Capítulo 1

Análisis sintáctico

Los límites de mi lenguaje son los límites de mi mente.

Ludwig Wittgenstein (1889-1951)

RESUMEN: En este tema se explican las tareas del analizador sintáctico y el propósito de las gramáticas incontextuales, que son las que se utilizan para especificar la sintaxis de los lenguajes. Se presentan dos tipos de analizadores, los descendentes y los ascendentes, basados respectivamente en las gramáticas $LL(k)$ y $LR(k)$. Se espera del alumno que sepa construir manualmente analizadores descendentes a partir de gramáticas apropiadas, así como analizadores ascendentes con ayuda de herramientas específicas.

1.1. Introducción

- ⋆ El **analizador sintáctico** (*parser*) recibe como entrada la secuencia de unidades léxicas reconocidas por el analizador léxico y valida la correción sintáctica del programa de entrada.

- ⋆ Si el programa tiene **errores sintácticos**, los detecta y se **recupera** de ellos para proseguir el análisis. El objetivo es detectar el mayor número posible de errores sintácticos en cada compilación.

- ⋆ El **alfabeto** de entrada es por tanto $\Sigma = \{\text{clases de unidades lexicas}\}$ y el lenguaje formal que reconoce es un cierto $L \subseteq \Sigma^*$. Llamaremos **símbolos** o **palabras** a los elementos de Σ, y llamaremos **frases** a los de L.

- ⋆ La descripción del lenguaje L se hace mediante una **gramática incontextual** y el reconocimiento de las frases válidas se realiza mediante un **autómata con pila**.

- ⋆ En general, los autómatas con pila son **indeterministas**. A diferencia de los autómatas finitos, no siempre existe un autómata determinista equivalente a uno no determinista. En el caso peor, el reconocimiento de una frase de longitud n tiene un coste en tiempo en $O(n^3)$.

- ⋆ Existen **subconjuntos** de las gramáticas incontextuales que admiten **analizadores deterministas**. Se llaman $LL(k)$ y $LR(k)$, con $k \geq 0$, y la relación entre ellas es:

$$LL(k) \subset LR(k) \subset \textit{gramaticas incontextuales}$$

- ⋆ El parámetro k indica el número de símbolos de **preanálisis** que se utilizan para decidir la regla a aplicar. La primera L, que la frase se procesa *left-to-right*; la segunda L, que el análisis se hace siguiendo una **derivación por la izquierda**; y la R, que se hace siguiendo una **derivación por la derecha**.

- ⋆ Los analizadores deterministas reconocen frases de longitud n en un tiempo en $O(n)$.

- ⋆ Los analizadores $LL(k)$ se llaman **descendentes** porque recorren el árbol sintáctico desde la raíz a las hojas. Los $LR(k)$ se llaman **ascendentes** porque proceden en sentido contrario.

1.2. Gramáticas incontextuales

- ⋆ Las expresiones y gramáticas regulares **no son suficientes** para expresar las estructuras anidadas que aparecen en los lenguajes de programación (LP). *Ejemplo:* $((\ldots)\ldots(((\ldots)\ldots)(\ldots))\ldots)$. La siguiente gramática incontextual genera todas las posibles cadenas con paréntesis anidados bien equilibrados:

$$S \to \underline{(}S\underline{)}S \mid \epsilon$$

- ⋆ Las gramáticas contextuales y las de tipo-0 son **más expresivas** que las incontextuales pero no tienen reconocedores eficientes. Las incontextuales representan un **buen equilibrio** entre expresividad y eficiencia.

- ⋆ Una **gramática incontextual** es una tupla $G = (V_N, V_T, P, S)$
  - V_N conjunto finito de símbolos **no terminales**. $A, B, S, X, \mathit{exp}, \mathit{instr}, \ldots \in V_N$.
  - V_T conjunto finito de símbolos **terminales**. $a, b, c, +, -, *, \mathbf{id}, \mathbf{begin}, \ldots \in V_T$.
  - $P \subseteq V_N \times (V_N \cup V_T)^*$ conjunto finito de **reglas de producción**.
    - $\alpha, \beta, \gamma, \ldots \in (V_N \cup V_T)^*$
    - $u, v, w, x, \ldots \in V_T^*$
    - $(A \to \alpha) \in P$. Abreviatura: $(A \to \alpha_1 \mid \ldots \mid \alpha_n) \subseteq P$.
  - $S \in V_N$ **símbolo inicial**, o **raíz**, de G.

Llamaremos $V = V_N \cup V_T$, con $N, M, O, \ldots \in V$.

⋆ *Ejemplo 1:* Expresiones con prioridad de $*$ respecto de $+$:

$$G_1 = (\{E, T, F\}, \{+, *, \underline{(}, \underline{)}, \mathbf{id}\}, P_1, E) \qquad P_1 = \begin{cases} E \to E + T \mid T \\ T \to T * F \mid F \\ F \to \underline{(}E\underline{)} \mid \mathbf{id} \end{cases}$$

⋆ *Ejemplo 2:* Expresiones sin prioridad entre $*$ y $+$:

$$G_2 = (\{E\}, \{+, *, \underline{(}, \underline{)}, \mathbf{id}\}, P_2, E) \qquad P_2 = \{\ E \to E + E \mid E * E \mid \underline{(}E\underline{)} \mid \mathbf{id}\ \}$$

⋆ Dada $G = (V_N, V_T, P, S)$, definimos **derivación en un paso** según G: $\alpha A \beta \Rightarrow_G \alpha\gamma\beta$, si existe $(A \to \gamma) \in P$.

⋆ La **derivación** en cero o más pasos según G, $\phi \Rightarrow_G^* \psi$, es el cierre reflexivo y transitivo de $\Rightarrow_G$.

⋆ **Lenguaje generado** por una gramática G:

$$L(G) = \{u \in V_T^* \mid S \Rightarrow_G^* u\}$$

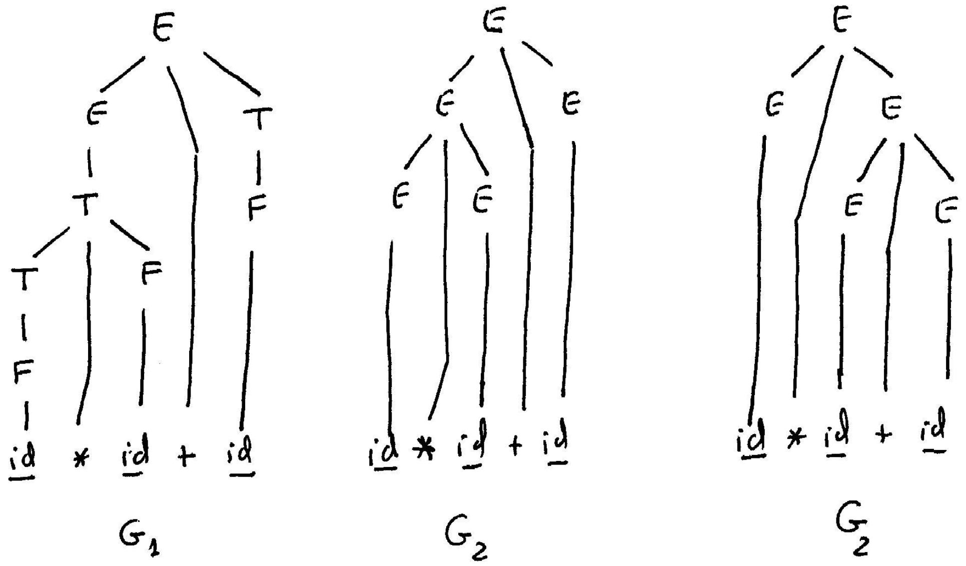
Decimos que las u son **frases** de G. Si $S \Rightarrow_G^* \alpha \in V^*$, decimos que α es una **forma de frase** de G.

⋆ *Ejemplo:* $\mathbf{id} * \mathbf{id} + \mathbf{id}$ es una frase tanto de G_1 como de G_2, $\mathbf{id} * \mathbf{id} + T$ es una forma de frase de G_1, y $E * \mathbf{id} + E$ lo es de G_2 (escribir las derivaciones).

⋆ Dada una forma de frase α de G, un **árbol sintáctico** para α es un árbol tal que:

1. S es el nodo raíz.
2. α es la **frontera** (recorrido de las hojas en sentido antihorario)
3. Cada nodo interior contiene $X \in V_N$ y sus hijos directos $N_1, \ldots, N_r \in V$ son tales que $(X \to N_1 \cdots N_r) \in P$, o tiene un único hijo ϵ y $(X \to \epsilon) \in P$.

⋆ *Ejemplo:* Para la frase $\mathbf{id} * \mathbf{id} + \mathbf{id}$ tenemos los siguientes árboles sintácticos:



- ⋆ Una (forma de) frase es **ambigua** si admite más de un árbol sintáctico. Una gramática es ambigua si contiene al menos una frase ambigua. Por ejemplo, la gramática G_2 es ambigua. Es posible demostrar que G_1 no lo es.

- ⋆ Para demostrar que una gramática es ambigua, basta exhibir una frase que lo sea. Demostrar que no es ambigua requiere razonamientos generales que impliquen toda frase posible generada (por ejemplo, razonamientos por inducción sobre la estructura de las frases). El problema de determinar si una gramática incontextual es ambigua o no, es **indecidible**.

- ⋆ Afortunadamente, ser $LL(k)$ o $LR(k)$ implica ser no ambigua.

- ⋆ Las gramáticas que describen LPs **no deben** ser ambiguas. Admitir más de un árbol sintáctico para una frase implicaría que hay varias formas distintas de interpretar la frase y por tanto varias formas distintas de traducirla a código máquina.

- ⋆ Un árbol sintáctico para una (forma de) frase α no expresa en qué orden se han reemplazado los no terminales por sus partes derechas, por lo que resume muchas posibles derivaciones para α. Dos de ellas tienen especial interés:

  derivación por la izquierda $S \Rightarrow^*_{iz} \alpha$ si en cada paso se reemplaza el no-terminal más a la izquierda. Equivale a recorrer el árbol sintáctico en **preorden**.

  derivación por la derecha $S \Rightarrow^*_{de} \alpha$ si en cada paso se reemplaza el no-terminal más a la derecha. Equivale a recorrer el árbol sintáctico en **preorden inverso**.

- ⋆ *Ejemplo:* Para la frase $\mathbf{id} * \mathbf{id} + \mathbf{id}$ y G_1 tendríamos:

$$E \Rightarrow_{iz} E + T \Rightarrow_{iz} T + T \Rightarrow_{iz} T * F + T \Rightarrow_{iz} F * F + T \Rightarrow_{iz} \mathbf{id} * F + T \ldots$$
$$E \Rightarrow_{de} E + T \Rightarrow_{de} E + F \Rightarrow_{de} E + \mathbf{id} \Rightarrow_{de} T + \mathbf{id} \Rightarrow_{de} T * F + \mathbf{id} \Rightarrow_{de} T * \mathbf{id} + \mathbf{id} \ldots$$

- ⋆ Los analizadores $LL(k)$ realizan un **derivación por la izquierda**, y los $LR(k)$ una **derivación por la derecha** en sentido inverso al de $\Rightarrow_{de}$.

1.3. Análisis de gramáticas

- ⋆ Para generar analizadores a partir de gramáticas, se requieren ciertos **análisis y transformaciones** de la gramática original:
  - eliminar no-terminales improductivos
  - eliminar no-terminales inalcanzables
  - calcular $prim_k(X)$ para cada no-terminal X
  - calcular $sig_k(X)$ para cada no-terminal X

- ⋆ X es **productivo** en $p \in P$ si $p = X \to \alpha$ y todos los no-terminales de α son productivos. En particular, si $\alpha = u \in V_T^*$, X es productivo en p. X es productivo en G, si es productivo en alguna regla de G.

- ⋆ *Ejemplo:* Z es improductivo en la gramática

$$\begin{aligned} &S \to aX \\ &X \to bS \mid aYbY \\ &Y \to ba \mid aZ \\ &Z \to aZX \end{aligned}$$

- ⋆ Eliminar no-terminales improductivos consiste en eliminar todas las reglas en las que aparecen, tanto en la izquierda como en la derecha.

- ⋆ X es **alcanzable** en $p \in P$ si $p = Y \to \alpha X \beta$, e Y es alcanzable. La raíz S de G siempre es alcanzable. X es alcanzable en G, si lo es en alguna regla de G.

- ⋆ *Ejemplo:* U, V son inalcanzables en la gramática

$$\begin{array}{ll} S \to Y & \quad Y \to YZ \mid Ya \mid b \\ U \to V & \quad X \to c \\ V \to Vd \mid d & \quad Z \to ZX \end{array}$$

- ⋆ Eliminar no-terminales inalcanzables consiste en eliminar todas las reglas en las que aparecen, tanto en la izquierda como en la derecha.

⋆ Determinar los no-terminales improductivos e inalcanzables es **decidible**. Primero se eliminan los improductivos y luego los inalzanzables. La gramática resultante se dice **reducida**. En adelante supondremos que nuestras gramáticas lo son.

⋆ Los analizadores deterministas $LL(k)$ y $LR(k)$ necesitan conocer los **primeros** k símbolos por los que pueden comenzar las frases generadas por cada no-terminal X, y los **siguientes** k símbolos que pueden ocurrir tras cada frase generada por X. El caso más frecuente es $k = 1$.

$$\begin{array}{l} prim_k(\alpha) = \{k : u \mid \alpha \Rightarrow_G^* u\} \\ sig_k(X) = \{w \mid S \Rightarrow_G^* \alpha X \beta, w \in prim_k(\beta \vdash)\} \end{array}$$

donde $0 : \alpha = \epsilon$, $k : \epsilon = \epsilon$, y $(k+1) : a\alpha = a(k : \alpha)$. El símbolo $\vdash$ se lee "fin-de-fichero", no pertenece a V_T, y asegura que ϵ **nunca pertenece** a $sig_k(X)$.

⋆ Definimos la **concatenación módulo** k de dos cadenas u, v y de dos conjuntos L, L' de cadenas:

$$u \oplus_k v = k : (uv) \qquad L \oplus_k L' = \{u \oplus_k v \mid u \in L, v \in L'\}$$

⋆ Algunas propiedades de $prim_k$:

1) $prim_k(u) = \{k : u\}$
2) $prim_k(u_0 X_1 u_1 \cdots X_n u_n) = \{u_o\} \oplus_k prim_k(X_1) \oplus_k \cdots \oplus_k prim_k(X_n) \oplus_k \{u_n\}$
3) $\emptyset \oplus_k L = \emptyset$
4) si $X \to \alpha_1 | \ldots | \alpha_n$ son todas las reglas de X, entonces $prim_k(X) = \bigcup_{i=1}^n prim_k(\alpha_i)$

⋆ Aplicación a G_1 con $k = 1$:

$$\begin{array}{l} prim(E) = (prim(E) \oplus_1 \{+\} \oplus_1 prim(T)) \cup prim(T) \\ prim(T) = (prim(T) \oplus_1 \{*\} \oplus_1 prim(F)) \cup prim(F) \\ prim(F) = prim(\underline{(}E\underline{)}) \cup prim(\mathbf{id}) = \{\underline{(}, \mathbf{id}\} \end{array}$$

⋆ Se genera un conjunto de ecuaciones **mutuamente recursivas** que se resuelven mediante el siguiente **algoritmo de punto fijo**:

1. Inicialmente asignar a todos los no-terminales $prim(X) = \emptyset$.
2. En cada iteración aplicar todas las ecuaciones de $prim$, usando para las apariciones de $prim(X)$ en la parte derecha el conjunto calculado en la iteración anterior.
3. Cuando en una iteración no cambie ninguno de los conjuntos, parar. El último conjunto alcanzado para X es el valor de $prim(X)$.

⋆ Aplicación a G_1:

Num. iter.	$prim(E)$	$prim(T)$	$prim(F)$
0	$\emptyset$	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$	$\{\underline{(}, \mathbf{id}\}$
2	$\emptyset$	$\{\underline{(}, \mathbf{id}\}$	$\{\underline{(}, \mathbf{id}\}$
3	$\{\underline{(}, \mathbf{id}\}$	$\{\underline{(}, \mathbf{id}\}$	$\{\underline{(}, \mathbf{id}\}$
4	$\boxed{\{\underline{(}, \mathbf{id}\}}$	$\boxed{\{\underline{(}, \mathbf{id}\}}$	$\boxed{\{\underline{(}, \mathbf{id}\}}$

Nótese la propagación de la información en sentido **ascendente** con respecto a la gramática.

⋆ Para calcular $sig(X)$ primero hay que asegurarse de que la raíz S de la gramática **no aparece** en la parte derecha de ninguna regla. Si apareciese, se **extiende** la gramática con una nueva raíz S' y con la regla $S' \to S$. Es claro que $sig(S') = \{\vdash\}$. Para todo otro no-terminal X se escribe la ecuación:

$$sig_k(X) = \bigcup_{(Y \to \alpha X \beta) \in P} prim_k(\beta) \oplus_k sig_k(Y)$$

⋆ Aplicación a $G_1 \cup \{S \to E\}$ con $k = 1$:

$$\begin{array}{ll}
sig(S) = \{\vdash\} & \\
sig(E) = (prim(+T) \oplus_1 sig(E)) \cup (prim(\underline{)}) \oplus_1 sig(F)) \cup sig(S) & = \{+, \underline{)}\} \cup sig(S) \\
sig(T) = (prim(*F) \oplus_1 sig(T)) \cup sig(E) & = \{*\} \cup sig(E) \\
sig(F) = sig(T) &
\end{array}$$

⋆ Algoritmo para calcular $sig(X)$:

1. Calcular previamente $prim(X)$ para todo X.
2. Inicialmente asignar a todos los no-terminales $sig(X) = \emptyset$ excepto a la raíz $sig(S') = \{\vdash\}$.
3. En cada iteración aplicar todas las ecuaciones de sig, usando para las apariciones de $sig(X)$ en la parte derecha el conjunto calculado en la iteración anterior.
4. Cuando en una iteración no cambie ninguno de los conjuntos, parar. El último conjunto alcanzado para X es el valor de $sig(X)$.

⋆ Aplicación a G_1 extendida con S:

Num. iter.	$sig(S)$	$sig(E)$	$sig(T)$	$sig(F)$
0	$\{\vdash\}$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\{\vdash\}$	$\{+, \underline{)}, \vdash\}$	$\{*\}$	$\emptyset$
2	$\{\vdash\}$	$\{+, \underline{)}, \vdash\}$	$\{*, +, \underline{)}, \vdash\}$	$\{*\}$
3	$\{\vdash\}$	$\{+, \underline{)}, \vdash\}$	$\{*, +, \underline{)}, \vdash\}$	$\{*, +, \underline{)}, \vdash\}$
4	$\boxed{\{\vdash\}}$	$\boxed{\{+, \underline{)}, \vdash\}}$	$\boxed{\{*, +, \underline{)}, \vdash\}}$	$\boxed{\{*, +, \underline{)}, \vdash\}}$

Nótese la propagación de la información en sentido **descendente** con respecto a la gramática.

1.4. Autómatas con pila

⋆ Para cada lenguaje incontextual existe un autómata con pila (AP) que lo reconoce. Además de un conjunto finito de estados, los APs poseen una **pila** potencialmente **infinita** donde pueden almacenar símbolos.

⋆ Un **autómata con pila** es una tupla $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$:

- Q conjunto finito de estados
- Σ alfabeto (finito) de entrada
- $\Gamma \supseteq \Sigma$ alfabeto (finito) de pila
- $\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times \Gamma \cup \{\epsilon\} \times Q \times \Gamma \cup \{\epsilon\}$ relación de transición
- $q_0 \in Q$ estado inicial
- $F \subseteq Q$ conjunto de estados finales

⋆ En una **transición** $(q, a, s; p, t) \in \delta$, ante una entrada a y cima de la pila s (ambos pueden ser ϵ), el estado interno pasa de q a p, el símbolo s es desapilado y el t apilado en su lugar. El **no determinismo** reside en que puede haber varias tuplas $(q, a, s; ?, ?)$.

⋆ Usaremos una **versión especializada** $P = (V_T, Q, \delta, q_0, F)$, en la que el alfabeto de entrada coincide con el de la gramática, el de pila coincide con Q, y $\delta \subseteq Q^+ \times V_T \cup \{\epsilon\} \times Q^*$ puede desapilar cadenas no vacías de la cima y apilar cadenas posiblemente vacías. El **estado en curso** es el situado en la cima.

⋆ Una **configuración** de un AP es un par $(\gamma, w) \in Q^+ \times V_T^*$ en el que γ representa el contenido de la pila (cima a la derecha) y w es la entrada por procesar (primer símbolo a la izquierda).

- (q_0, w) configuración inicial.
- $(\gamma q, \epsilon)$, con $q \in F$, configuración final.
- $(\gamma_1\gamma_2, aw) \vdash_P (\gamma_1\gamma_3, w)$, con $a \in V_T \cup \{\epsilon\}$ y $(\gamma_2, a, \gamma_3) \in \delta$, relación de transición entre configuraciones.
- $\vdash_P^*$ cierre reflexivo y transitivo de $\vdash_P$.
- lenguaje aceptado por P, $L(P) = \{w \in V_T^* \mid (q_0, w) \vdash_P^* (\gamma q, \epsilon),\ q \in F\}$

⋆ Dada $G = (V_N, V_T, P, S)$ incontextual, definimos el **autómata de items** K_G asociado a G:

$$K_G = (V_T, It_G, \delta, [S' \to \bullet S], \{[S' \to S\bullet]\})$$

donde S' es un símbolo no en V_N.

- un **item** es una terna $[A \to \alpha\bullet\beta]$ para toda regla $(A \to \alpha\beta) \in P$. A α se le llama **historia** del item. Si $\beta = \epsilon$, escribimos $[A \to \alpha\bullet]$ y el item se dice **completo**.

- Llamamos It_G al conjunto (finito) de todos los items posibles de G.
- regla de **expansión**: $\delta([X \to \beta \bullet Y\gamma], \epsilon) = \{[X \to \beta \bullet Y\gamma][Y \to \bullet\alpha] \mid Y \to \alpha \in P\}$
- regla de **desplazamiento**: $\delta([X \to \beta \bullet a\gamma], a) = \{[X \to \beta a \bullet \gamma]\}$
- regla de **reducción**: $\delta([X \to \beta \bullet Y\gamma][Y \to \alpha\bullet], \epsilon) = \{[X \to \beta Y \bullet \gamma]\}$

Nótese que K_G es en general **no-determinista** debido a la regla de expansión. Posiblemente habrá más de una regla $(Y \to \alpha) \in P$.

⋆ Dada G incontextual y su autómata de items asociado K_G, se cumple que:

$$L(G) = L(K_G)$$

⋆ En la Figura 1.1 se muestra el reconocimiento de una frase de la gramática G_1 por su correspondiente autómata de items. En los lugares donde se aplica la regla de expansión, nótese que se apila el item que conduce al estado final.

1.5. Analizadores descendentes $LL(1)$

⋆ Un analizador descendente $LL(k)$ es un autómata de items al que se ha eliminado el no-determinismo de la regla de expansión. Para ello, se le permite **inspeccionar anticipadamente** (sin consumirlos) los primeros k símbolos de la cadena remanente de entrada.

⋆ Supongamos la siguiente evolución del autómata de items:

$$([S' \to \bullet S], uv) \vdash_{K_G}^{*} ([S' \to \bullet S] \cdots [X \to \beta \bullet Y\gamma], v)$$

y que $Y \to \alpha_1 \mid \ldots \mid \alpha_n$. El autómata ha de elegir una de las alternativas para Y de forma que $Y \Rightarrow_G^* v_1$ y $v = v_1 v_2$. La idea intuitiva es que si $prim_k(\alpha_1), \ldots prim_k(\alpha_n)$ fueran conjuntos disjuntos, podría utilizar $k : v$ para elegir la alternativa i que cumpla $k : v \in prim_k(\alpha_i)$.

⋆ Una gramática es **LL(k)** si para todo par de derivaciones por la izquierda

$$S \Rightarrow_{iz}^* uY\alpha \Rightarrow_{iz} u\beta\alpha \Rightarrow_{iz}^* ux$$
$$S \Rightarrow_{iz}^* uY\alpha \Rightarrow_{iz} u\gamma\alpha \Rightarrow_{iz}^* uy$$

siempre que se cumple $k : x = k : y$, se cumple también $\beta = \gamma$.

⋆ En otras palabras, la alternativa elegida para expandir un no-terminal Y está **unívocamente determinada** por la historia u y los primeros k símbolos de la cadena remanente. Nótese que la definición no da lugar a un algoritmo, pues podría haber infinitas derivaciones por la izquierda.

⋆ Cuando $k = 1$, una condición **necesaria y suficiente** para que G sea $LL(1)$ es que cumpla para todo conjunto de reglas $Y \to \alpha_1 \mid \ldots \mid \alpha_n$:

Stack contents	Remaining input
$[S \to .E]$	$\mathbf{id} + \mathbf{id} * \mathbf{id}$
$[S \to .E][E \to .E + T]$	$\mathbf{id} + \mathbf{id} * \mathbf{id}$
$[S \to .E][E \to .E + T][E \to .T]$	$\mathbf{id} + \mathbf{id} * \mathbf{id}$
$[S \to .E][E \to .E + T][E \to .T][T \to .F]$	$\mathbf{id} + \mathbf{id} * \mathbf{id}$
$[S \to .E][E \to .E + T][E \to .T][T \to .F][F \to .\mathbf{id}]$	$\mathbf{id} + \mathbf{id} * \mathbf{id}$
$[S \to .E][E \to .E + T][E \to .T][T \to .F][F \to \mathbf{id}.]$	$+\mathbf{id} * \mathbf{id}$
$[S \to .E][E \to .E + T][E \to .T][T \to F.]$	$+\mathbf{id} * \mathbf{id}$
$[S \to .E][E \to .E + T][E \to T.]$	$+\mathbf{id} * \mathbf{id}$
$[S \to .E][E \to E. + T]$	$+\mathbf{id} * \mathbf{id}$
$[S \to .E][E \to E + .T]$	$\mathbf{id} * \mathbf{id}$
$[S \to .E][E \to E + .T][T \to .T * F]$	$\mathbf{id} * \mathbf{id}$
$[S \to .E][E \to E + .T][T \to .T * F][T \to .F]$	$\mathbf{id} * \mathbf{id}$
$[S \to .E][E \to E + .T][T \to .T * F][T \to .F][F \to .\mathbf{id}]$	$\mathbf{id} * \mathbf{id}$
$[S \to .E][E \to E + .T][T \to .T * F][T \to .F][F \to \mathbf{id}.]$	$*\mathbf{id}$
$[S \to .E][E \to E + .T][T \to .T * F][T \to F.]$	$*\mathbf{id}$
$[S \to .E][E \to E + .T][T \to T. * F]$	$*\mathbf{id}$
$[S \to .E][E \to E + .T][T \to T * .F]$	$\mathbf{id}$
$[S \to .E][E \to E + .T][T \to T * .F][F \to .\mathbf{id}]$	$\mathbf{id}$
$[S \to .E][E \to E + .T][T \to T * .F][F \to \mathbf{id}.]$	
$[S \to .E][E \to E + .T][T \to T * F.]$	
$[S \to .E][E \to E + T.]$	
$[S \to E.]$	

Figura 1.1: Reconocimiento de la frase $\mathbf{id} + \mathbf{id} * \mathbf{id}$ de G_1

1. $prim(\alpha_i) \cap prim(\alpha_j) = \emptyset$, para $1 \leq i < j \leq n$.
2. si existe i t.q. $\epsilon \in prim(\alpha_i)$, entonces $prim(\alpha_j) \cap sig(Y) = \emptyset$, para todo $j \neq i$.

Estas condiciones se pueden verificar algorítmicamente.

$\star$ *Ejemplo:* La siguiente gramática de expresiones es $LL(1)$:

$$G_3 = \left\{ \begin{array}{ll} S \to E & T \to FT' \\ E \to TE' & T' \to \, *T \mid \epsilon \\ E' \to \, +E \mid \epsilon & F \to \underline{(}E\underline{)} \mid \mathbf{id} \end{array} \right.$$

Si calculamos primeros y siguientes (hacerlo) obtenemos:

	S	E	E'	T	T'	F
$prim(X)$	$\{\underline{(},\mathbf{id}\}$	$\{\underline{(},\mathbf{id}\}$	$\{+,\epsilon\}$	$\{\underline{(},\mathbf{id}\}$	$\{*,\epsilon\}$	$\{\underline{(},\mathbf{id}\}$
$sig(X)$	$\{\vdash\}$	$\{\vdash,\underline{)}\}$	$\{\vdash,\underline{)}\}$	$\{+,\vdash,\underline{)}\}$	$\{+,\vdash,\underline{)}\}$	$\{*.+,\vdash,\underline{)}\}$

que cumple las condiciones exigidas:

$$\begin{array}{ll} 1) & prim(+E) \cap prim(\epsilon) = \emptyset \\ 1') & prim(*T) \cap prim(\epsilon) = \emptyset \\ 1'') & prim(\underline{(}E\underline{)}) \cap prim(\mathbf{id}) = \emptyset \\ 2) & prim(+E) \cap sig(E') = \emptyset \\ 2') & prim(*T) \cap sig(T') = \emptyset \end{array}$$

$\star$ Si $G = (V_N, V_T, P, S)$ es $LL(1)$, un **analizador LL(1)** para G es el autómata con pila K_G, provisto de una tabla adicional, llamada **tabla predictora**:

$$m : V_N \times V_T \longrightarrow P \cup \{error\}$$

que determina la regla a aplicar cuando el item en la cima es de la forma $[X \to \beta \bullet Y\gamma]$.

- Si el primer símbolo de la cadena remanente es a, $m[Y,a] = (Y \to \alpha)$ nos dice que hay que expandir Y con $Y \to \alpha$.
- Si $m[Y,a] = error$, la cadena de entrada no pertenece al lenguaje de G.

$\star$ Una vez comprobado que una gramática es $LL(1)$, el **algoritmo para rellenar la tabla** predictora m es el siguiente:

1. Para toda regla $(X \to \alpha) \in P$, para todo símbolo $a \in prim(\alpha)$, $a \neq \epsilon$, asignar $m[X,a] \leftarrow (X \to \alpha)$.
2. Para toda regla $(X \to \alpha) \in P$, si $\epsilon \in prim(\alpha)$, entonces para todo símbolo $b \in sig(X)$ asignar $m[X,b] \leftarrow (X \to \alpha)$.
3. Asignar al resto de las entradas $m[X,c] \leftarrow error$.

	(	)	+	*	**id**	#
E	$(E \rightarrow TE')$	*error*	*error*	*error*	$(E \rightarrow TE')$	*error*
E'	*error*	$(E' \rightarrow \varepsilon)$	$(E' \rightarrow +E)$	*error*	*error*	$(E' \rightarrow \varepsilon)$
T	$(T \rightarrow FT')$	*error*	*error*	*error*	$(T \rightarrow FT')$	*error*
T'	*error*	$(T' \rightarrow \varepsilon)$	$(T' \rightarrow \varepsilon)$	$(T' \rightarrow *T)$	*error*	$(T' \rightarrow \varepsilon)$
F	$(F \rightarrow (E))$	*error*	*error*	*error*	$(F \rightarrow \mathbf{id})$	*error*
S	$(S \rightarrow E)$	*error*	*error*	*error*	$(S \rightarrow E)$	*error*

Figura 1.2: Tabla predictora del analizador descendente $LL(1)$ para G_3

⋆ La tabla predictora de la Figura 1.2 corresponde a la gramática $LL(1)$ G_3. El símbolo # ha de ser interpretado como $\vdash$.

⋆ En la Figura 1.3 se muestra el análisis descendente $LL(1)$ de la frase $\mathbf{id} * \mathbf{id}$ con el analizador resultante de usar dicha tabla. Si el autómata escribe en orden cada regla de expansión utilizada, nótese que componen una derivación por la izquierda.

1.5.1. Transformación de gramáticas a $LL(1)$

⋆ Una gramática **ambigua no es** $LL(k)$ (por tanto $LL(k) \Rightarrow \neg ambigua$). Una frase ambigua ux admite dos o más árboles sintácticos, y por tanto más de una derivación por la izquierda:

$$S \Rightarrow^*_{iz} uY\alpha \Rightarrow_{iz} u\beta\alpha \Rightarrow^*_{iz} ux$$
$$S \Rightarrow^*_{iz} uY\alpha \Rightarrow_{iz} u\,\gamma\alpha \Rightarrow^*_{iz} ux$$

con $k : x = k : x$ y $\beta \neq \gamma$.

⋆ Una gramática **recursiva por la izquierda** no es $LL(k)$:

- G es recursiva por la izquierda si tiene una regla $A \rightarrow A\beta$.
- Como G es reducida, tendrá otra regla $A \rightarrow \gamma$ (si no, A sería improductivo). Entonces tenemos:

  $$S \Rightarrow^*_{iz} uA\alpha \Rightarrow^*_{iz} uA\beta^n\alpha$$

  Para poder elegir una de las alternativas $A\beta$ o γ de A, debería cumplirse $prim_k(A\beta^{n+1}\alpha) \cap prim_k(\gamma\beta^n\alpha) = \emptyset$, que a su vez implica $prim_k(\gamma\beta^{n+1}\alpha) \cap prim_k(\gamma\beta^n\alpha) = \emptyset$, lo cual es imposible para ningún k, escogiendo n suficientemente grande.

Stack contents	Input
$\#[S \rightarrow .E]$	$\mathbf{id} * \mathbf{id}\#$
$\#[S \rightarrow .E][E \rightarrow .TE']$	$\mathbf{id} * \mathbf{id}\#$
$\#[S \rightarrow .E][E \rightarrow .TE'][T \rightarrow .FT']$	$\mathbf{id} * \mathbf{id}\#$
$\#[S \rightarrow .E][E \rightarrow .TE'][T \rightarrow .FT'][F \rightarrow .\mathbf{id}]$	$\mathbf{id} * \mathbf{id}\#$
$\#[S \rightarrow .E][E \rightarrow .TE'][T \rightarrow .FT'][F \rightarrow \mathbf{id}.]$	$*\mathbf{id}\#$
$\#[S \rightarrow .E][E \rightarrow .TE'][T \rightarrow F.T']$	$*\mathbf{id}\#$
$\#[S \rightarrow .E][E \rightarrow .TE'][T \rightarrow F.T'][T' \rightarrow .*T]$	$*\mathbf{id}\#$
$\#[S \rightarrow .E][E \rightarrow .TE'][T \rightarrow F.T'][T' \rightarrow *.T]$	$\mathbf{id}\#$
$\#[S \rightarrow .E][E \rightarrow .TE'][T \rightarrow F.T'][T' \rightarrow *.T][T \rightarrow .FT']$	$\mathbf{id}\#$
$\#[S \rightarrow .E][E \rightarrow .TE'][T \rightarrow F.T'][T' \rightarrow *.T][T \rightarrow .FT'][F \rightarrow .\mathbf{id}]$	$\mathbf{id}\#$
$\#[S \rightarrow .E][E \rightarrow .TE'][T \rightarrow F.T'][T' \rightarrow *.T][T \rightarrow .FT'][F \rightarrow \mathbf{id}.]$	$\#$
$\#[S \rightarrow .E][E \rightarrow .TE'][T \rightarrow F.T'][T' \rightarrow *.T][T \rightarrow F.T']$	$\#$
$\#[S \rightarrow .E][E \rightarrow .TE'][T \rightarrow F.T'][T' \rightarrow *.T][T \rightarrow F.T'][T' \rightarrow \varepsilon.]$	$\#$
$\#[S \rightarrow .E][E \rightarrow .TE'][T \rightarrow F.T'][T' \rightarrow *.T][T \rightarrow FT'.]$	$\#$
$\#[S \rightarrow .E][E \rightarrow .TE'][T \rightarrow F.T'][T' \rightarrow *T.]$	$\#$
$\#[S \rightarrow .E][E \rightarrow .TE'][T \rightarrow FT'.]$	$\#$
$\#[S \rightarrow .E][E \rightarrow T.E']$	$\#$
$\#[S \rightarrow .E][E \rightarrow T.E'][E' \rightarrow \varepsilon.]$	$\#$
$\#[S \rightarrow .E][E \rightarrow TE'.]$	$\#$
$\#[S \rightarrow E.]$	$\#$
$\#$	$\#$

Output

$(S \rightarrow E)\,(E \rightarrow TE')\,(T \rightarrow FT')\,(F \rightarrow \mathbf{id})\,(T' \rightarrow *T)\,(T \rightarrow FT')\,(F \rightarrow \mathbf{id})$
$(T' \rightarrow \varepsilon)\,(E' \rightarrow \varepsilon)$

Figura 1.3: Análisis descendente $LL(1)$ para la frase $\mathbf{id} * \mathbf{id}$ de G_3

⋆ La recursión por la izquierda se puede transformar en **recursión por la derecha** con el siguiente algoritmo: **sustituir** cada regla de la forma $(A \to A\alpha \mid \beta) \in P$ por las reglas:

$$A \to \beta A' \qquad\qquad A' \to \alpha A' \mid \epsilon$$

El algoritmo se puede extender fácilmente a más de una regla recursiva y más de una regla no recursiva, así como a gramáticas con recursión **indirecta** por la izquierda (i.e. con derivaciones $A \Rightarrow^{+}_{G} A\alpha$).

⋆ *Ejemplo:* Aplicación a G_1

$$G_1 = \begin{cases} E \to E + T \mid T \\ T \to T * F \mid F \\ F \to \underline{(}E\underline{)} \mid \mathbf{id} \end{cases} \qquad G_4 = \begin{cases} E \to TE' & E' \to +TE' \mid \epsilon \\ T \to FT' & T' \to *FT' \mid \epsilon \\ F \to \underline{(}E\underline{)} \mid \mathbf{id} \end{cases}$$

⋆ Una gramática **no está factorizada por izquierda** si tiene reglas de la forma:

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \quad \text{con } \mathit{prim}(\alpha) \neq \{\epsilon\}$$

Claramente, este tipo de gramáticas **no** son $LL(1)$.

⋆ El siguiente algoritmo de **factorización por la izquierda** podría quizás eliminar este problema:

1. **Sustituir** cada regla de la forma

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \delta_1 \mid \cdots \mid \delta_m$$

por

$$A \to \alpha A' \mid \delta_1 \mid \cdots \mid \delta_m \quad \text{y} \quad A' \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

2. Aplicar (1) hasta que no haya dos alternativas con un prefijo común.

⋆ *Ejemplo:* G_5 no está factorizada por la izquierda y G_3 (que ya sabemos es $LL(1)$) es el resultado de aplicar el algoritmo de factorización a G_5:

$$G_5 = \begin{cases} E \to T + E \mid T \\ T \to F * T \mid F \\ F \to \underline{(}E\underline{)} \mid \mathbf{id} \end{cases} \qquad G_3 = \begin{cases} E \to TE' & E' \to +E \mid \epsilon \\ T \to FT' & T' \to *T \mid \epsilon \\ F \to \underline{(}E\underline{)} \mid \mathbf{id} \end{cases}$$

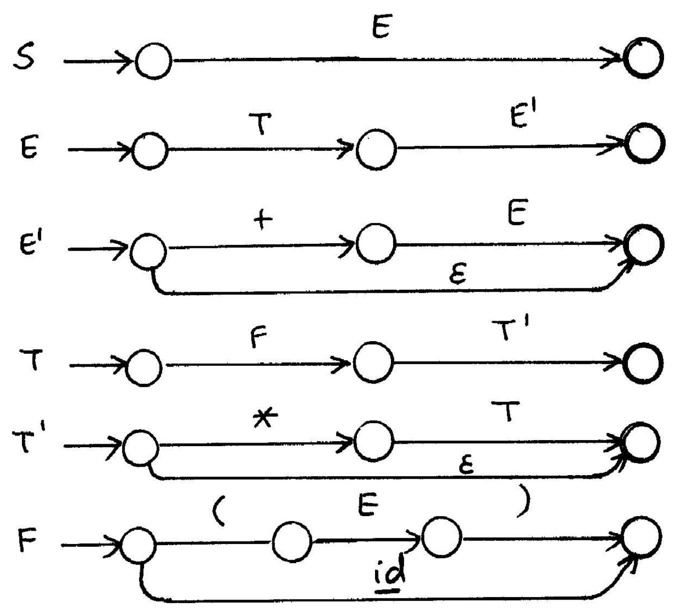
1.5.2. Analizadores recursivos descendentes

⋆ Una vez obtenida una gramática $LL(1)$, otra posibilidad para obtener un analizador descendente es la siguiente:

1. Se crea una función A por cada no-terminal A. Estas funciones podrán llamarse entre sí de forma **mutuamente recursiva**, según se explica a continuación.

2. Para cada regla $A \to \alpha$, la función A sigue en secuencia los símbolos de la parte derecha α:
   - Si encuentra un símbolo $a \in V_T$, comprueba que el siguiente símbolo de la entrada es a. Si lo es, lo consume y continua con el siguiente símbolo de α. Si no lo es, la parte derecha α falla. Si no hay más partes derechas para A, entonces A falla. Si hay más, ensaya otra parte derecha.
   - Si encuentra un símbolo $B \in V_N$, entonces llama a la función B. Si B falla, se procede como en el punto anterior.
3. Si hay una alternativa $A \to \epsilon$, solo se utiliza en caso de fallo de todas las demás.
4. Si hay alternativas que empiezan por terminales y otras que empiezan por no terminales, por eficiencia han de ensayarse antes las primeras.
5. Si la raíz consigue completar alguna de sus partes derechas, la frase es correcta. Si todas las ramas de la raíz fallan, es incorrrecta.

⋆ *Ejemplo:* Los siguientes diagramas para G_3 pueden entenderse como simples diagramas sintácticos al estilo de los que que aparecen en los libros, o como el conjunto de funciones mutuamente recursivas del analizador recursivo descendente:



- En sentido horizontal progresan las llamadas o comprobaciones de símbolos terminales de cada alternativa.
- En sentido vertical se expresa el orden en que se ensayan las mismas, una vez que han fallado las anteriores.

1.6. Analizadores ascendentes $LR(1)$

⋆ Los analizadores ascendentes leen símbolos de la entrada de izquierda a derecha y construyen el árbol sintáctico en sentido **ascendente**, siguiendo una derivación por la derecha en sentido inverso. Su problema consiste en reconocer las **partes derechas** de las reglas:

- si aun no es posible completar ninguna, proceden por **desplazamiento** consumiendo un símbolo de la entrada y almacenándolo en la pila.
- si consiguen completar una, proceden por **reducción** sustituyendo la parte derecha reconocida por el no terminal del que procede.

Se les llama también analizadores por desplazamiento y reducción.

⋆ *Ejemplo:* Sea la siguiente derivación de G_1.

$$S \Rightarrow_{de} E \Rightarrow_{de} T \Rightarrow_{de} T * F \Rightarrow_{de} T * \mathbf{id} \Rightarrow_{de} F * \mathbf{id} \Rightarrow_{de} \mathbf{id} * \mathbf{id}$$

En la tabla se muestran las acciones realizadas para reconocer la frase $\mathbf{id} * \mathbf{id}$, el estado de la pila, y el de la entrada:

Pila	Entrada	Acción
	$\underline{id} * \underline{id} \vdash$	D
$\underline{id}$	$* \underline{id} \vdash$	R
F	$* \underline{id} \vdash$	R
T	$* \underline{id} \vdash$	D
$T *$	$\underline{id} \vdash$	D
$T * \underline{id}$	$\vdash$	R
$T * F$	$\vdash$	R
T	$\vdash$	R
E	$\vdash$	R
S	$\vdash$	

⋆ Si seguimos una derivación por la derecha en sentido inverso, en cada paso se reduce una parte derecha α a un no-terminal A, aplicando una regla $A \to \alpha$. Llamamos **asidero** (*handle*) de una forma de frase derecha (ffd) a la cadena α que hay que reducir. En una gramática no ambigua, el asidero es **único**.

⋆ *Ejemplo:* en la ffd "$T * \mathbf{id}$", el asidero es **id**.

⋆ Dada una forma de frase derecha $\beta\alpha u$ con asidero α, se llama **prefijo viable** a cada prefijo de la cadena $\beta\alpha$, incluida ella misma.

- ⋆ *Ejemplo:* en la ffd "$T * \mathbf{id}$", son prefijos viables T, $T*$ y $T * \mathbf{id}$.
- ⋆ El trabajo del analizador ascendente consiste en reconocer prefijos viables:
  - mientras no ha llegado al asidero, el autómata procede por desplazamiento.
  - al llegar al final del prefijo viable procede por reducción, sustituyendo el asidero por su parte izquierda.
- ⋆ El lenguaje de prefijos viables es **regular**. El siguiente AFN, llamado **autómata finito característico** del autómata de items K_G, reconoce dicho lenguaje:

$$car(K_G) = (It_G, V_T \cup V_N, \Delta_c, [S' \to \bullet S], \{[X \to \alpha\bullet] \mid X \to \alpha \in P\})$$

  - $\Delta_c([X \to \alpha \bullet M\beta], M) = [X \to \alpha M \bullet \beta]$, para toda $X \to \alpha M\beta \in P$.
  - $\Delta_c([X \to \alpha \bullet Y\beta], \epsilon) = [Y \to \bullet \gamma]$, para todas $X \to \alpha Y\beta,\ Y \to \gamma \in P$.
- ⋆ La relación entre el AFN $car(K_G)$ y el autómata de items K_G es:
  - Las transiciones ϵ de $car(K_G)$ corresponden a la regla de expansión de K_G.
  - Las transiciones con $a \in V_T$ corresponden a la regla de desplazamiento de K_G.
  - Cuando $car(K_G)$ llega a un estado final $[Y \to \gamma\bullet]$, K_G utiliza la regla de reducción desapilando el item $[Y \to \gamma\bullet]$ y cambiando el item siguiente $[X \to \alpha \bullet Y\beta]$ por $[X \to \alpha Y \bullet \beta]$.
  - Este último cambio corresponde a una transición de $car(K_G)$ mediante el no terminal Y.
- ⋆ En la Figura 1.4 se muestra el AFN $car(K_G)$ de G_1. Notese el no determinismo de las transiciones ϵ. Debido a ellas, no es sencillo determinar el asidero. Por ejemplo, E es un prefijo viable tanto de la ffd $E + F$ con asidero F, como de la ffd E con asidero E. En el primer caso hay que desplazar y en el segundo hay que reducir.
- ⋆ Dado un prefijo viable $\gamma\alpha$, diremos que el item $[X \to \alpha \bullet \beta]$ es **válido** para $\gamma\alpha$ si existe:

$$S \Rightarrow^*_{de} \gamma X u \Rightarrow_{de} \gamma\alpha\beta u$$

- ⋆ *Teorema:* $([S' \to \bullet S], \psi) \vdash^*_{car(K_G)} (q, \epsilon)$ si y solo si q es un item válido para ψ. Más aún, q es de la forma $[X \to \alpha \bullet \beta]$ y $\psi = \gamma\alpha$. Si además $q \in F$, entonces $\beta = \epsilon$ y α es un asidero.
- ⋆ Dado que $car(K_G)$ es no determinista, en general habrá **varios** items válidos $[X \to \alpha \bullet \beta]$ para un mismo prefijo viable ψ. Cada uno representa un análisis distinto de ψ. Si convertimos $car(K_G)$ en determinista, cada estado del AFD equivalente contendrá **todos** los items válidos para ψ.
- ⋆ El AFD equivalente a $car(K_G)$ se denomina $AFD\text{-}LR(0)_G$:

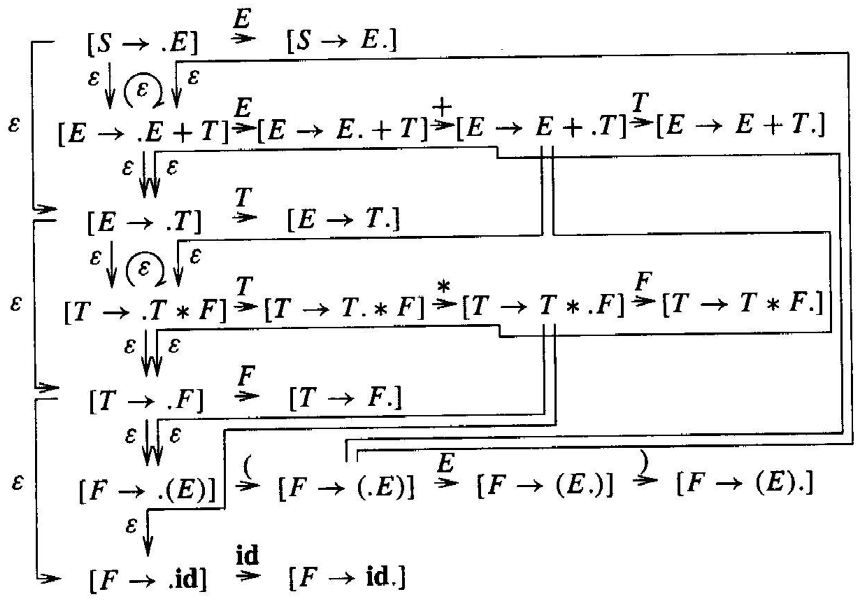


Figura 1.4: Transiciones del AFN $car(K_G)$ de G_1

- Sus estados son **conjuntos de items**.
- El estado inicial contiene el item $[S' \to \bullet\, S]$.
- Los estados finales contienen al menos un item de la forma $[X \to \alpha\bullet]$.

El lenguaje regular reconocido por $AFD\text{-}LR(0)_G$ es el de los prefijos viables de G que terminan en un asidero. Si todos los estados fueran finales, el lenguaje sería el de los prefijos viables de G.

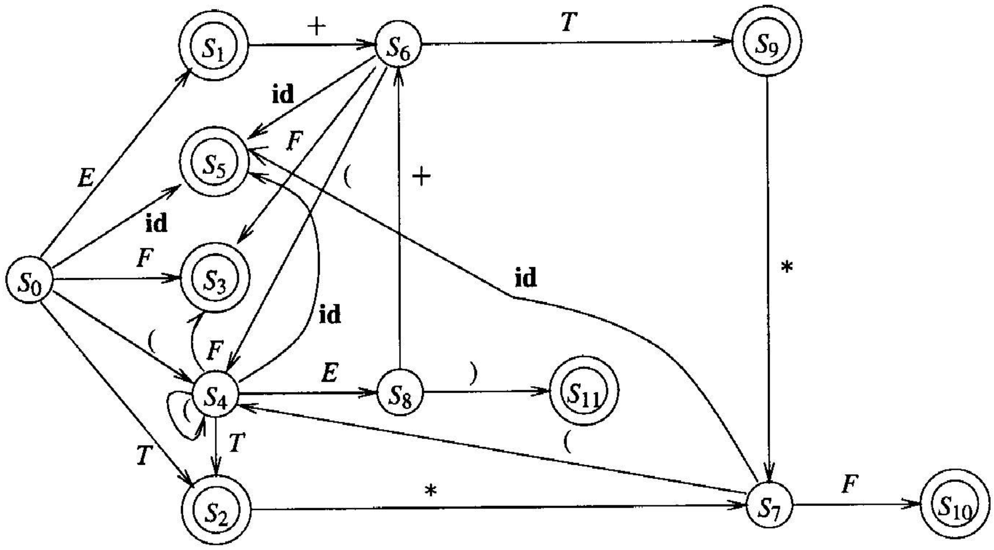
⋆ En la Figura 1.5 se muestra el $AFD\text{-}LR(0)$ de G_1. Los siguientes prefijos viables terminan en el asidero F:

$$\begin{array}{lll} F & (F & ((F\ldots \\ T*(F & T*((F & T*(((F\ldots \\ E+F & E+(F & E+((F\ldots \\ E+(E+(F & E+(E+(E+(F & \ldots \end{array}$$

⋆ El camino seguido para construir el $AFD\text{-}LR(0)_G$ puede esquematizarse como:

$$G \quad -\text{items de } G\to \quad car(K_G) \quad -\text{conversión AFN/AFD}\to \quad AFD\text{-}LR(0)_G$$

En la Figura 1.6 se muestra un algoritmo que permite construir **directamente** el $AFD\text{-}LR(0)_G$ a partir de G.



$$
\begin{aligned}
S_0 = \{\ & [S \to .E], \\
& [E \to .E + T], \\
& [E \to .T], \\
& [T \to .T * F], \\
& [T \to .F], \\
& [F \to .(E)], \\
& [F \to .\mathbf{id}]\}
\end{aligned}
$$

$$
\begin{aligned}
S_1 = \{\ & [S \to E.], \\
& [E \to E. + T]\}
\end{aligned}
$$

$$
\begin{aligned}
S_2 = \{\ & [E \to T.], \\
& [T \to T. * F]\}
\end{aligned}
$$

$$
S_3 = \{\ [T \to F.]\}
$$

$$
\begin{aligned}
S_4 = \{\ & [F \to (.E)], \\
& [E \to .E + T], \\
& [E \to .T], \\
& [T \to .T * F] \\
& [T \to .F] \\
& [F \to .(E)] \\
& [F \to .\mathbf{id}]\}
\end{aligned}
$$

$$
S_5 = \{\ [F \to \mathbf{id}.]\}
$$

$$
\begin{aligned}
S_6 = \{\ & [E \to E + .T], \\
& [T \to .T * F], \\
& [T \to .F], \\
& [F \to .(E)], \\
& [F \to .\mathbf{id}]\}
\end{aligned}
$$

$$
\begin{aligned}
S_7 = \{\ & [T \to T * .F], \\
& [F \to .(E)], \\
& [F \to .\mathbf{id}]\}
\end{aligned}
$$

$$
\begin{aligned}
S_8 = \{\ & [F \to (E.)], \\
& [E \to E. + T]\}
\end{aligned}
$$

$$
\begin{aligned}
S_9 = \{\ & [E \to E + T.], \\
& [T \to T. * F]\}
\end{aligned}
$$

$$
S_{10} = \{\ [T \to T * F.]\}
$$

$$
S_{11} = \{\ [F \to (E).]\}
$$

Figura 1.5: AFD-LR(0) equivalente a $car(K_G)$ de G_1

Algorithm LR-DFA:
Input: context-free grammar $G = (V'_N, V_T, P', S')$.
Output: LR-DFA$(G) = (Q_d, V_N \cup V_T, q_d, \delta_d, F_d)$.
Method: The states and transitions of the LR-DFA(G) are constructed
in steps using the following three auxiliary functions
Start, Closure and *Succ*.

var q, q': **set of item**;
function *Start:* **set of item**;
  return($\{[S' \to .S]\}$);
    (∗ if S' and S are the new and old start symbols of G, respectively ∗)
function *Closure*(s : **set of item**) : **set of item**;
    (∗ corresponds to the ε successor states of Def. 7.7 ∗)
begin
  $q := s$;
  while *exist.* $[X \to \alpha.Y\beta]$ **in** q **and** $Y \to \gamma$ **in** P
      and $[Y \to .\gamma]$ **not in** q **do**
    add $[Y \to .\gamma]$ to q
  od;
  return(q)
end;

function *Succ*(s : **set of item**, Y : $V_N \cup V_T$) : **set of item**;
    (∗ corresponds to the (S) transitions in K_G ∗)
  return($\{[X \to \alpha Y.\beta] \mid [X \to \alpha.Y\beta] \in s\}$);

begin
  $Q_d := \{Closure(Start)\}$;
  $\delta_d := \emptyset$;
  foreach q **in** Q_d **and** X **in** $V_N \cup V_T$ **do**
    let $q' = Closure(Succ(q, X))$ **in**
      if $q' \neq \emptyset$
      then
        if q' **not in** Q_d
        then $Q_d := Q_d \cup \{q'\}$
        fi;
        $\delta_d := \delta_d \cup \{q \xrightarrow{X} q'\}$
      fi
    tel
  od
end

Figura 1.6: Algoritmo para calcular el *AFD-LR*(0) de una gramática

⋆ Al igual que el AFN $car(K_G)$ está asociado al autómata de items K_G, se puede asociar a *AFD-LR*$(0)_G$ un autómata con pila $(V_T, Q, \Delta, q_0, \{q_f\})$ en el que los estados de Q son conjuntos de items. Se le llama **autómata LR**(0) de G.

⋆ Los estados q de *AFD-LR*(0) son muy informativos sobre el tipo de transiciones que puede hacer el autómata con pila $LR(0)$:

- Si en q existe un item $[X \to \alpha \bullet a\beta]$, es posible una transición de **desplazamiento** en Δ con $a \in V_T$.
- Si en q existe un item $[X \to \alpha\bullet]$ es posible una transición de **reducción** en Δ con $X \to \alpha \in P$.
- Si en q existen dos items $[X \to \alpha \bullet a\beta]$ y $[Y \to \gamma\bullet]$ se dice que hay en q un **conflicto desplazamiento-reducción**.
- Si en q existen dos items $[X \to \alpha\bullet]$ y $[Y \to \gamma\bullet]$ se dice hay en q un **conflicto reducción-reducción**.
- En ambos casos los estados se llaman **inadecuados**.
- Si en *AFD-LR*(0) no hay estados inadecuados, el autómata con pila $LR(0)$ es **determinista**.

⋆ *Ejemplo:* En el *AFD-LR*$(0)_{G_1}$ de la Figura 1.5, los estados S_1, S_2 y S_9 son inadecuados.

⋆ Sea G incontextual y *AFD-LR*$(0)_G = (Q, V_T \cup V_N, \delta, q_0, F)$. Definimos el **autómata con pila LR**(0) de G:
$$(V_T, Q, \Delta, q_0, \{q_f\})$$

- $Q \subseteq \mathcal{P}(It_G)$ coincide con el conjunto de estados de *AFD-LR*$(0)_G$. Por tanto, la pila será una pila de conjuntos de items.
- q_0 es el único estado que contiene el item $[S' \to \bullet S]$.
- $q_f = \{[S' \to S\bullet]\}$.
- si existe $[X \to \ldots \bullet a \ldots] \in q$ y $\delta(q, a) = q'$, entonces $(q, a, qq') \in \Delta$. Llamamos de **desplazamiento** a esta transición.
- si existe $[X \to \alpha\bullet] \in q_n$ y $|\,\alpha\,| = n$, entonces
$$(q\,\underbrace{q_1 \cdots q_n}_{|\,\alpha\,|}, \epsilon, qq') \in \Delta$$
donde $\delta(q, X) = q'$. Llamamos de **reducción** a esta transición.

⋆ Si *AFD-LR*$(0)_G$ no tiene estados inadecuados, el autómata $LR(0)$ es **determinista**:

- En las transiciones de desplazamiento, fijados q y a, y debido a que *AFD-LR*$(0)_G$ es determinista, solo hay un estado q' tal que $(q, a, qq') \in \Delta$.

- No son posibles dos reducciones en el mismo estado q, ni una reducción y un desplazamiento, debido a la inexistencia de estados inadecuados en $AFD\text{-}LR(0)_G$.
- La **construcción** del autómata $LR(0)$ es sencilla: una vez tenemos el $AFD\text{-}LR(0)_G$, se asigna un natural único a cada estado $q \in Q$ y se tabula $\delta : Q \times (V_N \cup V_T) \to Q$. Es decir, no es necesario conservar explícitamente los conjuntos de items. Basta con marcar los estados finales, e indicar la regla $X \to \alpha$ que hay que aplicar en la correspondiente reducción.

1.6.1. Gramáticas y autómatas $LR(k)$

⋆ Una gramática incontextual G, extendida si es necesario con $S' \to S$, **es LR(k)**, con $k \geq 0$, si siempre que se cumple

$$\begin{array}{l} S' \Rightarrow^*_{de} \alpha X u \Rightarrow_{de} \alpha\beta u \\ S' \Rightarrow^*_{de} \gamma Y v \Rightarrow_{de} \alpha\beta w \\ k : u = k : w \end{array}$$

también se cumple $v = w$, $X = Y$ y $\alpha = \gamma$. Es decir, en toda ffd, los primeros k símbolos a la derecha del asidero determinan unívocamente la regla utilizada para reducir.

⋆ *Ejemplo 1:* Sea G definida por

$$S \to A \mid B \qquad A \to aAb \mid 0 \qquad B \to aBbb \mid 1$$

que no es $LL(k)$ para ningún k (comprobarlo) y es $LR(0)$. Las ffd's posibles son:

$$\underline{A}, \quad \underline{B}, \quad a^n\underline{aAb}b^n, \quad a^n\underline{aBbb}b^{2n}, \quad a^n\underline{0}b^n, \quad a^n\underline{1}b^{2n}$$

donde el asidero está subrayado. En esta gramática, la forma del asidero determina unívocamente la regla.

⋆ Una gramática G **es LR(0)** si y solo si el autómata $AFD\text{-}LR(0)_G$ **no tiene estados inadecuados**.

⋆ *Ejercicio:* Construir el $AFD\text{-}LR(0)$ del *Ejemplo 1* y comprobar que no tiene estados inadecuados. Analizar la frase $aa0bb$ con el autómata $LR(0)$ correspondiente.

⋆ *Ejemplo 2:* La siguiente gramática es $LR(1)$.

$$S \to aAc \qquad A \to bbA \mid b$$

Sus ffd's son: $\underline{aAc}$, $ab^{2n}\underline{bbA}c$, $ab^{2n}\underline{b}c$. Sea la frase $ab^m w$ donde los símbolos ab^m ya han sido consumidos. Si $1 : w = c$, entonces el asidero es la última b y hay que reducir con $A \to b$. Si $1 : w = b$, entonces hay que desplazar (consumir dicha b) y buscar el asidero más adelante en w.

⋆ *Ejemplo 3:* La siguiente gramática no es $LR(k)$ para ningún k.

$$S \rightarrow aAc \qquad A \rightarrow bAb \mid b$$

Elegimos un k y las dos siguientes derivaciones con $n \geq k$:

$$S \Rightarrow^*_{de} ab^nAb^nc \Rightarrow_{de} ab^n\underline{b}|b^nc$$
$$S \Rightarrow^*_{de} ab^{n+1}Ab^{n+1}c \Rightarrow_{de} ab^{n+1}|\underline{b}b^{n+1}c$$

La linea vertical marca el límite de los símbolos consumidos. En el primer caso hay que reducir con $A \rightarrow b$ y en el segundo hay que desplazar un símbolo más, pero no es posible determinar la elección mirando k símbolos a la derecha.

⋆ Cuando el *AFD-LR*(0) tiene estados inadecuados, todavía es posible construir un analizador **determinista** si se le permite inspeccionar sin consumirlos hasta k símbolos anticipados de la entrada. La idea consiste en dotar a los items de una información adicional llamada **anticipo** (*lookahead*) que permite discriminar entre dos reducciones, o entre un desplazamiento y una reducción, cuando ambas son posibles en el mismo estado.

⋆ Dada G y $k > 0$, **un item LR(k)** es una tupla $[X \rightarrow \alpha_1 \bullet \alpha_2, L]$, con $(X \rightarrow \alpha_1\alpha_2) \in P$ y $L \subseteq (V_T \cup \{\vdash\})^+$ un conjunto de cadenas posiblemente terminadas por $\vdash$ y cada una a lo sumo de longitud k. Si $k = 1$, entonces L es simplemente un conjunto de símbolos que puede incluir $\vdash$. La parte $X \rightarrow \alpha_1 \bullet \alpha_2$ se llama **núcleo** y la parte L, **anticipo**.

⋆ Dado un prefijo viable $\gamma\alpha_1$, diremos que el item $[X \rightarrow \alpha_1 \bullet \alpha_2, L]$ es **válido** para $\gamma\alpha_1$ si para toda $w \in L$ existe

$$S \Rightarrow^*_{de} \gamma X u \Rightarrow_{de} \gamma\alpha_1\alpha_2 u$$

con $w = k : (u \vdash)$.

⋆ **Una gramática** G **es LR(k)** si para todo prefijo viable $\gamma\alpha_1$ de G e item $LR(k)$ $[X \rightarrow \alpha_1\bullet, L_1]$ válido para $\gamma\alpha_1$, si existe otro item $LR(k)$ $[Y \rightarrow \alpha_2 \bullet \alpha_3, L_2]$ que también es válido para $\gamma\alpha_1$, entonces se cumple $L_1 \cap \mathit{prim}_k(\alpha_3 L_2) = \emptyset$.

⋆ Dada G, el algoritmo de la Figura 1.7 **construye** un autómata finito determinista, que llamaremos $\mathbf{AFD\text{-}LR(k)_G}$, y que es muy similar al *AFD-LR*(0) excepto por el hecho de que los items son $LR(k)$. La expresión ϵ-*ffi*(α) ha de entenderse como $\mathit{prim}_k(\alpha)$.

⋆ *Teorema:* El *AFD-LR*$(k)_G$ reconoce el lenguaje de prefijos viables de G y además cada estado contiene **todos** los items $LR(k)$ válidos para los prefijos viables que conducen a él.

⋆ Un estado q del *AFD-LR*$(k)_G$ se dice **adecuado** si:

```
var q, q': set of item;
var Q: set of  set of item;
var δ: set of item × (V_N ∪ V_T) → set of item;
function Start: set of item;
    return ({[S' → .S, {#}]});
function Closure(q : set of item) : set of item;
begin
    foreach [X → α.Yβ, L] in q and Y → γ inP do
        if exists [Y → .γ, L'] in q
        then replace [Y → .γ, L'] by [Y → .γ, L' ∪ ε-ffi(βL)]
        else q := q ∪ {[Y → .γ, ε-ffi(βL)]}
        fi
    od;
    return(q)
end;
function Succ(q : set of item, Y : V_N ∪ V_T) : set of item;
    return({[X → αY.β, L] | [X → α.Yβ, L] ∈ q})
begin
    Q := {Closure(Start)};  δ := ∅;
    foreach q in Q and X in V_N ∪ V_T do
        let q' = Closure(Succ(q, X)) in
            if q' ≠ ∅
            then
                if q' not in Q
                then Q := Q ∪ {q'}
                fi;
                δ := δ ∪ {q --X--> q'}
            fi
        tel
    od
end.
```

Figura 1.7: Algoritmo para calcular el $AFD\text{-}LR(k)$ de una gramática

- siempre que $[X \to \alpha\bullet, L_1], [Y \to \beta\bullet, L_2] \in q$. se cumple $L_1 \cap L_2 = \emptyset$.
- y siempre que $[X \to \alpha\bullet, L_1], [Y \to \beta\bullet\gamma, L_2] \in q$, se cumple $L_1 \cap prim_k(\gamma L_2) = \emptyset$.

En caso contrario se dice **inadecuado**.

⋆ Si aplicamos dicho algoritmo con $k = 1$ a G_1, obtenemos el $AFD\text{-}LR(1)_G$ de la Figura 1.8. Puede obervarse que tiene más estados que el $AFD\text{-}LR(0)_G$ correspondiente, aunque hay algunos estados que sólo se diferencian en los anticipos de los items.

⋆ El el caso del $AFD\text{-}LR(1)_G$ la definición de estado q **adecuado** puede simplificarse:

- siempre que $[X \to \alpha\bullet, L_1], [Y \to \beta\bullet, L_2] \in q$, se cumple $L_1 \cap L_2 = \emptyset$.
- y siempre que $[X \to \alpha\bullet, L_1], [Y \to \beta \bullet a\gamma, L_2] \in q$, se cumple $a \notin L_1$.

⋆ *Teorema:* G es $LR(k)$ si y solo si el autómata $AFD\text{-}LR(k)_G$ no contiene estados inadecuados.

⋆ *Ejemplo:* En la Figura 1.8, los estados 10, 14, 15, 20 y 21 son potencialmente conflictivos porque contienen a la vez items de reducción e items de desplazamiento. Nótese sin embargo que todos ellos son adecuados. Concluimos por tanto que G_1 es $LR(1)$.

⋆ Un **analizador** $LR(k)$ es un autómata con pila, que es **determinista** si la gramática es $LR(k)$, muy similar al autómata $LR(0)$ descrito más arriba. Supongamos que δ es la función de transición del $AFD\text{-}LR(k)_G$.

- Si en el estado q hay un item $[X \to \alpha\bullet, L_1]$ y los siguientes k símbolos de la entrada forman una cadena en L_1, entonces realiza una **reducción** con $X \to \alpha$ y pasa al estado $q' = \delta(q'', X)$, donde q'' es el estado que queda en la cima tras desapilar $\mid \alpha \mid$ estados.
- Si en el estado q hay un item $[Y \to \beta\bullet a\gamma, L_2]$ y los siguientes k símbolos de la entrada forman un cadena en $prim_k(a\gamma L_2)$, entonces realiza un **desplazamiento** consumiendo el primer símbolo a de la entrada y pasa al estado $q' = \delta(q, a)$.

⋆ El algoritmo completo se muestra en la Figura 1.9.

- La función *goto* es la δ del $AFD\text{-}LR(k)_G$.
- las funciones *push* y *pop* manejan la pila del autómata.
- la **tabla de accciones** *action* contiene para cada estado q y anticipo w la acción a realizar: desplazar, reducir, terminar cuando el estado de llegada es $\{[S' \to S\bullet]\}$, o reportar error.
- La acción *output* genera la lista de reglas utilizadas para construir el árbol sintáctico de la frase analizada.

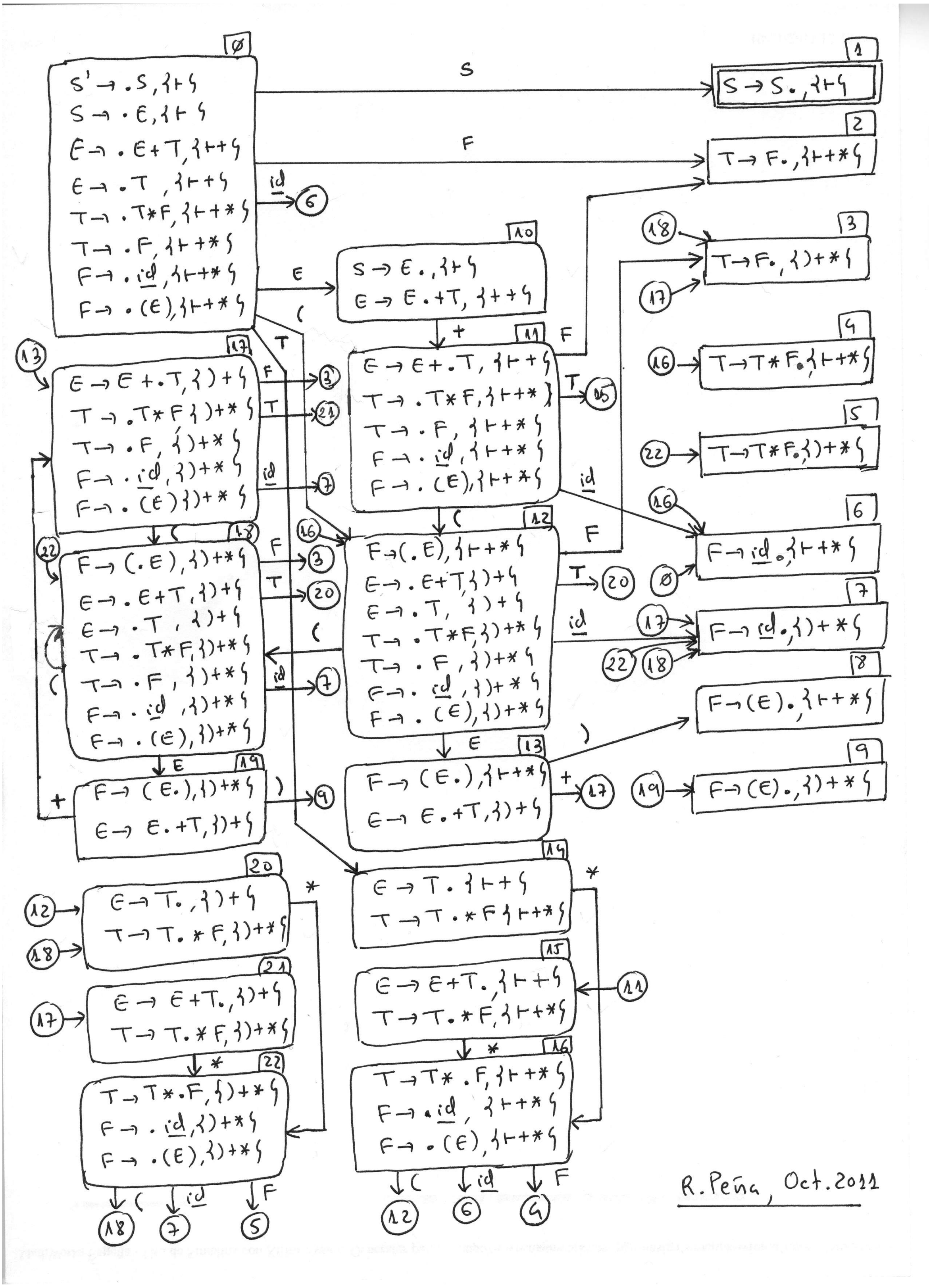


Figura 1.8: Autómata *AFD-LR*(1) de G_1

```
Algorithm LR(k)-PARSER:

type   state = set of item;
var    lookahead: seq of symbol;
              (* the next k lexically analysed,
                         but not yet consumed input symbols *)
       S : stack of state;
proc scan;
       (* analyse another symbol lexically,
         append it after lookahead *)
proc acc;
       (* report successful end of the syntax analysis; stop *)
proc err(message: string);
       (* report error; stop *)
scan^k;
push(S, q0);
forever do
    case action[top(S), lookahead] of
       shift: begin    push(S, goto(top(S), hd(lookahead)));
                       lookahead := tl(lookahead);
                       scan
              end;
       reduce (X → α): begin  pop^|α|(S); push(S, goto(top(S), X));
                              output("X → α")
                       end;
       accept:  acc;
       error:   err("...");
    end case
od
```

Figura 1.9: Algoritmo de un analizador $LR(k)$

1.6.2. Métodos simplificados $SLR(1)$ y $LALR(1)$

- ⋆ En la práctica, el autómata $AFD\text{-}LR(1)_G$ de un lenguaje de programación real puede tener varios miles de estados. Muchos de estos estados tienen items con los mismos núcleos y distintos anticipos. Estos estados se llaman **homólogos**.

- ⋆ *Ejemplo:* En el autómata $AFD\text{-}LR(1)$ de la Figura 1.8 los siguientes pares de estados son homólogos:

$2-3$	$18-12$
$4-5$	$19-13$
$6-7$	$20-14$
$8-9$	$21-15$
$17-11$	$23-16$

- ⋆ Los métodos $SLR(1)$ y $LALR(1)$ utilizan como punto de partida el autómata finito $AFD\text{-}LR(0)_G$, cuyos estados serán en general muchos menos que los del $AFD\text{-}LR(1)_G$. Se basan en **añadir un anticipo** a cada item de reducción. Nótese que los anticipos de los items de desplazamiento del $AFD\text{-}LR(1)_G$) en realidad no se usan. Se diferencian en la forma de calcular dicho anticipo.

- ⋆ En el método $SLR(1)$ (*Simple* $LR(1)$), cada item $[X \to \alpha\bullet]$ en un estado del $AFD\text{-}LR(0)_G$ se transforma a $[X \to \alpha\bullet, sig(X)]$.

- ⋆ *Ejemplo:* Recordemos que en el autómata $AFD\text{-}LR(0)$ de la Figura 1.5, los estados S_1, S_2 y S_9 eran inadecuados. Recordemos que el cálculo de $sig(X)$ realizado en la Sección 1.3 para G_1 daba como resultado:

$$sig(S) = \{\vdash\} \qquad sig(E) = \{\vdash,), +\}$$

El método $SLR(1)$ genera entonces los siguientes estados con anticipos:

S_1	S_2	S_9
$[S \to E\bullet,\ \{\vdash\}]$	$[E \to T\bullet,\ \{\vdash,), +\}]$	$[E \to E + T\bullet,\ \{\vdash,), +\}]$
$[E \to E \bullet +T]$	$[T \to T \bullet *F]$	$[T \to T \bullet *F]$

Los tres son adecuados en sentido $LR(1)$. Decimos que G_1 **es SLR(1)**.

- ⋆ En el método $LALR(1)$ (*Look Ahead* $LR(1)$), cada item $[X \to \alpha\bullet]$ en un estado del $AFD\text{-}LR(0)_G$ se transforma en $[X \to \alpha\bullet, \bigcup_i L_i]$, donde los L_i son los anticipos de los items con núcleo $X \to \alpha\bullet$ de los estados del autómata $AFD\text{-}LR(1)_G$, homólogos entre sí, que contienen dicho item. Estos items tendrán pues la forma $[X \to \alpha\bullet, L_i]$. Una forma sencilla para calcular el autómata $LALR(1)$ es calcular primero el $AFD\text{-}LR(1)_G$ y luego “fundir” sus estados homólogos, uniendo los anticipos de los items de reducción que tengan el mismo núcleo. Es posible, no obstante, aplicar métodos más refinados que eviten el cákculo explícito de $AFD\text{-}LR(1)_G$.

- ⋆ *Ejemplo:* En el autómata *AFD-LR*(1) de la Figura 1.8 los estados 10, 14, 15, 20, 21 son potencialmente conflictivos, de los cuales los pares $(14, 20)$ y $(15, 21)$ son homólogos. El estado 10 es equivalente al S_1 del método $SLR(1)$; al fundir los estados 14 y 20 resulta un estado equivalente al S_2 del método $SLR(1)$; y al fundir 15 y 21 resulta uno equivalente a S_9. En este ejemplo, el método $LALR(1)$ genera el mismo autómata que el $SLR(1)$. Decimos que G_1 también **es LALR(1)**.

- ⋆ En general se cumple:

$$\text{Gramáticas } SLR(1) \quad \subset \quad \text{Gramáticas } LALR(1) \quad \subset \quad \text{Gramáticas } LR(1)$$

1.6.3. Construcción eficiente del automáta LALR(1)

Una forma eficiente (y simple) de construir el autómata LALR(1), que evita la construcción explícita de *AFD-LR*$(1)_G$, es amplicar el algoritmo de construcción de *AFD-LR*$(0)_G$ para incluir información en los ítems que permita calcular sus anticipos una vez finalizada dicha construcción. Para ello:

- Los estados se identifican unívocamente (mediante un número natural). Los ítems dentro de un estado también se identifican unívocamente (mediante otro número natural). De esta forma, es posible referir unívocamente a ítems dentro de estados. Para ello se utilizará la notación $\#_{i,j}$: referencia al ítem i en el estado j.

- En el estado inicial, en lugar de $[S' \to \bullet S]$ se incluye $[0, S' \to \bullet S, \{\vdash\}]$.

- Si en el estado i el ítem $[k, B \to \bullet \gamma, \Gamma]$ se ha generado por cierre a partir del ítem $[j, A \to \alpha \bullet B\beta, \Delta]$, entonces $\Gamma \supseteq prim_1(\beta) \oplus_1 \#_{i,j}$ (es decir, deberá añadirse a los anticipos de $[B \to \bullet \gamma]$ los $prim_1(\beta) - \{\epsilon\}$, y, en caso de que β sea anulable, una referencia al ítem $[A \to \alpha \bullet B\beta]$ en el estado i).

- De forma análoga, si el ítem $[n, A \to \alpha X \bullet \beta, \Gamma]$ en el estado k se genera a partir del ítem $[j, A \to \alpha \bullet X\beta, \Delta]$ en el estado i, entonces $\Gamma \supseteq \{\#_{i,j}\}$ (es decir, deberá añadirse a los anticipos de $[A \to \alpha X \bullet \beta]$ en el estado k una referencia al ítem $[A \to \alpha \bullet X\beta]$ en el estado i).

De esta forma, los ítems pueden ir "acumulando" referencias a aquellos ítems que los generan, si ello es necesario (nótese que un mismo ítem puede ser generado de más de una manera). Una vez que termina la construcción del autómata *AFD-LR*$(0)_G$ ampliado para incluir la información sobre los anticipos, los anticipos en sí pueden calcularse planteando un sistema de ecuaciones en el que las referencias sean las incógnitas. Más concretamente, para cada ítem j en cada estado i, dicho sistema contendrá una ecuación de la forma $\#_{i,j} = \Theta \cup \#_{i_0,j_0} \ldots \cup \#_{i_{n(i,j)},j_{n(i,j)}}$, donde:

- Θ es la parte constante de la información de anticipo del ítem.

- $\#_{i_0,j_0} \ldots \#_{i_{n(i,j)},j_{n(i,j)}}$ son las referencias incluidas en dicha información de anticipo.

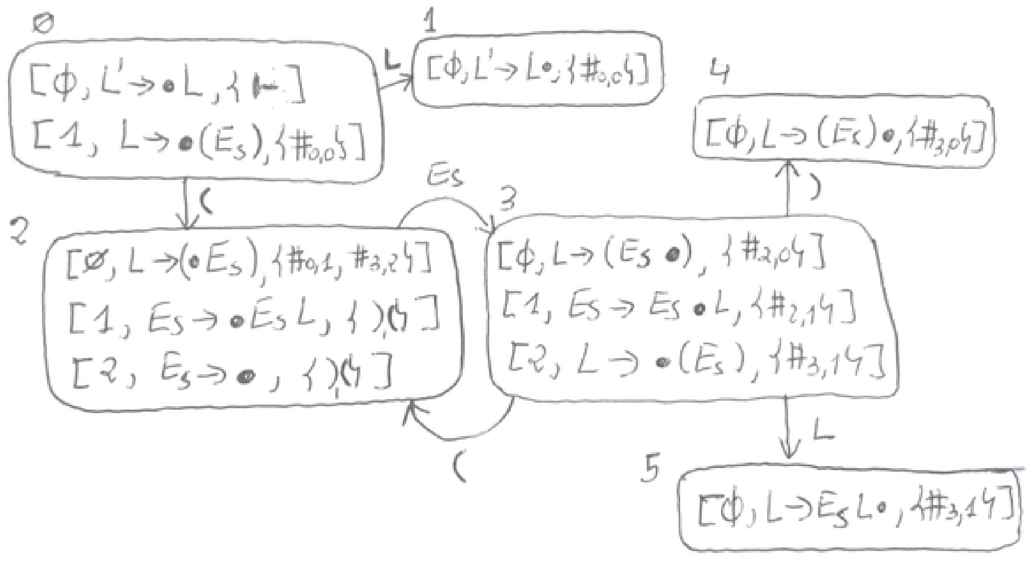
Una vez planteado dicho sistema de ecuaciones, los anticipos de cada ítem pueden determinarse resolviendo el mismo mediante un algoritmo de punto fijo similar al usado para el cálculo de $prim_k$ o sig_k.

Ejemplo

Considérese la gramática

$$\begin{aligned} L' &\longrightarrow L \\ L &\longrightarrow (Es) \\ Es &\longrightarrow Es\ L|\epsilon \end{aligned}$$

El *AFD-LR*(0) ampliado con información de anticipo para dicha gramática es el siguiente:



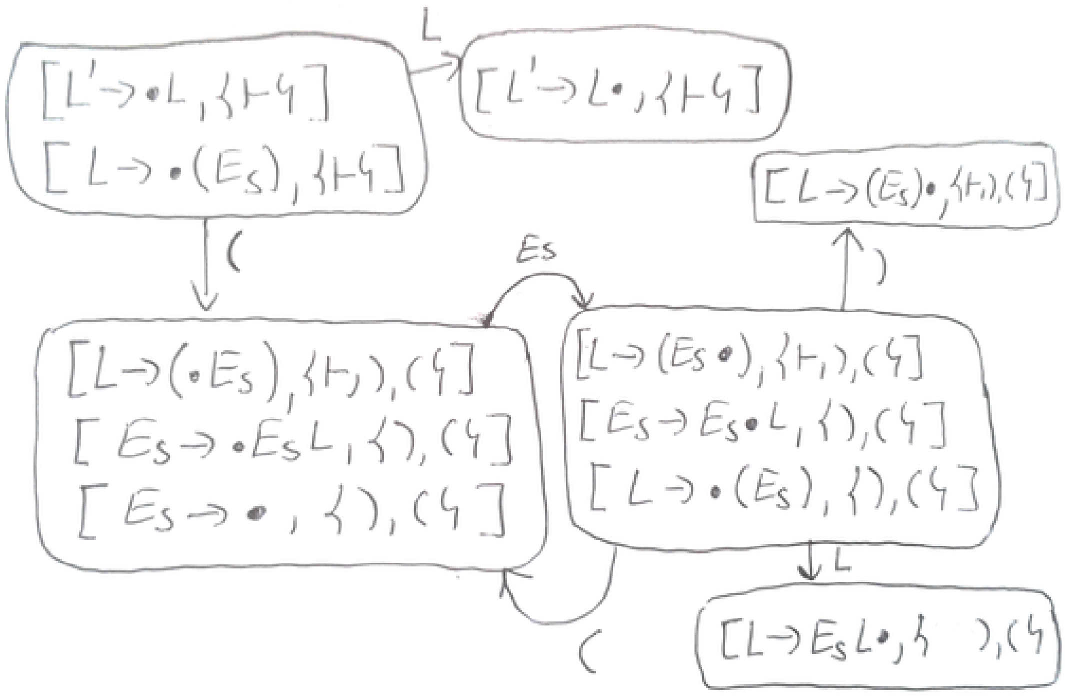
(en dicha construcción es necesario saber que $prim_1(L) = \{(\}$, hecho que puede constatarse directamente en la gramática). De esta forma, para determinar los anticipos de cada ítem debe resolverse el siguiente sistema de ecuaciones:

$$\#_{0,0} = \{\vdash\}$$
$$\#_{0,1} = \#_{0,0}$$
$$\#_{1,0} = \#_{0,0}$$
$$\#_{2,0} = \#_{0,1} \cup \#_{3,2}$$
$$\#_{2,1} = \{),(\}$$
$$\#_{2,2} = \{),(\}$$
$$\#_{3,0} = \#_{2,0}$$
$$\#_{3,1} = \#_{2,1}$$
$$\#_{3,2} = \#_{3,1}$$
$$\#_{4,0} = \#_{3,0}$$
$$\#_{5,0} = \#_{3,1}$$

Este sistema de ecuaciones puede resolverse de manera iterativa mediante un sencillo algoritmo de punto fijo:

	IT. 0	IT 1	IT 2	IT 3	IT 4	IT 5	IT 6	IT 7
$\#_{0,0}$	$\emptyset$	$\{\vdash\}$	$\{\vdash\}$	$\{\vdash\}$	$\{\vdash\}$	$\{\vdash\}$	$\{\vdash\}$	$\{\vdash\}$
$\#_{0,1}$	$\emptyset$	$\emptyset$	$\{\vdash\}$	$\{\vdash\}$	$\{\vdash\}$	$\{\vdash\}$	$\{\vdash\}$	$\{\vdash\}$
$\#_{1,0}$	$\emptyset$	$\emptyset$	$\{\vdash\}$	$\{\vdash\}$	$\{\vdash\}$	$\{\vdash\}$	$\{\vdash\}$	$\{\vdash\}$
$\#_{2,0}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{\vdash\}$	$\{\vdash,),(\}$	$\{\vdash,),(\}$	$\{\vdash,),(\}$	$\{\vdash,),(\}$
$\#_{2,1}$	$\emptyset$	$\{),(\}$	$\{),(\}$	$\{),(\}$	$\{),(\}$	$\{),(\}$	$\{),(\}$	$\{),(\}$
$\#_{2,2}$	$\emptyset$	$\{),(\}$	$\{),(\}$	$\{),(\}$	$\{),(\}$	$\{),(\}$	$\{),(\}$	$\{),(\}$
$\#_{3,0}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{\vdash\}$	$\{\vdash,),(\}$	$\{\vdash,),(\}$	$\{\vdash,),(\}$
$\#_{3,1}$	$\emptyset$	$\emptyset$	$\{),(\}$	$\{),(\}$	$\{),(\}$	$\{),(\}$	$\{),(\}$	$\{),(\}$
$\#_{3,2}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{),(\}$	$\{),(\}$	$\{),(\}$	$\{),(\}$	$\{),(\}$
$\#_{4,0}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{\vdash\}$	$\{\vdash,),(\}$	$\{\vdash,),(\}$
$\#_{5,0}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{),(\}$	$\{),(\}$	$\{),(\}$	$\{),(\}$	$\{),(\}$

La solución de dicho sistema de ecuaciones determina directamente los anticipos de los elementos del autómata $LALR(1)$:



Notas bibliográficas

Se debe ampliar el contenido de estas notas en (Scott, 2009, Capítulo 2) y en (Wilhelm y Maurer, 1995, Capítulo 8), en los cuales están parcialmente basadas.

Ejercicios

1. Dada la siguiente gramática incontextual,

$$S \longrightarrow SS+ \mid SS* \mid a$$

   Escribir, si existe, el árbol sintáctico de la cadena `aa+a*`. Explicar verbalmente el lenguaje generado por la gramática.

2. Dada la gramática,

$$\begin{array}{rcl} \textit{bexpr} & \longrightarrow & \textit{bexpr}\ \textbf{or}\ \textit{bterm} \mid \textit{bterm} \\ \textit{bterm} & \longrightarrow & \textit{bterm}\ \textbf{and}\ \textit{bfactor} \mid \textit{bfactor} \\ \textit{bfactor} & \longrightarrow & \textbf{not}\ \textit{bfactor} \mid (\textit{bexpr}) \mid \textbf{true} \mid \textbf{false} \end{array}$$

   Construir una derivación por la izquierda para la frase **not (true or false)**. ¿Genera esta gramática todas las expresiones booleanas? ¿Es ambigua?

3. Calcular las funciones *prim* y *sig* para la siguiente gramática:

$$\begin{array}{lcllcl} S' & \longrightarrow & S & S & \longrightarrow & LB \\ B & \longrightarrow & ;S;L \mid := L & E & \longrightarrow & a \mid L \\ J & \longrightarrow & ,EJ \mid) & L & \longrightarrow & (EJ \end{array}$$

4. Calcular las funciones *prim* y *sig* para la siguiente gramática:

$$\begin{array}{lcl} S & \longrightarrow & AaAb \mid BbBa \\ A & \longrightarrow & \epsilon \\ B & \longrightarrow & \epsilon \end{array}$$

5. Calcular las funciones *prim* y *sig* para la siguiente gramática:

$$\begin{array}{lcl} S & \longrightarrow & Aa \mid bAc \mid dc \mid bda \\ A & \longrightarrow & d \end{array}$$

6. Escribir las ecuaciones recursivas que definen la *productividad* de los símbolos no terminales de la siguiente gramática, y ejecutar manualmente un algoritmo de punto fijo para determinar qué símbolos son no productivos.

$$\begin{array}{l} S \rightarrow aX \\ X \rightarrow bS \mid aYbY \\ Y \rightarrow ba \mid aZ \\ Z \rightarrow aZX \end{array}$$

7. Escribir las ecuaciones recursivas que definen la *alcanzabilidad* de los símbolos no terminales de la siguiente gramática, y ejecutar manualmente un algoritmo de punto fijo para determinar qué símbolos son no alcanzables.

$$\begin{array}{ll} S \rightarrow Y & Y \rightarrow YZ \mid Ya \mid b \\ U \rightarrow V & X \rightarrow c \\ V \rightarrow Vd \mid d & Z \rightarrow ZX \end{array}$$

8. Comprobar si la siguiente gramática es $LL(1)$:

$$\begin{array}{lcllcl} S & \longrightarrow & AB \mid BC & A & \longrightarrow & BA \mid a \\ B & \longrightarrow & CC \mid b & C & \longrightarrow & AB \mid a \end{array}$$

9. Decir si la gramática del Ejercicio 2 es $LL(1)$. Si no lo fuera, transformarla a una equivalente que lo sea.

10. En la gramática transformada del Ejercicio 9 escribir los pasos seguidos por el analizador $LL(1)$ para reconocer la frase **not (true or false)**.

11. Dada la siguiente gramática incontextual

$$\begin{array}{lcllcl} S & \longrightarrow & A \mid F \mid (\, L\,) & A & \longrightarrow & a \\ F & \longrightarrow & a\,(\, L\,) & L & \longrightarrow & S \mid L\,,S \end{array}$$

   a) Explicar informalmente el lenguaje que genera.
   b) Transformarla en una gramática $LL(1)$ equivalente.

12. Dada la gramática,

$$\begin{array}{lcl} S & \longrightarrow & (L) \mid a \\ L & \longrightarrow & L,S \mid S \end{array}$$

   a) Eliminar la recursión por la izquierda y comprobar que la gramática resultante es $LL(1)$.
   b) Construir un analizador $LL(1)$.
   c) Escribir los pasos seguidos por el analizador $LL(1)$ para reconocer la frase $(a,(a,a),a)$.

13. Contruir el autómata de items de la siguiente gramática G:

$$S \longrightarrow a \mid \textbf{if}\ b\ \textbf{then}\ S \mid \textbf{if}\ b\ \textbf{then}\ S\ \textbf{else}\ S$$

   a) Dar una derivación de aceptación para la frase **if** b **then** **if** b **then** a **else** a.
   b) Mostrar que G es ambigua.
   c) Dar una grammática no ambigua equivalente.

14. Mostrar que la siguiente gramática es SLR (1) y dar la tabla de acciones

$$\begin{array}{lcllcl} S & \longrightarrow & E & E & \longrightarrow & E+T \mid T \\ T & \longrightarrow & T*P \mid P & P & \longrightarrow & F \uparrow P \mid F \\ F & \longrightarrow & \textbf{id} \mid (E) & & & \end{array}$$

15. Mostrar que la siguiente gramática es LL(1), pero no es SLR(1):

$$\begin{array}{lcl} S & \longrightarrow & AaAb \mid BbBa \\ A & \longrightarrow & \epsilon \\ B & \longrightarrow & \epsilon \end{array}$$

16. Mostrar que la siguiente gramática es LALR(1), pero no es SLR(1):

$$\begin{array}{lcl} S & \longrightarrow & Aa \mid bAc \mid dc \mid bda \\ A & \longrightarrow & d \end{array}$$

17. Dada la siguiente gramática G incontextual:

$$\begin{array}{lllllll} S & \longrightarrow & A & \quad & A & \longrightarrow & b\,B \mid a \\ B & \longrightarrow & c\,C \mid c\,C\,e & \quad & C & \longrightarrow & d\,A \end{array}$$

   a) Calcular el $AFD\text{-}LR(0)$.
   b) Decir si G es $SLR\,(1)$.
   c) Decir si G es $LR\,(1)$, calculando el autómata correspondiente.
   d) Decir si G es $LALR\,(1)$, calculando el autómata correspondiente.

18. Mostrar que la siguiente gramática es LR(1), pero no es LALR(1):

$$\begin{array}{lll} S & \longrightarrow & Aa \mid bAc \mid Bc \mid bBa \\ A & \longrightarrow & d \\ B & \longrightarrow & d \end{array}$$

19. Dada la siguiente gramática G incontextual:

$$\begin{array}{lllllll} S & \longrightarrow & A & \quad & A & \longrightarrow & b\,B \mid a \\ B & \longrightarrow & c\,C \mid c\,C\,e & \quad & C & \longrightarrow & d\,A \end{array}$$

   a) Calcular $AFD\text{-}AFD\text{-}LR(G)$.
   b) Decir si G es $SLR\,(1)$.
   c) Decir si G es $LR\,(1)$, calculando el autómata correspondiente.
   d) Decir si G es $LALR\,(1)$, calculando el autómata correspondiente.

20. Mostrar que la siguiente gramática es LR(1), pero no es LALR(1):

$$\begin{array}{lll} S & \longrightarrow & Aa \mid bAc \mid Bc \mid bBa \\ A & \longrightarrow & d \\ B & \longrightarrow & d \end{array}$$

Bibliografía

Durante mucho tiempo se creyó que esos libros impenetrables correspondían a lenguas pretéritas o remotas.

Jorge Luis Borges

Scott, M. L. *Programming Language Pragmatics*. 3rd edition. Morgan Kaufmann, 2009.

Wilhelm, R. y Maurer, D. *Compiler Design*. Addison-Wesley, 1995.