Procesadores de Lenguajes I:

Una introducción a la fase de análisis

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

Capítulo 3

Análisis sintáctico descendente

En este capítulo se realiza la presentación de las técnicas de análisis sintáctico empleadas en la construcción de un compilador. Tras una breve introducción al análisis sintáctico descendente y ascendente, se presentan los conceptos fundamentales que permiten la utilización del formalismo de los autómatas de pila para la especificación del analizador sintáctico. El resto del capítulo está dedicado a mostrar la técnica de análisis sintáctico descendente predictiva LL y los métodos de recuperación de errores.

3.1. Conceptos fundamentales

3.1.1. Especificación sintáctica de los lenguajes de programación

Mientras que para la especificación de las unidades léxicas de un lenguaje se requiere un formalismo como las Expresiones Regulares, en el caso de la especificación de las estructuras sintácticas de un lenguaje de programación se necesita también un formalismo. El formalismo que utilizaremos para este último propósito es el de las Gramáticas Independientes del Contexto (GIC).

Podemos apuntar diferentes propiedades de este formalismo que justifican su utilización como herramienta de especificación sintáctica de los lenguajes de programación:

1. Es precisa y fácil de entender.

2. Para algunos lenguajes hay algoritmos de análisis eficientes basados en este formalismo.

3. La ampliación de nuevas estructuras del lenguaje es sencilla.

Una ***Gramática Independiente del Contexto (o Gramática Incontextual)*** se define como: $G = (N, \Sigma, P, S)$, donde N representa el conjunto de símbolos no-terminales, Σ representa el conjunto de símbolos terminales, P el conjunto de reglas de producción donde cada regla $(A \rightarrow \alpha)$

$\in$ P es tal que A $\in$ N y $\alpha \in (N \cup \Sigma)^*$ **(**siendo $N \cap \Sigma = \varnothing$); y S es el símbolo inicial de la gramática tal que S $\in$ N.

Ejemplo 3.1.- G=({E, O}, {+, -, *, /, (,), id }, P, E) donde P es:

$E \rightarrow E\ O\ E \mid (\ E\ ) \mid -\ E \mid id$

$O \rightarrow + \mid - \mid * \mid /$

Estas reglas de producción pueden verse como reglas de reescritura. Así, la regla de producción $O \rightarrow +$ **,** representa que se reemplaza una instancia de O por +.

Respecto a la aplicación de las reglas de producción aparecen los siguientes conceptos:

Derivación Directa. Representa la aplicación de una regla de producción.

$$\delta\ A\ \gamma \rightarrow \delta\ \beta\ \gamma \quad \underline{sii} \quad \exists\ (A \rightarrow \beta) \in P; \qquad \delta, \gamma \in (N \cup \Sigma)^*$$

Derivación. Representa la aplicación a una cadena de 0 o más derivaciones directas.

$$\alpha \rightarrow^* \beta \qquad \underline{sii} \qquad \exists\ \alpha_0, \ldots, \alpha_m \in (N \cup \Sigma)^*\ ;\ \alpha_0 = \alpha \rightarrow \alpha_1 \rightarrow \ldots\ \alpha_{m-1} \rightarrow \alpha_m = \beta$$

Forma Sentencial. Es una cadena que puede derivarse a partir del símbolo inicial de la gramática:

$$\alpha \in (N \cup \Sigma)^*\ \underline{sii} \qquad \exists\ S \rightarrow^* \alpha$$

Sentencia. Es una cadena formada por terminales que puede derivarse a partir del símbolo inicial de la gramática:

$$\alpha \in \Sigma^* \qquad \underline{sii} \qquad \exists\ S \rightarrow^* \alpha$$

Lenguaje Generado por una gramática G. Lenguaje formado por el conjunto de cadenas de símbolos terminales (o sentencias) que pueden derivarse a partir del símbolo inicial de la gramática.

$$L(G) = \{x / x \in \Sigma^*: \quad S \rightarrow^* x\}$$

Ejemplo 3.2.- Dadas las siguientes reglas de producción de una gramática:

$$S \rightarrow a\ b \ \mid \ a\ S\ b,$$

podemos obtener distintas formas sentenciales por sucesivas derivaciones como

$$S \rightarrow a\ S\ b \rightarrow a\ a\ S\ b\ b \rightarrow a\ a\ a\ b\ b\ b$$

donde la última cadena (a a a b b b) es una *sentencia* por estar formada únicamente por símbolos terminales.

3.1.2. El papel del analizador sintáctico

El objetivo fundamental del analizador sintáctico es construir el árbol sintáctico conocidas las hojas (terminales) y el símbolo inicial de la gramática.

Dada una gramática $G = (N, \Sigma, P, S)$, un árbol sintáctico, también denominado *árbol de derivación*, se define de la siguiente forma:

1. La raíz está etiquetada con el símbolo inicial S.
2. Cada hoja está etiquetada con un símbolo de $\Sigma \cup \{ \varepsilon \}$.
3. Cada nodo interior está etiquetado con un símbolo de N.
4. Sea $A \in N$ la etiqueta de un nodo, y $x_1,...,x_n$ las etiquetas de sus nodos hijos (de izquierda a derecha). Entonces: $A \rightarrow x_1 \ldots x_n \in P$

A la cadena obtenida por concatenación de las hojas del árbol de derivación se le denomina *frontera*.

Si identificamos cada regla de la gramática por un número, el ***análisis*** o "***parse***" de una cadena consiste en una secuencia de números resultante de la numeración sucesiva de cada una de las reglas utilizadas en el proceso de derivación, representando internamente el árbol de derivación,

Si en el proceso de derivación se sustituye siempre el símbolo no-terminal que se encuentra más a la izquierda de la forma sentencial decimos que se ha realizado una *derivación a izquierdas*. Si se sustituye siempre el no-terminal más a la derecha, decimos que se ha realizado una *derivación a derechas*.

Ejemplo 3.3.- Dadas las siguientes reglas de producción de una gramática:

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

y siendo S el símbolo inicial de ésta, analizaremos la cadena a * (a + b) siguiendo las dos estrategias antes mencionadas, derivación a izquierdas (DI) y derivación a derechas (DD):

DI: $S \rightarrow^2 T \rightarrow^3 T * F \rightarrow^4 F * F \rightarrow^6 a * F \rightarrow^5 a * (S) \rightarrow^1 a * (S + T)$

$\rightarrow^2 a * (T + T) \rightarrow^4 a * (F + T) \rightarrow^6 a * (a + T) \rightarrow^4 a * (a + F) \rightarrow^7 a * (a + b)$

siendo por tanto el "parse" izquierdo = 2-3-4-6-5-1-2-4-6-4-7

DD: $S \rightarrow^2 T \rightarrow^3 T * F \rightarrow^5 T * (S) \rightarrow^1 T * (S + T) \rightarrow^4 T * (S + F) \rightarrow^7 T * (S + b)$

$\rightarrow^2 T * (T + b) \rightarrow^4 T * (F + b) \rightarrow^6 T * (a + b) \rightarrow^4 F * (a + b) \rightarrow^6 a * (a + b)$

siendo el "parse" derecho = 6-4-6-4-2-7-4-1-5-3-2. En ambos casos, como puede verse en la Fig. 3.1 el árbol de derivación es el mismo aunque se ha obtenido por métodos de análisis distintos.

Teorema.

Un árbol tiene un único "parse" izquierdo y un único "parse" derecho.

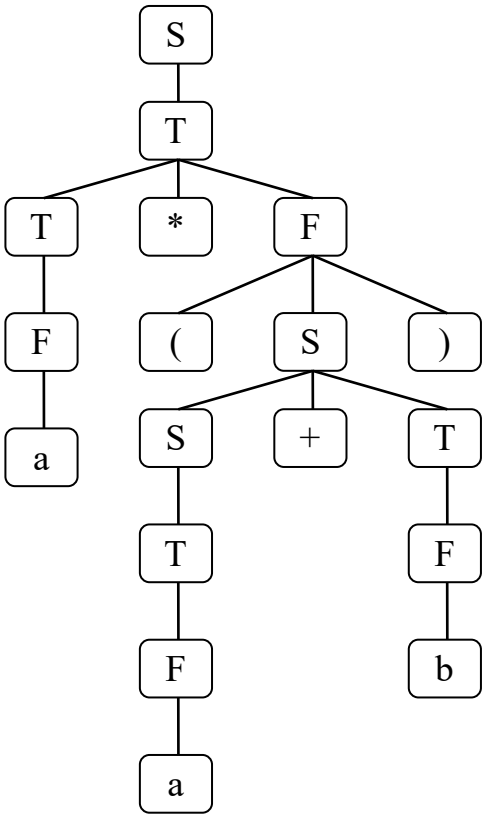


Figura 3.1. Arbol sintáctico correspondiente a a*(a+b).

Teorema.

Dada G(S), $S \rightarrow^* \alpha$ (con $\alpha \in \Sigma^*$) si y solo si existe un árbol de derivación que produce α.

Gramática Ambigua. Una gramática es ambigua si y solo si existe una cadena $x \in \Sigma^*$ / $x \in L(G)$ para la cual hay más de un árbol de derivación que produce x. En la Fig. 3.2 puede verse un ejemplo de gramática ambigua para una instrucción condicional if-then-else.

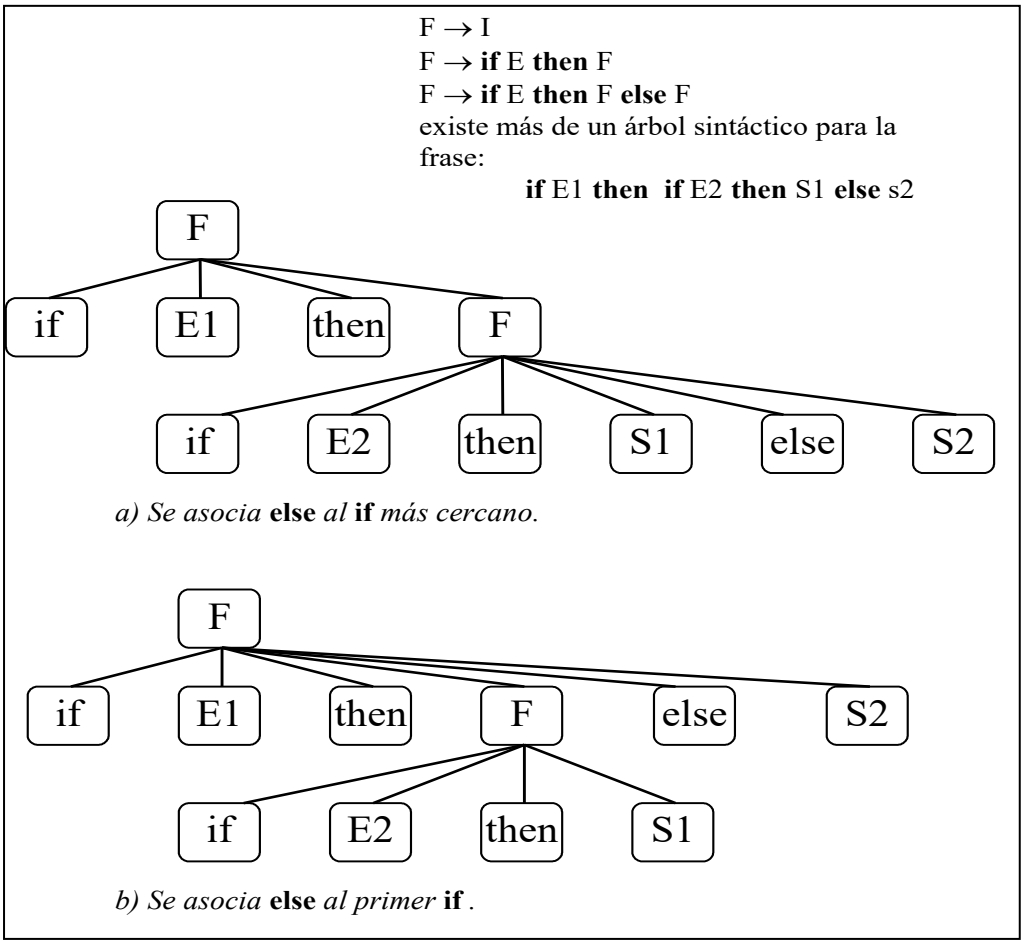


Figura 3.2. Gramática Ambigua.

Proposición.

Determinar si una gramática es ambigua, es un problema indecidible.

Es decir, no existe un algoritmo que determine en un tiempo finito si una gramática es ambigua o no. De la misma manera, tampoco existe un algoritmo general que produzca una gramática no ambigua a partir de una gramática ambigua, pero en algunos casos se puede transformar la gramática para eliminar la ambigüedad[1].

Una gramática ambigua crea problemas para la traducción de los programas ya que distintos árboles sintácticos pueden tener distintas interpretaciones semánticas.

[1] De hecho hay lenguajes intrínsicamente ambiguos, es decir, lenguajes para los que no existe una gramática no ambigua.

Ejemplo 3.4.- La gramática de la Fig. 3.2 se puede transformar de tal manera que produciendo el mismo lenguaje deja de ser ambigua.

$$F \rightarrow F1 \mid F2$$
$$F2 \rightarrow \text{if E then F2 else F2} \mid I$$
$$F1 \rightarrow \text{if E then F} \mid \text{if E then F2 else F1}$$

3.1.3. Aceptores: Reconocedores y Traductores

Una máquina reconocedora o autómata permite definir un lenguaje por medio de la vía de aceptación de las cadenas que lo componen.

Las máquinas reconocedoras deciden si una cadena pertenece o no a un lenguaje, en contraposición con las gramáticas formales que generan las cadenas a partir de las reglas de producción.

En la Fig. 1.1 del capítulo 1 se presentaron los distintos tipos de reconocedores junto a la fase de análisis en la que se utilizan cada uno de ellos. En el análisis sintáctico se utilizan los autómatas a pila como reconocedores de lenguajes independientes del contexto.

Autómata a Pila

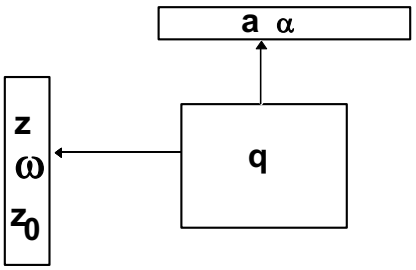
Un autómata a pila se define como:

$$A_p = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$$

Donde Q representa un conjunto finito de estados, Σ representa el alfabeto de entrada, Γ representa el alfabeto de la pila, δ es la función de transición tal que

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow P(Q \times \Gamma^*)$$, q_0 es el estado inicial tal que $q_0 \in Q$, z_0 representa el símbolo inicial de la pila, y F es el conjunto de estados finales: $F \subset Q$.

- ***Estructura***:



Configuración.

$(q, a\ \alpha,\ z\ \omega\ z_0)$

En esta configuración podemos ver que q representa el estado actual del autómata, α la entrada de la cadena que queda por analizar, y z es el símbolo que se encuentra en la cima de la pila.

- ***Movimiento***.

$(q, a\ \alpha, z\ \omega\ z_0) \vdash (q', \alpha, \beta\ \omega\ z_0)$ es un movimiento válido sii $\exists\ \delta(q, a, z) = (q', \beta)$

con $q,q' \in Q$; $a \in \Sigma \cup \{\varepsilon\}$; $\alpha \in \Sigma^*$; $z \in \Gamma$; $\beta, \omega \in \Gamma^*$

- ***Lenguaje Aceptado a Pila Vacía***

$$L_v(A_p) = \{x / x \in \Sigma^* : (q_0, x, z_0) \vdash^* (q, \varepsilon, \varepsilon)\}$$

- ***Lenguaje Aceptado a Estado Final***

$$L_F(A_p) = \{x / x \in \Sigma^* : (q_0, x, z_0) \vdash^* (q, \varepsilon, a);\quad q \in F\}$$

Si una máquina reconocedora es además capaz de emitir cadenas de algún alfabeto determinado, entonces a esta máquina se le denomina máquina traductora. Así, si un Autómata a Pila es capaz de emitir cadenas de un determinado alfabeto, éste se denominará *Traductor a pila*.

3.1.4. Introducción general al análisis sintáctico

En la construcción del árbol de derivación se puede seguir una estrategia de análisis *ascendente* o *descendente*.

El ***análisis descendente*** puede verse como una derivación a izquierdas. En términos generales, podemos decir que las operaciones en este análisis consisten en realizar sucesivas *derivaciones* hasta que el símbolo de la cima de la pila corresponda con el primer símbolo que hay en la cadena a analizar, en cuyo caso se *consume* este símbolo de la cadena y se *extrae* el correspondiente de la *cima de la pila*. Equivale a construir el árbol de derivación de la cadena a analizar desde la raíz a las hojas.

El ***análisis ascendente*** puede verse como una derivación a derechas recorrida al revés. En la derivación a derechas del Ejemplo 3.3 vimos que pasamos de la forma sentencial T * F a T aplicando al revés la regla de producción 3. En este caso se dice que T * F se ha *reducido*. En términos generales, en el análisis ascendente las operaciones que se realizan son: *reducir* por la parte izquierda de una determinada regla de producción, si la cadena que se encuentra en la cima de la pila coincide con la parte derecha de dicha regla, o bien, si no es posible la reducción entonces *desplazar* a la pila un símbolo de la cadena a analizar. Esta técnica es equivalente a construir el árbol de derivación de la cadena a analizar desde las hojas a la raíz.

Durante el proceso de análisis sintáctico, puede presentarse el problema del indeterminismo. Por ejemplo, el ***indeterminismo*** se nos plantea cuando en la GIC se presentan varias reglas para un mismo símbolo auxiliar como $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots . \mid \alpha_n$. Podemos enfrentarnos a este problema aplicando distintos tipos de estrategias:

- Algoritmos basados en backtracking (o vuelta atrás).

  Es una estrategia muy general que sirve para cualquier gramática independiente del contexto, pero no son los más adecuados para analizar los lenguajes de programación por su complejidad exponencial. La sección 4.1 de [Aho 72] está dedicada a discutir este tipo de técnicas.

- Algoritmos generales o tabulares, como el algoritmo de Early o el de Cocke-Younger-Kasami.

  Tienen una complejidad más baja $O(n^3)$, que puede llegar a ser de $O(n^2)$ en el algoritmo de Early si la gramática no es ambigua. Sin embargo este coste sigue siendo muy elevado si tenemos en cuenta que el tamaño del problema viene determinado por el número de símbolos que aparecen en el programa fuente a analizar. En la sección 4.2 de [Aho 72] se pueden consultar estos dos algoritmos.

- Algoritmos deterministas.

  Algoritmos descendentes y ascendentes deterministas que se caracterizan por encontrar una única alternativa a aplicar dada una determinada cadena de entrada y un no-terminal. Consecuentemente tendrán un coste O(n). En este último tipo de algoritmos es en el que nos vamos a centrar en los próximos apartados y en el siguiente capítulo.

Lamentablemente, estos últimos métodos solo son apropiados para un subconjunto de GIC. Por esta razón, para poder realizar un análisis determinista de coste lineal nos centraremos en un subconjunto de gramáticas incontextuales: aquellas en las que desde cada configuración sólo hay un posible movimiento adecuado.

3.2. Análisis sintáctico descendente

Un analizador LL(1) analiza la cadena de izquierda a derecha (Left-to-right), siguiendo un proceso de derivación a izquierdas (Leftmost-derivation). Para decidir la regla de producción a utilizar es suficiente con examinar sólo un símbolo (cadena de símbolos terminales de longitud 1) de la cadena de entrada.

Ejemplo 3.5.- En la siguiente gramática podemos saber qué regla aplicar para el auxiliar S (la número 2) en el análisis de la cadena "bd" observando el primer símbolo de la cadena ("b"),

$$S \rightarrow a\ B\ B \qquad (1)$$
$$S \rightarrow b\ E \qquad (2)$$
$$B \rightarrow c \qquad (3)$$
$$E \rightarrow d \qquad (4)$$

Para construir un analizador sintáctico descendente predictivo LL(1), nos apoyaremos en dos funciones: Primeros y Siguientes.

El dominio de la función Primeros es: $(N \cup \Sigma)^* \rightarrow P(\Sigma \cup \{\varepsilon\})$ donde primeros de una cadena α de auxiliares y terminales se define como:

Primeros (α) = $\{ a \in \Sigma \mid \alpha \rightarrow^* a\,\beta \} \cup \{ \varepsilon \mid \alpha \rightarrow^* \varepsilon \}$

$$\text{con} \quad \alpha, \beta \in (N \cup \Sigma)^*$$

Se puede observar que se puede calcular Primeros de cadenas de terminales y no-terminales de cualquier tamaño, y el resultado será un conjunto de símbolos terminales, (incluyendo en algunos casos la cadena vacía). El algoritmo 3.1 muestra una forma de calcular el conjunto de elementos pertenecientes a Primeros de una cadena x.

```
Función Primeros (x ∈ (N ∪ Σ)*): Conjunto de (Σ∪{ε})
Dada  G = (N, Σ, P, S); con   C: Conjunto de (Σ∪{ε})

Comienzo
      C := ∅;
      Si x ∈ (Σ ∪ ε) ent C := {x}

      Si x ∈ N ent
            Para toda (x → α) ∈ P hacer C := C ∪ Primeros(α)

      Si x = x1 x2 ... xm  ∧ m > 1  ent
            i := 1;

            Mientras (i < m) ∧ (ε ∈ Primeros(xi)) hacer
                  C := C ∪ (Primeros(xi) - {ε});
                  i := i + 1;
            C := C ∪ (Primeros(xi) - {ε});
            Si (i = m) ∧ (ε ∈ Primeros(xm)) entonces  C := C ∪ {ε}

      Devolver    C
Fin
```

Algoritmo 3.1. Función Primeros.

El algoritmo 3.1 calcula primeros de una cadena x en un conjunto C.

a) Si se trata de un símbolo terminal o de la cadena vacía, C será igual al mismo símbolo terminal o la cadena vacía.

b) Si x es un símbolo no-terminal, C será el conjunto de Primeros de los lados derechos de las producciones del símbolo no-terminal x.

c) Finalmente, si x es una cadena, Primeros(x) incluirá a Primeros del primer símbolo de la cadena x. Si la cadena vacía pertenece a Primeros de este primer símbolo de la cadena x_1, no se incluirá en C, pero se añadirá a C el conjunto de Primeros del siguiente símbolo de la cadena x_2. Se puede observar en el algoritmo que debe hacerse lo mismo con el resto de símbolos de la cadena x hasta que Primeros del símbolo que se está considerando (x_i) no incluya a la cadena vacía, o se haya llegado al final de la cadena. De esta manera, la cadena vacía pertenecerá a Primeros de una cadena x si y solo sí la cadena vacía pertenece a Primeros de todos los símbolos que forman la cadena x.

Ejemplo 3.6.- Calcular el conjunto Primeros de los símbolos no-terminales de la gramática

$E \rightarrow T\ E'$

$E' \rightarrow + T\ E' \mid \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow * F\ T' \mid \varepsilon$

$F \rightarrow (\ E\ ) \mid a$

Primeros (E) = Primeros (T) – {ε} = Primeros (F) – {ε} = { (, a }

Primeros (E') = { +, ε } Primeros (T') = {*, ε }

La otra función que se usa para construir un analizador LL(1) es la función Siguientes. El dominio de la función Siguientes es: $N \longrightarrow P\ (\Sigma \cup \{\$\})$ donde Siguientes de un auxiliar A se define como:

Siguientes(A) = $\{ a \in \Sigma \mid S \rightarrow^* \alpha\ A\ a\ \beta \} \cup \{ \$ \mid S \rightarrow^* \alpha\ A \}$

con $\alpha, \beta \in (N \cup \Sigma)^*; A \in N$

El algoritmo 3.2 muestra una forma de calcular el conjunto de elementos pertenecientes a Siguientes de los símbolos no-terminales de una gramática.

```
Función Siguientes:  Conjunto de (Σ ∪ {$})
Dada G = (N, Σ, P, S) ;  con C = vector [N] de conjunto de (Σ ∪ {$})

Comienzo
      Para todo A ∈ N hacer  C [A] = ϕ
      C [S] = { $ }

      Para toda ( B → γ )  ∈ P hacer
            Para todo A ∈ N    ∧   γ = αAβ  hacer
                  C [A] = C [A] ∪ (Primeros(β) - { ε })

      Repetir
            Para toda ( B → γ )  ∈ P hacer
                  Para todo A ∈ N   ∧  γ = αAβ  ∧  β →* ε hacer
                        C [A] = C [A] ∪ C [B]
      hasta que C [A] no se modifique para ningún A ∈ N
```

```
      Para toda A ∈ N hacer Siguientes (A) := C [A]
      Devolver Siguientes
Fin
```

Algoritmo 3.2. Función Siguientes.

El algoritmo 3.2 va calculando Siguientes de los no-terminales de la gramática en un array indexado por los símbolos no-terminales: C. Inicialmente los conjuntos de Siguientes para todos los símbolos de la gramática están vacios (excepto para el símbolo inicial de la gramática, ya que se le incluye un símbolo especial $). Si en el lado derecho de una producción ($B \rightarrow \alpha A \beta$) aparece un no-terminal A seguido de una cadena β, se añaden los Primeros de β a los siguientes del no-terminal A[2]. Si Primeros de dicha cadena β incluye a la cadena vacía, se añaden al conjunto C[A] los Siguientes del símbolo no-terminal del lado izquierdo de la producción (B).

El conjunto Siguientes de los símbolos no-terminales de la gramática del Ejemplo 3.6 sería:

Siguientes (E') = Siguientes (E) = { $,) }

Siguientes (T') = Siguientes (T) = { +, $,) }

Siguientes (F) = { *, +, $,) }

3.2.1. Condición LL(1)

Es posible construir un analizador sintáctico descendente predictivo usando un símbolo de anticipación para cualquier GIC que cumpla la siguiente condición, denominada Condición LL(1):

Definición

Una gramática independiente del contexto es LL(1), si para cualquier par de producciones de un mismo auxiliar A ($A \rightarrow \alpha \ \wedge \ A \rightarrow \beta$) se cumple la condición:

Primeros (α Siguientes (A)) $\cap$ Primeros (β Siguientes (A)) = $\varnothing$

Para comprobar si una gramática cumple la condición LL(1) se comprueba si la cumple para todos sus no-terminales. Dado un no-terminal A, se calcula Primeros del lado derecho de cada una de sus producciones. Si alguno de estos conjuntos incluye a la cadena vacía, se eliminará ésta, y en su lugar se añadirá Siguientes del no-terminal A. Si ninguno de los conjuntos calculados de esta forma para los lados derechos de las producciones de A incluye ningún elemento incluido

[2] Notad que no se incluye la cadena vacía porque ésta nunca pertenece al conjunto de Siguientes.

en otro de los conjuntos (es decir, si los conjuntos son disjuntos tomados dos a dos), la gramática cumple la condición LL(1) para ese no-terminal (A). La gramática cumplirá la condición LL(1) (también decimos que la gramática es LL(1)) si cumple la condición para todos sus no-terminales.

Ejemplo 3.7- La gramática del Ejemplo 3.6 es LL(1) ya que cumple la condición LL(1):

Primeros (+TE' Siguientes (E')) $\cap$ Primeros (ε Siguientes (E')) = {+} $\cap$ { $,) } = $\varnothing$

Primeros (*FT' Siguientes (T')) $\cap$ Primeros (ε Siguientes (T')) = {*} $\cap$ { +, $,) } = $\varnothing$

Primeros ((E) Siguientes (F)) $\cap$ Primeros (a Siguientes (F)) = {(} $\cap$ { a } = $\varnothing$

3.2.2. Transformaciones de las gramáticas

Factorización:

Cuando en una GIC, cualquier par de producciones de un mismo no-terminal comienzan por el mismo prefijo α la gramática no puede ser LL(1):

$$A \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2 \mid \ldots\ldots \mid \alpha\ \beta_n \mid \gamma$$

Una solución podría ser factorizar, transformando la gramática en

$$A \rightarrow \alpha\ A' \mid \gamma$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots\ldots \mid \beta_n$$ donde A' es un nuevo no-terminal.

Ejemplo 3.8.- Dadas las siguientes reglas de producción de una gramática,

$$I \rightarrow a\ B \mid a\ B\ c \mid d$$
$$B \rightarrow e$$

se puede observar que no es LL(1),

Primeros(aB Siguientes(I))= {a}

Primeros(aBc Siguientes(I))= {a}

$\{a\} \cap \{a\} \neq \varnothing$

Si factorizamos por la izquierda esta gramática se convierte en,

$$I \rightarrow a\ B\ I' \mid d$$
$$I' \rightarrow c \mid \varepsilon$$
$$B \rightarrow e$$

que si que es LL(1):

Primeros(aBI' Siguientes(I))= {a}

Primeros(d Siguientes(I))= {d}	$\{a\} \cap \{d\} = \varnothing$
Primeros(c Siguientes(I'))= {c} Primeros(ε Siguientes(I'))= {$}	$\{c\} \cap \{\$\} = \varnothing$

Eliminación de la recursión a izquierdas:

Definición

Una GIC es recursiva a izquierdas sii $\exists A \in N: A \rightarrow^* A \alpha$.

Teorema

Una GIC LL(1) no es ambigua ni recursiva a izquierdas.

Según este teorema, una gramática que sea recursiva a izquierdas ($\exists A \in N: A \rightarrow^* A \alpha$) no puede ser LL(1). No obstante, frecuentemente se puede realizar un análisis descendente predictivo, transformando la gramática en otra equivalente que genere el mismo lenguaje.

Un caso particular de gramática con recursión a la izquierda es el de las gramáticas con recursión a izquierdas inmediata. Una GIC tiene recursión a izquierdas inmediata si y solo si $\exists$ una producción de la forma $A \rightarrow A \alpha \in P$.

En general, las producciones de un auxiliar con recursión a izquierdas inmediata son de la forma:

$$A \rightarrow A \alpha_1 \mid A \alpha_2 \mid \ldots. \mid A \alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots.. \mid \beta_p$$

donde los β_i no comienzan por A , $\forall i$.

Una solución para eliminar la recursión a izquierdas inmediata es transformar las reglas genéricas anteriores en,

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots.. \mid \beta_p A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$$

donde A' es un nuevo no-terminal.

En la sección 4.4 de [Aho 90] aparece un algoritmo (algoritmo 4.1) para transformar una gramática con recursión a izquierdas no inmediata en otra sin recursión.

Ejemplo 3.9.- Dadas las siguientes reglas de producción de una gramática recursiva a izquierdas,

$$E \rightarrow E + T \mid T$$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid a$

Se puede eliminar la recursividad a izquierdas inmediata aplicando la transformación vista anteriormente, obteniendo una gramática equivalente:

$E \rightarrow T\ E'$

$E' \rightarrow + T\ E' \mid \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow * F\ T' \mid \varepsilon$

$F \rightarrow (E) \mid a$

Ejemplo 3.10.- Dadas las siguientes reglas de producción de una gramática en la que se observa el problema de la ambigüedad que introduce el *else*,

$I \rightarrow$ if E then I | if E then I else I | a

$E \rightarrow b$

Si factorizamos por la izquierda esta gramática se convierte en,

$I \rightarrow$ if E then I I' | a

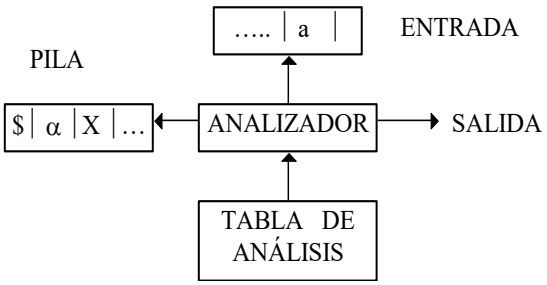
$I' \rightarrow$ else I | ε

$E \rightarrow b$

Es evidente, que a pesar de la transformación la gramática, no es LL(1), ya que sigue siendo ambigua: para I' no se sabrá que alternativa elegir cuando la entrada sea *else.* Una solución a este problema se vio en el Ejemplo 3.4.

3.2.3. Modelo de un analizador sintáctico descendente predictivo.

***Estructura**:*



***Configuración**:*

(x, β, π) siendo $x \in \Sigma^*$; $\beta \in (N \cup \Sigma \cup \{\$\})^*$; $\pi \in \{1, 2, \ldots, N\}^*$

donde "x " representa la cadena de entrada que queda por analizar, "β" representa el contenido de la pila, y "π " la cadena formada por la concatenación de los números de las reglas que se han utilizado en las sucesivas derivaciones durante el proceso de análisis (parse izquierdo).

Algoritmo de Análisis:

El algoritmo de análisis LL(1) es el que se muestra en 3.4. Como se observa hace uso de una tabla de análisis que previamente debe construirse según el algoritmo 3.3.

```
ALGORITMO   Construcción de la T.A. LL(1)
ENTRADA     G = (N, Σ, P, S);
SALIDA      TA:(N ∪ Σ ∪ {$})×(Σ ∪ {$})⟶{(r: A → β),sacar,aceptar,error}
METODO
1) Inicializar TA con la acción "error";
2) para toda (r: A → β) ∈ P hacer
      para todo a ∈ (Σ ∪ {$}): a ∈ PRIMEROS (β SIGUIENTES(A)) hacer
            TA [A, a] := (r: A → β);
3) para todo a ∈ Σ hacer   TA [a, a] := sacar;
4) TA [$, $] := aceptar;
FIN
```

Algoritmo 3.3. Construcción de la Tabla de análisis.

Una tabla de análisis LL(1) tendrá una fila por cada símbolo de la gramática, y una columna por cada terminal y otra para el símbolo $. Para completar la fila del símbolo no-terminal A se calcula, para cada una de las producciones de A (A → β), Primeros(β Siguientes (A). Para cada uno de los terminales resultantes, se añadirá en su columna la acción correspondiente a aplicar la producción A → β.

Teorema

Una gramática es LL(1) si y solo si el algoritmo de construcción de la tabla de análisis no genera ningún conflicto, es decir, si ∀ X, Y / X ∈ V ∧ Y ∈ V se cumple que | TA[X, Y] | = 1.

Ejemplo 3.11.- La Tabla de análisis correspondiente a la gramática del Ejemplo 3.9, siguiendo el algoritmo 3.3 es la siguiente:

	a	**+**	*****	**(**	**)**	**$**
E	(TE',1)			(TE',1)		
E'		(+TE',2)			(ε,3)	(ε,3)
T	(FT',4)			(FT',4)		
T'		(ε,6)	(*FT',5)		(ε,6)	(ε,6)

F	(a,8)			((E),7)		
a	sacar					
+		sacar				
*			sacar			
(				sacar		
)					sacar	
$						aceptar

```
ALGORITMO   A.S.D: basado en la T.A.  LL(1)
ENTRADA     ω ∈ Σ*; TA, para una G = (N, Σ, P, S);
SALIDA      Si ω ∈ L(G) entonces π else error( )
METODO

apilar (S$);   sim = obtsym;    π := ε; fin = falso;
repetir
      caso  TA [cima,sim] sea
            "(r: A → β)": desapilar;   apilar(β);  π := π · r;
            "sacar":      desapilar;   sim := obtsym;
            "aceptar":    fin := verdad;
            "error":      error( );
      end;
hasta fin

FIN
```

Algoritmo 3.4. Algoritmo de análisis LL(1).

Como se puede apreciar en el Algoritmo 3.4, las entradas que pueden aparecer en una tabla de análisis LL(1) son:

r:A $\rightarrow$ β que consiste en aplicar la producción indicada. Para ello se desapila de la cima del analizador el no-terminal A, y se apilan los símbolos que forman el lado derecho de la producción (β).

Sacar: que consiste en quitar de la cima de la pila un terminal, y pasar a considerar el siguiente símbolo de la cadena de entrada.

Aceptar: Fin del análisis, habiendo reconocido la cadena de la entrada.

Error (normalmente indicado por una celda vacía): La cadena de la entrada no pertenece al lenguaje reconocido por el autómata LL(1), o lo que es lo mismo, no puede ser generada por la gramática usada para construir la tabla de análisis.

Aplicando el algoritmo de análisis a la cadena ‘a + a $’, tomando la tabla de análisis del Ejemplo 3.11, obtenemos la traza del análisis LL(1):

(a + a $, E $, ε) ⊢ (a + a $, T E' $, 1)
⊢ (a + a $, F T’ E' $, 14)
⊢ (a + a $, a T’ E' $, 148)
⊢ (+ a $, T’ E' $, 148)
⊢ (+ a $, E' $, 1486)
⊢ (+ a $, + T E' $, 14862)
⊢ (a $, T E' $, 14862)
⊢ (a $, F T’ E' $, 148624)
⊢ (a $, a T’ E' $, 1486248)
⊢ ($, T’ E' $, 1486248)
⊢ ($, E' $, 14862486)
⊢ ($, $, 148624863)

3.2.4. Recuperación de errores en el análisis sintáctico predictivo

Durante el análisis sintáctico descendente predictivo se detecta un error cuando el primer símbolo *a* de la cadena que queda por analizar no concuerda con el símbolo terminal *b* que se encuentra en la cima de la pila, o bien, cuando en la cima de la pila tenemos un símbolo no-terminal A y en la tabla de análisis para la entrada TA [*A, a*] tenemos *error* (o blanco).

Cuando un compilador que emplea un analizador LL(1) detecta un error, informa del error y, normalmente se “recupera” del error para continuar con el análisis sintáctico y mostrar el rsto de errores sintácticos del programa fuente analizado. A este proceso se de denomina “*recuperación del error sintáctico*”. Una posible técnica de recuperación de error es la denominada ***recuperación en modo pánico*** que consiste en saltarse símbolos de la cadena de entrada hasta que aparezca un símbolo que pertenezca a un conjunto especial de símbolos terminales denominada conjunto de ***símbolos de sincronización***.

Como conjunto de símbolos de sincronización de un no-terminal *A* deben seleccionarse aquellos que permitan al analizador sintáctico recuperarse de los errores y así continuar el análisis. Hay varias formas de definir el conjunto de símbolos de sincronización. Por ejemplo, podemos definir como conjunto de símbolos de sincronización para el no-terminal A al conjunto de Primeros(A). De esta manera, si durante el análisis está el no-terminal A en la cima de la pila, el terminal x en la cadena de entrada, y en la celda [A,x] aparece la acción error (celda en blanco), se saltarán (quitarán) símbolos de la cadena de entrada hasta encontrar uno que pertenezca a

Sinc(A), definido como Primeros(A). En ese momento se continuará el análisis de la cadena con normalidad en busca de otros errores sintácticos.

De igual manera se puede definir como conjuntos símbolos de sincronización de A (Sinc(A)) al conjunto de Siguientes(A). En este caso, cuando durante el proceso de sincronización se termine de saltar símbolos porque se ha alcanzado uno que pertenece a Siguientes(A), se desapilará el símbolo A de la pila. Esto se hace porque al sincronizar con un símbolo perteneciente a Siguientes(A), estamos supuestamente llegando a un estado equivalente a haber aplicado una producción para el no-terminal A.

Ejemplo 3.12.- Partiendo de la Tabla de análisis correspondiente al Ejemplo 3.11, podemos marcar los símbolos de sincronización con *sinc* (siempre que en la casilla correspondiente no exista ya una acción de derivación). Definimos el conjunto de símbolos de sincronización de cada no-terminal como la unión de los Primeros y Siguientes. Los símbolos de sincronización correspondientes a Primeros(A) ya están marcados con una acción (aplicar la producción correspondiente), así que únicamente marcamos las casillas correspondientes a Siguientes(A) que estuviesen en blanco todavía. Si el analizador sintáctico al buscar la entrada TA [*A, a*] se encuentra la acción de error (o blanco) entonces informará del error y comenzará la sincronización en modo pánico. En este proceso de sincronización, se saltará símbolos de la cadena de entrada hasta alcanzar uno (x) cuya celda (TA[A,x]) esté marcada con una acción (habrá sincronizado con un símbolo perteneciente a Primeros(A)), o con sinc (habrá sincronizado con Siguientes(A)). Si es una sincronización con Primeros(A), se aplicará la producción indicada en la celda, si es una sincronización con Siguientes(A), deberá desapilarse el símbolo A de la pila y continuar el análisis.

	a	+	*	**(**	**)**	**$**
E	(TE',1)			(TE',1)	sinc	sinc
E'		(+TE',2)			(ε,3)	(ε,3)
T	(FT',4)	sinc		(FT',4)	sinc	sinc
T'		(ε,6)	(*FT',5)		(ε,6)	(ε,6)
F	(a,8)	sinc	sinc	((E),7)	sinc	sinc
a	sacar					
+		sacar				
*****			sacar			
(				sacar		
)					sacar	
$						aceptar

(a * + a $, E $, ε) ⊢ (a * + a $, T E' $, 1)

⊢ (a * + a $, F T' E' $, 14)

⊢ (a * + a $, a T’ E' $, 148)

⊢ (* + a $, T’ E' $, 148)

⊢ (* + a $, * F T’E' $, 1485)

⊢ (+ a $, F T’ E' $, 1485) ERROR (sincronizado)

⊢ (+ a $, T’ E' $, 1485)

⊢ (+ a $, E' $, 14856)

⊢ (+ a $, + T E' $, 148562)

⊢ (a $, T E' $, 148562)

⊢ (a $, F T’ E' $, 148562)

⊢ (a $, a T’ E' $, 1485628)

⊢ ($, T’ E' $, 1485628)

⊢ ($, E' $, 14856286)

⊢ ($, $, 1485624863)