	2		4
4	desplazar			5	
5	reducir $A \to (A)$				

Hacer notar que no hemos necesitado conocer el tipo de *token* que teníamos en la entrada para decidir la acción a realizar.

Veamos el contenido de la pila, la entrada y la acción a realizar para la entrada `((a))#`

PILA	ENTRADA	ACCION
#0	((a))#	desplazar
#0(3	(a))#	desplazar
#0(3(3	a))#	desplazar
#0(3(3a2	))#	reducir $A \to a$
#0(3(3A4	))#	desplazar
#0(3(3A4)5	)#	reducir $A \to (A)$
#0(3A4	)#	desplazar
#0(3A4)5	#	reducir $A \to (A)$
#0A1	#	aceptar

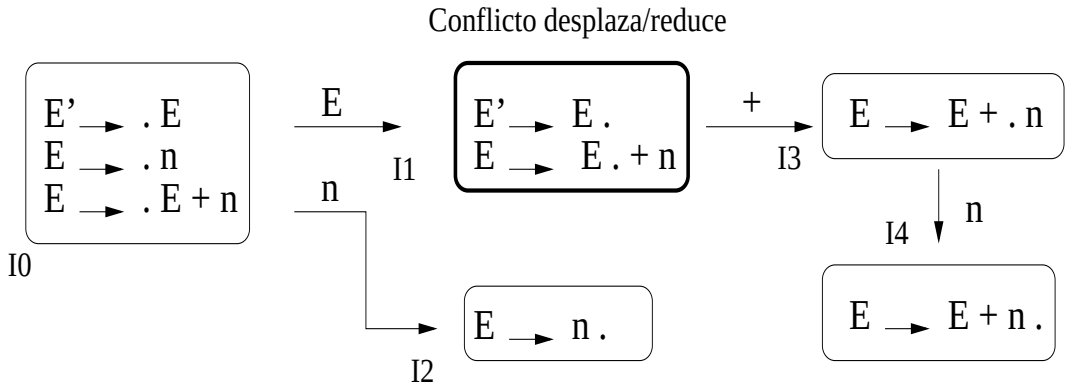
Obtenemos la derivación más a la derecha de la cadena de entrada:
$A \Rightarrow (A) \Rightarrow ((A)) \Rightarrow ((a))$.

Limitaciones del análisis LR(0):

Supongamos la gramática siguiente:

$$E \to E + n | n$$

El AFD de elementos LR(0) correspondiente es:



En el estado I1 tenemos un item que indica desplazar $E \to E \bullet +n$ y otro que indica reducir $E' \to E\bullet$. Tenemos un conflicto *desplaza/reduce*, según el algoritmo LR(0).

Existen determinadas construcciones de los lenguajes de programación

que no pueden ser reconocidas por este tipo de método. Para solucionar este problema se introduce el análisis SLR(1).

5.5 ANÁLISIS SLR(1)

Este tipo de análisis usa el AFD construido a partir de elementos LR(0) y usa el *token* de la cadena de entrada para determinar el tipo de acción a realizar. Este método consulta el *token* de la entrada antes de realizar un desplazamiento para tener seguro que existe una transición correspondiente en el estado en el que se encuentra el analizador (sino sería un error) y en el caso de que se tenga que hacer una reducción se comprueba que el *token* actual pertenece al conjunto de SIG del no-terminal al que se quiere reducir (sino sería un error).

Algoritmo de análisis sintáctico SLR(1)

Sea S el estado de la cima de la pila. Entonces:

- Si el estado S contiene cualquier item de la forma $A \rightarrow \alpha \bullet X\beta$ con X terminal ($X \in V_T$) y X es el siguiente terminal en la entrada, entonces la acción es **desplazar** X a la pila. Como nuevo estado al que pasa el analizador apilamos aquel que contenga un item de la forma $A \rightarrow \alpha X \bullet \beta$.
- Si el estado S contiene un item de la forma $A \rightarrow \alpha\bullet$ y el siguiente *token* en la entrada está en SIG(A), entonces la acción es **reducir** aplicando la producción $A \rightarrow \alpha$. Sacamos $2||\alpha||$ elementos de la pila y el estado que queda al descubierto debe contener un item de la forma $B \rightarrow \gamma \bullet A\sigma$, metemos el no-terminal A en la pila y como nuevo estado, aquel que contenga el item de la forma $B \rightarrow \gamma A \bullet \sigma$.
- Si el símbolo en la entrada es tal que no se pueden realizar ninguna de las acciones anteriores entonces error.

Se dice que una **gramática es SLR(1)** si la aplicación de las reglas anteriores no son ambiguas. Es decir, una gramática es SLR(1) sii se cumple para cada estado una de las siguientes condiciones:

- Para cada item de la forma $A \rightarrow \alpha \bullet X\beta$ en S con X terminal, no existe un item completo $B \rightarrow \gamma\bullet$ en S, con $X \in SIG(B)$. Sino tendríamos un conflicto *desplaza-reduce*.
- Para cada par de items completos $A \rightarrow \alpha\bullet$ y $B \rightarrow \beta\bullet$ en S, entonces $SIG(A) \bigcap SIG(B) = \emptyset$. De lo contrario se produciría un conflicto *reduce-reduce*.

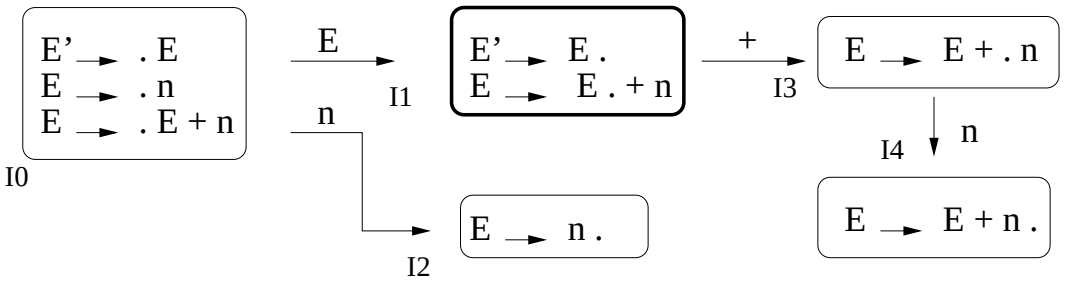
El aspecto de la tabla de análisis sintáctico cambia respecto al análisis LR(0), puesto que un estado puede admitir desplazamientos o reducciones dependiendo del *token* en la entrada. Por tanto, cada entrada debe tener una etiqueta de desplazamiento o reducción.

Ejemplo:

Para la gramática: $E \rightarrow E + n | n$
Reglas:

	Regla
0	E' → E
1	E → E + n
2	E → n

El AFD de elementos LR(0) correspondiente es:



Comprobemos que según el análisis SLR(1), ahora no existe conflicto desplaza/reduce en el estado I_1. En efecto: $\{+\} \notin SIG(E') = \{\#\}$.

La tabla de análisis sintáctico es:

# estado	Entrada			Ir_a
	n	+	#	E
0	d2			1
1		d3	aceptar	
2		r2	r2	
3	d4			
4		r1	r1	

Tabla 5.1: Tabla de análisis sintáctico

Consideremos la entrada n+n+n# y veamos el contenido de la pila, la entrada y la acción a realizar.

PILA	ENTRADA	ACCION
#0	n+n+n#	desplazar d2
#0n2	+n+n#	reducir r2 E $\rightarrow$ n
#0E1	+n+n#	desplazar d3
#0E1+3	n+n#	desplazar d4
#0E1+3n4	+n#	reducir r1 E $\rightarrow$ E + n
#0E1	+n#	desplazar d3
#0E1+3	n#	desplazar d4
#0E1+3n4	#	reducir r1 E $\rightarrow$ E + n
#0E1	#	reducir r0
#0E'	#	aceptar

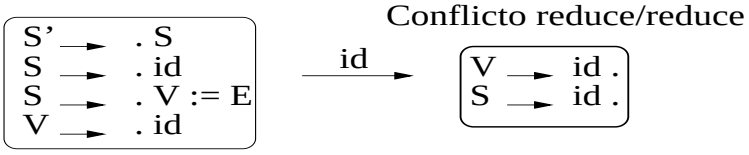
Limitaciones del análisis SLR(1):

El análisis SLR(1) es simple y eficiente. Es capaz de describir prácticamente todas las estructuras de los lenguajes de programación. Pero existen algunas situaciones donde este tipo de análisis no es suficientemente poderoso y falla. Veamos un ejemplo simplificado de la gramática en *Pascal* que

genera las llamadas a procedimientos y las asignaciones de expresiones a variables.

Supongamos la gramática siguiente:

$$S \rightarrow \textbf{id} \mid V{:=}E$$
$$V \rightarrow \textbf{id}$$
$$E \rightarrow V \mid \textbf{num}$$



Existe un conflicto *reduce/reduce*, según el algoritmo SLR(1), porque $SIG(V) = \{:=, \#\}$ y $SIG(S) = \{\#\}$, de manera que $SIG(V) \cap SIG(S) \neq \emptyset$. Imaginemos que estamos observado $\#$ por tanto no sabemos si es el final de la llamada a un procedimiento o es el final de una de una expresión (una expresión es una variable).

Los conflictos reduce/reduce no son habituales y suelen ser un síntoma de un mal diseño de la gramática.

5.6 ANÁLISIS LR(1)

Para solucionar este problema se introduce el método más general que existe, el llamado *análisis sintáctico LR(1)*. Este es el método más poderoso que existe, a costa de que se incrementa la complejidad (un factor de 10 respecto a los métodos que usan el AFD de items LR(0)).

Las limitaciones del método LR(0) y SLR(1) radican en que tienen en cuenta el símbolo de preanálisis después de la construcción del AFD de items LR(0), que por sí misma ignora los símbolos siguientes en la entrada.

5.6.1 Definición de Elemento LR(1) y Construcción del Autómata finito de elementos LR(1)

Un **elemento de análisis sintáctico LR(1)** de una gramática G, es un par consistente de un item LR(0) y un símbolo de preanálisis.

$$[A \to \alpha \bullet \beta, a]$$

siendo $A \to \alpha \bullet \beta$ un item LR(0) y a el *token* de preanálisis.

Un **item completo** $[A \to \alpha\beta\bullet, a]$, indica que ya se ha reconocido por completo una cadena derivable de A y reduciré siempre que en la entrada tenga el *token* a.

Autómata finito de elementos LR(1)

Los *estados* son los propios items LR(1).
Las *transiciones* son de la forma:
Si $X \in V_T$ tenemos una transición

$$[A \to \alpha \bullet X\gamma, a] \xrightarrow{X} [A \to \alpha X \bullet \gamma, a]$$

Si $X \in V_{NT}$ tenemos una transición

$$[A \to \alpha \bullet X\gamma, a] \xrightarrow{X} [A \to \alpha X \bullet \gamma, a]$$

y además ϵ-transiciones

$$[A \to \alpha \bullet X\gamma, a] \xrightarrow{\epsilon} [X \to \bullet\beta, b] \quad \forall X \to \beta \quad y \quad \forall b \in PRIM(\gamma a)$$

El estado inicial del AFD se obtiene ampliando la gramática, como para el caso de items LR(0), y poniendo como símbolo de preanálisis $\#$.

$$[S' \to \bullet S, \#]$$

que significa que reconoceremos una cadena derivable de S seguida del símbolo $\#$.

Algoritmo de análisis sintáctico LR(1)

Es básicamente igual al algoritmo de análisis SLR(1), excepto que usa el símbolo de preanálisis en vez del conjunto de SIG.

Sea S el estado de la cima de la pila. Entonces:

- Si el estado S contiene cualquier item de la forma $[A \rightarrow \alpha \bullet X\beta, a]$ con X terminal ($X \in V_T$) y X es el siguiente terminal en la entrada, entonces la acción es **desplazar** X a la pila. Como nuevo estado al que pasa el analizador apilamos aquel que contenga un item de la forma $[A \rightarrow \alpha X \bullet \beta, a]$.
- Si el estado S contiene un item de la forma $[A \rightarrow \alpha\bullet, a]$ y el siguiente *token* en la entrada es a, entonces la acción es **reducir** aplicando la producción $A \rightarrow \alpha$. Sacamos $2||\alpha||$ elementos de la pila y el estado que queda al descubierto debe contener un item de la forma $[B \rightarrow \gamma \bullet A\sigma, b]$, metemos el no-terminal A en la pila y como nuevo estado, aquel que contenga el item de la forma $[B \rightarrow \gamma A \bullet \sigma, b]$.
- Si el símbolo en la entrada es tal que no se pueden realizar ninguna de las acciones anteriores entonces error.

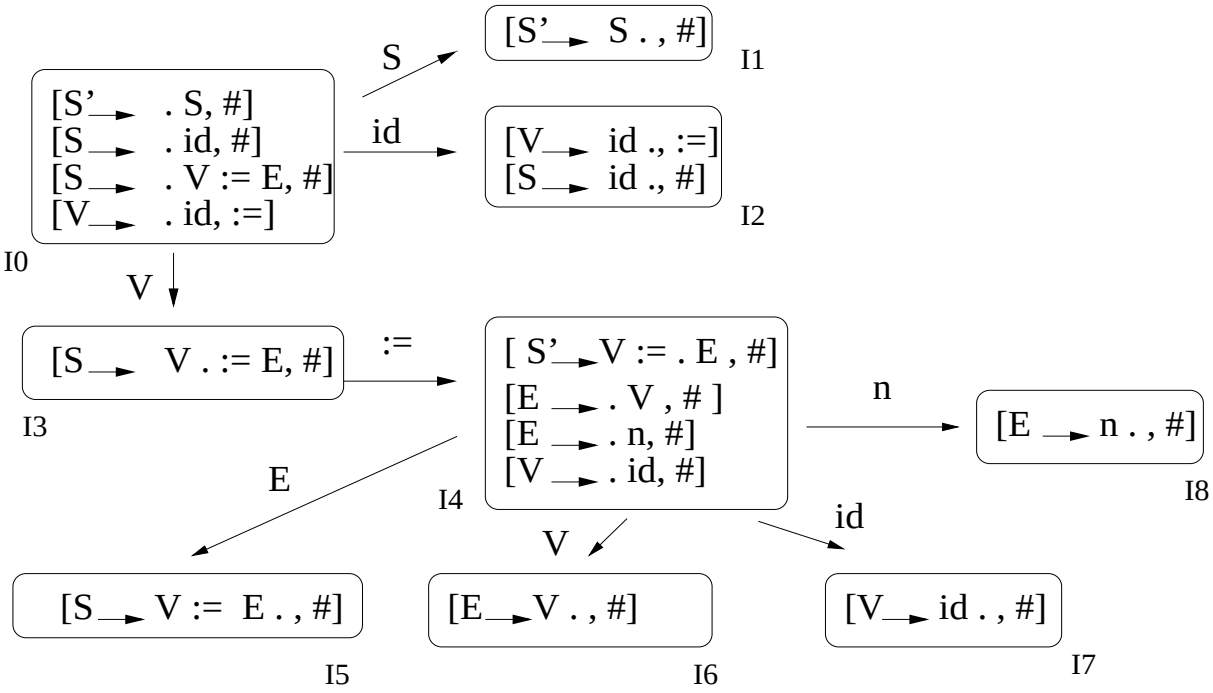
Se dice que una **gramática es LR(1)** si la aplicación de las reglas anteriores no son ambiguas. Es decir, una gramática es LR(1) sii se cumple para cada estado una de las siguientes condiciones:

- Para cada item de la forma $[A \rightarrow \alpha \bullet X\beta, a]$ en S con X terminal, entonces no existe un item completo $[B \rightarrow \gamma\bullet, X]$ en S. Sino tendríamos un conflicto *desplaza-reduce*.
- No existen dos items completos de la forma $[A \rightarrow \alpha\bullet, a]$ y $[B \rightarrow \beta\bullet, a]$ en S. De lo contrario se produciría un conflicto *reduce-reduce*.

La tabla de análisis sintáctico tiene el mismo aspecto que para el caso SLR(1).

Supongamos la gramática y calculemos el AFD de items LR(1)

$$S \rightarrow \textbf{id} \mid V{:=}E$$
$$V \rightarrow \textbf{id}$$
$$E \rightarrow V \mid \textbf{num}$$



Se ha evitado el conflicto que aparecía en el estado I_2.

5.6.2 Análisis LALR(1)

Una modificación del método LR(1), llamada *método LALR(1)* mantiene el poder del LR(k) y preserva la eficiencia del SLR(1). El análisis LALR(1) se basa en la observación de que en muchos casos el tamaño grande del AFD de items LR(1) se debe a la existencia de muchos estados diferentes que tienen igual la primera componente, los items LR(0), y difieren sólo de la segunda componente, los símbolos de preanálisis.

Lo que haremos es identificarlos como un único estado combinando los

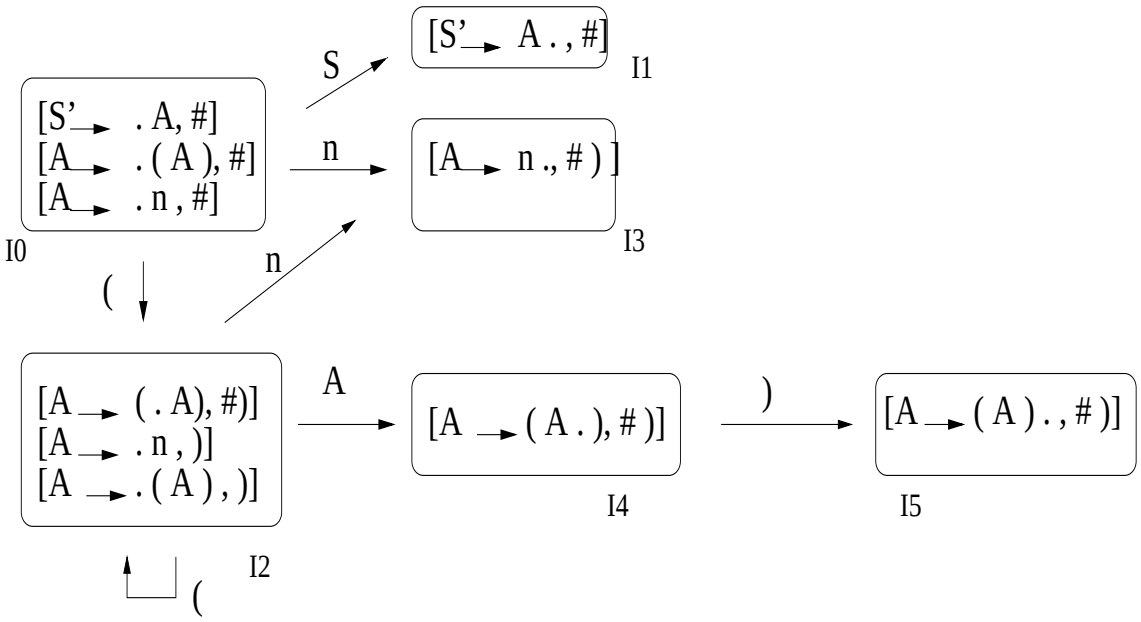
símbolos de preanálisis. Si el proceso está bien hecho, llegaremos a un autómata como el de items LR(0), excepto que cada estado tiene símbolos de preanálisis.

El algoritmo de análisis LALR(1) es idéntico al LR(1).

Por ejemplo para la gramática para generar paréntesis anidados:

$$A \rightarrow (A)|a$$

El AFD de elementos LR(1) resultante después de combinar estados es:



5.7 ANÁLISIS SINTÁCTICO POR PRECEDENCIA DE OPERADORES

Existe una clase especial de gramáticas, las **gramáticas de operadores**, para las cuales se puede construir con facilidad a mano eficientes analizadores sintácticos. Se dice que una gramática es de operadores si:

- No contiene no-terminales adyacentes.

- No contiene ϵ- producciones.

Ejemplo

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

El análisis sintáctico por precedencia de operadores es útil para el análisis semántico ya que el valor de la expresión es un atributo sintetizado (de hijos a padres) que se calcula en forma ascendente, pero existe el problema de la ambigüedad. Para resolverlo introducimos precedencias entre los terminales, que servirán para guiar la selección de la subcadena a reducir (el mango).

Existen tres tipos de relaciones de precedencia que representaremos por los símbolos:

- $a \lessdot b$, significa que a cede la precedencia a b.
- $a \doteq b$, significa que a tiene la misma precedencia que b.
- $a \gtrdot b$, significa que a tiene mayor precedencia que b.

La tabla de análisis sintáctico es una tabla con dos entradas. Las filas indican el operador que ya está en la pila y las columnas indican el operador en la entrada. Para construir la tabla de análisis sintáctico tendremos en cuenta que:

- Si θ_1, θ_2 son dos operadores tales que θ_1 tiene mayor precedencia que θ_2, hagáse $\theta_1 \gtrdot \theta_2$ y $\theta_2 \lessdot \theta_1$. Por ej: $* \gtrdot +, + \lessdot *$.
- Si θ_1, θ_2 son dos operadores de igual precedencia entonces hagáse: $\theta_1 \gtrdot \theta_2$ y $\theta_2 \gtrdot \theta_1$, si son asociativos por la izquierda. $\theta_1 \lessdot \theta_2$ y $\theta_2 \lessdot \theta_1$, si son asociativos por la derecha. Por ej: $+ \gtrdot +, + \lessdot +$.
- Respecto a los otros no terminales: $id \gtrdot \theta$ y $\theta \lessdot id \; \forall \theta$.
- Respecto al final de cadena, $\# \lessdot id$ y $id \gtrdot \#$, $\# \lessdot \theta$ y $\theta \gtrdot \#$. En general, $\# \lessdot a, a \gtrdot \#, \forall a \in V_T$.

Veamos como las relaciones de precedencia nos van a servir para guiar la selección de las cadenas a reducir. EL símbolo $\lessdot$ va a marcar el extremo izquierdo del mango, el símbolo $\gtrdot$ va a marcar el extremo derecho y $\doteq$, en caso de que aparezca, será siempre en el interior del mango. Entonces, la cadena a reducir se puede encontrar mediante el siguiente proceso:

1 Examínese la cadena desde el principio hasta encontrar el $\gtrdot$.

2 Después examínese la cadena hacia atrás, saltando los $\doteq$ hasta encontrar $\lessdot$.

3 El mango es todo lo que está a la izquierda del $\gtrdot$ y a la derecha del $\lessdot$.

Por ejemplo para la entrada `large od + id * id`

$$\# \lessdot id \gtrdot + \lessdot id \gtrdot * \lessdot id \gtrdot \#$$
$$\# \lessdot E + \lessdot E * E \gtrdot \#$$
$$\# \lessdot E + E \gtrdot \#$$
$$\#E\#$$

Algoritmo de análisis sintáctico por precedencia de operadores

Entrada: Una cadena de componentes léxicos w y una tabla de relaciones de precedencia.

Salida: Si w está bien formada, una estructura de árbol de análisis sintáctico sino una indicación de error.

Método: Tenemos dos variables: a (un puntero al terminal más cerca de la cima de la pila) y b (un puntero al primer símbolo de la entrada que queda por analizar).

Inicialmente, la pila contiene $\#$ y en el buffer de entrada la cadena $w\#$.
terminar=falso

repetir

si PILA==#S y b==# **entonces**

terminar=verdadero

sino
si $a \lessdot b$ o $a \doteq b$ **entonces**
desplazar b a la pila
sino
si $a \gtrdot b$ **entonces**
reducir, el mango está formado un extremo por
la cima de la pila y el otro se desapila hasta encontrar
un símbolo terminal s, tal que existe
un símbolo σ por debajo de s, tal que $\sigma \lessdot s$.
sino error
finsi
finsi
finsi
hasta terminar==verdadero

Ejemplo

$$E \to E + E \mid E * E \mid (E) \mid E \hat{} E \mid id$$

	+	*	(	)		id	#
+	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$
*	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$	$\lessdot$	error4()
)	$\gtrdot$	$\gtrdot$	error1()	$\gtrdot$	$\gtrdot$	error1()	$\gtrdot$
	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$
id	$\gtrdot$	$\gtrdot$	error1()	$\gtrdot$	$\gtrdot$	error1()	$\gtrdot$
#	$\lessdot$	$\lessdot$	$\lessdot$	error2()	$\lessdot$	$\lessdot$	error3()

Recuperación de errores: Recuperación a nivel de frase: al descubrir el error el analizador sintáctico realiza una corrección local de la entrada, para ello: inserta, elimina o sustituye un símbolo de la cadena de entrada a partir del punto donde se ha encontrado el error por otro que le permita continuar con el análisis (por ejemplo: `(, +, id`).

Procedimientos de error:

```
error1()
{
printf(‘‘Falta operador:  inserto + ’’);
}

error2()
{
printf(‘‘Paréntesis no equilibrado:  inserto ( ’’);
}

error3()
{
printf(‘‘Falta expresión’’);
}

error4()
{
printf(‘‘Paréntesis no equilibrado:  inserto ) ’’);
}
```

5.8 RECUPERACIÓN DE ERRORES SINTÁCTICOS

Los programas pueden contener errores de muy diverso tipo. En general se trata de: errores de ortografía por no tener el cuidado suficiente al programar, errores por desconocer o no comprender bien los requisitos formales del lenguaje o por la confusión con otro lenguaje que el programador utiliza habitualmente (por ejemplo al usar a la vez Pascal y Modula-2). Dependiendo de la fase del compilador en la cual se detectan, los errores se clasifican en:

- *léxicos*: pueden ocasionarse por usar un carácter inválido, que no pertenezca al vocabulario del lenguaje de programación, al escribir mal

PILA	ENTRADA	ACCION
#	id*(id+id)#	desplazar
# $\lessdot$ id	* (id+id)#	reducir
#E	*(id+id)#	desplazar
# $\lessdot$ E *	(id+id) #	desplazar
# $\lessdot$ E * $\lessdot$ (	id+id) #	desplazar
# $\lessdot$ E * $\lessdot$ ($\lessdot$ id	+ id)#	reducir
# $\lessdot$ E * $\lessdot$ ($\lessdot$ E	+ id)#	desplazar
# $\lessdot$ E * $\lessdot$ ($\lessdot$ E +	id)#	desplazar
# $\lessdot$ E * $\lessdot$ ($\lessdot$ E +$\lessdot$ id	)#	reducir
# $\lessdot$ E * $\lessdot$ ($\lessdot$ E + E	)#	reducir
# $\lessdot$ E * $\lessdot$ (E	)#	desplazar
# $\lessdot$ E * $\lessdot$ ($\doteq$ E)	#	reducir
# $\lessdot$ E * E	#	reducir
# E	#	aceptar

un identificador, palabra reservada u operador.

- *sintácticos*: se producen cuando el analizador sintáctico espera un símbolo terminal en cierta posición de la gramática que no corresponde con él que se acaba de leer. Por ejemplo, en una expresión aritmética con paréntesis no equilibrados.

- *semánticos*: corresponden a la semántica del lenguaje de programación la cual no está descrita en la gramática. Por ejemplo: un operador aplicado a un operando incompatible o uso de una variable no declarada.

- *de ejecución*: son errores de ejecución por ejemplo el uso de un índice en una matriz fuera del rango especificado, usar un puntero no inicializado o intentar una división por cero o la raíz cuadrada de un número negativo. Son detectados por el sistema de ejecución cuando el programa fuente se ha compilado con una determinada opción en la que se generan ciertas acciones automáticamente para tratar estos casos. Lo usual es que se informe del error y se detenga la ejecución.

- *lógicos*: especialmente en entornos de enseñanza, un compilador debe también informar de errores lógicos que pueden ser potenciales errores como variables declaradas pero no usadas, instrucciones que no son alcanzables, y operaciones aritméticas sin significado, como la suma de un valor cero a una variable.

La mayor parte de la detección de errores en un compilador se centra en la fase de análisis sintáctico. La razón es porque muchos errores son de naturaleza sintáctica. En esta sección se presentan algunas técnicas básicas para recuperarse de errores sintácticos y su implantación en los diferentes métodos de análisis sintáctico estudiados.

La mayoría de las especificaciones de los lenguajes de programación no describen cómo debe responder un compilador a los errores; la respuesta se deja al diseñador del compilador. Un buen gestor de errores:

- Debe informar de la presencia de errores con claridad y exactitud, de manera que permita al programador encontrar y corregir fácilmente los errores en su programa.
- Se debe recuperar de cada error con la suficiente rapidez y sin perder demasiada información de la entrada para detectar errores posteriores.
- No debe retrasar de manera significativa la compilación de programas correctos.

El gestor de errores debe imprimir la línea y la columna donde se ha detectado el error. También se debe incluir un mensaje de diagnóstico suficientemente informativo, por ejemplo “ se espera punto y coma en esa posición”. En la mayoría de los casos no es conveniente que el proceso de análisis sintáctico se detenga después de encontrar el primer error, ya que podrían existir más errores en la entrada. Además, puesto que la compilación de un programa puede tardar varios minutos, el programador espera que la compilación detecte todos los errores posibles en el mismo

momento. Se debe entonces implantar una *recuperación del error*, donde el analizador sintáctico intenta volver él mismo a un estado consistente en el que el procesamiento de la entrada pueda continuar.

Los métodos LL y LR detectan un error lo antes posible, se dice que tienen la propiedad del *prefijo viable*, lo cual quiere decir que detectan el error nada más ver un prefijo de la entrada que no es un prefijo de ninguna cadena del lenguaje. En los analizadores *LR* los errores sintácticos se detectan cuando a partir del estado que actualmente se encuentra en la cima de la pila no existe ninguna transición definida con el carácter actual de entrada. El contenido de la pila representa el contexto a la izquierda del error y el resto de la cadena de entrada el contexto a su derecha. La recuperación de errores se lleva a cabo modificando la pila y/o la entrada de forma que el analizador puede llegar a un estado adecuado para poder continuar con el análisis de la entrada. En los analizadores sintácticos LL los errores de sintaxis se detectan cuando un terminal en la cima de la pila no coincide con el símbolo actual en la entrada o cuando el símbolo de preanálisis y el no-terminal de la cima de la pila nos lleva a una entrada de la tabla de análisis sintáctico que está vacía.

Hay que hace notar que cualquier ’reparación’ efectuada por el compilador tiene el fin único de continuar la búsqueda de otros errores, no de corregir el código fuente.

5.8.1 Estrategias de recuperación de errores sintácticos

Aunque hay muchas estrategias que se pueden emplear para la recuperación de errores, ninguna de ellas ha demostrado ser universal (que sirva para todos los lenguajes de programación y métodos de análisis). El proceso de recuperación siempre debe basarse en hipótesis (estrategias heurísticas) acerca sobre el tipo de errores más comunes y esto hace que siempre dependa del lenguaje. Sin embargo, algunos métodos tienen amplia aplicabilidad.

1 *Recuperación de emergencia (o en modo de pánico)*: al detectar un error, el analizador desecha símbolos en la entrada hasta que encuentra uno perteneciente a un conjunto de componentes léxicos de sincronización. Estos componentes de sincronización son generalmente delimitadores de sentencia como el punto y coma, la palabra `end` de final de bloque o cualquier palabra clave que pueda ser inicio de una proposición nueva. Es el método más sencillo de implantar y puede ser usado por la mayoría de los métodos de análisis sintáctico, pero sólo reconoce un error por proposición, unidad sintáctica. Esto no es necesariamente una desventaja ya que no es probable que ocurran varios errores en la misma proposición. Es tarea del diseñador del compilador elegir un conjunto adecuado de componentes de sincronización.

2 *Recuperación a nivel de frase*: al descubrir el error el analizador sintáctico realiza una corrección local de la entrada, para ello: inserta, elimina o sustituye un símbolo de la cadena de entrada a partir del punto donde se ha encontrado el error por otro que le permita continuar con el análisis (por ejemplo: insertar un punto y coma faltante al final de una sentencia). Es el diseñador del compilador él que debe elegir que tipo de modificación se debe realizar en cada caso.

3 *Recuperación mediante producciones de error (o expansión de la gramática)*: si se tiene una buena idea de los errores más comunes que se pueden encontrar, se puede aumentar la gramática del lenguaje con producciones que generan las construcciones erróneas, *producciones de error*, los errores quedan así legalizados, que no quiere decir que no sean detectados, ya que se incluyen las acciones correspondientes para informar de la detección. Se usa entonces esta gramática ampliada para construir el analizador sintáctico.

4 *Recuperación mediante corrección global*: dada una cadena de entrada con errores, `x`, y una gramática se trata de encontrar otra cadena, `y`, correcta tal que el número de inserciones, supresiones y sustituciones para transformar `x` en `y` sea el mínimo posible. Estos métodos son

demasiado costosos y su uso queda restringido a entornos experimentales.

La recuperación en modo de pánico se utiliza normalmente en analizadores escritos 'a mano', como el analizador recursivo descendente predictivo, mientras que la recuperación mediante producciones de error se utilizan en analizadores generados automáticamente. Por ejemplo: Bison y Yacc. Los métodos de recuperación a nivel de frase son los menos usados y los de corrección global sólo se usan en proyectos de investigación.

Los tres primeros métodos son métodos 'ad hoc' que no se pueden extrapolar a gramáticas de otros lenguajes. La idea es que cuando el analizador encuentra un error en la cadena de entrada llama a una rutina de manejo específico de ese error. Los diseñadores son libres de decidir las acciones más oportunas para llevar a cabo en cada situación. Se puede conseguir una recuperación de errores bastante buena, siempre que los errores que se presentan encajen en alguna de las categorías de errores que los creadores del compilador han anticipado. Esto es difícil de anticipar en la práctica. Otra desventaja, es que debemos codificar rutinas adecuadas para las entradas de la tabla lo cual implica un riguroso conocimiento de la tabla de análisis sintáctico y se adapta con dificultad a cambios que se realicen en la gramática, ya que implicaría la modificación de la tabla.

5.8.2 Recuperación en modo de pánico en analizadores LR

En caso de que se produzca un error se desechan símbolos de la pila hasta encontrar un estado s con un valor de *ir_a* para un determinado noterminal A. Entonces se desechan cero o más símbolos de la entrada hasta encontrar un símbolo que pueda seguir legalmente a A. De esta forma el analizador intenta aislar la frase que tiene el error. En el momento de la detección del error parte de la cadena ya se ha metido en la pila y el resto queda en la entrada. Por tanto, el analizador echa marcha atrás saca

símbolos de la pila hasta donde empezó a recocer la cadena que tenia el error, es decir, encontrar un estado, A (estado al descubierto), que contenga un valor para la función *ir_a* y desecha símbolos en la entrada hasta encontrar un símbolo a que pueda seguir a A.

Veamos un ejemplo para la gramática para generar expresiones aritméticas:

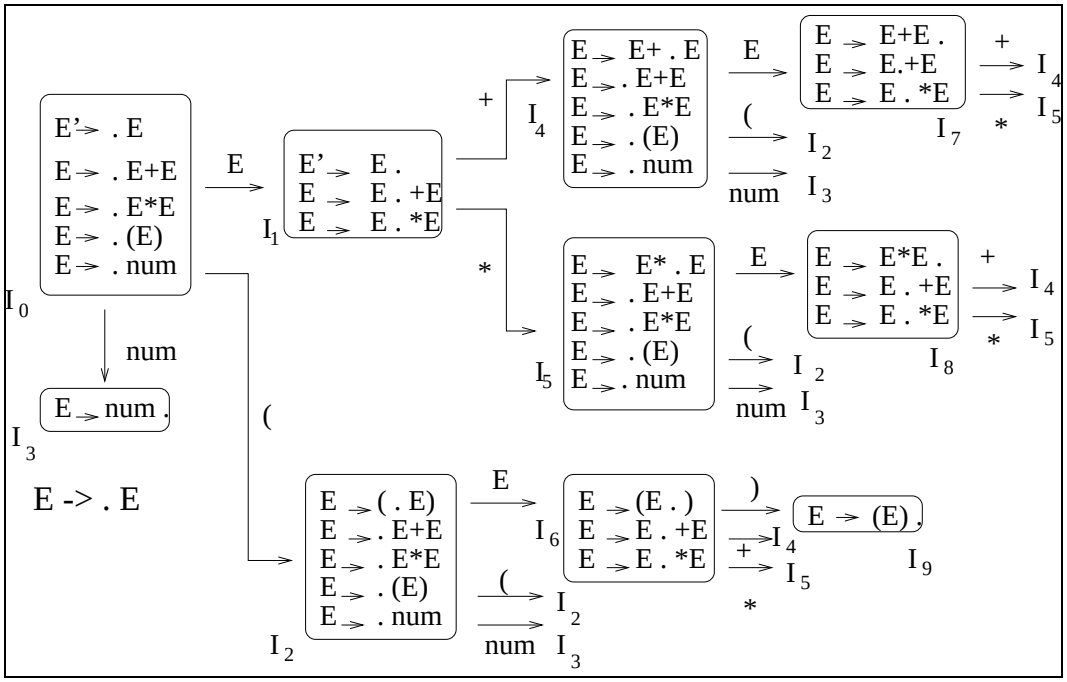


Figura 5.1: Autómata de elementos LR

#	Regla
	S' → E
1	E → E + E
2	E → E * E
3	E → (E)
4	E → num

Tabla de análisis sintáctico:

Veamos ahora cual sería el contenido de la pila y el estado de la entrada al reconocer la cadena `num * num num`.

He reconocido $num + num$.

# estado	num	+	*	(	)	#	E
0	d3			d2			1
1		d4	d5			aceptar	
2	d3			d2			6
3		r4	r4		r4	r4	
4	d3			d2			7
5	d3			d2			8
6		d4	d5		d9		
7		r1	d5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Tabla 5.2: Tabla de análisis sintáctico

Pila	Entrada	Acción	Mensaje
0	num * num num #	desplazar	
0num3	* num num #	reducir	
0E1	* num num #	desplazar	
0E1*5	num num #	desplazar	
0E1*5num3	num #	error, saco símbolos de pila, desplazar	espero +,*,),#
0E1*5E8	#	reducir	
0E1	#	aceptar	

Tabla 5.3: Recuperación en modo de pánico: Análisis sintáctico para la entrada `num * num num #`

5.8.3 Recuperación a nivel de frase en analizadores LR

La *recuperación a nivel de frase* se implanta examinando cada entrada de error en la tabla de análisis sintáctico y decidiendo, basándose en el uso del lenguaje, cual es el procedimiento de recuperación más apropiado. El proceso de recuperación puede implicar la modificación de los estados de la pila y/o la cadena de entrada.

Hay que evitar que no se produzcan lazos infinitos, esto se garantiza si siempre al menos un símbolo de la entrada es eliminado o desplazado a la pila o aplicando una reducción en la pila si se ha alcanzado el final de un

mango.

- e1: esta rutina se llama desde los estados 0,2,4,5 que esperan el comienzo de un operando, es decir, `id, (`. En vez de eso se ha encontrado un `+,*,#`. En este caso se introduce un `num` ficticio en la pila y se cubre con el estado 3. Emítase el mensaje `‘‘Falta operando’’`.
- e2: esta rutina se llama desde los estados 0,1,2,4,5 al encontrar un paréntesis derecho. En este caso se elimina el paréntesis derecho. Emítase el mensaje `‘‘Paréntesis ) no equilibrado’’`.
- e3: esta rutina se llama desde los estados 1,6 que esperan un operador y se encuentra con un `id, )`. En este caso se introduce un `+` ficticio en la pila y se cubre con el estado 4. Emítase el mensaje `‘‘Falta operador’’`.
- e4: esta rutina se llama desde el estado 6, se espera un operador o un paréntesis derecho y se encuentra con un `#`,. En este caso se introduce un `)` ficticio en la pila y se cubre con el estado 9. Emítase el mensaje `‘‘Falta paréntesis derecho )’’`.

La tabla de análisis sintáctico queda ahora de la forma:

# estado	num	+	*	(	)	#	E
0	d3	e1	e1	d2	e2	e1	1
1	e3	d4	d5	e3	e2	aceptar	
2	d3	e1	e1	d2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	d3	e1	e1	d2	e2	e1	7
5	d3	e1	e1	d2	e2	e1	8
6	e3	d4	d5	e3	d9	e4	
7	r1	r1	d5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

Tabla 5.4: Tabla de análisis sintáctico

IMPORTANTE: Se ha sustituido las entradas de error en estados que indican una reducción por la reducción correspondiente, esto lo único que hará es que se retrase la detección hasta haber hecho las reducciones, pero el error será siempre detectado antes de que tenga lugar cualquier movimiento de desplazamiento.

Veamos el ejemplo con la entrada `id + )#`

Pila	Entrada	Acción	Mensaje
0	num+)#	desplazar	
0num3	+)#	reducir regla	
0E1	+)#	desplazar	
0E1+4	)#	error, elimina)	‘‘Paréntesis) no equilibrado’’
0E1+4num3	#	error, mete operando, reducir	‘‘Falta operando’’
0E1+4E7	#	reducir	
0E1	#	aceptar	

Tabla 5.5: Análisis sintáctico para la entrada `num +)#`

5.8.4 Recuperación mediante producciones de error en analizadores LR

Este método se basa en la incorporación en la gramática del lenguaje lo que se llama *producciones de error*. Desde el punto de vista formal se trata de producciones normales en las que se ha incorporado un símbolo terminal especial, χ, que indica al algoritmo de análisis dónde se desea realizar una recuperación de error. Por ejemplo, dada la gramática para generar expresiones aritméticas podemos introducir las siguientes producciones de error:

$$
\begin{aligned}
&\texttt{E} \rightarrow \texttt{E + E} \\
&\mid \texttt{E}\ \chi\ \texttt{E} \\
&\mid \chi\ \texttt{+}\ \chi \\
&\mid \chi
\end{aligned}
$$

Estas producciones se tratan como las otras a la hora de construir el autómata de análisis LR. Mientras no se detecte ningún error en la entrada, no se activarán ya que al ser χ un símbolo ficticio no puede aparecer en la cadena de entrada de forma natural y nunca dará una transición en condiciones normales. En el momento en que se detecta un error el método hace lo siguiente:

1 Si el estado que se encuentra en la cima de la pila tiene una transición en χ entonces se introduce en la entrada delante del símbolo conflictivo y se continua con el análisis como si nada hubiera ocurrido.

2 Si por el contrario χ no es un símbolo válido en el estado de la cima, entonces se elimina el símbolo actual de la entrada, se sustituye por χ y se eliminan de la pila tantos estados y símbolos como sea necesario hasta encontrar uno en el que sí sea válido el carácter χ o hasta vaciar la pila, en este caso el método falla y no es capaz de recuperarse del error que se ha encontrado.

Veamos como se aplicaría el método al caso sencillo de una gramática que genera sumas de números naturales

```
E → E + E
 | num
 | χ
```

en la que se ha introducido una producción de error genérica. Construyamos el autómata de elementos LR y la tabla de análisis asociada.

#	Regla
1	S' → E
2	E → E + E
3	E → num
4	E → χ

Tabla de análisis sintáctico:

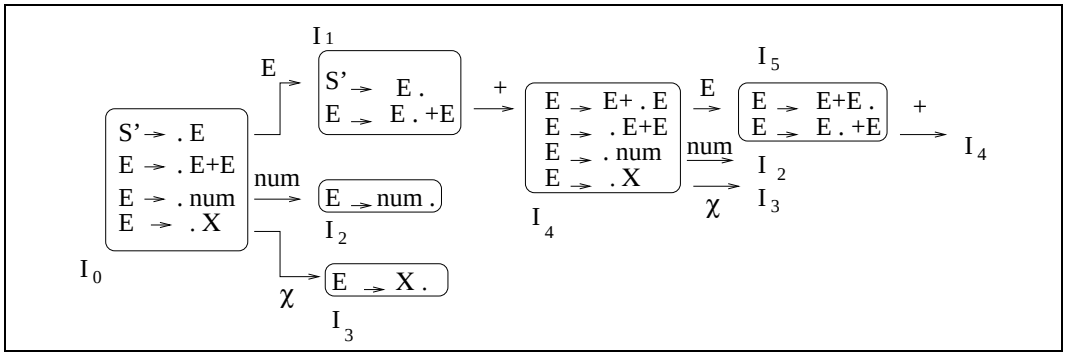


Figura 5.2: Autómata de elementos LR

# estado	+	num	χ	#	E
0		d2	d3		1
1	d4			aceptar	
2	r3			r3	
3	r4			r4	
4		d2	d3		5
5	r2			r2	

Tabla 5.6: Tabla de análisis sintáctico

A continuación veremos ejemplos que ilustran el comportamiento del algoritmo en dos situaciones diferentes.

χ como símbolo ficticio

Vamos a estudiar de qué forma reacciona el analizador sintáctico ante la cadena errónea $n++n\#$. Veamos la evolución de la pila, la entrada y la acción que se lleva a cabo en cada momento:

Pila	Entrada	Acción
0	num++num#	desplazar
0num2	++num#	reducir regla 3
0E1	++num#	desplazar
0E1+4	+num#	error

No existe ninguna transición válida con el carácter $+$ en el estado 4 por tanto se declara un error, el analizador introduce el símbolo ficticio χ en la entrada y como el estado 4 tiene una transición para ese carácter, se lleva

a cabo un desplazamiento al estado 3 y el análisis continua como si nada hubiera pasado:

Pila	Entrada	Acción
0E1+4	χ+num#	desplazar
0E1+4χ3	+num#	reducir regla 4
0E1+4E5	+num#	reducir regla 2
0E1	+num#	desplazar
0E1+4	num#	desplazar
0E1+4num2	#	reducir regla 3
0E1+4E5	#	reducir regla 2
0E1	#	aceptar

Tabla 5.7: Análisis sintáctico para la entrada `num + + num#`

El analizador se ha recuperado del error introduciendo en la cadena de entrada el símbolo ficticio χ, de forma que la expresión que realmente se ha reconocido ha sido `num+`χ`+num`. Ahora la pregunta es: ¿ Qué es χ? Tenemos que sustituirlo por un símbolo de la gramática adecuado en esa posición. En este caso el único que es posible es num, de forma que la cadena una vez recuperada es `num+num+num`, se ha corregido el error. Se podría haber sustituido también por `num+num`, pero siempre se suele sustituir por el símbolo que resulte más simple en el contexto de la regla que ha introducido χ en la entrada.

χ para ignorar parte de la entrada

Veamos ahora un ejemplo de como se comporta el analizador cuando es necesario ignorar símbolos en la entrada. En este caso se dice que el analizador 'tira' parte de la entrada antes de continuar con el reconocimiento. Supongamos la cadena de entrada `num num + num#`.

Pila	Entrada	Acción
0	num num+num#	desplazar
0num2	num+num#	error

No existe ninguna transición a partir del estado 2 con el carácter `num` y

tampoco ninguna para χ. En esta situación se sustituye el carácter conflictivo de la entrada por χ y se eliminan símbolos de la pila hasta encontrar un estado que admita una transición con este símbolo de error. En este caso sería necesario desapilar hasta que el estado 0 quedase en la cima. A partir de entonces la evolución del analizador sería:

Pila	Entrada	Acción
0	χ+num#	desplazar
0χ3	+num#	reducir regla 4
0E1	+num#	desplazar
0E1+4	num#	desplazar
0E1+4num2	#	reducir regla 3
0E1+4E5	#	reducir regla 2
0E1	#	aceptar

Tabla 5.8: Análisis sintáctico para la entrada `num num + num#`

Hemos reconocido $\chi + num$.

5.8.5 Heurísticas para obtener buenas producciones de error

Tal y como hemos visto la eficiencia del método depende en gran medida de la adecuada elección de las reglas de error y del punto en donde se coloca el símbolo ficticio χ. Desgraciadamente no existen reglas de uso general que podamos aplicar en todas las situaciones. Las reglas se definen en base a la experiencia introduciendo reglas heurísticas basadas en los errores más comunes y que funcionan de forma razonable en la mayoría de los casos. A continuación se enumeran algunos consejos que se pueden tener en cuenta:

1 Obtener una gramática sin producciones de error lo más compacta posible teniendo una idea intuitiva clara de los elementos del lenguaje.

2 A partir de la gramática anterior diseñe un analizador sintáctico y recopile una colección de programas con errores sintácticos y obtenga

una clasificación de los errores que han aparecido y con qué frecuencia.

3 Una vez clasificados los errores en función de su frecuencia, establecer una cota inferior, de forma que tan sólo introduciremos producciones de error para tratar de forma específica aquellos errores que se presentan con una frecuencia superior a la fijada. La forma de tratar estos errores es:

- Si el error ocurre con mucha frecuencia de forma sistemática. Por ejemplo en Pascal: olvidar el `;` en una declaración de un procedimiento o escribir detrás del `end` del programa principal un punto y coma `;` en vez del punto `.`. Lo mejor es entonces introducir acciones concretas para su tratamiento, por ejemplo:

  `bloque_ppal` $\rightarrow$ `BEGIN lista_inst END punto_bloque_ppal`

  `punto_bloque_ppal` $\rightarrow$ `. | ; yyerror(‘‘Se esperaba un ’.’ y se encontró un ’;’)`

  Aunque puede parecer una técnica un poco burda lo cierto es que muchos errores frecuentes se pueden tratar de una forma muy cómoda y sencilla con este método.

- Si el error es frecuente pero no puede describirse como la omisión o la adición de símbolos incorrectos, sino que se trata realmente de un conjunto errores que se suelen presentar en una posición específica de la entrada sin que ninguno de ellos se de con mayor frecuencia que los otros, entonces se utiliza una producción de error con el símbolo χ colocado en el punto conflictivo, $A \rightarrow \alpha\chi\gamma$ donde el prefijo α y el sufijo β son cadenas correctas. Por ejemplo, en una gramática para generar expresiones se suele producir un error con bastante frecuencia que consiste en la omisión de un operador o uso de un operador no existente en el lenguaje, entonces una producción adecuada de error sería:

  `E` $\rightarrow$ `E` χ `E`

4 No intentar introducir reglas de recuperación específica para los errores que no superen cierta frecuencia de corte mínima establecida.

Estos errores se tratan con las reglas de forma genérica, llamadas 'trágatelo-todo' (*catch-all*). En general, los lenguajes constan de subestructuras sintácticas delimitadas por ciertos símbolos de la gramática. Por ejemplo: un bloque de instrucciones empieza con el símbolo `begin` y termina con el símbolo `end`. De forma general una regla 'trágatelo-todo' es una regla de la forma $A \rightarrow t_1 \chi t_2$ donde A es el no-terminal que describe la estructura a la que queremos asociar el tratamiento de errores y t_1 y t_2 son los terminales respectivos símbolos de inicio y final.

5.8.6 Ejemplo

A continuación veamos un ejemplo sencillo para ilustrar la metodología. Supongamos un lenguaje para una pequeña calculadora que genera instrucciones del tipo:

```
print 1 +2
y:=3
x:= y + [y=0?1:4]
```

Tenemos sólo dos instrucciones: para imprimir y para asignar. Las expresiones son las aritméticas elementales y consta de un operador condicional. La primera tarea es diseñar una gramática que genere dicho lenguaje:

```
P → P I | I
I → PRINT E | ID ASIG E
E → E + E | E = E | ( E ) | [ E ? E : E ]
```

A continuación se realiza un estudio sobre los posibles errores más comunes obteniéndose los resultados:

1 Se omitió la palabra `print` (25%).

2 Se confundió el signo `=` con la asignación `:=` (15%).

3 Se olvidó cerrar el paréntesis en las expresiones (20%).

4 Se olvidó escribir un operador o se utilizó uno incorrecto (20%).

5 Se escribió `(?b:c)` en vez de `[a?b:c]` (5%).

6 Se olvidó cerrar un corchete (5%).

7 El resto fueron errores variados.

La gramática ampliada queda para este ejemplo:

```
P → P I
    | I
I → PRINT E
    | ID ASIG E
    | ID = E { yyerror(''Confundió ell = con el :=''); }
    | E { yyerror(''Olvidó el PRINT '');}
E → E + E
    | E = E
    | ( E )
    | [ E ? E : E ]
    | ( E { yyerror(''Olvidó el ) '');}
    | E χ E { yyerror(''Error en el operador '');}
```

Una vez introducidas estas reglas debemos completarlas con las reglas 'trágatelo-todo' para contemplar el posible resto de errores. Tenemos tres estructuras básicas: las instrucciones, las subexpresiones entre paréntesis y la expresión condicional encerrada entre corchetes, que faltaban por tratar. Por tanto, añadimos las reglas:

```
I → χ
E → ( χ )
    | [ χ ]
```

5.9 EJERCICIOS

1 (0.25 ptos) La siguiente gramática corresponde a algunas de las listas válidas en PROLOG. Se pide: (a) Construir la tabla de análisis SLR(1). (b) Eliminar los posibles conflictos que aparecen, considerando que el operador **,** es asociativo por la derecha.

Lista → [ ] | [Termino]
Termino → Termino **,** Termino | Lista | **id**

2 (0.25 ptos) Dada la siguiente gramática que genera bloques de código que pueden contener declaraciones y sentencias. Se pide: (a) Construir la colección de elementos LR(1). (b) Construir la tabla de análisis sintáctico LALR(1). (c) Obtener la traza del análisis LALR(1) para la cadena de entrada: **begin d; d; s; s end** (a) Construir la colección de elementos LR(1). (b) Construir la tabla de análisis sintáctico LALR(1). (c) Obtener la traza del análisis LALR(1) para la cadena de entrada: **begin d; d; s; s end**

B → **begin** D **;** S **end**
D → **d** | D **; d**
S → **s** | S **; s**

3 (0.25 ptos) Dada la siguiente gramática

S → **if** E S | **if** E S **else** S | **id**
E → E **and** E | E **or** E | **id** | **(** E **)**

(a) Construir la tabla de análisis sintáctico SLR(1). (b) En el caso de que haya conflictos elegir la acción adecuada para que el operador disyunción **or** tenga menor prioridad que el operador conjunción **and**, y que ambos tengan asociatividad por la derecha; y que toda parte del **else** de una sentencia condicional debe estar asociada al **if** más próximo.

4 (0.25 ptos) Demostrar que la siguiente gramática no es LR(1). ¿ Es LR(k) para algún k?

$$S \rightarrow X\ Y\ a$$
$$X \rightarrow a \mid Y\ a\ b$$
$$Y \rightarrow c \mid \epsilon$$

5 (0.25 ptos) Construir la tabla de precedencias de operadores para la siguiente gramática de operadores de forma que la parte del **else** esté asociada al **if** más cercano.

$$S \rightarrow \textbf{if}\ C\ \textbf{then}\ S \mid \textbf{if}\ C\ \textbf{then}\ S\ \textbf{else}\ S \mid \textbf{id}$$
$$C \rightarrow 0 \mid 1$$

Indica el contenido de la pila, la entrada y la acción a realizar para la entrada **if 0 then id else id**.

6 (0.25 ptos) Considerar la siguiente gramática. Se pide: (a) Construir el AFD de items LR(0). (b) Construir la tabla de análisis sintáctico SLR(1). (c) Mostrar el contenido de la pila, la entrada y la acción para la cadena **s; s; s**. (d) ¿ Es esta gramática LR(0)? Sino, describe el tipo de conflicto.

$$\text{stmt\_seq} \rightarrow \text{stmt\_seq}\ \textbf{;}\ \text{stmt} \mid \text{stmt}$$
$$\text{stmt} \rightarrow \textbf{s}$$