TEMA 4

ANÁLISIS SINTÁCTICO DESCENDENTE

4.1 GRAMÁTICAS LL(1)

Backtracking

El análisis descendente intenta encontrar entre las producciones de la gramática una derivación por la izquierda del símbolo inicial para una cadena de entrada. El análisis sintáctico descendente (ASD) puede incluir retrocesos en el análisis (*backtracking*), lo que implica varios exámenes de la entrada. En la práctica no existen muchos A.S. con retroceso, pues casi nunca es necesario.

EJEMPLO:

S → cAd
A → ab | a Analicemos la cadena de entrada: "cad"

1. En la situación en la que el símbolo analizado es el primero: "cad" la única producción que se puede escoger es la primera, luego: S → cAd , y el primer símbolo de la entrada "**c**" queda emparejado con la primera hoja izquierda del árbol que también es "c", con lo que se puede seguir adelante con el análisis.
2. Se avanza la marca de entrada al segundo símbolo: "c**a**d" y se considera la siguiente hoja del árbol (siempre de izquierda a derecha), etiquetada con A. Entonces se expande el árbol con la primera producción de A, luego cAd → cabd , y como el segundo símbolo de la entrada "**a**" queda emparejado con la segunda hoja por la izquierda, podemos avanzar la marca de análisis.
3. Se avanza la marca de entrada al símbolo siguiente: "ca**d**" y se compara con la siguiente hoja etiquetada con "b". Como no concuerda con "**d**" se indica el error y se vuelve a A para ver si hay otra alternativa no intentada, restableciendo la marca de entrada a la posición que ocupaba entonces.
4. Con la marca en "c**a**d" se expande la producción no intentada A → a, que hace que cAd → cad. Ahora el símbolo de entrada "**a**" coincide con la nueva hoja "a" y se puede proseguir el análisis.
5. Se avanza de nuevo la marca a "ca**d**" y coincide con la hoja de la derecha que quedaba por visitar en el árbol, por lo que se da por finalizado el análisis con éxito.

❑

Análizadores Sintácticos Predictivos (ASP)

Para que el algoritmo tenga una complejidad lineal, siempre debe saber qué regla se debe aplicar, ya que si hace *backtracking*, la complejidad ya no sería lineal (en el peor de los casos sería exponencial). Por tanto, es necesario que el analizador realice una predicción de la regla a aplicar.

Para ello, se debe conocer, dado el *token* de la entrada, *a*, que esté siendo analizado, y el no terminal a expandir *A*, cuál de las alternativas de producción $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ es la única posible que da lugar a que el resto de la cadena que se está analizando empiece por *a*. Dicho de

otra manera, la alternativa apropiada debe poderse predecir sólo con ver el primer símbolo que produce (como así sucede en la mayoría de lenguajes de programación). Veremos qué forma deben tener las gramáticas a las que se puede aplicar esta metodología.

EJEMPLO:

Sent → if Expres then Sent | while Expres do Sent | begin Sent end

En esta gramática sólo existe siempre una posibilidad de derivación, según que el primer *token* que haya en la entrada en el momento de tomar esa decisión sea *if*, *while* o *begin*.

❑

Según la nomenclatura que ya hemos introducido, las gramáticas que son susceptibles de ser analizadas sintácticamente de forma descendente y mediante un análisis predictivo, pertenecen al conjunto de gramáticas denominado LL(1). En un análisis de izquierda a derecha de la cadena de entrada y haciendo derivaciones por la izquierda, debe bastar con ver un solo *token* en la cadena para saber en cada caso qué producción escoger. A partir de las gramáticas de tipo LL(1) se pueden construir automáticamente *analizadores sintácticos descendentes predictivos* (ASDP), que no son otra cosa que ASD sin retroceso.

Cualquier gramática recursiva por la izquierda o con símbolos comunes por la izquierda en algunas producciones, seguro que no será LL(1). Si en una gramática se elimina su recursividad por la izquierda y se factoriza por la izquierda (si tuviera factores comunes por ese lado), la gramática modificada resultante podría ser analizable por un ASDP si no presenta ambigüedades, lo cual motivaría el problema de que en algún momento el analizador no sabrá qué producción seleccionar, puesto que hay, por lo menos, dos posibles análisis.

Conjuntos de predicción

Los conjuntos de predicción son conjuntos de *tokens* que ayudan a predecir qué regla se debe aplicar para la variable que hay que derivar. Se construyen, como veremos a continuación, a partir de los símbolos de las partes derechas de las producciones de la gramática.

Para saber qué regla se debe aplicar en cada caso, el analizador consulta el siguiente *token* en la entrada y si pertenece al conjunto de predicción de una regla (de la variable que hay que derivar, por supuesto), aplica esa regla. Si no puede aplicar ninguna regla, se produce un mensaje de error.

EJEMPLO:

Supóngase la siguiente gramática:

A → aBc | xC | B
B → bA
C → c

y la entrada "`babxcc`". Supongamos que el análisis a progresado a lo largo de los símbolos marcados en negrita: "**`bab`**`xcc`". En esta situación, la cadena de derivaciones habrá sido esta:

A → B → bA → baBc → babAc

La cuestión en este momento es: ¿qué producción tomar para seguir el análisis? Para seguir analizando, hay que desarrollar la variable A, que puede hacerlo según tres posibles opciones. Es fácil, observando las producciones de A en la gramática, darse cuenta que para escoger la primera opción el resto de la cadena debería empezar por *a*; para escoger la segunda, por *x* y para la tercera, por *b*. Como el resto de la cadena es "`xcc`", no hay duda que hay que tomar la segunda opción. Hemos hecho uso de los conjuntos de predicción.

❑

La condición LL(1) está basada en el contenido de estos conjuntos e implica las siguientes propiedades o características:

- LL(1) ⇒ las gramáticas pertenecientes a este conjunto satisfarán:
  - ♦ La secuencia de *tokens* se analiza de izquierda a derecha.
  - ♦ Utilizaremos la derivación del no terminal que aparezca más a la izquierda.
  - ♦ Sólo tendremos que ver un *token* de la secuencia de entrada para saber qué producción seguir.

EJEMPLO:

La siguiente gramática (de la que se muestran sólo las producciones de la variable A, pues el resto da igual para esto) no cumple los requisitos para ser LL(1):

```
...
A  →  aBc | aC | B
...
```

Si tenemos que desarrollar la variable A, no podemos saber, viendo un único símbolo en la entrada, cuál es la opción a escoger, pues si aparece "a" en la entrada tenemos dos posibles opciones: la primera y la segunda. Luego el análisis no es predictivo y la gramática no es LL(1).

❑

4.2 CÁLCULO DE LOS CONJUNTOS DE PREDICCIÓN

Los conjuntos de predicción de una regla se calculan en función de los primeros símbolos que puede generar la parte derecha de esa regla, y a veces (cuando esa parte derecha puede generar la cadena vacía) en función de los símbolos que pueden aparecer a continuación de la parte izquierda de la regla en una forma sentencial. Para especificar formalmente cómo se calculan los conjuntos de predicción es necesario estudiar antes cómo se calculan los primeros símbolos que genera una cadena de terminales y no terminales (conjunto de primeros) y cómo obtener los símbolos que pueden seguir a una variable en una forma sentencial (conjunto de siguientes).

Los conjuntos de PRIMEROS y SIGUIENTES contienen símbolos terminales, están asociados a una gramática determinada y facilitan la construcción de un ASDP, además de servir para comprobar que dicha construcción es factible (es decir, verificar el carácter LL(1) de una gramática dad). Vamos a definirlos a continuación.

Cálculo del conjunto de PRIMEROS:

La función *PRIMEROS* se aplica a cadenas de símbolos de la gramática ($\alpha \in (T \cup N)^*$) y devuelve un conjunto que puede contener cualesquiera terminales de la gramática y además puede contener a la cadena vacia, ε. A partir de ahora, cuando aparezca *PRIMEROS*(α), siendo α una cadena de $(T \cup N)^*$, nos referiremos al conjunto resultado de aplicar la función *PRIMEROS* a la cadena α.

Definición:

Si α es una forma sentencial compuesta por una concatenación de símbolos, PRIMEROS(α) es el conjunto de terminales (o ε) que pueden aparecer iniciando las cadenas que pueden derivar (en cero o más pasos) de α.

Definición formal:

$a \in$ PRIMEROS(α) si $a \in (T \cup \{\varepsilon\})$ / $\exists\ \alpha \Rightarrow^* a\beta$ para alguna tira β.

Reglas para su cálculo:

6. Si $\alpha \in T$, PRIMERO(α) = {α}
7. Si $\alpha \in N$:
   - Inicialmente, PRIMEROS(α) =$\varnothing$
   - Si aparece la producción $\alpha \rightarrow \varepsilon$, PRIMEROS($\alpha$) = PRIMEROS($\alpha$) $\cup$ {ε}
   - si $\alpha \rightarrow a_1\ a_2\ ...\ a_n$ entonces PRIMEROS(α) = PRIMEROS(α) $\cup$ PRIMEROS($a_1\ a_2\ ...\ a_n$) y para el cálculo de PRIMEROS($a_1\ a_2\ ...\ a_n$) pueden darse dos casos:

   - Si $\varepsilon \notin$ PRIMEROS(a_1) entonces
     PRIMEROS(α) = PRIMEROS(α) $\cup$ PRIMEROS(a_1)

   - Si $\varepsilon \in$ PRIMEROS(a_1) entonces
     PRIMEROS(α) = PRIMEROS(α) $\cup$ (PRIMEROS(a_1)$-${ε}) $\cup$ PRIMEROS($a_2 ... a_n$)
     y de nuevo pueden darse estos dos casos para PRIMEROS($a_2\ ...\ a_n$) y siguientes, hasta a_n.
     Si $\forall i$, $\varepsilon \in$ PRIMEROS(a_i) entonces
     PRIMEROS(α) = PRIMEROS(α) $\cup$ {ε}
8. Para recoger todos los casos posibles habría que considerar que
   Si $\alpha \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$ entonces $\text{PRIMEROS}(\alpha) = \bigcup_{i=1}^{n} \text{PRIMEROS}(\alpha_i)$

EJEMPLO:

Sea la siguiente gramática para expresiones aritméticas con sumas y multiplicaciones:

E → T E'
E' → + T E' | ε
T → F T'
T' → ∗ F T' | ε
F → (E) | ident

Cálculo de los PRIMEROS de todos los no terminales de esta gramática:

PRIMEROS(E') = { + , ε }

PRIMEROS(T') = { ∗ , ε }

PRIMEROS(F) = { (, ident }

PRIMEROS(E) $=^{(E \rightarrow TE',\ T \in N)}$ PRIMEROS(T) $=^{(T \rightarrow FT',\ F \in N)}$ PRIMEROS(F) = { (, ident }

❑

EJEMPLO:

Sea la gramática siguiente:

A → A a
A → B C D
B → b
B → ε
C → c
C → ε
D → d
D → C e

Calculamos los conjuntos de PRIMEROS de todas las variables de esta gramática:

PRIMEROS(C) = { c , ε }

PRIMEROS(B) = { b , ε }

PRIMEROS(D) = PRIMEROS(d) ∪ PRIMEROS(Ce) =
= { d } ∪ ((PRIMEROS(C) – {ε}) ∪ PRIMEROS(e)) = { d , c , e }

PRIMEROS(A) = PRIMEROS(Aa) ∪ PRIMEROS(BCD) =$^{(*)}$ PRIMEROS(BCD)
$^{(*)}$ Puesto que el miembro se calcula como PRIMEROS(A) que aún es ∅.
= { b } ∪ PRIMEROS(CD) = { b , c } ∪ PRIMEROS(D) = { b , c , d , e }

- ¿Y si añadimos a esta gramática la regla D → ε ?

Entonces hay que cambiar los cálculos, pues habría que añadir ε a PRIMEROS(D) y eso cambiaría el cálculo de PRIMEROS(A):
PRIMEROS(BCD) = { b } ∪ PRIMEROS(CD) = {b, c} ∪ PRIMEROS(D) = { b, c, d, e, ε }

Entonces PRIMEROS(A) = { b, c, d, e, ε }, pero puesto que ahora la regla A → BCD puede derivar a ε, eso implica que A puede desaparecer de la primera posición de la regla A → Aa y, por tanto, también hay que añadir "a" al conjunto de PRIMEROS(A).

PRIMEROS(A) = { b , c , d , e , ε , a }

❑

Cálculo del conjunto de SIGUIENTES:

La función *SIGUIENTES* se aplica a variables de la gramática (A ∈ *N*) y devuelve un conjunto que puede contener cualesquiera terminales de la gramática y además puede contener un símbolo especial, "$", que representa el final de la cadena de entrada. Esto es equivalente a añadir una producción adicional a la gramática en la que el axioma, S, es definido como X → S$.

Definición:
Si A es un símbolo no terminal de la gramática, SIGUIENTES(A) es el conjunto de terminales que pueden aparecer a continuación de A en alguna forma sentencial derivada del símbolo inicial.

Definición formal:

a ∈ SIGUIENTES(*A*) si *a* ∈ (*T* ∪ {$}) / ∃ $S \Rightarrow^* \alpha A a \beta$ para algún par de tiras α,β.

Reglas para su cálculo:

1. Inicialmente, SIGUIENTES(*A*) =∅
2. Si A es el símbolo inicial, entonces SIGUIENTES(*A*) = SIGUIENTES(*A*) ∪ {$}
3. **(S1)** Para cada regla de la forma:
   B → α*A*β entonces SIGUIENTES(A) = SIGUIENTES(A) ∪ PRIMEROS(β) – {ε}
4. **(S2)** Para cada regla de la forma:
   B → α*A* o bien B → α*A*β en la que ε ∈ PRIMEROS(β)
   entonces SIGUIENTES(A) = SIGUIENTES(A) ∪ SIGUIENTES(B)
5. Repetir los pasos 3 y 4 hasta que no se puedan añadir más símbolos a SIGUIENTES(A).

NOTA:
Las reglas **(S1)** y **(S2)** no son excluyentes. Primero habrá que intentar aplicar **(S1)** y luego **(S2)** Sólo en el caso de producciones del tipo B → α*A*, no tendrá sentido intentar aplicar **(S1)**.

EJEMPLO:

En la gramática de las expresiones aritméticas del ejemplo anterior calcularemos los conjuntos SIGUIENTES de todos los símbolos no terminales:

Primero se añade la producción previa a la inicial: $X \rightarrow E\$$

$SIGUIENTES(E) = {}_{\mathbf{S1}}^{F\rightarrow(E) \text{ y } X\rightarrow E\$}$ { **)** , **$** }

$SIGUIENTES(E') = {}_{\mathbf{S2}}^{E'\rightarrow+TE' \text{ y } E\rightarrow TE'}$ $SIGUIENTES(E) \cup \underline{SIGUIENTES(E')}_{=\varnothing}$ = { **)** , **$** }

$SIGUIENTES(T) = {}_{\mathbf{S1}}^{E'\rightarrow+TE'}$ $PRIMEROS(E')-\{\varepsilon\} \cup SIGUIENTES(E) \cup SIGUIENTES(E')$ =

---------- **S2** también porque $E' \rightarrow \varepsilon$ ---------

= {**+**} ∪ SIGUIENTES(E) ∪ SIGUIENTES(E') = { **+** , **)** , **$** }

$SIGUIENTES(T') = {}_{\mathbf{S2}}^{T\rightarrow FT' \text{ y } T\rightarrow *FT'}$ $SIGUIENTES(T) \cup \underline{SIGUIENTES(T')}_{=\varnothing}$ = { **+** , **)** , **$** }

$SIGUIENTES(F) = {}_{\mathbf{S1}}^{T\rightarrow FT' y T\rightarrow *FT'}$ $PRIMEROS(T')-\{\varepsilon\} \cup SIGUIENTES(T) \cup SIGUIENTES(T')$ =

---------- **S2** también porque $T' \rightarrow \varepsilon$ ----------

= { ∗ } ∪ { **+** , **)** , **$** } = { ∗ , **+** , **)** , **$** }

❑

Cálculo del conjunto PREDICT:

La función *PREDICT* se aplica a producciones de la gramática ($A \rightarrow \alpha$) y devuelve un conjunto, llamado *conjunto de predicción*, que puede contener cualesquiera terminales de la gramática y el símbolo "$", pero nunca puede contener ε. Cuando el análisis ASDP tiene que derivar una variable, consulta el símbolo de la entrada que espera a ser analizado y lo busca en los conjuntos de predicción de cada regla de esa variable (ver ejemplo en la página 2 de este tema). De esta forma y siempre que los conjuntos de predicción de todas las reglas de cada variable por separado sean disjuntos entre sí (aunque puede suceder que dos conjuntos de predicción de variables distintas tengan símbolos comunes), el analizador sintáctico puede construir una derivación por la izquierda de la cadena de entrada.

Regla para su cálculo:

PREDICT($A \rightarrow \alpha$) =

Si $\varepsilon \in$ PRIMEROS(α) entonces = (PRIMEROS(α) – {ε}) ∪ {SIGUIENTES(A)}

sino = PRIMEROS(α)

EJEMPLO:

Supóngase la siguiente gramática:

S → AB | s
A → aSc | eBf | ε
B → bAd | ε

Calculamos los conjuntos de predicción utilizando la regla adecuada en cada caso:

PREDICT(S → AB) = (PRIMEROS(AB)–{ε}) ∪ SIGUIENTES(S) = { a , e , b , c , $ }
PREDICT(S → s) = PRIMEROS(s) = { s }
PREDICT(A → aSc) = PRIMEROS(aSc) = { a }
PREDICT(A → eBf) = PRIMEROS(eEf) = { e }
PREDICT(A → ε) = (PRIMEROS(ε)–{ε}) ∪ SIGUIENTES(A) = { b , d , c , $ }
PREDICT(B → bAd) = PRIMEROS(bAd) = { b }
PREDICT(B → ε) = (PRIMEROS(ε)–{ε}) ∪ SIGUIENTES(B) = { f , c , $ }

La pregunta ahora sería: ¿podía construirse un ASDP a la vista de estos conjuntos?

❑

4.3 LA CONDICIÓN LL(1)

Para que una gramática pertenezca al conjunto de gramáticas LL(1) ha de cumplir la condición LL(1). Esta condición no "salta a la vista" a partir del aspecto de las producciones de la gramática, sino que tiene que ver con el contenido de los conjuntos de predicción de las reglas que derivan de un mismo no terminal.

Para que la regla a aplicar sea siempre única, se debe exigir que los conjuntos de predicción de las reglas de cada no terminal sean disjuntos entres sí; es decir, no puede haber ningún símbolo (*token*) que pertenezca a dos o más conjuntos de predicción de las reglas de una misma variable. Si se cumple esta condición, la gramática es LL(1) y se puede realizar su análisis sintáctico en tiempo lineal.

La condición LL(1) es necesaria y suficiente para poder construir un ASDP para una gramática.

Vamos a ver formalmente dicha condición.

- **Condición LL(1):**

  Dadas todas las producciones de la gramática para un mismo terminal:

  $$A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n \quad \forall A \in N$$

  se debe cumplir la siguiente condición:

  $$\forall\, i, j \;\; (i \neq j) \quad \text{PREDICT}(A \rightarrow \alpha_i) \cap \text{PREDICT}(A \rightarrow \alpha_j) = \varnothing$$

EJEMPLO:

Sea la siguiente gramática con sus conjuntos de predicción ya calculados para cada regla:

```
A  →  a b B    { a }
A  →  B b      { b , c }
B  →  b        { b }
B  →  c        { c }
```

Se puede afirmar que es LL(1) porque los conjuntos de predicción de las dos reglas de la variable A son disjuntos entre sí, y los conjuntos de predicción de las reglas de B también lo son. Como se puede comprobar, los símbolos b y c pertenecen a varios conjuntos de predicción de reglas de diferentes variables, y sin embargo la gramática sigue siendo LL(1). Si añadimos la regla

```
B  →  a        { a }
```

los conjuntos de predicción quedan de la siguiente manera:

```
A  →  a b B    { a }
A  →  B b      { a , b , c }
B  →  a        { a }
B  →  b        { b }
B  →  c        { c }
```

Con esta nueva regla, los conjuntos de predicción de las reglas de la variable A ya no son disjuntos (el símbolo a pertenece a ambos) y por tanto la gramática no es LL(1), independientemente de si los conjuntos de predicción de las reglas de B son o no disjuntos.

❑

EJEMPLO:

Sea la siguiente gramática:

E → E + T | T
T → T ∗ F | F
F → num | (E)

Se trata de estudiar si se cumple la condición LL(1). Para ello se calculan los PREDICT:

PREDICT(E → E+T) = PRIMEROS(E+T) = PRIMEROS(E) = { num , (}
PREDICT(E → T) = PRIMEROS(T) = PRIMEROS(F) = { num , (}
PREDICT(T → T∗F) = PRIMEROS(T∗F) = PRIMEROS(T) = { num , (}
PREDICT(T → F) = PRIMEROS(F) = { num , (}
PREDICT(F → num) = PRIMEROS(num) = { num }
PREDICT(F → (E)) = PRIMEROS((E)) = { (}

Para el símbolo F, la intersección los conjuntos de predicción de todas las reglas en las que se desarrolla es:

PREDICT(F → num) ∩ PREDICT(F → (E)) = { num } ∩ { (} = ∅

pero no sucede lo mismo con los PREDICT de las producciones de T y de E, que son iguales, por lo tanto, no disjuntos, por lo que la gramática no cumple la condición LL(1).

❑

EJEMPLO:

E → T E'
E' → + T E' | ε
T → F T'
T' → ∗ F T' | ε
F → num | (E) ¿Cumple la condición LL(1)?

No hace falta calcular los conjuntos de predicción de aquellas variables que no tienen más que una opción para su desarrollo. Si sólo hay una opción, no se planteará nunca dudas sobre qué opción elegir. Sólo aquellas variables con dos o más alternativas son las que hay que estudiar para ver si sus conjuntos de predicción son disjuntos. Por lo tanto, en este ejemplo no hace falta calcular los PREDICT(E → TE') ni PREDICT(T → FT'). Vamos a ver qué ocurre con los restantes.

PREDICT(E' → +TE') = { + }
PREDICT(E' → ε) = SIGUIENTES(E') = {) , $ }

PREDICT(T' → ∗FT') = { ∗ }
PREDICT(T' → ε) = SIGUIENTES(T') = { + ,) , $ }

PREDICT(F → num) = { num }
PREDICT(F → (E)) = { (}

Ahora, para cada uno de estos no terminales con alternativas, comprobamos si los conjuntos de predicción son disjuntos entre sí:

PREDICT(E' → +TE') ∩ PREDICT(E' → ε) = { + } ∩ {) , $ } = ∅
PREDICT(T' → ∗FT') ∩ PREDICT(T' → ε) = { ∗ } ∩ { + ,) , $ } = ∅
PREDICT(F → num) ∩ PREDICT(F → (E)) = { num } ∩ { (}= ∅

Luego esta gramática cumple la condición LL(1).

❑

4.4 MODIFICACIÓN DE GRAMÁTICAS NO LL(1)

Existen algunas características que, en el caso de ser observadas en una gramática, garantizan que no es LL(1) (sin necesidad de calcular los conjuntos de predicción); sin embargo, si ninguna de estas características aparece, la gramática puede que sea LL(1) o puede que no lo sea (en este caso sí que hay que calcular los conjuntos de predicción para comprobarlo). También, si la gramática no es LL(1) no necesariamente debe tener alguna de estas características; puede que tenga alguna, puede que las tenga todas o puede que no tenga ninguna de ellas.

Por otra parte, veremos que si nos encontramos una gramática que no sea LL(1) existen métodos para modificarlas para convertirlas en LL(1).

4.4.1 ELIMINACIÓN DE LA AMBIGÜEDAD

Cualquier gramática ambigua no cumple LL(1) pues, por la propia definición de esta propiedad, en algún caso no sabremos qué producción tomar a la vista de un único símbolo, pues habrá más de un posible árbol de análisis sintáctico. Esto no quiere decir que si la gramática no es ambigua entonces será LL(1), pues puede presentar otros problemas, como veremos a continuación.

El proble ma de la ambigüedad es el más difícil de resolver pues no existe una metodología para eliminarla y tampoco hay otra fórmula para saber que una gramática es ambigua mas que la de encontrar alguna sentencia que tenga dos árboles de análisis sintáctico distintos.

La mejor opción cuando se presenta una gramática ambigua es replantearse el diseño de la misma para encontrar una gramática no ambigua equivalente (que genere el mismo lenguaje).

4.4.2 FACTORIZACIÓN IZQUIERDA

Si dos producciones alternativas de un símbolo A empiezan igual, no se sabrá por cuál de ellas seguir. Se trata de reescribir las producciones de A para retrasar la decisión hasta haber visto lo suficiente de la entrada como para elegir la opción correcta.

EJEMPLO:

Sean las producciones:

Sent → if Expr then Sent else Sent
Sent → if Expr then Sent
Sent → Otras

al ver “if” no se sabe cuál de las dos producciones hay que tomar para expandir Sent. De hecho, un analizador sintáctico descendente no sabría qué hacer hasta superar el cuarto símbolo de esta producción. Si entonces llega else ya sabe que se trataba de la primera y si entra cualquier otro *token* entonces se trataba de la segunda. Pero esto es ya demasiado *tarde* para el análisis sintáctico predictivo.

❑

Nos enfrentamos, pues, al problema de producciones que tienen símbolos comunes por la izquierda; es decir, si son del tipo: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$. En estos casos, ante una entrada del prefijo α, no sabemos si expandir $\alpha\beta_1$ o por $\alpha\beta_2$. La solución pasa por modificar cada producción afectada de la siguiente forma:

cambiar $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ por dos producciones: $A \rightarrow \alpha A'$
$A' \rightarrow \beta_1 \mid \beta_2$

Con esto se retrasa el momento de la decisión hasta después de analizar los símbolos comunes, y se soluciona uno de los problemas que tenía esa gramática para no cumplir la condición LL(1).

Veamos la regla general para factorizar por la izquierda una gramática. Para todos los no terminales, *A*, de la gramática, encontrar el prefijo α más largo común a dos o más alternativas suyas. Si $\alpha \neq \varepsilon$ entonces sustituir las producciones

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

donde γ_i representa a todas las alternativas que no empiezan por α, por:

$$A \rightarrow \alpha A' \mid \gamma_i$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Repetir el paso anterior hasta que no haya 2 alternativas de un no terminal con un prefijo común.

EJEMPLO:

Sent → if Expr then Sent else Sent
Sent → if Expr then Sent
Sent → Otras

Fijándonos en la regla para factorizar, podemos identificar: $\alpha \equiv$ if Expr then Sent
$\gamma_i \equiv$ Otras

Con lo que la gramática factorizada queda
Sent → if Expr then Sent Sent'
Sent → Otras
Sent' → else Sent
Sent' → ε

Adicionalmente, supóngase que el no terminal Otras deriva en un terminal cualquiera "*x*". Compruébese que esta gramática es ahora LL(1).

❑

4.4.3 ELIMINACIÓN DE LA RECURSIVIDAD IZQUIERDA

Como se dijo en el tema anterior, una gramática es recursiva por la izquierda si tiene alguna producción que sea recursiva por la izquierda; es decir:

$$\exists\ A \in N\ /\ \exists\ A \Rightarrow^* A\alpha \quad \text{para alguna cadena } \alpha$$

Los analizadores sintácticos descendentes no pueden manejar estas gramáticas pues entrarían en ciclos de recursividad infinita al ejecutarse. Cualquier gramática recursiva por la izquierda no cumple la condición LL(1), por ejemplo, para gramáticas de la forma:

$A \rightarrow A\alpha \mid \beta$ ⟶ genera el lenguaje: β, $\beta\alpha$, $\beta\alpha\alpha$, $\beta\alpha\alpha\alpha$...
No cumple LL(1)

Existe una regla para modificar este tipo de producciones para que deje de ser recursivas por la izquierda:

$A \rightarrow \beta A'$ ⟶ Nuevo no terminal auxiliar.
$A' \rightarrow \alpha A' \mid \varepsilon$
⟶ recursiva por la derecha

Esta gramática modificada ya no es recursiva por la izquierda, sino que lo es por la derecha y ya no presenta el problema que le impedía ser LL(1).

EJEMPLO:

Eliminar la recursividad izquierda de la gramática de las expresiones aritméticas.

1º $E \rightarrow E + T \mid E - T \mid T$
2º $T \rightarrow T * F \mid T / F \mid F$
$F \rightarrow (E) \mid$ núm

1º $E \rightarrow E + T \mid T$; $A \rightarrow A\ \alpha \mid \beta$ } $E \rightarrow T\ E'$, $E' \rightarrow + T\ E' \mid \varepsilon$

2º $T \rightarrow T * F$, $T \rightarrow F$ } $T \rightarrow F\ T'$, $T' \rightarrow * F\ T' \mid \varepsilon$

Por tanto nos queda la siguiente gramática:

$E \rightarrow T\ E'$
$E' \rightarrow + T\ E' \mid - T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow * F\ T' \mid / F\ T' \mid \varepsilon$
$F \rightarrow (E) \mid num$

❑

En el caso general, independientemente de cuántas producciones alternativas haya de A, se puede eliminar la recursividad por la izquierda utilizando las siguientes reglas:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid ... \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid ... \mid \beta_n$$

Se sustituye por:

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid ... \mid \alpha_m A' \mid \varepsilon$$

Este algoritmo no elimina la recursividad izquierda que incluya alguna derivación que sea recursiva tras 2 o más pasos (recursividad indirecta). Por ejemplo, la siguiente gramática tiene recursividad directa por la izquierda, y además otro problema de recursividad indirecta:

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \varepsilon$

Tenemos: $S \rightarrow Aa \rightarrow Sda$ que presenta recursividad izquierda indirecta $S \Rightarrow Sda$

Para resolver esto se proporciona el siguiente algoritmo capaz de eliminarlas sistemáticamente. Funciona si la gramática no tiene ciclos ($A \Rightarrow^{+} A$) o producciones vacías ($A \rightarrow \varepsilon$).

Pasos:
1º: Ordenar los no terminales según $A_1, A_2, \ldots, A_n$
2º: DESDE $i \leftarrow 1$ HASTA n HACER
  DESDE $j \leftarrow 1$ HASTA $i-1$ HACER
    Sustituir cada $A_i \rightarrow A_j\gamma$ por $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$
      donde $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ son las producciones actuales de A_j
    Eliminar la recursividad izquierda directa de la producción de A_i
  HECHO
HECHO

EJEMPLO:

Vamos a deshacer la recursividad izquierda que presentaba el ejemplo anterior:

$S \rightarrow A\ a \mid b$
$A \rightarrow A\ c \mid S\ d \mid \varepsilon$

1º: $A_1 = S$, $A_2 = A$

2º. Para *i*=1: desde *j*:=1 hasta 0 hacer: *nada*

Para *i*=2: desde *j*:=1 hasta 1 hacer:

tomamos el segundo y lo sustituimos por el primero en todos los lugares donde éste aparezca a la derecha:

$S \rightarrow Aa \mid b$, $A \rightarrow Ac \mid Sd \mid \varepsilon$ $\vdash\!\!\rightarrow$ $S \rightarrow Aa \mid b$, $A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

(Si hubiera otra regla: $X \rightarrow Sa \mid Ab \mid ...$

$X \rightarrow$ sustituir S y A por su definición cuando aparezca)

A esto le quitamos la recursividad directa. Sólo modificar A, quedando

$S \rightarrow A\,a \mid b$
$A \rightarrow b\,d\,A' \mid A'$
$A' \rightarrow c\,A' \mid a\,d\,A' \mid \varepsilon$

❑

EJEMPLO:

Sea la siguiente gramática:

$S \rightarrow S\ inst \mid T\ R\ V$
$T \rightarrow tipo \mid \varepsilon$
$R \rightarrow blq\ V\ fblq \mid \varepsilon$
$V \rightarrow id\ S\ fin \mid id\ ; \mid \varepsilon$

Es evidente que esta gramática no es LL(1) pues la primera producción es recursiva por la izquierda y las dos primeras de V tienen factores comunes por la izquierda. A continuación se presenta la gramática a la cual se han eliminado estos problemas mediante las técnicas descritas:

$S \rightarrow T\ R\ V\ S'$
$S' \rightarrow inst\ S'$
$S' \rightarrow \varepsilon$
$T \rightarrow tipo$
$T \rightarrow \varepsilon$
$R \rightarrow blq\ V\ fblq$
$R \rightarrow \varepsilon$
$V \rightarrow id\ V'$
$V \rightarrow \varepsilon$
$V' \rightarrow S\ fin$
$V' \rightarrow ;$

Ahora ya la gramática no presenta los problemas anteriores, pero no se puede asegurar que sea LL(1) mientras no se aplique la condición que verifica dicha propiedad. Nótese que todos los no terminales excepto el símbolo inicial, S, tienen más de una alternativa.

PREDICT(S'→TRVS') = { tipo , blq , id , inst , fin , $ }
PREDICT(S'→ε) = { inst } } ∩ = ∅

PREDICT(T→tipo) = { tipo }
PREDICT(T→ε) = { blq } } ∩ = ∅

PREDICT(R→blq V fblq) = { blq }
PREDICT(R→ε) = { id , inst , fin , $ } } ∩ = ∅

PREDICT(V→id V') = { id }
PREDICT(V→ε) = { inst , fin , $ } } ∩ = ∅

PREDICT(V'→S fin) = { tipo , blq , id , inst , fin }
PREDICT(V'→ ;) = { ; } } ∩ = ∅ luego ahora <u>sí es LL(1)</u>.

❑

4.5 IMPLEMENTACIÓN DE UN ANALIZADOR SINTÁCTICO DESCENDENTE RECURSIVO

Es un método de análisis sintáctico descendente que sólo se puede aplicar en gramáticas LL(1) y por tanto es un analizador sintáctico descendente predictivo (ASDP). Consiste en un conjunto de funciones recursivas (una por cada variable de la gramática) que son diseñadas a partir de los elementos que definen cada una de las producciones de la gramática. La secuencia de llamadas al procesar la cadena de entrada define implícitamente su árbol de análisis sintáctico.

Veremos a continuación cómo implementar un analizador sintáctico descendente recursivo (ASDR) a partir de las producciones de la gramática que genera las frases del lenguaje que se pretenden analizar. Vamos a describir la implementación de un ASDR en un lenguaje de alto nivel (en C), y vamos a definir los elementos necesarios para ello

Símbolo de preanálisis

Si esta metodología es aplicable a las gramáticas LL(1) es porque basta ver un único *token* para saber qué hacer. Ese *token* se llama símbolo de preanálisis o *LookAhead* y será pedido al analizador léxico cada vez que se necesite. Será este *token* el que se busque en los conjuntos de predicción de las diferentes reglas para escoger aquélla en la que aparezca

Función de emparejamiento

La función de emparejamiento o *Match* es la encargada de comprobar si el símbolo de preanálisis coincide con el terminal de la gramática que, de acuerdo con los elementos de la producción escogida debería aparecer en esa posición. Esta función también se encarga de otra misión fundamental como es la petición del siguiente *token* al analizador léxico si se ha producido la coincidencia o invocar la función de error en caso contrario.

Su estructura en metalenguaje algorítmico sería la siguiente:

FUNCIÓN *Emparejar* (Parámetro de entrada: *token*; Usa variable global *preanálisis*)
SI *preanálisis* coincide con *token* ENTONCES
    Pedir el siguiente *preanálisis* al analizador léxico
SINO
    Error Sintáctico(Encontrado *preanálisis*, esperaba *token*)

Su implementación en C, si t es el *token* que el ASDR espera encontrar en la entrada y, por tanto el que la función *Emparejar* recibe como parámetro, sería:

```
void Emparejar ( int t )
{
   if ( t == preanalisis )
      preanalisis = analex();     /* analex es el nombre que hemos dado al Analizador Léxico */
   else
      ErrorSintáctico(preanalisis,t);
}
```

Para construir un ASDR, además de utilizar estos elementos auxiliares, hay que hacer lo siguiente:

1. Escribir una función por cada símbolo no terminal de la gramática. Cada una de estas funciones llevará a cabo el análisis de las producciones de dicho no terminal, como se indicará más adelante.
2. Cuando este no terminal tenga distintas alternativas en la gramática, para decidir durante su ejecución cuál de las producciones utilizar, se optará por aquella alternativa a cuyo conjunto de predicción pertenezca el *token de preanálisis*.

```
EN CASO DE QUE Preanálisis PERTENEZCA A
   PREDICT(α1): ... proceder según alternativa α1 ...
   PREDICT(α2): ... proceder según alternativa α2 ...
   ...
FIN
```

Si el token de preanálisis no pertenece a ninguno de los PREDICT, entonces se considerará error sintáctico.

3. Si una de las alternativas para el no terminal que se está analizando es la cadena vacía ($A \rightarrow \varepsilon$), en ese caso no se hará nada.

4. Para analizar cada alternativa, se aplican distintas metodologías a los símbolos de la parte derecha de la producción, en el mismo orden en el que aparecen, según si son terminales o no:
   - Para cada $A \in N$ hacemos una llamada a su función correspondiente.
   - Para cada $a \in T$ hacemos una llamada a la función *Emparejar* con *a* como parámetro.

5. La invocación o puesta en marcha del ASDR se realiza mediante una llamada al símbolo inicial de la gramática. Para hacer esa llamada se supone que el *token* de preanálisis habrá sido inicializado por una llamada previa al analizador léxico.

EJEMPLO:

Sea la siguiente gramática LL(1) con sus conjuntos de predicción:

```
S  →  A      {a,$}
S  →  s      {s}
A  →  aSc    {a}
A  →  ε      {c,$}
```

Vamos a ver la implementación de un ASDR para ella. Supondremos siempre en este tipo de problemas que tenemos definidos los *tokens* (a, c y s) y además una variable lexema como un *array* de caracteres. Supondremos también en estos ejemplos que ya tenemos implementada previamente la función *emparejar* que acabamos de definir.

```
void S(void)
{
  if ( preanalisis == a || preanalisis = FINFICHERO )
    A();
  else if ( preanalisis == s )
    emparejar(s);
  else ErrorSintactico(lexema,a,s,FINFICHERO);
      /* encontrado 'lexema', esperaba 'a', 's' o fin de fichero */
}

void A(void)
{
  if ( preanalisis == a )
    { emparejar(a); S(); emparejar(c); }
  else if ( preanalisis == c || preanalisis = FINFICHERO )
    ;  /* producción epsilon */
  else ErrorSintactico(lexema,a,c,FINFICHERO);
}
```

A la función de error sintáctico se le suele pasar como argumentos el lexema que ha producido el error (el del *token* de preanálisis) y los que esperaba en su lugar (los terminales que aparezcan en la unión de todos los conjuntos de predicción de las reglas del no terminal al que pertenece la función).

❑

EJEMPLO:

Implementación de un ASDR para la gramática modificada LL(1) de las expresiones aritméticas del apartado anterior, usando la siguiente definición de *tokens*: `(MAS, MENOS, POR, DIV, LPAR, RPAR, NUM, FDF)` y que en los no terminales con *prima* sustituiremos ésta por un 2.

```
void E()
{
  T(); E2();
}

void E2()
{
  switch ( preanalisis ) {
    case MAS   : Emparejar(MAS); T(); E2(); break;
    case MENOS : Emparejar(MENOS); T(); E2(); berak;
    case RPAR  : case FDF : /* E' → ε */ break;
    default    : ErrorSintactico(lexema,MAS,MENOS,RPAR,FDF);
  }
}

void T()
{
   F(); T2();
}

void T2()
{
  switch ( preanalisis ) {
    case POR : Emparejar(POR); F(); T2(); break;
    case DIV : Emparejar(DIV); F(); T2(); break;
    case MAS : case MENOS :
    case RPAR : case FDF : /* T' → ε */ break;
    default  : ErrorSintactico(lexema,POR,DIV,MAS,MENOS,RPAR,FDF);
  }
}

void F()
{
  switch ( preanalisis ) {
    case NUM  : Emparejar(NUM); break;
    case LPAR : Emparejar(LPAR); E(); Emparejar(RPAR); break;
    default   : ErrorSintactico(lexema,NUM,LPAR);
  }
}
```

Nótese que se han permitido algunas licencias en el lenguaje C utilizado, que en una implementación real habría que realizar con más cuidado.

❑

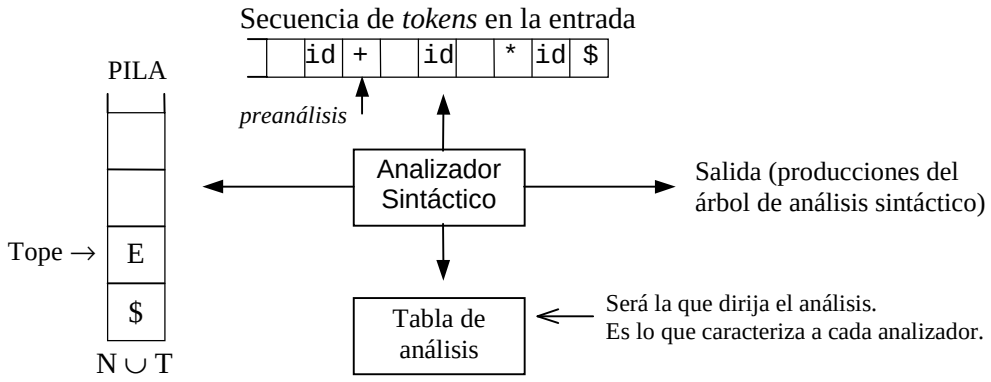
Comentarios adicionales para la implementación

Tras la devolución del control a la función main que inició el análisis con una llamada al símbolo inicial de la gramática, se debe comprobar que la cadena de entrada se ha analizado en su totalidad para dar como exitoso el análisis. Como se indica a continuación:

```
token = analex();
S();
if (token != FINFICHERO) error(token,lexema,{FINFICHERO});
   /* encontrado 'lexema', esperaba fin de fichero */
```

4.5 ANALIZADORES SINTÁCTICOS DESCENDENTES DIRIGIDOS POR TABLA

Se puede construir un analizador sintáctico descendente predictivo mediante una técnica completamente distinta a la descrita en el apartado anterior, utilizando una pila de símbolos (terminales y no terminales) en vez de haciendo llamadas recursivas. Para determinar qué producción debe aplicarse a la vista de un terminal (*token* de preanálisis), se buscará en una tabla llamada *tabla de análisis*.



El proceso de análisis es siempre aplicar el mismo algoritmo que describiremos más adelante, pero previamente se tendrá que construir la tabla, que es lo que diferencia un analizador de otro y que depende de la gramática a ser analizada. El procedimiento para construir tablas de análisis LL(1) es el siguiente:

1. Las filas de la tabla se etiquetan con las variables de la gramática.
2. Las columnas se etiquetan con los terminales y el símbolo de fin de entrada, $.
   - [En cada celda de la tabla se va a colocar la regla que se debe aplicar para analizar la variable de esa fila cuando el terminal de esa columna (o $) aparezca en la entrada (ese terminal o $ debe pertenecer al conjunto de predicción de esa regla)].
3. Se calculan los conjuntos de predicción para cada regla.
4. Para cada regla $A \rightarrow \alpha$ de la gramática, hacer:
   Para cada terminal $a \in$ PREDICT($A \rightarrow \alpha$), añádase $A \rightarrow \alpha$ a Tabla[A,*a*].
5. Cada entrada no definida de Tabla será *error sintáctico*.

Aunque hemos dicho que la tabla de análisis contiene las reglas, al implementar el algoritmo no es necesario almacenar la regla completa en la tabla sino sólo los símbolos de su parte derecha (que es conveniente guardar al revés) o incluso es más eficiente almacenar un número de regla que haga referencia a otra tabla con los símbolos de las partes derechas de las reglas.

Mensajes de error

- En el tope de la pila hay una variable; en este caso, el conjunto de símbolos que se esperaban en lugar del que se ha leído se calcula recorriendo la fila correspondiente a la variable en la tabla de análisis, anotando todos los símbolos para los que la entrada correspondiente en la tabla contiene un número de regla. Se puede comprobar fácilmente que ese conjunto es la unión de los conjuntos de predicción de las reglas de esa variable.
- En el tope de la pila hay un terminal; en este caso, el mensaje de error debe decir que se esperaba ese terminal en lugar del lexema leído.

EJEMPLO:

Sea de nuevo la gramática de las expresiones aritméticas (ya modificada para que sea LL(1)):

X → E $
E → T E'
E' → + T E' | – T E' | ε
T → F T'
T' → ∗ F T' | / F T' | ε
F → (E) | id

Veamos cada una de las producciones (recuérdese que las producciones con varias alternativas son realmente una producción de la variable por cada una de las alternativas):

1: PREDICT(E → T E') = { (, id } ⇒ en las celdas [E,(] y [E,id] añadir E→TE'

2: PREDICT(E'→ + T E') = {+} ⇒ en la celda [E',+] añadir E'→+TE'

3: PREDICT(E'→ – T E') = {–} ⇒ en la celda [E',–] añadir E'→–TE'

4: PREDICT(E'→ ε) = {) , $ } ⇒ en la celda [E',$] añadir E'→ ε

5: PREDICT(T → F T') = { (, id } ⇒ en las celdas [F,(] y [F,id] añadir T→FT'

6: PREDICT(T' → ∗ F T') = {∗} ⇒ en la celda [T',∗] añadir F'→ ∗FT'

7: PREDICT(T' → / F T') = {/} ⇒ en la celda [T',/] añadir F'→ /FT'

8: PREDICT(T'→ ε) = {+,–,),$} ⇒ en las celdas [T',+] [T', –] [T',)] [T',$] añadir T'→ ε

9: PREDICT(F→ (E)) = { (} ⇒ en la celda [F,)] añadir F → (E)

10: PREDICT(F→ id) = {id} ⇒ en la celda [F,id] añadir F → id

- Todas las celdas vacías se marcan como *error sintáctico.*

Con estos cálculos, la tabla de análisis para esta gramática resulta así:

N \ T+{$}	id	+	–	∗	/	(	)	$
E	E → TE'	*e*	*e*	*e*	*e*	E → TE'	*e*	*e*
E'	*e*	E' → +TE'	E' → –TE'	*e*	*e*	*e*	E' → ε	E' → ε
T	T → FT'	*e*	*e*	*e*	*e*	T → FT'	*e*	*e*
T'	*e*	T' → ε	T' → ε	T' → ∗FT'	T' → /FT'	*e*	T' → ε	T' → ε
F	F → id	*e*	*e*	*e*	*e*	F → (E)	*e*	*e*

❑

Una importante observación a tener en cuenta es que este tipo de analizador, al ser predictivo, sólo podrá construirse si la gramática a analizar es LL(1). Esto se refleja en la tabla en el hecho de que no aparezcan dos producciones en la misma casilla. Si en alguna de ellas aparecieran dos producciones significaría que, llegando a esa situación, el analizador no sabría qué decisión tomar, lo cual está en contra del concepto de gramática LL(1).

Así pues, la no aparición de casillas con dos o más producciones puede considerarse como una demostración de que la gramática analizada es LL(1).

Veámoslo con un ejemplo.

EJEMPLO:

Sea una gramática para las sentencias IF en Pascal:

Sent → if Expr then Sent Sent' | otras
Sent' → else Sent | ε
Expr → lógico

Se puede comprobar que la tabla resultante es la siguiente:

	otras	lógico	else	if	then	$
P	P→otras	*e*	*e*	Sent→if Expr then Sent Sent'	*e*	*e*
P'	*e*	*e*	Sent'→else Sent Sent' → ε	*e*	*e*	Sent'→ε
E	*e*	E→lógico	*e*	*e*	*e*	*e*

Luego la gramática es ambigua debido a la existencia de dos producciones en la casilla señalada. Esta ambigüedad se podría resolver durante el análisis aplicando sistemáticamente una de las dos. La producción Sent' → else Sent equivale a asociar el ELSE al IF más cercano. ❑

Una vez que se tiene la tabla de análisis de una gramática ya se puede construir un analizador sintáctico descendente dirigido por esa tabla. A continuación se presenta el algoritmo que ejecuta dicho analizador:

- **Algoritmo de ejecución del Analizador Descendente Dirigido por Tabla:**

  Entrada: cadena de elementos léxicos devueltos por el A.L.

  Salida: producciones que construyen su árbol de análisis sintáctico

  Pasos:

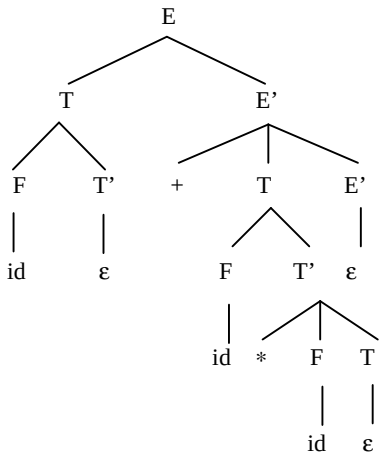
```
push($)
push(S)
REPETIR
   A := Pila[tope]    /* el símbolo que haya en el tope de la pila */
   a := analex( )     /* a es el símbolo de preanálisis */
   SI A es un terminal o $ ENTONCES
      SI A = a ENTONCES
         pop A
         a := analex( )
      SINO
         Error Sintáctico ( encontrado lexema de a, esperaba A )
      FINSI
   SINO   /* A es un no terminal */
      SI Tabla[A,a] = A → c1 c2 ... ck ENTONCES
         pop(A)
         DESDE i := k HASTA 1 HACER push(ci) FINDESDE
         /* A efectos de traza se puede indicar que se usa A → c1 c2 ... ck */
      SINO
         Error Sintáctico(encontrado lexema de a, esperaba ⋃_{i=1}^{n} PREDICT( A → αi ) )
      FINSI
   FINSI
HASTA A = $              /* La pila está vacía */
```

EJEMPLO:

Para hacer una traza de este analizador sintáctico sobre la gramática LL(1) de las expresiones aritméticas tomamos la tabla de analisis del ejemplo anterior y aplicaremos el algoritmo anterior a la siguiente entrada: "`id + id * id $`".

PILA	ENTRADA	SALIDA
$ E	id + id * id $	E → TE'
$ E' T	id + id * id $	T → FT'
$ E' T' F	id + id * id $	F → id
$ E' T' id	id + id * id $	emparejar id
$ E' T'	+ id * id $	T → ε
$ E'	+ id * id $	E' → +TE'
$ E' T +	+ id * id $	emparejar +
$ E' T	id * id $	T → FT'
$ E' T' F	id * id $	F → id
$ E' T' id	id * id $	emparejar id
$ E' T'	* id $	T' → *FT
$ E' T' F *	* id $	emparejar *
$ E' T' F	id $	F → id
$ E' T' id	id $	emparejar id
$ E' T'	$	T' → ε
$ E'	$	E' → ε
$	$	aceptar!

ÁRBOL DE ANÄLISIS SINTÁCTICO



La columna de la izquierda muestra el contenido de la pila en cada momento, quedando el fondo a su izquierda y la cima a su derecha, donde se van colocando los símbolos de las partes derechas de las producciones en orden inverso a como aparecen en dichas producciones.

La columna central representa lo que en cada paso resta por analizar de la cadena de entrada. El primer símbolo de la izquierda representa el símbolo de preanálisis. Cuando este símbolo coincide el terminal de la pila se eliminan ambos (el tope y el preanálisis) avanzando el análisis al siguiente símbolo de la entrada.

La columna de salida muestran los emparejamientos de *tokens* que se realizan y las producciones que se van aplicando de la tabla de análisis para ir actualizando el contenido de la pila y construyendo el árbol de análisis sintáctico que se muestra a la derecha de la tabla.

❑

EJEMPLO:

Sea la gramática siguiente con sus conjuntos de predicción ya calculados

S → C A B { c }
S → a C b { a }
A → a S d { a }
A → ε { b , d , $ }
B → b { b }
B → ε { d , $ }
C → c { c }

La tabla de análisis resulta

	a	b	c	d	$
S	S → a C b	*e*	S → C A B	*e*	*e*
A	A → a S d	A → ε	*e*	A → ε	A → ε
B	*e*	B → b	*e*	B → ε	B → ε
C	*e*	*e*	C → c	*e*	*e*

Y la traza del análisis para la cadena "`cacdb`" sería:

PILA	ENTRADA	ACCIÓN
$ S	c a c d b $	S → C A B
$ B A C	c a c d b $	C → c
$ B A c	c a c d b $	emparejar c
$ B A	a c d b $	A → a S d
$ B d S a	a c d b $	emparejar a
$ B d S	c d b $	S → C A B
$ B d B A C	c d b $	C → c
$ B d B A c	c d b $	emparejar c
$ B d B A	d b $	A → ε
$ B d B	d b $	B → ε
$ B d	d b $	emparejar d
$ B	b$	B → b
$ b	b$	emparejar b
$	$	aceptar!

❑

4.6 MANIPULACIÓN DE ERRORES

Durante la ejecución de un ASDR, se detecta un error cuando durante la ejecución se llega a una llamada al procedimiento de *Error* y en el dirigido por tabla, cuando el terminal del tope de la pila no coincide con el símbolo de preanálisis o si se accede a una celda vacía (por tanto, etiquetada como *error*).

Ya se ha indicado que los mensajes de error constan típicamente del último lexema que entregó el analizador léxico (el del *token* de preanálisis), como causante del error, y una indicación de los lexemas de los *tokens* que esperaba en su lugar (los terminales que aparezcan en la unión de todos los conjuntos de predicción de las reglas del no terminal al que pertenece la función).

La posibilidad más sencilla en el diseño de todo compilador es detener el proceso de compilación al detectar el primer error, pero también existen muchas técnicas para recuperarse de un error y seguir analizando, lo cual es de gran interés en compiladores en línea, como se vio en la introducción.

Conjuntos de sincronización

Una posible estrategia para recuperarse de un error es seguir avanzando por la entrada hasta encontrar algún símbolo que pertenezca a un determinado conjunto de símbolos de sincronización. La construcción de este conjunto se basa en diversas técnicas empíricas:

- Para el no terminal A que se define en la producción en la que se ha producido el error poner en el conjunto todo los símbolos de SIGUIENTES(A);
- Poner también los de PRIMEROS(A);
- Si se puede generar la cadena vacía, tomarla por omisión; etc.

En general, los sistemas de recuperación de errores están llenos de reglas empíricas y soluciones *ad hoc*, por lo que hay mucho escrito sobre estas técnicas, pero nada que sea definitivo en cuanto a robustez y fiabilidad.

EJERCICIOS RESUELTOS Y PROPUESTOS

Ejercicio 1: Compruébese que la siguiente gramática es LL(1) sin modificarla (en esta y en el resto de gramáticas de estos ejercicios se supondrá que el símbolo inicial es el primero).

A → B C D
B → a C b
B → ε
C → c A d
C → e B f
C → g D h
C → ε
D → i

Ejercicio 2: ¿La gramática A → B C D
B → b | ε
C → c | ε
D → d | ε es LL(1)?

Ejercicio 3: De un simple vistazo se puede comprobar que la siguiente gramática no es LL(1):

S → S inst
S → S var D
S → ε
D → D ident E
D → D ident sep
D → int
D → float
E → S fproc

Elimínese la recursividad izquierda y los factures comunes por la izquierda y compruébese si la gramática equivalente resultante cumple la condición LL(1).

Ejercicio 4: Dada la siguiente gramática:

S → A B
A → begin S end B theend
A → ε
B → var L : tipo
B → B fvar
B → ε
L → L , id
L → id

- Háganse las transformaciones necesarias para eliminar la recursividad izquierda.
- Calcúlense los conjuntos de PRIMEROS y SIGUIENTES de cada no terminal.
- Compruébese que la gramática modificada cumple la condición LL(1).
- Constrúyase la tabla de análisis sintáctico LL(1) para esa nueva gramática.
- Finalmente hágase la traza del análisis de la cadena: "`begin var a,b:tipo var c:tipo fvar end var d:tipo theend`", comprobando que las derivaciones son correctas mediante la construcción del árbol de análisis sintáctico.

Ejercicio 5: Dada la siguiente gramática:

S → S inst
S → T R V
T → tipo
T → ε
R → blq V fblq
R → ε
V → ε
V → id S fin
V → id ;

Constrúyase un analizador sintáctico descendente recursivo (ASDR) para la gramática LL(1) equivalente a esta gramática. Supóngase que ya existen las funciones `analex` (analizador léxico), `emparejar` (función de emparejamiento) y `error` (emisión de error sintáctico). La función `error` se debe llamar con una cadena de caracteres que indique exactamente qué lexema ha provocado el error y qué símbolos se esperaban en lugar del encontrado en la entrada.

Ejercicio 6: Háganse las mismas operaciones que en el ejercicio anterior para la gramática:

E → [L
E → a
L → E Q
Q → , L
Q →]

Con los mismos supuestos y condiciones que en aquel caso. Escríbase la secuencia de llamadas recursivas a las funciones que haría el ASDR durante su ejecución, incluidas las llamadas a la función de emparejamiento, para la cadena de entrada "`[a,a]`".

Ejercicio 7: Dada la siguiente gramática:

P → D S
D → D V
D → ε
S → S N
S → ε
V → decl id ;
V → decl id (P) ;
V → decl [D] id ;
N → id ;
N → begin P end

- Háganse las transformaciones necesarias para que cumpla la condición LL(1).
- Constrúyase la tabla de análisis sintáctico LL(1) para esa nueva gramática.
- Finalmente hágase la traza del análisis de las cadenas: "`decl id ( decl [ decl id ; ] id ; ) ; id ;`" y "`decl id ( begin id ; )`". Si durante el análisis se produjera algún error sintáctico indíquese qué símbolos podrían esperarse en lugar del *token* de preanálisis que entró.