Tema 3.
Análisis Sintáctico.

Todo lenguaje de programación tiene reglas que describen la estructura sintáctica de programas bien formados. En Pascal, por ejemplo, un programa se compone de bloques, un bloque de proposiciones, una proposición de expresiones, una expresión de componentes léxicos, y así sucesivamente. Se puede describir la sintaxis de las construcciones de los lenguajes de programación por medio de gramáticas de contexto libre o notación BNF (Backus-Naur Form).

Las gramáticas ofrecen ventajas significativas a los diseñadores de lenguajes y a los desarrolladores de compiladores.

- Las gramáticas son especificaciones sintácticas y precisas de lenguajes de programación.
- A partir de una gramática se puede generar automáticamente un analizador sintáctico.
- El proceso de construcción puede llevar a descubrir ambigüedades.
- Una gramática proporciona una estructura a un lenguaje de programación, siendo más fácil generar código y detectar errores.
- Es más fácil ampliar/modificar el lenguaje si está descrito con una gramática.

La mayor parte de este tema está dedicada a los métodos de análisis sintáctico de uso típico en compiladores. Primero se introducen los conceptos básicos, después las técnicas adecuadas para la aplicación manual. Además como los programas pueden contener errores sintácticos, los métodos de análisis sintáctico se pueden ampliar para que se recuperen de los errores sintácticos más frecuentes.

✯ ¿Qué es el analizador sintáctico ?

Es la fase del analizador que se encarga de chequear el texto de entrada en base a una gramática dada. Y en caso de que el programa de entrada sea válido, suministra el árbol sintáctico que lo reconoce.

En teoría, se supone que la salida del analizador sintáctico es alguna representación del árbol sintáctico que reconoce la secuencia de tokens suministrada por el analizador léxico.

En la práctica, el analizador sintáctico también hace:

- Acceder a la tabla de símbolos (para hacer parte del trabajo del analizador semántico).
- Chequeo de tipos (del analizador semántico).
- Generar código intermedio.
- Generar errores cuando se producen.

En definitiva, realiza casi todas las operaciones de la compilación. Este método de trabajo da lugar a los métodos de compilación dirigidos por sintaxis.

✯ Manejo de errores sintácticos

Si un compilador tuviera que procesar sólo programas correctos, su diseño e implantación se simplificarían mucho. Pero los programadores a menudo escriben programas incorrectos, y un buen compilador debería ayudar al programador a identificar y localizar errores. Es más, considerar desde el principio el manejo de errores puede simplificar la estructura de un compilador y mejorar su respuesta a los errores.

Los errores en la programación pueden ser de los siguientes tipos:

- Léxicos, producidos al escribir mal un identificador, una palabra clave o un operador.
- Sintácticos, por una expresión aritmética o paréntesis no equilibrados.
- Semánticos, como un operador aplicado a un operando incompatible.
- Lógicos, puede ser una llamada infinitamente recursiva.

El manejo de errores de sintaxis es el más complicado desde el punto de vista de la creación de compiladores. Nos interesa que cuando el compilador encuentre un error, se recupere y siga buscando errores. Por lo tanto el manejador de errores de un analizador sintáctico debe tener como objetivos:

- Indicar los errores de forma clara y precisa. Aclarar el tipo de error y su localización.
- Recuperarse del error, para poder seguir examinando la entrada.
- No ralentizar significativamente la compilación.

Un buen compilador debe hacerse siempre teniendo también en mente los errores que se pueden producir; con ello se consigue:

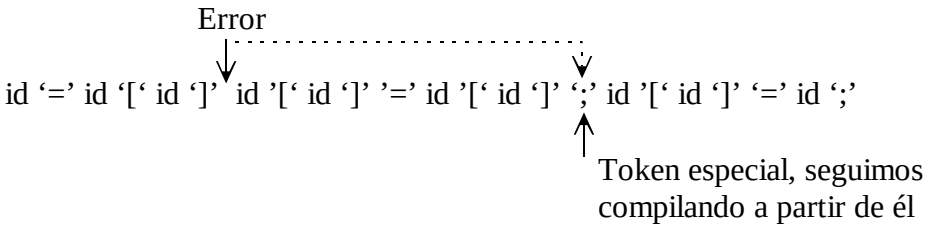
- Simplificar la estructura del compilador.
- Mejorar la respuesta ante los errores.

Tenemos varias estrategias para corregir errores, una vez detectados:

- *Ignorar el problema (Panic mode)*: Consiste en ignorar el resto de la entrada hasta llegar a una condición de seguridad. Una condición tal se produce cuando nos encontramos un token especial (por ejemplo un ‘;’ o un ‘END’).A partir de este punto se sigue analizando normalmente.

Ejemplo:

```
aux = a[i]
a[i] = a[j];
a[j] = aux;
```



- *Recuperación a nivel de frase*: Intenta recuperar el error una vez descubierto. En el caso anterior, por ejemplo, podría haber sido lo suficientemente inteligente como para insertar el token ';' . Hay que tener cuidado con este método, pues puede dar lugar a recuperaciones infinitas.

- *Reglas de producción adicionales para el control de errores*: La gramática se puede aumentar con las reglas que reconocen los errores más comunes. En el caso anterior, se podría haber puesto algo como:

  sent_errónea → sent_sin_acabar sentencia_acabada ';'
  sentencia_acabada → sentencia ';'
  sent_sin_acabar → sentencia

  Lo cual nos da mayor control en ciertas circunstancias

- *Corrección Global* : Dada una secuencia completa de tokens a ser reconocida, si hay algún error por el que no se puede reconocer, consiste en encontrar la secuencia completa más parecida que sí se pueda reconocer. Es decir, el analizador sintáctico le pide toda la secuencia de tokens al léxico, y lo que hace es devolver lo más parecido a la cadena de entrada pero sin errores, así como el árbol que lo reconoce.

Análisis Sintáctico.

✯ Tipo de gramática que acepta un analizador sintáctico

Nosotros nos centraremos en el análisis sintáctico para lenguajes basados en gramáticas formales, ya que de otra forma se hace muy difícil la comprensión del compilador, y se pueden corregir, quizás más fácilmente, errores de muy difícil localización, como es la ambigüedad en el reconocimiento de ciertas sentencias.

La gramática que acepta el analizador sintáctico es una gramática de contexto libre:

- *Gramática* : G (N, T, P, S)
  N = No terminales.
  T = Terminales.
  P = Reglas de Producción.
  S = Axioma Inicial.

Ejemplo : Se considera la gramática que reconoce las operaciones aritméticas.

① E → E + T
② | T
③ T → T * F
④ | F
⑤ F → ID
⑥ | NUM
⑦ | (E)

En el que: N = {E, T, F} están a la izquierda de la regla.
T = {ID, NUM, (,) ,+ ,*}
P = Son las siete reglas de producción.
S = Axioma inicial. Podría ser cualquiera, en este caso es E.

- *Derivaciones* : La idea central es que se considera una producción como una regla de reescritura, donde el no terminal de la izquierda es sustituido por la cadena del lado derecho de la producción.

De un modo más formal:

$\alpha \Rightarrow \beta$ con $\alpha, \beta \in (N \cup T)^*$, si

$$\left.\begin{array}{l} \alpha \equiv \sigma A \delta \\ \beta \equiv \sigma \tau \delta \end{array}\right\} \text{ con } A \in N;\ \sigma, \tau, \delta \in (N \cup T)^*; \text{ y}$$

$\exists$ regla de producción $A \rightarrow \tau$

- Derivación por la izquierda : Derivación donde solo el no terminal de más a la izquierda de cualquier forma de frase se sustituye en cada paso.
- Derivación por la derecha o Derivación canónica: Derivación donde el no terminal más a la derecha se sustituye en cada paso.

Ejemplo: Sea la gramática

$E \rightarrow E + E$
$\quad | \; E - E$
$\quad | \; E * E$
$\quad | \; E / E$
$\quad | \; E \wedge E$
$\quad | \; id$

Constrúyase una derivación por la izquierda y por la derecha para la siguiente frase:
a * c + b ($id_1 * id_2 + id_3$)

Derivación más a la izquierda:

$E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow id_1 * E + E \Rightarrow id_1 * id_2 + E \Rightarrow id_1 * id_2 + id_3$
↑ ↑ ↑ ↑ ↑

Derivación más a la derecha::

$E \Rightarrow E + E \Rightarrow E + id_3 \Rightarrow E * E + id_3 \Rightarrow E * id_2 + id_3 \Rightarrow id_1 * id_2 + id_3$
↑ ↑ ↑ ↑ ↑

✯ Árbol sintáctico de una sentencia de un lenguaje

Es una representación que se utiliza para describir el proceso de derivación de dicha sentencia.

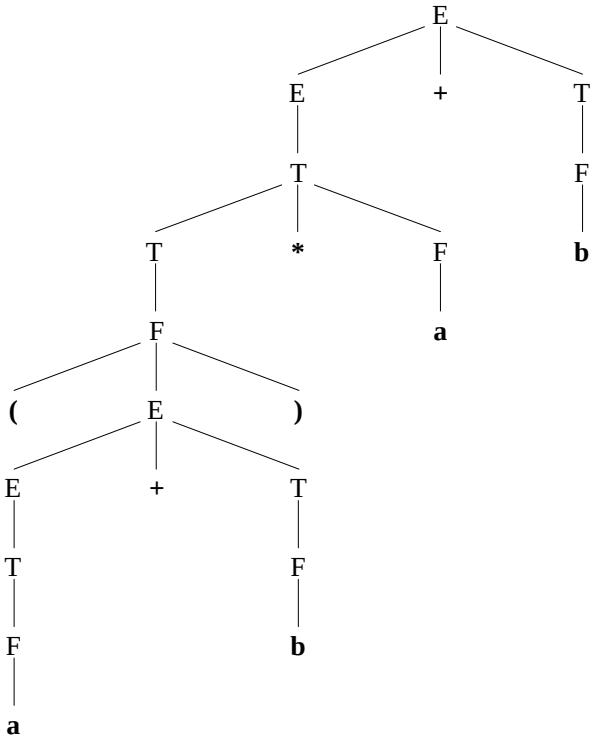
Como nodos internos del árbol, se sitúan los elementos no terminales de las reglas de producción que vayamos aplicando, y tantos hijos como símbolos existan en la parte derecha de dichas reglas.

Veamos un ejemplo: Sea la gramática anterior.

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid a \mid b$

Supongamos que hay que reconocer: (a + b) * a + b

Si el árbol puede construirse, es que la sentencia es correcta:

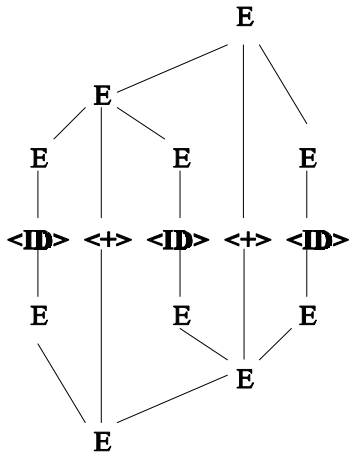


✯ **Ambigüedad**: Una gramática es ambigua si derivando de forma diferente con el mismo tipo de derivación se llega al mismo resultado.

Ejemplo: Considérese la gramática

E → E + E
E → E * E
E → (E)
E → id | num

si tenemos aux + cont + i
<ID><+><ID><+><ID>



Forma Sentencial
Es cualquier secuencia de terminales y no terminales obtenida mediante derivaciones a partir del axioma inicial.

✯ Tipos de Análisis

De la forma de construir el árbol sintáctico se desprenden dos tipos o clases de analizadores sintácticos. Pueden ser descendentes o ascendente

- *Descendentes* : Parten del axioma inicial, y van efectuando derivaciones a izquierda hasta obtener la secuencia de derivaciones que reconoce a la sentencia.

  Pueden ser:
  - Con retroceso.
  - Con recursión.
  - LL(1)

- *Ascendentes*: Parten de la sentencia de entrada, y van aplicando reglas de producción hacia atrás (desde el consecuente hasta el antecedente), hasta llegar al axioma inicial.

  Pueden ser:
  - Con retroceso.
  - LR(1)

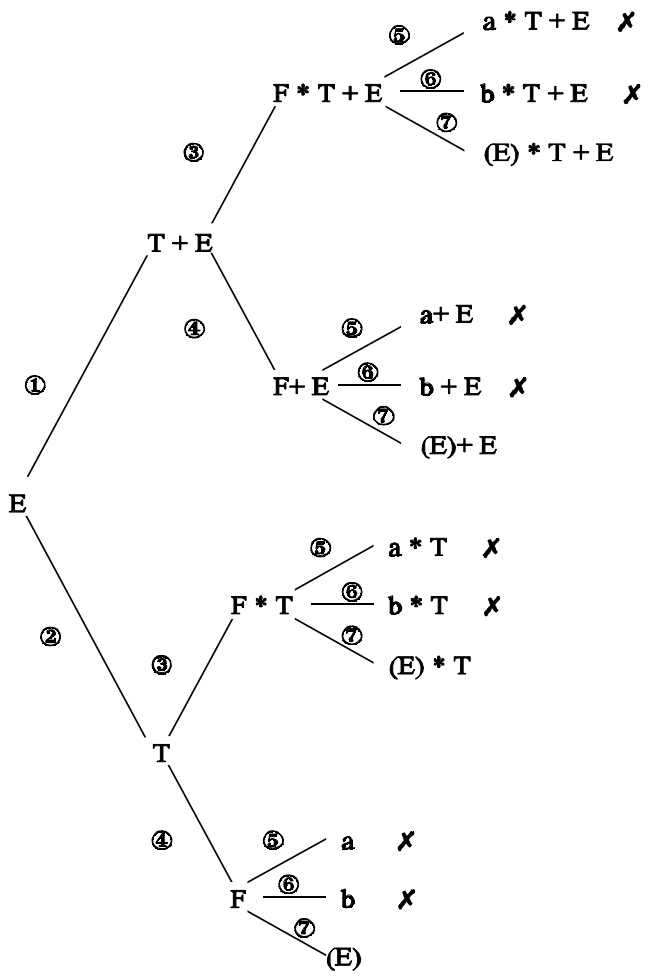
☆ **Análisis descendente con retroceso.**

- *Objetivo* : El método parte del axioma inicial y aplica todas las posibles reglas al no terminal más a la izquierda.

- *Ejemplo*: Utilizaremos la siguiente gramática (No recursiva por la izquierda)

① E → T + E
② E → T
③ T → F * T
④ T → F
⑤ F → a
⑥ F → b
⑦ F → (E)

para reconocer la cadena de entrada: (a + b) * a + b



Mediante este árbol se pueden derivar todas las posibles sentencias reconocibles por esta gramática y el objetivo de este algoritmo es hacer una búsqueda en este árbol de la rama que culmine en la sentencia a reconocer. El mecanismo funciona mediante una búsqueda primero en profundidad.
Mira si todos los tokens a la izquierda de un No Terminal coincide con la cabeza de la secuencia a reconocer.
En todo el árbol de derivaciones, se pretende profundizar por cada rama hasta llegar a encontrar una forma sentencial que no puede coincidir con lo que se busca, en cuyo caso se desecha, o que coincide con lo buscado, momento en que se acepta la sentencia. Si por ninguna rama se puede reconocer, se rechaza la sentencia.

- *Algoritmo* :

Sea $k \in N$ el no terminal más a la izquierda de la forma sentencial.
Sea $\tau \in T^*$ la secuencia de tokens en la izquierda de k.

```
Ensayar (forma_sentencial, entrada)
for {p_i ∈P / p_i ≡ k → α}
        forma_sentencial' = Aplicar(p_i, k, forma_sentencial)
        if τ' = Parte_izquierda (entrada)
                if forma_sentencial == entrada
                        ¡ Sentencia reconocida !
                else
                        Ensayar (forma_sentencial', entrada)
                endif
        endif
endfor;
```

En el programa principal

```
        Ensayar (S, cadena a reconocer)
        if not ¡ Sentencia reconocida !
                ¡¡ Sentencia no reconocida !!
        endif;
```

- *Problemas* : Este método no funciona con gramáticas recursivas a la izquierda, ya que puede ocurrir que entre en un bucle infinito.
  No existen muchos analizadores sintácticos con retroceso. En parte, porque casi nunca se necesita el retroceso para analizar sintácticamente las construcciones de los lenguajes de programación. En casos como el análisis sintáctico del lenguaje natural, el retroceso tampoco es muy eficiente, y se prefieren otros métodos.

Reconocer con el algoritmo (a+b)*a+b

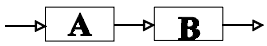
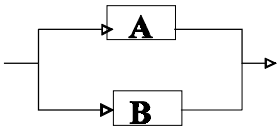
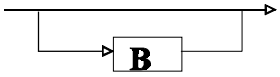
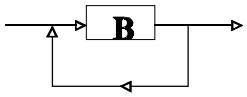
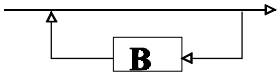
forma_sentencial	Pila (de reglas utilizadas)
E	1
T + E	1-3
F * T +E	1-3-5
a * T + E	1-3
F * T +E	1-3-6
b * T + E	1-3
F * T +E	1-3-7
(E) * T +E	1-3-7-1
(T + E) * T + E	1-3-7-1-3
(F * T + E) * T + E	1-3-7-1-3-5
(a * T + E) * T + E	1-3-7-1-3
(F * T + E) * T + E	1-3-7-1-3-6
(b * T + E) * T + E	1-3-7-1-3
(F * T + E) * T + E	1-3-7-1-3-7
((E) * T + E) * T + E	1-3-7-1-3
(F * T + E) * T + E	1-3-7-1
(T + E) * T + E	1-3-7-1-4
(F + E) * T + E	1-3-7-1-4-5
(a + E) * T + E	1-3-7-1-4-5-1
(a + T + E) * T + E	1-3-7-1-4-5-1-3
(a + F * T + E) * T + E	1-3-7-1-4-5-1-3-5
(a + a * T + E) * T + E	1-3-7-1-4-5-1-3
(a + F * T + E) * T + E	1-3-7-1-4-5-1-3-6
(a + b * T + E) * T + E	1-3-7-1-4-5-1-3
(a + F * T + E) * T + E	1-3-7-1-4-5-1-3-7
(a + (E) * T + E) * T + E	1-3-7-1-4-5-1-3
(a + F * T + E) * T + E	1-3-7-1-4-5-1
(a + T + E) * T + E	1-3-7-1-4-5-1-4
(a + F + E) * T + E	
....	
(a + E) * T + E	1-3-7-1-4-5-2
(a + T) * T + E	
...	

☆ **Análisis descendente con recursión. Diagramas de Conway**.

Una gramática de contexto libre puede expresar un lenguaje al igual que puede hacerlo la notación BNF, y los diagramas de Conway.

- *Definición*: Un diagrama de Conway es un grafo dirigido donde los elementos no terminales aparecen como rectángulos, y los terminales como círculos.

Para demostrar que permite representar las mismas gramáticas que la BNF, se hace por inducción sobre las operaciones básicas de BNF:

Operación	BNF	Diagrama de Comway
Yuxtaposición	AB	
Opción	A|B	
	ε | B	
Repetición	1 o más veces {B}	
	0 o más veces [B]	

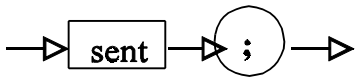
De esta forma todos los posibles caminos desde el inicio del grafo hasta el final, representan formas sentenciales válidas.
En todo diagrama de Conway hay un origen y un destino.

Vamos a intentar establecer una comparación entre el flujo de un programa y el camino que puedo establecer en los diagramas de Conway

Ejemplo: **Yuxtaposición**

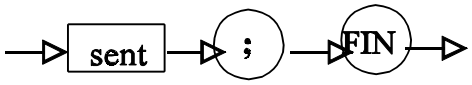
Es importante el ejercicio de programación consistente en seguir la evolución detallada de las llamadas de los procedimientos entre sí.

prog: sent ;



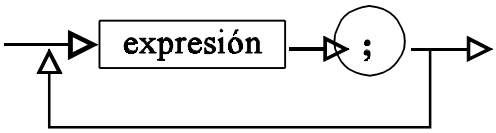

```
Prog ( );
        sent ( );
        if el siguiente token es ';' entonces
                consumir el token
        else
                !Error sintáctico¡
        fi
FinProg
```

prog: sent ; FIN




```
Prog ( );
        sent ( );
        if el siguiente token es ';' entonces
                Consumir el token
        else
                !Error sintáctico¡
        fi
        if el siguiente token es FIN entonces
                Consumir el token
        else
                !Error sintáctico¡
        fi
FinProg
```

Secuencia

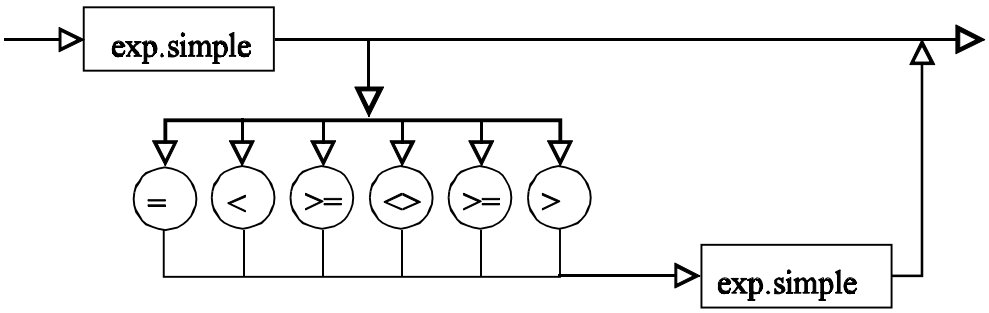



```
main ( ) {
        get_token ( );
        do {
             expresión ( );
             while (token != PUNTOYCOMA) {
                !Error en expresión¡
                get_token ( );
             };
             get_token( );
         }while (token != EOF);
};
```

En este caso se considera al ';' como un token de seguridad, lo que permite hacer una recuperación de errores mediante el método panic mode.

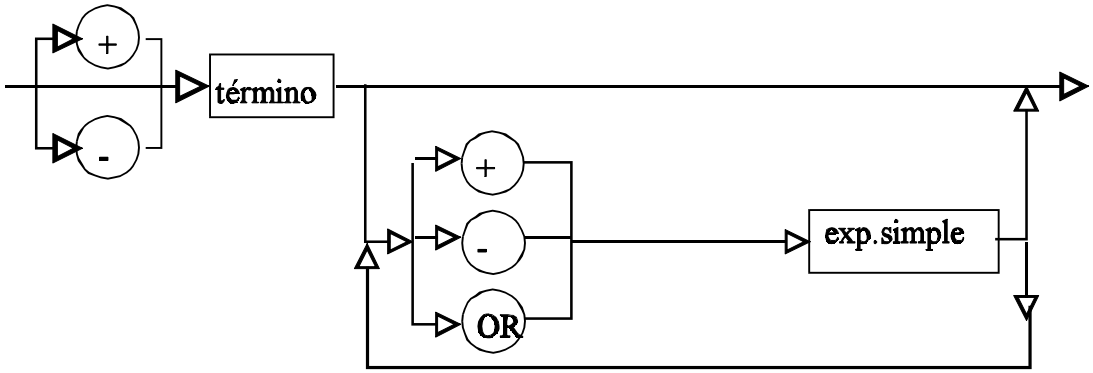
Se supone que el sintáctico pide al lexicográfico tokens a través de get_token ( );, y que el lexicográfico deja el token actual en la variable global token.
Antes de entrar a una función, en token debemos tener el token de lookahead, que esa función necesita consultar.

expresión



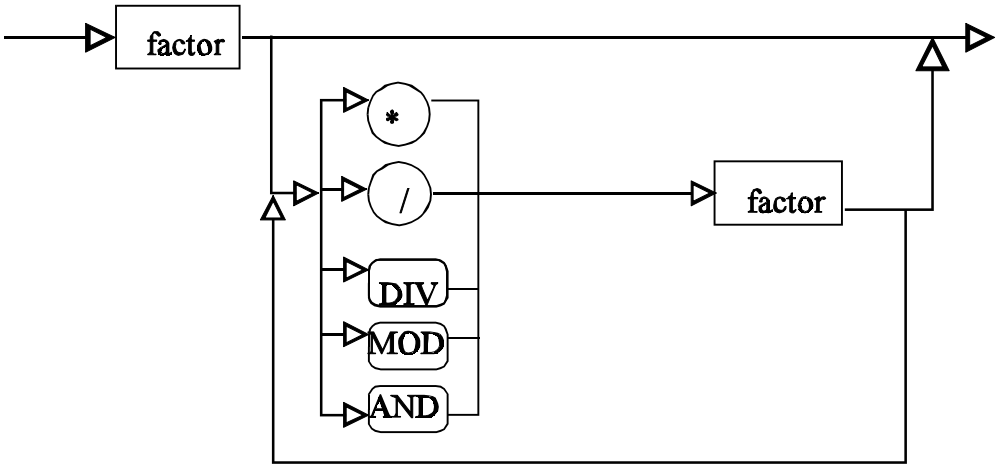

```
expresión ( ){
        expr_simple ( );
        if ((token == IGUAL)|| (token == ME)|| (token == MEI)||
            (token == DIST)|| (token == MAI)|| (token == MA))
                get_token ( ) ;
                expr_simple ( ) ;

        }
}
```

expr_simple



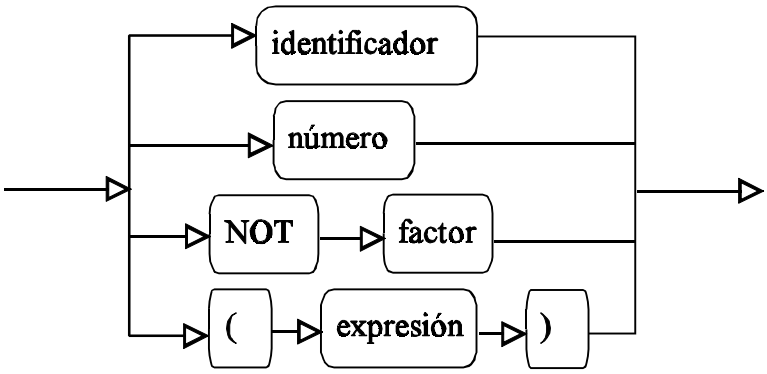

```
expr_simple ( ) {
        if ((token == IGUAL) || (token == MENOS)) {
          get_token( );
        }
        término ( );
        while ((token == MAS) || (token == MENOS) || (token == OR)) {
          get_token( );
          término ( );
        }
}
```

término



```
término ( ){
        factor ( );
        while ((token == POR) || (token == DIV) || (token == DIV_ENT) ||
                (token == MOD) || (token == AND) {
                   get_token( );
                   factor( );
}
```

factor




```
factor ( ){
        switch (token) {
           case ID : get_token ( ); break;
           case NUM : get_token ( ); break;
           case NOT : get_token ( ); factor ( ); break;
           case AB_PARID : get_token ( ); expresión ( );
                                if (token != CE_PAR) {Error: Paréntesis de cierre}
                                else get_token ( );
                                 break;
           default : Error : Expresión no válida.
        }
}
```

✯ Análisis descendente de gramáticas LL(1)

Una gramática LL(1) es aquella en la que su tabla de chequeo de sintaxis no posee entradas múltiples, o sea, es suficiente con examinar sólo un símbolo a la entrada, para saber qué regla aplicar. Toda gramática reconocible mediante el método de los diagramas de Conway es LL(1)

El método consiste en seguir un algoritmo partiendo de:

- La cadena a reconocer, junto con un apuntador, que nos indica cual es el token actual.
- Una pila de símbolos (terminales y no terminales)
- Una tabla asociada de forma unívoca a una gramática. En esta asignatura no vamos a ver como calcular dicha tabla.

La cadena de entrada acabará en el símbolo $, que consideramos como si fuese un EOF(End Of File - Fin de Fichero).

Sea **X** el elemento encima de la pila, y **a**, el apuntado en la entrada. El algoritmo consiste en:

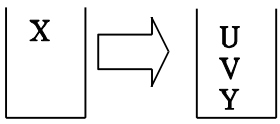
1.- Si X = a = $ entonces ACEPTAR.

2.- Si X = a $\neq$ $ entonces
- se quita X de la pila
- y se avanza el apuntador.

3.- Si X $\in$ T y X $\neq$ a entonces RECHAZAR.

4.- Si X $\in$ N entonces consultamos la entrada M[X,a] de la tabla:
- M[X,a] es vacia : RECHAZAR.
- M [X,a] no es vacia, se quita a X de la pila y se inserta el consecuente en orden inverso.

Ejemplo: Si M[X,a] = {X $\rightarrow$ UVY}, se quita a X de la pila, y se meten UVY en orden inverso :



5.- Ir al paso 1.

Una vez aplicada una regla, no será desaplicada por ningún tipo de retroceso.

El algoritmo comienza con $ y con el axioma inicial metidos en la pila.

Este tipo de análisis tiene el inconveniente de que muy pocas gramáticas son LL(1), aunque muchas pueden traducirse a LL(1), tras un adecuado estudio. Por ejemplo:

E → E + T | T
T → T * F | F
F → (E) | id

Esta gramática no es LL(1). La sustituimos por:

E → T E'
E'→ + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

que es equivalente y LL(1)

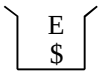
Tabla M :

N \ T	**id**	**+**	*****	**(**	**)**	**$**
E	E → TE'			E → TE'		
E'		E'→ +TE'			E' → ε	E' → ε
T	T → F T'			T → F T'		
T'		T' → ε	T→ * F T'		T' → ε	T' → ε
F	F → id			F → (E)		

Reconocer a* (b + c) ≡ id * (id + id) $

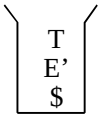
Análisis Descendentes

Inicialmente

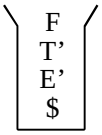


id * (id + id) $
↑

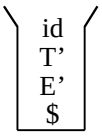
M[E, id] = E → T E’



M[T, id] = T → F T’

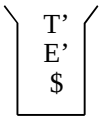


M[F, id] = F → id

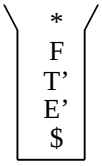


id = id Avanzar apuntador

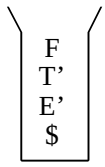
id * (id + id) $
↑



M[T’, *] = T → * F T’

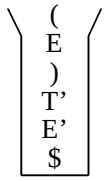


* = * Avanzar apuntador

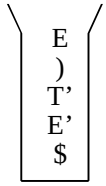


id * (id + id) $
↑

M[F, (] = F → (E)

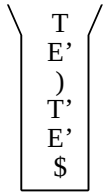


(= (Avanzar apuntador

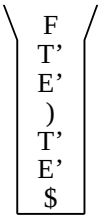


id * (id + id) $
↑

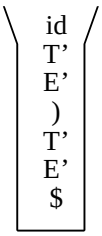
M[E,id] = E → T E’



M[T,id] = T → F T’



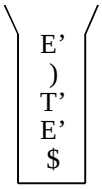
M[F, id] = F →id



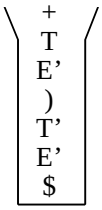
id = id Avanzar apuntador

id * (id + id) $
↑

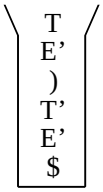
M[T',+] = T' → ε



M[E',+] = E' →+ T E'

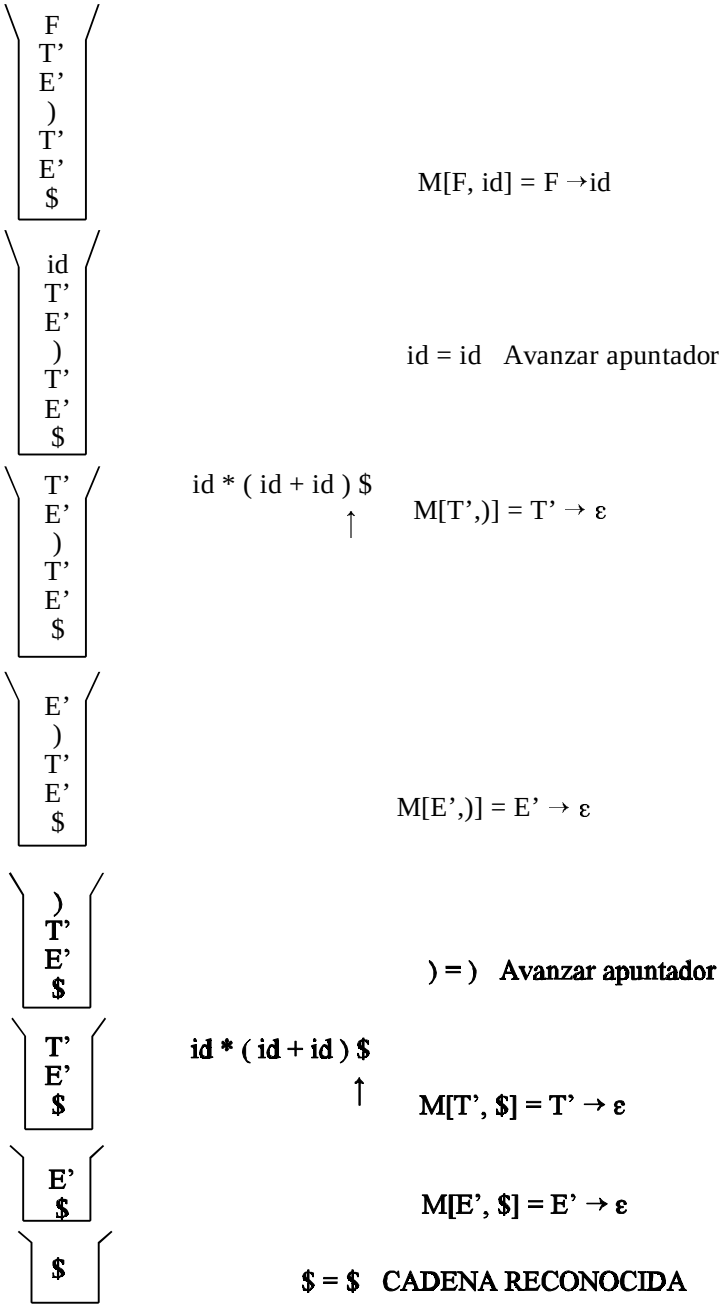


+ = + Avanzar apuntador



id * (id + id) $
↑

M[T,id] = T →F T'



Las reglas que hemos ido aplicando en cada caso, nos van dando el parse izquierdo que reconoce la sentencia.

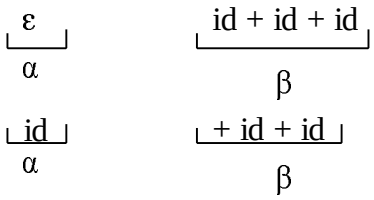
Análisis Ascendentes

Aquí se construye el árbol sintáctico de abajo hacia arriba, lo cual disminuye el número de reglas mal aplicadas con respecto al caso descendente (si hablamos del caso con retroceso).
Tanto si hay retroceso como si no, en un momento dado, la cadena de entrada estará dividida en dos partes α y β:

β: El trozo de la cadena de entrada (secuencia de tokens) por reconocer. Coincidirá siempre con algún trozo de la parte derecha de la cadena de entrada :β ∈ T*.Vamos consumiendo tokens, y todos los tokens que nos queden por consumir constituyen β.

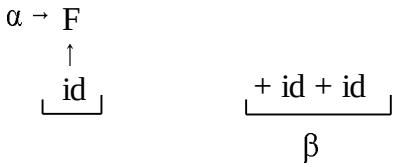
α: coincidirá siempre con el resto de la cadena de entrada, trozo al que se habrán aplicado algunas reglas de producción, α ∈ (N∪T)* en sentido inverso.

Ejemplo: Comienza el análisis sintáctico.

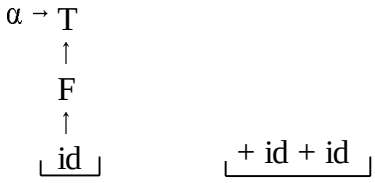


Consumimos el primer token

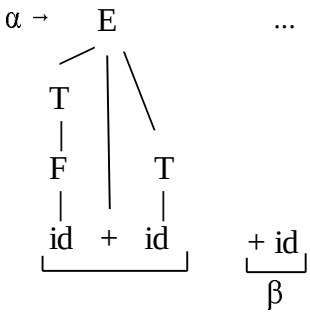
Ahora, α es F



Aplicamos una regla al revés



Forma sentencial β Secuencia de tokens



...

En un momento dado, el analizador sintáctico se encuentra en con un par α, β concreto, al que se llama configuración.

El analizador sintáctico para poder trabajar puede realizar una de las cuatro operaciones siguientes:

- *Aceptar* : Cadena reconocida.
- *Rechazar* : La entrada no es válida.
- *Reducir* : Aplicar una regla de producción a los elementos de α

Por ejemplo si tenemos una configuración:

$$\underbrace{X_1\, X_2 \ldots X_{p+1} \ldots X_m}_{\alpha} \qquad \underbrace{a_{m+1} \ \ldots \ a_n}_{\beta}$$

Donde $X_i \in (N \cup T)$ y $a_i \in T$

Si hay una regla de forma $A_k \rightarrow X_{p+1} \ldots X_m$, se reducirá a

$$\underbrace{X_1\, X_2 \ldots A_k}_{\alpha} \qquad \underbrace{a_{m+1} \ \ldots \ a_n}_{\beta}$$

Volviéndose a aplicar el método.

- Desplazar : Se desplaza el terminal más de la izquierda de β a la derecha de α

$$\underbrace{X_1\, X_2 \ldots X_m}_{\alpha} \qquad \underbrace{X_{m+1}\, X_{m+2} \ldots X_n}_{\beta}$$

Tras desplazar

$$\underbrace{X_1\, X_2 \ldots X_m\, X_{m+1}}_{\alpha} \qquad \underbrace{X_{m+2} \ \ldots \ X_n}_{\beta}$$

Es posible reducir por las reglas ε.

$$A \rightarrow \varepsilon \qquad \underbrace{X_1\, X_2 \ldots X_p\, X_{p+1} \ldots X_m\, \varepsilon}_{\alpha}$$

$$\underbrace{X_1\, X_2 \ldots X_p\, X_{p+1} \ldots X_m\, A}_{\alpha'}$$

Mediante reducciones y desplazamientos, tenemos que llegar a aceptar o rechazar la cadena de entrada. Antes de hacer los desplazamientos tenemos que hacerles todas las reducciones posibles a α. Cuando α es el axioma inicial y β es la tira nula, se acepta la cadena de entrada. Cuando β no es la tira nula o α no es el axioma inicial y no se puede aplicar ninguna regla, entonces se rechaza la cadena de entrada.

✯ Análisis Ascendente con retroceso.

Cuando se da cuenta que llega a una situación en la que no puede continuar, entonces vuelve atrás deshaciendo todos los cambios.
En el análisis con retroceso no se permiten las reglas ε, puesto que estas se podrán aplicar de forma indefinida.

El algoritmo es el siguiente:

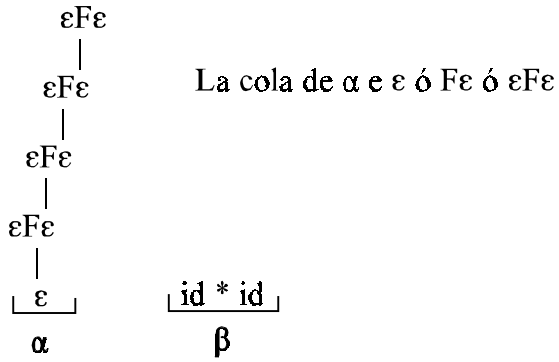
```
Ensayar (α, β)
    for p_i ∈ P hacer
        if consecuente (p_i) == cola(α)
                α' = Reducir la cola de α por p_i
                if (α' == Axioma inicial) AND (β == ε)
                        ACEPTAR
                else
                        Ensayar (α', β)
                Endif
        endif
    endfor
    if (β ≠ε)
        α', β' = Desplazar de β a α
        Ensayar (α',  β')
    endif
endEnsayar
```

En el programa principal pondremos:

```
Ensayar(ε, cadena a reconocer);
if NOT se ha aceptado then
        RECHAZAR
endif;
```

Ejemplo: (Vamos a ver por que no se permiten las reglas ε). Supongamos la siguiente gramática.

① E → T + E
② E → T
③ T → F * T
④ T → F
⑤ F → (E)
⑥ F → id
⑦ F → num
⑧ F → ε



No puede aparecer ε en una gramática ascendente con retroceso porque da lugar a recursión infinita.

Ejemplo: Reconocer a*a ≡ id * id, dada la siguiente gramática:

① E → E + T
② E → T
③ T → T * F
④ T → F
⑤ F → (E)
⑥ F → id
⑦ F → num

(Llevaremos una pila para el Backtraking)

Pila	**α**	**β**	**Acción**
-	ε	id * id	Desplazar
-	id	* id	F→ id
6	F	* id	T→ F
6-4	T	* id	E→ T
6-4-2	E	* id	Desplazar
6-4-2	E *	id	Desplazar
6-4-2	E * id	ε	F→ id
6-4-2-6	E * F	ε	T→ F
6-4-2-6-4	E * T	ε	E→ T
6-4-2-6-4-2	E * E	ε	Retroceso
6-4	T	* id	Desplazar
6-4	T *	id	Desplazar
6-4	T * id	ε	F→ id
6-4-6	T * F	ε	T→ T * F
6-4-6-3	T	ε	E→ T
6-4-6-3-2	E	ε	Aceptar

✯ Analizadores LR

Vamos a analizar una técnica eficiente de análisis sintáctico ascendente que se puede utilizar para analizar una amplia clase de gramáticas de contexto libre. La técnica se denomina análisis sintáctico **LR(k)**; la "**L**" es por el examen de la entrada de izquierda a derecha (en inglés, left-to-right), la "**R**" por construir una derivación por la derecha (en inglés, rightmost derivation) en orden inverso, y la *k* por el número de símbolos de entrada de examen por anticipado utilizados para tomar las decisiones del análisis sintáctico. Cuando se omite, se asume que *k*, es 1. El análisis LR es atractivo por varias razones.

- Pueden reconocer la inmensa mayoría de los lenguajes de programación que puedan ser generados mediante gramáticas de contexto-libre.
- El método de funcionamiento de estos analizadores posee la ventaja de localizar un error sintáctico en el mismo instante que se produce con lo que se adquiere una gran eficiencia de tiempo de compilación frente a procedimientos menos adecuados como puedan ser los de retroceso.

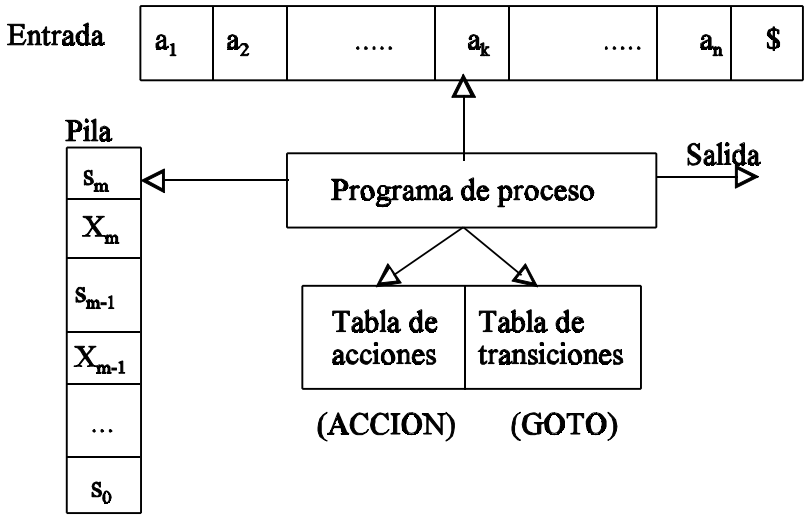
El principal inconveniente del método es que supone demasiado trabajo construir un analizador sintáctico LR a mano para una gramática de un lenguaje de programación típico. Se necesita una herramienta especializada - un generador de analizadores sintácticos LR - . Por fortuna, existen disponibles varios de estos generadores. Más adelante estudiaremos el diseño y uso de uno, el programa YACC. Con este generador se puede escribir una gramática de contexto libre y el generador produce automáticamente un analizador sintáctico de dicha gramática. Si la gramática contiene ambigüedades u otras construcciones difíciles de analizar en un examen de izquierda a derecha de la entrada, el generador puede localizar dichas construcciones e informar al diseñador del compilador de su presencia.

Existen tres técnicas para construir una tabla de análisis sintáctico LR para una gramática. El primer método, llamado LR sencillo (SLR, en inglés) es el más fácil de implantar, pero el menos poderoso de los tres. Puede que no consiga producir una tabla de análisis sintáctico para algunas gramáticas que otros métodos si consiguen. El segundo método, llamado LR canónico, es el más poderoso y costoso. El tercer método, llamado LR con examen por anticipado (LALR, en inglés), está entre los otros dos en cuanto a poder y costo. El método LALR funciona con las gramáticas de la mayoría de los lenguajes de programación y, con un poco de esfuerzo, se puede implantar en forma eficiente.

Funcionalmente hablando, un analizador LR consta de dos partes diferenciadas, un *programa de proceso* y una *tabla del análisis.*

El programa de proceso posee como veremos seguidamente un funcionamiento muy simple y permanece invariable de analizador a analizador. Según sea la gramática a procesar deberá variarse el contenido de la tabla de análisis que es la que identifica plenamente al analizador.

La figura muestra un esquema sinóptico de la estructura general de una analizador LR.



Como puede apreciarse en la figura, el analizador consta de una tira de entrada donde se encuentra la cadena a reconocer finalizada con el símbolo $ que representa el delimitador. Esta tira lee de izquierda a derecha un símbolo cada vez en el proceso de reconocimiento.

El contenido de la pila tiene la forma

$$s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m$$

donde el símbolo s_m se encuentra en la cabeza tal y como se muestra en la figura. Cada uno de los X_i son símbolos de la gramática y a los s_i vamos a denominarlos estados del analizador.

Los estados se utilizan para representar toda la información contenida en la pila y situada antes del propio estado. Es mediante el estado en cabeza de la pila por el que se decide qué reducción ha de efectuarse o bien qué desplazamiento.

Tradicionalmente, una tabla de análisis para un reconocedor LR consta de dos partes claramente diferenciadas entre sí que representan dos funciones, la función GOTO y la función ACCION. Seguidamente estudiaremos los cometidos de ambas acciones.

El funcionamiento del analizador LR es el siguiente

1.- Se determina el estado s_m en cabeza de la pila y el símbolo actual a_i en el instante de la cadena de entrada.

2.- Se consulta en la tabla de análisis la función acción con los parámetros anteriores y que puede dar como resultado.

$$\text{Acción } (s_m, a_i) = \begin{cases} \text{Desplazar S} \\ \text{Reducir A} \rightarrow \beta \\ \text{Aceptar} \\ \text{Rechazar} \end{cases}$$

por su parte la función GOTO actúa igualmente con un estado y un símbolo de la gramática produciendo un nuevo estado.

Análogamente puede definirse una *configuración de un analizador LR* como un par de la forma

$$(s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_m\ s_m\ \ ,\ \ a_i\ a_{i+1}\ \ldots\ a_n\ \$)$$

es decir, el primer componente es el contenido actual de la pila, y el segundo la subtira de entrada que resta por reconocer, a_i es el símbolo de entrada actual de análisis.

El movimiento del analizador se realiza teniendo en cuenta:

1. El símbolo leído a_i.
2. El símbolo en cabeza de la pila s_m.

Actuando con la función acción y dependiendo de las cuatro posibles alternativas pueden obtenerse las configuraciones que seguidamente se detallan.

1. Si *acción* (s_m, a_i) = *desplazar s.*
entonces se introducen en la pila el símbolo actual analizado de la cadena de entrada y en la cabeza de la pila el nuevo estado obtenido mediante la función GOTO(s_m, a_i) = S.

La configuración así obtenida es la mostrada seguidamente.

$$(s_0\ X_1\ s_1\ X2\ s_2\ \ldots\ X_m\ s_m\ a_i\ \ s\ \ ,\ \ \ a_{i+1}\ \ldots\ a_n\ \$)$$

pasando s a estar situado en cabeza de la pila y a_{i+1} el siguiente símbolo a explorar en la cinta de entrada.

2. Si *acción* (s_m, a_i) = *reducir* $A \rightarrow \beta$
entonces el analizador ejecuta la reducción oportuna donde el nuevo estado en cabeza de la pila se obtiene mediante la función GOTO(s_{m-r}, a_i) = *s* donde r es precisamente la longitud de la cadena β reducida.

Aquí el analizador extrajo primero *2r* símbolos de la pila (r símbolos de estados y r símbolos de la gramática), exponiendo el estado s_{m-r}. Luego introdujo A, el lado izquierdo de la regla de producción, y s, la entrada de GOTO($s_{m-r,}$A), en la pila.

La configuración así obtenida es la mostrada seguidamente.

$$(s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_{m-r}\ s_{m-r}\ A\ \ s\ \ ,\ \ \ a_i\ a_{i+1}\ \ldots\ a_n\ \$)$$

donde *s* es el nuevo estado en cabeza de la pila y no se ha producido variación en la tira de entrada que aun queda por analizar.

3. Si *acción (s_m, a_i) = aceptar*
entonces se ha llegado a la finalización en el proceso de reconocimiento y el análisis termina reconociendo la tira de entrada.

4. Si *acción (s_m, a_i) = error*
entonces es muestra de que el analizador LR ha descubierto un error sintáctico y procederá en consecuencia activando las rutinas de corrección de errores. Una de las ventajas de este tipo de análisis es que , cuando se produce una acción de error, el token erróneo suele estar al final de α o al principio de β, lo que permite depurar con cierta facilidad las cadenas de entrada (programas).

La configuración inicial del analizador es

$$(s_0 \ , \ a_1 \ a_2 \ ... \ a_n \ \$)$$

donde s_0 es el estado inicial del reconocedor.

Los sucesivos movimientos se realizan en base a los cuatro puntos anteriores hasta que se acepta la cadena de entrada o bien hasta la aparición de un error.

Ejemplo: Expresiones aritméticas y tablas LR
Sea la gramática ya conocida de generación de expresiones aritméticas.

① S → S + T
② S → T
③ T → T * F
④ T → F
⑤ F → (S)
⑥ F → id

La figura siguiente muestra la tabla de análisis con las funciones ACCIÓN y GOTO para la gramática anterior.

	función ACCIÓN						función GOTO		
ESTADO	id	+	*	(	)	$	S	T	F
0	D5			D4			1	2	3
1		D6				ACEP			
2		R2	D7		R2	R2			
3		R4	R4		R4	R4			
4	D5			D4			8	2	3
5		R6	R6		R6	R6			
6	D5			D4				9	3
7	D5			D4					10
8		D6			D11				
9		R1	D7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

El significado de las entradas de la tabla anterior es la siguiente :

1. Di significa desplazar y meter en la pila el estado i,

2. Rj significa reducir por la regla de producción con número j,

3. ACEP significa aceptar,

4. las entradas en blanco significan un error sintáctico.

En la siguiente figura muestra un ejemplo de reconocimiento para la cadena de entrada

a * (a + a) ≡ id * (id + id)

(suponemos que el estado s_0 queda representado por 0).

PASO	PILA	CADENA DE ENTRADA
1	0	id * (id + id)$
2	0 a 5	* (id + id) $
3	0 F 3	* (id + id) $
4	0 T 2	* (id + id) $
5	0 T 2 * 7	(id + id) $
6	0 T 2 * 7 (4	id + id) $
7	0 T 2 * 7 (4 a 5	+ id) $
8	0 T 2 * 7 (4 F 3	+ id) $
9	0 T 2 * 7 (4 T 2	+ id) $
10	0 T 2 * 7 (4 S 8	+ id) $
11	0 T 2 * 7 (4 S 8 + 6	id) $
12	0 T 2 * 7 (4 S 8 + 6 a 5	) $
13	0 T 2 * 7 (4 S 8 + 6 F 3	) $
14	0 T 2 * 7 (4 S 8 + 6 T 9	) $
15	0 T 2 * 7 (4 S 8	) $
16	0 T 2 * 7 (4 S 8) 11	$
17	0 T 2 * 7 F 10	$
18	0 T 2	$
19	0 S 1	$
20	aceptación de la cadena	

Análisis Ascendentes

Hemos estudiado como funcionan los analizadores LR mediante la utilización de sus correspondientes tablas de análisis. En las líneas de reconocimiento anteriores puede observarse como el estado que siempre se encuentra en cabeza de la pila señala en todo momento la información necesaria para la reducción, si esto procede.

Hemos dejado al margen intencionadamente la estructura del programa de proceso puesto que se trata esencialmente de un autómata finito con pila.

En general puede afirmarse que, dada la estructura de los analizadores LR, con la sola inspección de k símbolos de la cadena de entrada a la derecha del símbolo actual puede decidirse con toda exactitud cual es el movimiento (reducción, desplazamiento, etc) a realizar. Es por este motivo por lo que suele denominarse a este tipo de gramáticas como LR(k). Como ya se comentó, en la práctica casi todos los lenguajes de programación pueden ser analizados mediante gramáticas LR(0) o LR(1).

Las tablas LR(1) ideadas por Knuth en 1965 son demasiado grandes para las gramáticas de los lenguajes de programación. En 1969 De Remer y Korenjack descubrieron formas de compactar estas tablas, haciendo práctico y manejable este tipo de parser. El algoritmo es el mismo ya visto. Hay para ellos las gramáticas Simples- LR (SLR) o bien Look-Ahead LR (LALR) estando las gramáticas incluidas de la siguiente forma:

$$LR(k) \supset LALR(k) \supset SLR(k)$$

pero no sus lenguajes respectivos que coinciden en sus conjuntos.

El metacompilador YACC utiliza el análisis LALR(1). El autómata debe de mantener información de su configuración (α), y para mantener información sobre β se comunica con LEX, quién se encarga de la metacompilación a nivel léxico.

- **Consideraciones sobre el análisis ascendente.**

  ¿Cuando usar recursión a derecha o a izquierda?

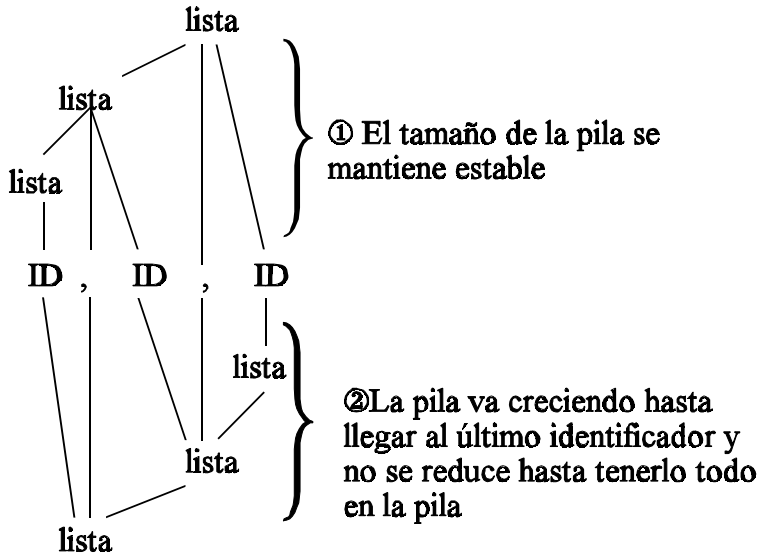
  Ejemplo : Reconocer identificadores separados por comas

  id, id, id, ... , id

  podríamos optar por 2 gramáticas diferentes (① y ②):

  ① lista → ID
  | lista ',' ID
  ② lista → ID
  | ID ',' lista

  ¿Qué diferencia hay entre ① y ②?



① Alternativamente se va desplazando y reduciendo con lo que el tamaño de la pila se mantiene siempre estable (es más conveniente, pero no siempre se puede aplicar la recursión a la izquierda).

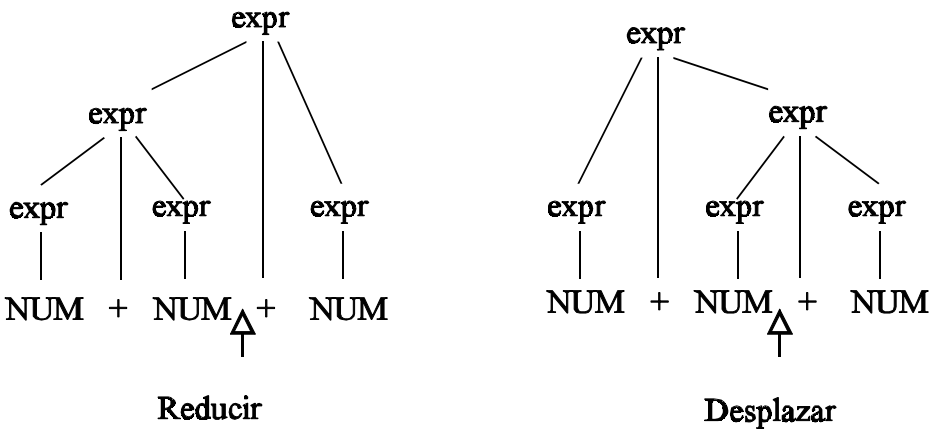
② Si hacemos la recursión a derecha, siempre existe la posibilidad de que se desborde la pila.

Por la forma de construir las tablas pueden aparecer conflictos :

- Shift/Reduce
- Reduce/Reduce

- *Shift/Reduce* :aparece cuando en la tabla de acciones hay que poner una R de reducir y una D de desplazar, el conflicto es que el programa no sabe si reducir o desplazar.

Ej : expr → expr '+' expr
| NUM



En gramáticas con ambigüedad se produce el conflicto shift/reduce (relacionado con el hecho de considerar la gramática como asociativa a izquierda o asociativa a derecha).

- Reduce/Reduce : Se pueden utilizar dos reglas para reducir y no sabe cual elegir

Ejemplo: S → aaBdd
S → aCd
B → a
C → aa

Esta gramática reconoce estas dos secuencia de tokens

aaadd
aaad

Suponemos la secuencia aaad

Aquí aunque lo correcto sería reducir por C, como el reconocimiento LALR(1) sólo permite ver un token a la entrada, pues sólo se ve la d. Como no se sabe si detrás hay otra d o no, pues es imposible tomar una decisión. Estamos ante una gramática LALR(2).

Los conflictos reduce/reduce se pueden eliminar en la mayoría de los casos. Ej:

S → aaBdd S → aCd B → a C → aa	S → aaadd S → aaad

En este caso lo que se ha hecho ha sido añadir las secuencias que producen el conflicto como parte de la gramática.