Análisis sintáctico

Funciones del análisis sintáctico

Analizar sintácticamente una *tira* o cadena de tokens no es más que encontrar para ella el árbol sintáctico o de derivación que tiene como raíz el axioma de la gramática, y como nodos terminales la sucesión ordenada de símbolos que componen la cadena analizada. En caso de no existir este árbol sintáctico, la cadena no pertenecerá al lenguaje, y el analizador sintáctico ha de emitir el correspondiente mensaje de error.

Existen dos formas de analizar sintácticamente una cadena:

- **Análisis descendente**: partiendo del axioma inicial de la gramática se va descendiendo utilizando las derivaciones izquierdas hasta llegar a construir la cadena analizada.
- **Análisis ascendente**: se va construyendo el árbol desde sus nodos terminales. Es decir, se construye desde los símbolos de la cadena hasta llegar al axioma de la gramática. En este caso, se emplean normalmente las derivaciones más a la derecha hasta la localización de la raíz.

  Los principales métodos de análisis sintáctico son:

  - Análisis descendente
    - Análisis descendente con retroceso
    - Análisis de gramáticas **LL**
  - Análisis ascendente
    - Análisis ascendente con retroceso
    - Análisis de gramáticas de precedencia simple
    - Análisis de gramáticas de precedencia generalizada.
    - Análisis de gramáticas por el método mixto
    - Análisis de gramáticas de precedencia de operador
    - Análisis de gramáticas **LR**

Los *análisis con retroceso* se basan en la prueba sistemática de todas las alternativas posibles, dando marcha atrás tan pronto como se detecte que el camino seguido es erróneo. Pueden usarse para cualquier gramática de contexto libre, aunque tienen tres grandes inconvenientes:

1. Emplean mucho más tiempo para el análisis que los demás analizadores, dependiendo éste incluso de la ordenación de las reglas gramaticales.
2. No dan un buen diagnóstico de los errores que encuentran
3. Complican la generación de código cuando ésta se realiza al par que el análisis sintáctico.

Los métodos más eficientes de análisis (tanto ascendente como descendente) no funcionan para todas las gramáticas de contexto libre, sino sólo para las gramáticas que cumplen unas determinadas condiciones.

Afortunadamente, en la mayoría de los casos, pueden encontrase para los lenguajes de programación gramáticas de tipo **LL** o **LR** que los generen.

Para representar el árbol sintáctico que conduce hasta una cadena se asigna a cada regla de la gramática un número. Se define el *parse* como la secuencia ordenada de números (de reglas) aplicadas para construir dicho árbol.

Hay dos tipos de *parse*, que son:

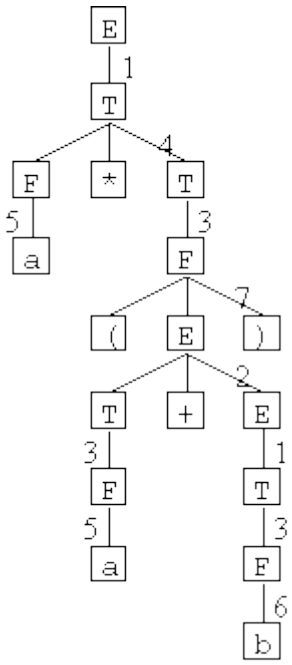
- El **parse-izquierdo**: son los números de las reglas de derivación izquierda utilizadas para generar la cadena a partir del axioma, por tanto correspondiente a un *análisis descendente*.
- El **parse-derecho**: son los números de las reglas de derivación derecha utilizadas para generar la cadena a partir del axioma, en orden inverso. El tomar el orden inverso viene condicionado por ser el análisis ascendente el que normalmente utiliza las reglas de derivación derecha, con lo que el orden en el que aparecen al realizar el análisis es **invertido**.

Ejemplos:

Dada la gramática

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

y la sentencia "a*(a+b)"



El parse izquierdo es:

1-4-5-3-7-2-3-5-1-3-6

y el derecho:

5-5-3-6-3-1-2-7-3-4-1

Simultáneamente a la fase de análisis sintáctico, además de reconocer las secuencias de tokens, y analizar su estructura, pueden realizarse una serie de tareas adicionales, como:

- Recopilar información de los distintos tokens y almacenarla en la tabla de símbolos.
- Realizar algún tipo de análisis semántico, tal como la comprobación de tipos.
- Generar código intermedio.
- Avisar de los errores que se detecten.

Tipos de análisis sintáctico. Determinismo.

Análisis descendente con retroceso.

Si bien en la práctica nunca se usa, se estudia a continuación el análisis descendente con retroceso, por ser el más general de los análisis descendentes, y por su valor didáctico.

Para la realización del análisis descendente con retroceso contamos con una gramática, cuyas reglas se enumeran:

$$
\begin{aligned}
&A \dashrightarrow \alpha_1\\
&A \dashrightarrow \alpha_2\\
&\ldots\\
&B \dashrightarrow \beta_1\\
&B \dashrightarrow \beta_2\\
&B \dashrightarrow \beta_3\\
&\ldots\\
&C \dashrightarrow \gamma_1\\
&\ldots
\end{aligned}
$$

y de una cadena, supuestamente perteneciente al lenguaje definido por la gramática:

$$a_1a_2a_3a_4....$$

Paso 1. Inicialmente se parte de la forma sentencial **s**, que contiene únicamente el axioma de la gramática.

Paso 2. Dada una forma sentencial generada anteriormente, se sustituye el símbolo no terminal situado más a la izquierda por la primera de las alternativas posibles para dicho símbolo, generando así una nueva forma sentencial. Anotamos en una pila el número de la regla empleada.

En caso de que no existiera ningún símbolo no terminal en la forma sentencial, se habría llegado a producir la sentencia que estábamos analizando, estando almacenado su parse izquierdo en la pila; si no, se continúa en el **paso 4**.

Paso 3. Se anula la última producción utilizada (situada en la cima de la pila), volviendo a obtener la forma sentencial anterior, y se retira de la pila el último numero. En esta forma sentencial se utiliza ahora la siguiente de las alternativas disponibles para el símbolo no terminal situado más a la izquierda, anotando en la pila el número de la regla empleada. En caso de que no haya más alternativas, se repite el **paso 3** para la forma sentencial obtenida.

Si llega el momento en que se agotan todas las posibles alternativas del axioma inicial, (la pila se habría quedado vacía), entonces se puede garantizar que la cadena analizada no pertenece al lenguaje.

Paso 4. Se comprueba si todos los símbolos terminales consecutivos que hayan aparecido a la izquierda de la forma sentencial encajan con alguna subcadena izquierda de la tira analizada. En caso afirmativo, se va al **paso 2**. Si no, se va al **paso 3**.

Ejemplo:

Sea la gramática:

```
(1) S --> cAd
(2) A --> bcB
(3) A --> a
(4) B --> b
```

A continuación, se va a realizar el análisis (paso a paso) de la sentencia `cad`:

Paso 1: Inicialmente se parte de la forma sentencial `S`.

Paso 2: El símbolo no terminal situado más a la izquierda es `S`, en las reglas de la gramática la primera (y única) alternativa para el símbolo `S` es `S-->cAd`. Empleando esta regla se obtiene la forma sentencial `cAd`. La pila contiene el número `1`. Ir al **paso 3**.

Paso 3: La subcadena de símbolos terminales consecutivos que han aparecido a la izquierda de la forma sentencial es `c` y la tira a analizar es `cad`. Por tanto, se supone válida esta producción y se continúa por el **paso 2**.

Paso 2: Ahora la forma sentencial es `cAd`, cuyo primer símbolo no terminal es `A`, y la primera de las producciones para `A` es `A-->bcB`. Aplicándola se obtiene la nueva forma sentencial `cbcBd`. La pila de análisis contiene ahora el parse `1-2`.

Paso 4: La subcadena izquierda de símbolos terminales es ahora `cbc`, que no encaja con el principio de la tira analizada. Por tanto, vamos al **paso 3**.

Paso 3: Se anula la producción anterior, obteniendo nuevamente la forma sentencial `cAd`. Retiramos el símbolo superior de la pila de análisis, quedando ésta como antes (es decir, `1`). Se aplica ahora la segunda de las alternativas del símbolo no terminal `A`, que es `A-->a`, obteniéndose la forma sentencial `cad`. En la pila de análisis se introduce el número de la regla aplicada quedando ésta como `1-3`.

Paso 4: La subcadena izquierda de símbolos terminales es `cad`, que coincide exactamente con la tira analizada.

Paso 2: La forma sentencial no incluye ningún símbolo no terminal, por lo que el proceso termina satisfactoriamente. El parse izquierdo de la cadena está contenido en la pila.

* * *

Es evidente que una condición necesaria para poder realizar este tipo de análisis, tal como se ha descrito anteriormente, es que la gramática no sea recursiva por la izquierda y, en general, que no tenga ciclos por la izquierda. Por ejemplo, supongamos la gramática:

A --> Aa | a

Según el método que se sigue para el análisis, se generaría la siguiente secuencia de formas sentenciales:

$$A \Longrightarrow Aa \Longrightarrow Aaa \Longrightarrow Aaaa \Longrightarrow \ldots$$

Igualmente sucede si la gramática tiene ciclos, ya que por definición de ciclo:

$$A \Longrightarrow^{+} A\ldots$$

Para solucionar este problema podemos hallar una gramática equivalente no recursiva por la izquierda y sin ciclos, o bien modificar ligeramente el algoritmo.

La implementación en Pascal de este algoritmo, mediante el uso de un procedimiento recursivo, puede encontrase en el programa ejemplo RPM. Su ejecución para la gramática:

```
1. E --> T
2. E --> T+E
3. T --> F
4. T --> F*T
5. F --> a
6. F --> b
7. F --> (E)
```

y para el reconocimiento de la cadena

a*(a+b)

da como resultado la siguiente secuencia:

Pila	**Formas Sentenciales**
1-	T
1-3-	F
1-3-5-	a
1-3-6-	b
1-3-7-	(E)
1-4-	F*T
1-4-5-	a*T
1-4-5-3-	a*F
1-4-5-3-5-	a*a
1-4-5-3-6-	a*b
1-4-5-3-7-	a*(E)
1-4-5-3-7-1-	a*(T)
1-4-5-3-7-1-3-	a*(F)
1-4-5-3-7-1-3-5-	a*(a)
1-4-5-3-7-1-3-6-	a*(b)
1-4-5-3-7-1-3-7-	a*((E))
1-4-5-3-7-1-4-	a*(F*T)
1-4-5-3-7-1-4-5-	a*(a*T)
1-4-5-3-7-1-4-6-	a*(b*T)
1-4-5-3-7-2-	a*(T+E)
1-4-5-3-7-2-3-	a*(F+E)
1-4-5-3-7-2-3-5-	a*(a+E)
1-4-5-3-7-2-3-5-1-	a*(a+T)
1-4-5-3-7-2-3-5-1-3-	a*(a+F)
1-4-5-3-7-2-3-5-1-3-5-	a*(a+a)
1-4-5-3-7-2-3-5-1-3-6-	a*(a+b)

En la columna derecha se muestran las formas sentenciales que se van obteniendo durante el análisis, y en la izquierda, los números de las reglas utilizadas para producirlas (es decir, el parse izquierdo de cada una de estas formas senteciales).

El último valor obtenido en la columna izquierda es el parse izquierdo de la cadena analizada.

Análisis ascendente con retroceso.

El análisis ascendente, como ya se ha visto, consiste en construir el árbol sintáctico de una cadena dada, partiendo de los nodos terminales del mismo, hasta llegar a su raíz. Estos nodos terminales no son otros que los distintos símbolos terminales de la cadena de entrada.

En todos los análisis ascendentes se plantea la necesidad de reducir una subcadena de nodos, para obtener así una *metanoción*, o símbolo no terminal, común a todos los símbolos de la subcadena. A esta operación se le llama `reducción`.

Por otra parte, a las sucesivas lecturas de un símbolo de la cadena de entrada se les llama `desplazamientos`.

Cualquier análisis ascendente utilizará estas dos operaciones, además de las operaciones `aceptar` y `error`, que determinan el final del proceso.

El primero de los métodos ascendentes que vamos a ver es el análisis ascendente con retroceso. Este análisis, al igual que todos los métodos con retroceso, se basa en la prueba sistemática de todas las combinaciones posibles de reducciones y desplazamientos, continuando con el proceso siempre que sea posible, y dando marcha atrás cuando no quede otra alternativa.

En la cadena de entrada, y en las distintas formas sentenciales que se van a ir generando para su reconocimiento, se distinguen dos subcadenas: la primera de ellas corresponde a los símbolos que se han leído hasta el momento; y la segunda será la subcadena que queda por leer. Cada vez que se lee un símbolo, se realiza el desplazamiento de un símbolo de la subcadena derecha a la subcadena izquierda, o lo que es lo mismo, se desplaza una posición la separación entre las subcadenas izquierda y derecha.

Inicialmente, se parte de la cadena de entrada, de la cuál se ha leído el primer símbolo. Por tanto, la subcadena izquierda contiene un solo símbolo, y la subcadena derecha el resto.

El algoritmo siguiente se repite hasta que se obtiene el axioma en la subcadena izquierda, estando la derecha vacía.

La forma sentencial que se está analizando en un instante dado se puede representar como $\alpha\beta$, siendo α la subcadena izquierda y β la subcadena derecha.

Si tomamos $\alpha = X_1X_2....X_p...X_n$, y $\beta = Y_1Y_2....Y_m$, entonces

$$\alpha\beta = X_1X_2....X_p...X_nY_1Y_2....Y_m$$

sería la forma sentencial analizada. Se va a considerar que las reglas gramaticales son de la forma:

$$\begin{array}{llllll} A_1 & \rightarrow & Z_{1\,1} & Z_{1\,2} & .... & Z_{1\,s1} \\ A_2 & \rightarrow & Z_{2\,1} & Z_{2\,2} & .... & Z_{2\,s2} \\ & & & ... & & \end{array}$$

A_k -> $Z_{k\,1}$ $Z_{k\,2}$ .... $Z_{k\,Sk}$

Como ya se ha anticipado, llamamos reducción de la cadena α, según la regla *k*, a la sustitucion de los (n-p+1) símbolos derechos de la cadena α que coinciden exactamente con los S_k símbolos del consecuente de la regla *k*, por el antecedente de dicha regla A_k. Por supuesto, sólo será posible efectuar la reducción en el caso de que todos los símbolos del consecuente encajen con el extremo derecho de la subcadena α. El resultado de la reducción sería:

$$\alpha = X_1X_2...A_k$$

Se llama desplazamiento al paso del símbolo izquierdo de la cadena β al extremo derecho de la cadena α. Por supuesto, esto sólo es posible si la longitud de β es mayor que cero (es decir, si la cadena β es distinta de la tira nula ε. El resultado sería:

$$\alpha = X_1X_2....X_p...X_nY_1$$

$$\beta = Y_2....Y_m$$

Se describe a continuación, en pseudo-codigo, el algoritmo recursivo de reconocimiento con retroceso que realiza el análisis ascendente:

```
procedure Ensayar (α,β);
begin
        for k:=1 to Numero de reglas do
                if consecuente de regla k = Parte derecha de α then begin
                        Realizar Reduccion k en la subcadena α

                        if α=Axioma and β=ε then Aceptar
                        else Ensayar(α,β)

                        Anular Reduccion
                end

                if β<>ε then begin
                        Realizar un desplazamiento de un símbolo de β a α

                        if α=Axioma and β=α then Aceptar
                        else Ensayar(α,β)

                        Anular Desplazamiento
                end
        end
end
```

En el algoritmo anterior se ha supuesto que el procedimiento Aceptar conlleva la finalización del proceso de análisis.

La implementación práctica de dicho algoritmo corresponde al programa ejemplo RPMASC. El siguiente es un ejemplo de la ejecución de este programa.

Ejemplo: Sea la gramática gramática:

1. E -> E+T
2. E -> T
3. T -> T*F
4. T -> F
5. F -> (E)
6. F -> a

cuyo axioma es `E`. Los símbolos no terminales de la gramática son `{E,F,T}`, y los símbolos terminales son `{(,),*,+,a}`. Tomando como entrada la cadena `a+a*a`, la traza de ejecución del análisis ascendente con retroceso para esta cadena es:

Pila	**Expresión Regular**	**Operación**
	a|+a*a	Desplazamiento
6-	F|+a*a	Reducción
6-4-	T|+a*a	Reducción
6-4-2-	E|+a*a	Reducción
6-4-2-	E+|a*a	Desplazamiento
6-4-2-	E+a|*a	Desplazamiento
6-4-2-6-	E+F|*a	Reducción
6-4-2-6-4-	E+T|*a	Reducción
6-4-2-6-4-1-	E|*a	Reducción
6-4-2-6-4-1-	E*|a	Desplazamiento
6-4-2-6-4-1-	E*a|	Desplazamiento
6-4-2-6-4-1-6-	E*F|	Reducción
6-4-2-6-4-1-6-4-	E*T|	Reducción
6-4-2-6-4-1-6-4-2-	E*E|	Reducción
6-4-2-6-4-2-	E+E|*a	Reducción
6-4-2-6-4-2-	E+E*|a	Desplazamiento
6-4-2-6-4-2-	E+E*a|	Desplazamiento
6-4-2-6-4-2-6-	E+E*F|	Reducción
6-4-2-6-4-2-6-4-	E+E*T|	Reducción
6-4-2-6-4-2-6-4-2-	E+E*E|	Reducción
6-4-2-6-4-	E+T*|a	Desplazamiento

6-4-2-6-4-	E+T*a|	Desplazamiento
6-4-2-6-4-6-	E+T*F|	Reducción
6-4-2-6-4-6-3-	E+T|	Reducción
6-4-2-6-4-6-3-1-	E|	Reducción
6-4-2-6-4-6-3-1-	E|	Fin de proceso. Correcto

Análisis sintáctico

Procedimientos de análisis LL(1)

Análisis descendente sin retroceso.

En el análisis descendente con retroceso se generan formas sentenciales a partir del axioma, dando marcha atrás en cuanto se detecta que la forma generada no es viable, (es decir, no conduce a ninguna sentencia del lenguaje). Este proceso de vuelta atrás es lento. Para mejorar la eficiencia del mismo, sería muy útil saber *a priori* qué alternativa del símbolo no terminal es más conveniente usar.

Veamos de nuevo el ejemplo del apartado anterior:

Ejemplo: Sea la gramática

(1) S --> cAd
(2) A --> bcB
(3) A --> a
(4) B --> b

y la sentencia `cad`. Partiendo del axioma, sólo se puede aplicar la regla 1, obteniendo la forma sentencial `cAd`. Si se compara con la sentencia `cad`, se observa que ambas comienzan con el caracter `c`. Por tanto, la subcadena `Ad` ha de generar el resto de la sentencia, o sea, `ad`. En este instante existen dos alternativas que se pueden emplear para modificar la forma sentencial, que corresponden a la aplicación de las reglas 2 y 3.

La aplicación de la regla 2 provoca la aparición del carácter `b` al principio de la subcadena restante, mientras que la regla 3 provoca la aparición del carácter `a`. Por tanto, como la subcadena que falta por generar para producir la sentencia final es `ad` (empieza por `a`), puede deducirse que en este instante la regla que debe emplearse es la regla 3, y no la 2.

El método de análisis que hemos seguido consiste en leer la cadena de entrada de izquierda a derecha, (**L**: *Left to rigth*) utilizando reglas de producción izquierda (**L**: *Left most*) e inspeccionando un (**1**) solo símbolo de la entrada para elegir la regla conveniente. Este análisis se denomina **LL(1)**.

Desafortunadamente, hay casos en los que este procedimiento no sirve. Supóngase, por ejemplo, que la gramática fuese:

(1) S --> cAd
(2) A --> aB
(3) A --> a
(4) B --> b

Al analizar la tira de entrada `cad`, tras realizar la primera producción obtendríamos la forma sentencial `cAd`, quedando como subcadena a analizar `ad` (que comienza con `a`). Pero ahora hay

dos reglas aplicables que comienzan por `a` (las reglas número 2 y 3). Por tanto, no es posible decidir de forma automática qué regla debe emplearse.

Por otra parte, si se pretende que el análisis sea sin retroceso, es indispensable que la gramática no tenga ciclos por la izquierda (y consiguientemente, que no sea recursiva por la izquierda).

No todas las gramáticas admiten un análisis descendente sin retroceso en el que se pueda predecir la alternativa que debe usarse. En el siguiente apartado se verá una condición necesaria y suficiente para que una gramática admita un análisis **LL(1)**.

Análisis sintáctico

Funciones de CABECERA y SIGUIENTE

A continuación, vamos a definir dos funciones que llamaremos **CABECERA** y **SIGUIENTE**.

- **CABECERA**

Se define la función `CABECERA` de una forma sentencial α, como el conjunto de símbolos terminales que pueden aparecer a la izquierda en alguna forma sentencial derivada de α. El símbolo nulo (ε) pertenecerá a `CABECERA` de α solamente si puede producirse a partir de α mediante la aplicación de las reglas de la gramática.

La definición formal de `CABECERA` es:

$$\text{CABECERA}: (N \cup T)^* \rightarrow P((T \cup \varepsilon))$$
$$\alpha \rightarrow \text{CABECERA}(\alpha) = \{a/\alpha \Rightarrow^* a\beta\} \cup \{\varepsilon/\alpha \Rightarrow^* \varepsilon\}$$

En donde se han hecho las consideraciones habituales, que son:

- α,β pertenecen a `{N∪ T}*`
- a pertenece a `T`
- P(X) = conjunto de las partes de X.

Ejemplo: Sea la gramática

S --> AaBc
A --> Bbc | b
B --> d

y sea la forma sentencial α= "`AaBc`"

Las formas sentenciales que pueden producirse a partir de α son:

- **Formas sentenciales de orden 0.**

  `AaBc` Sin sustituir ningún símbolo

- **Formas sentenciales de orden 1.**

  `BbcaBc` Sustituyendo la primera opción de `A`.
  `baBc` Sustituyendo la segunda opción de `A`.
  `Aadc` Sustituyendo la primera opción de `B`.

- **Formas sentenciales de orden 2.**

`dbcaBc` De segundo orden ....
`Bbcadc`
`badc`
`Bbcadc`
`badc`

etc ...

Los símbolos terminales que aparecen a la izquierda de todas estas formas sentenciales son `{b,d}`; por tanto:

CABECERA(α) = CABECERA(AaBc) = {b,d}

El problema se complica sensiblemente cuando la gramática contiene reglas ε. Por ejemplo, supongamos que existe una regla más en el caso anterior:

S --> AaBc
A --> Bbc | b | ε
B --> d

Ahora, las formas sentenciales que se pueden generar a partir de `S` serían:

- **Formas sentenciales de orden 1.**

  `AaBc` Sin sustituir ningún símbolo en esta forma sentencial.

- **Formas sentenciales de orden 2.**

  `BbcaBc` Sustituyendo la primera opción de `A`.
  `baBc` Sustituyendo la segunda opción de `A`.
  `aBc` Sustituyendo la tercera opción de `A`.
  `Aadc` Sustituyendo la primera opción de `B`.

- **Formas sentenciales de orden 3.**

  `dbcaBc` De segundo orden para `AaBc`....
  `Bbcadc`
  `badc`
  `adc`
  `Bbcadc`
  `badc`
  `...`

  En este caso:

CABECERA(α) = CABECERA(AaBc) = {a,b,d}

A continuación se verá qué ocurre en esta gramática con la cadena `AA`. Hallamos las formas sentenciales que se derivan de ella, que son:

AA

BbcA,bA,A,ABbc,Ab,A

dbcA,BbcBbc,Bbcb,Bbc,bBbc,bb,b,Bbc,b,ε,
BbcBbc,bBbc,Bbc,Bbcb,bb,b,Bbc,b,ε

etc ....

En este caso, los símbolos terminales que se encuentran a la izquierda de las formas sentenciales que se han producido son `b` y `d`. Pero también se ha producido la tira nula ε, por lo que:

$$\text{CABECERA}(\alpha) = \text{CABECERA}(AA) = \{ b , d , e\}$$

Afortunadamente, no hay que generar todas las formas sentenciales posibles para hallar la cabecera de una cadena. El siguiente algoritmo recursivo resuelve el problema:

1. Supongamos que la cadena α consta de un solo símbolo (es decir, α=`X`). El algoritmo para hallar `CABECERA(X)` consiste en la aplicacion de los tres pasos siguientes:

   1.1. Si `X` es terminal entonces `CABECERA(X) := {X}`. **Fin**.

   1.2. Si existe una regla de producción en la gramática de la forma `X --> ε`, entonces ε pertenece a `CABECERA(X)`. Por tanto, `CABECERA(X):={ε}`

   1.3. Para todas las reglas de producción de la gramática de la forma

   $$\begin{aligned} X &\dashrightarrow \alpha_1 \\ X &\dashrightarrow \alpha_2 \\ &\vdots \quad \vdots \\ X &\dashrightarrow \alpha_p \end{aligned}$$

   se añade a `CABECERA(X)` la unión de las funciones `CABECERA` de cada uno de los consecuentes:

   $$\text{CABECERA}(X) := \text{CABECERA}(X) \cup \left[\bigcup_{i=1}^{p} \text{CABECERA}(\alpha_i) \right]$$

   Fin.

• *Si la cadena tiene más de un símbolo terminal*. Sea la cadena $\alpha = X_1X_2X_3...X_n$. A partir de `k:=1`, y hasta que no se llegue al final, se ejecutan los siguientes pasos :

   2.1. Se calcula $\text{CABECERA}(X_k)$ con el algoritmo del paso 1.

   2.2. Si ε no pertenece a $\text{CABECERA}(X_k)$, entonces se ha terminado. Pero como `k<=n`, ε no pertenece a `CABECERA(`α`)`

$$\text{CABECERA}(\alpha) := \left[\bigcup_{i=1}^{k} \text{CABECERA}(X_i)\right] - \{\varepsilon\}$$

Fin.

2.3. Si ε pertenece a $\text{CABECERA}(X_k)$, entonces $k:=k+1$. Si $k>n$, vamos al paso 2.4. Si no, se vuelve al paso 2.1.

2.4. *Todos los símbolos de* α *son anulables*. Por tanto, ε pertenece a $\text{CABECERA}(\alpha)$ y

$$\text{CABECERA}(\alpha) := \left[\bigcup_{i=1}^{n} \text{CABECERA}(X_i)\right] \cup \{\varepsilon\}$$

- Ejemplo:

  Sea la gramática :

  S --> Aa
  A --> BCda
  B --> b
  B --> d
  B --> ε
  C --> c
  C --> ε

  Se va a calcular CABECERA(Aa).

Cálculo de CABECERA(Aa): Como el parámetro de entrada tiene más de un símbolo, se comienza con el paso 2.1., que dice que hay que aplicar el algoritmo para calcular CABECERA(A).

Cálculo de CABECERA(A):

Los pasos 1.1, y 1.2. no son aplicables. El paso 1.3. indica que para calcular CABECERA(A) hay que calcular CABECERA(BCda)

Cálculo de CABECERA(BCda):

Como la cadena tiene cuatro símbolos (más de uno), se entra en el paso 2 del algoritmo, tomando inicialmente k:=1. El paso 2.1. dice que hay que calcular CABECERA(B)

Cálculo de CABECERA(B):

El paso 1.1. no es aplicable, pero el paso 1.2. sí, ya que existe la regla B-->ε. Por tanto, ε pertenece a CABECERA(B). Se continúa con el paso 1.3., que hace calcular CABECERA(b) y

CABECERA(d), ya que existen las dos reglas B-->b , y B-->d

Cálculo de CABECERA(b):

Como b es un símbolo terminal, entonces, por el paso 1.1. CABECERA(b):={b}

Cálculo de CABECERA(d):

Como d es un símbolo terminal, entonces, por el paso 1.1. CABECERA(d):={d}

Se retoma el cálculo de CABECERA(B), en el que se había hallado que CABECERA(B) contenía el símbolo ε y todos los contenidos en CABECERA(b) y en CABECERA(d). Por tanto:

CABECERA(B):={ε} ∪{b} ∪{d} = {ε,b,d}

Se vuelve ahora al paso 2.3., ya que ε pertenece a CABECERA(B). Por tanto, se hace k:=2, y se vuelve al paso 2.1. En este momento se tiene que calcular CABECERA(C), de forma similar a como se había calculado CABECERA(B)

Cálculo de CABECERA(C):

...

CABECERA(C):={ε,c}

Como ε pertenece a CABECERA(C), el paso 2.3. vuelve a enviarnos al paso 2.1., pero ahora con k:=3. Hay que calcular, por tanto, la cabecera del tercer símbolo. Es decir, CABECERA(d)

Cálculo de CABECERA(d):

.....

CABECERA(d):={d}

Ahora, ε no pertenece a la cabecera del símbolo que se está calculando. Por tanto, se puede aplicar el paso 2.2., que dice que:

CABECERA(BCda):= CAB(B) ∪CAB(C) ∪CAB(d) - {ε}

= {ε,b,d} ∪{ε,c} ∪{d} - {ε}

CABECERA(BCda):={b,c,d}

Se retoma el cálculo de `CABECERA(A)`. Como la única regla de la gramática para el símbolo `A` es `A-->BCda`, la cabecera de `A` coincide con la de la cadena `BCda`

CABECERA(A):={b,c,d}

Finalmente, en el cálculo de `CABECERA(Aa)`, se ha analizado el primer símbolo, `A`, obteniéndose que ε no pertenece a su conjunto `CABECERA`, por lo que es aplicable el paso 2.2., que dice que `CABECERA(Aa)=CABECERA(A)`, y por tanto, finalmente:

CABECERA(Aa):={b,c,d}

- **SIGUIENTE**

Se define la función `SIGUIENTE` de un símbolo no terminal `A` como el conjunto de símbolos terminales que pueden aparecer inmediatamente a la derecha del símbolo `A` en cualquier forma sentencial que sea posible generar a partir de la gramática. Si `A` puede ser el símbolo situado más a la derecha en alguna forma sentencial, entonces `$` pertenece a `SIGUIENTE(A)`.

Formalmente:

$$\texttt{SIGUIENTE: N} \rightarrow P(\texttt{T} \cup \$)$$

$$\texttt{A} \rightarrow \texttt{SIGUIENTE(A)} = \{a / S \Rightarrow^* \alpha A a \beta\} \cup \{\$ / S \Rightarrow^* \alpha A\}$$

Ejemplo:

Sea la gramática :

S --> ABC
A --> Bd
B --> b | ε
C --> c | ε

Las formas sentenciales de esta gramática que se consiguen a partir de `S` son:

- **Formas sentenciales de orden 1.**

ABC

- **Formas sentenciales de orden 2.**

BdBC,AbC,AC,ABc,AB

- **Formas sentenciales de orden 3.**

bdBC,dBC,BdbC,BdC,BdBc,BdB,BdbC,Abc,
Ab,BdC,Ac,A,BdBc,Abc,Ac,BdB,Ab,A

etc ...

* * *

En este ejemplo, los símbolos terminales que pueden aparecer inmediatamente a la derecha de `A` en alguna forma sentencial son `b` y `c`. Además, el símbolo `A` aparece como símbolo más a la derecha en alguna forma sentencial, por lo que:

SIGUIENTE(A) = {$,b,c}

El siguiente algoritmo recursivo calcula eficientemente el conjunto `SIGUIENTE` de un símbolo no terminal `A`.

Paso 1. Si no está incluida ya, se añade a la gramática la regla `S'-->S$`, siendo `S` el axioma de la gramática.

Paso 2. Para todas las reglas de la gramática que contengan el símbolo `A` en el consecuente (de la forma `B-->αAβ`), es decir:

$$\begin{aligned} B_1 &\dashrightarrow \alpha_1 A \beta_1 \\ B_2 &\dashrightarrow \alpha_2 A \beta_2 \\ &\vdots \quad \vdots \\ B_k &\dashrightarrow \alpha_k A \beta_k \end{aligned}$$

$$\mathtt{SIGUIENTE(A)} := \left[\bigcup_{i=1}^{k} \mathtt{CABECERA}(X_i) \right] - \{\varepsilon\}$$

Paso 3. Tras ejecutar el paso anterior, si existen *p* reglas:

$$\begin{aligned} B_1 &\dashrightarrow \alpha_1 A \beta_1 \\ B_2 &\dashrightarrow \alpha_2 A \beta_2 \\ &\vdots \quad \vdots \\ B_p &\dashrightarrow \alpha_p A \beta_p \end{aligned}$$

tales que el símbolo nulo ε pertenece a `CABECERA(`β_j`)`, entonces:

$$\mathtt{SIGUIENTE(A)} := \mathtt{SIGUIENTE(A)} \cup \left[\bigcup_{j=1}^{p} \mathtt{SIGUIENTE}(B_j) \right]$$

Fin.

Ejemplo: Sea la gramática

S --> ABC
A --> aAa
A --> Bd
B --> b | ε
C --> c | ε

Aplicando el algoritmo para el cálculo de SIGUIENTE(A):

Paso 1: Se añade la regla S'-->S$ a la gramática, quedando ésta como:

S'--> S$
S --> ABC
A --> aAa
A --> Bd
B --> b | ε
C --> c | ε

Paso 2: Se buscan en la gramática todas las formas sentenciales en donde aparezca el símbolo A en el consecuente. Las dos únicas reglas que cumplen esta condición son S-->ABC y A-->aAa, por lo que:

$$\text{SIGUIENTE}(A) := \text{CABECERA}(BC) \cup \text{CABECERA}(a) - \{\varepsilon\}$$

Se calcula CABECERA(BC) según el algoritmo descrito anteriormente, obteniéndose que CABECERA(BC)={ε,b,c}

Igualmente CABECERA(a)={a}

Hasta el momento SIGUIENTE(A)={a,b,c}

Paso 3: Sean las dos subcadenas de las que hemos obtenido la cabecera anteriormente. La cabecera de la subcadena BC incluye el símbolo nulo ε. Esta subcadena pertenece a la regla S-->ABC. Por lo tanto, para completar el cálculo de SIGUIENTE(A) hay que calcular SIGUIENTE(S):

$$\text{SIGUIENTE}(A) := \{a,b,c\} \cup \text{SIGUIENTE}(S)$$

Se calcula SIGUIENTE(S) aplicando una vez más el algoritmo. En este caso es muy fácil ver que SIGUIENTE(S)={$}

Así que, finalmente SIGUIENTE(A):={$,a,b,c}

* * *

Antes de enunciar la condición **LL(1)**, se va a ampliar la definición de CABECERA para su aplicación a conjuntos:

Sea A un conjunto de cadenas, $A=\{\alpha_1,\alpha_2,\ldots,\alpha_n\}$, se define CABECERA(A) como:

$$\text{CABECERA}(A) := \bigcup_{i=1}^{n} \text{CABECERA}(\alpha_i)$$

Asimismo, vamos a definir la operación externa de concatenación entre cadenas y conjuntos de cadenas:

Sea β una forma sentencial, y Q un conjunto de cadenas de la forma:

$$Q=\{\alpha_1,\alpha_2,...,\alpha_n\}$$

Se define la operación · (producto externo) como:

$$\beta \cdot Q = \{\beta\alpha_1, \beta\alpha_2, \ldots, \beta\alpha_n\}$$

Análogamente, se definen los productos externos de símbolos por conjuntos de símbolos, etc...

Dada una regla gramatical A-->α, perteneciente a P, se define el conjunto de *símbolos directores* (SD) asociados a dicha regla como:

$$\mathtt{SD:}\ \ \mathtt{P} \rightarrow P(\{\mathtt{T} \cup \$\})$$

$$(\mathtt{A} \rightarrow \alpha) \longrightarrow \mathtt{SD}(\mathtt{A} \rightarrow \alpha) = \mathtt{CABECERA}(\alpha \cdot \mathtt{SIGUIENTE}(\mathtt{A}))$$

Condición LL(1): Se dice que una gramática cumple la *condición LL(1)* si para cada par de reglas de la gramática que tengan el mismo antecedente la intersección de sus símbolos directores es **vacía**.

Análisis sintáctico

Construcción de tablas LL(1)

Para realizar el análisis sintáctico de una cadena generada por una gramática **LL(1)** se puede definir el siguiente procedimiento basado en el empleo de una pila y una tabla de doble entrada. Este procedimiento consiste en asignar a un par **(símbolo leído** *(terminal)***, símbolo de pila** *(no terminal)***)** una regla de la gramática. La tabla de análisis se obtiene mediante el siguiente algoritmo:

Se define la tabla:

$$TABLA : N \times [T \cup \{\$\}] \longrightarrow [N \cup T \cup \{\$\}]^*$$

Para todas las reglas de la gramática, que son del tipo $A \rightarrow \alpha$:

1. Se calcula $SD(A \rightarrow \alpha)$. Este cálculo proporciona todos los símbolos terminales de $CABECERA(\alpha)$, y si ε pertenece a $CABECERA(\alpha)$, además, todos los símbolos terminales incluidos en $SIGUIENTE(A)$.
2. Para todo símbolo terminal a perteneciente al conjunto $SD(A \rightarrow \alpha)$ se hace la asignación:

$$TABLA[A,a] := \alpha$$

Puede verse fácilmente que si la gramática es **LL(1)**, existe como mucho una asígnación para cada par de valores (A,a), ya que los conjuntos de símbolos directores son disjuntos. El recíproco también es cierto; es decir, si se obtiene como máximo una regla en cada casilla de esta tabla, entonces la gramática es **LL(1)**. Nótese que pueden quedar en la tabla parejas de valores (A,b) a los cuáles no se asigne ninguna regla. Esto supondría que el símbolo b no pertenece a ninguno de los conjuntos de símbolos directores de ninguna regla que tenga a A por antecedente.

El programa ejemplo LL1 calcula dicha tabla. Se da su ejecución para la gramática del ejemplo anterior:

Gramática:

```
S  -> E$E  -> TE'E' -> +TE'E' -> εT  -> FT'T' -> *FT'T' -> εF  -> (E)F  -> a
```

Símbolos Terminales = {$ ( ) * + a }

Símbolos No terminales = {E' E F T' S T }

SD(S -> E$)

```
          Cabecera  (E$) =
               Head      (E) =
                    Cabecera  (TE') =
                         Head      (T) =
                              Cabecera  (FT') =
                                   Head      (F) =
                                        Cabecera  ((E)) =Cabecera  ((E)) = {( }Cabecera  (a)
                                   Head      (F) = {( a }
                              Cabecera  (FT') = {( a }
                         Head      (T) = {( a }
                    Cabecera  (TE') = {( a }
               Head      (E) = {( a }
          Cabecera  (E$) = {( a }
```

SD(E -> TE')

```
          Cabecera  (TE') =
               Head      (T) =
```

```
                    Cabecera  (FT') =
                         Head      (F) =
                              Cabecera  ((E)) =Cabecera  ((E)) = {( }Cabecera  (a)   =Cabecera
                         Head      (F) = {( a }
                    Cabecera  (FT') = {( a }
               Head      (T) = {( a }
          Cabecera  (TE') = {( a }
```

SD(E' -> +TE')

```
          Cabecera  (+TE') =Cabecera  (+TE') = {+ }
```

SD(E' -> ε)

```
          Cabecera  (ε)  =Cabecera  (ε)  = {ε }Siguiente (E') =
               Siguiente (E) =
                    Cabecera  ($) =Cabecera  ($) = {$ }Cabecera  ()) =Cabecera  ()) = {) }
               Siguiente (E) = {$ ) }
          Siguiente (E') = {$ ) }
```

SD(T -> FT')

```
          Cabecera  (FT') =
               Head      (F) =
                    Cabecera  ((E)) =Cabecera  ((E)) = {( }Cabecera  (a) =Cabecera  (a) = {a
               Head      (F) = {( a }
          Cabecera  (FT') = {( a }
```

SD(T' -> *FT')

```
          Cabecera  (*FT') =Cabecera  (*FT') = {* }
```

SD(T' -> ε)

```
          Cabecera  (ε)  =Cabecera  (ε)  = {ε }Siguiente (T') =
               Siguiente (T) =
                    Cabecera  (E') =
                         Head      (E') =
                              Cabecera  (+TE') =Cabecera  (+TE') = {+ }Cabecera  (ε) =Cabece
                         Head      (E') = {+ ε }Cabecera  (ε) =Cabecera  (ε) = {ε }
                    Cabecera  (E') = {+ ε }Siguiente (E) =
                         Cabecera  ($) =Cabecera  ($) = {$ }Cabecera  ()) =Cabecera  ()) = {)
                    Siguiente (E)  = {$ ) }Cabecera  (E') =
                         Head      (E') =
                              Cabecera  (+TE') =Cabecera  (+TE') = {+ }Cabecera  (ε)    =Cab
                         Head      (E') = {+ ε }Cabecera  (ε)  =Cabecera  (ε)  = {ε }
                    Cabecera  (E') = {+ ε }Siguiente (E') =Siguiente (E') = {}
               Siguiente (T) = {$ ) + }
          Siguiente (T') = {$ ) + }
```

SD(F -> (E))

```
          Cabecera  ((E)) =Cabecera  ((E)) = {( }
```

SD(F -> a)

```
          Cabecera  (a) =Cabecera  (a) = {a }
```

Tabla(S ,()= E$
Tabla(S ,a)= E$
Tabla(E ,()= TE'
Tabla(E ,a)= TE'
Tabla(E',+)= +TE'
Tabla(E',$)= ε
Tabla(E',))= ε
Tabla(T ,()= FT'
Tabla(T ,a)= FT'
Tabla(T',*)= *FT'
Tabla(T',$)= ε

Tabla(T',))= ε
Tabla(T',+)= ε
Tabla(F ,()= (E)
Tabla(F ,a)= a

La tabla resultante es:

	a	**+**	*****	**(**	**)**	**$**
S	E$			E$		
E	TE'			TE'		
E'		+TE'			ε	ε
T	FT'			FT'		
T'			*FT'		ε	ε
F	a			(E)		

Una vez construida la tabla, el siguiente procedimiento se utiliza para el reconocimiento de cadenas:

Paso 1. Inicialmente se sitúa el axioma de la gramática en la pila. Se lee el primer símbolo de la entrada.

Paso 2. Mientras que la pila no se quede vacía y queden símbolos por leer:

Paso 2.1. Si el símbolo superior de la pila es un símbolo no terminal `A` de la gramática, se sustituye dicho símbolo por la cadena de símbolos correspondiente a la entrada de la tabla:

TABLA[*símbolo de pila*,*símbolo leído*]

Si esta entrada esta vacía, se ha producido un error. La cadena no pertenece al lenguaje generado por la gramática.

Paso 2.2. Si el símbolo de la pila es un símbolo terminal, y coincide con el símbolo leído, entonces suprimimos dicho símbolo de la pila, y se lee un nuevo símbolo de la cadena de entrada.

Si el símbolo superior de la pila no coincide con el símbolo leído, se ha producido un error sintáctico. La cadena no pertenece al lenguaje generado por la gramática.

Ejemplo:

Para la cadena de entrada `a+a$`, perteneciente al lenguaje generado por la gramática anterior, se obtiene la siguiente traza al aplicar este algoritmo.

Pila antes	**Símbolo leído**	**Acción**	**Pila después**

E	a+a$	SUSTITUIR	TE'
TE'	a+a$	SUSTITUIR	FT'E'
FT'E'	a+a$	SUSTITUIR	aT'E'
aT'E'	a+a$	AVANZAR	T'E'
T'E'	a+a$	SUSTITUIR	E'
E'	a+a$	SUSTITUIR	+TE'
+TE'	a+a$	AVANZAR	TE'
TE'	a+a$	SUSTITUIR	FT'E'
FT'E'	a+a$	SUSTITUIR	aT'E'
aT'E'	a+a$	AVANZAR	T'E'
T'E'	a+a$	SUSTITUIR	E'
E'	a+a$	SUSTITUIR	ε
ε		FIN	

* * *

Se denomina a este analizador así construido **reconocedor LL(1)**, y al análisis que realiza **análisis LL(1)** o análisis predictivo. Desde el punto de vista teórico, se demuestra que el algoritmo descrito constituye un análisis sintáctico determinista.

Teorema: Dado un reconocedor **LL(1)**, es posible construir un autómata con pila determinista equivalente (es decir, que acepte el mismo lenguaje). La demostración es constructiva:

Sea la gramática G={N,T,P,S} donde se tiene que:

- **Conjunto de símbolos no terminales:** N = { A,B,C, ... }
- **Conjunto de símbolos terminales:** T = { a,b,c, ... }
- **Conjunto de reglas de producción:**

P = { A -> α_1 A -> α_2 : : B -> β_1 B -> β_2 : : }

- **El axioma:** S = A

Se define el *autómata con pila* AP={Q,T_e,T_p,δ,q_0,Z_0,F}, en el que:

- **Conjunto de estados del autómata:** $Q = \{q, q_a, q_b, \ldots\}$. Este conjunto tiene un elemento (estado) por cada símbolo terminal de la gramática, más un estado adicional q.
- **Símbolos de entrada:** $T_e = T$
- **Símbolos de pila:** $T_p = N \cup T$
- **Estado inicial del autómata:** $q_0 = q$
- **Estado inicial de la pila:** $Z_0 = S$ (axioma de la gramática)
- $F = \emptyset$

  La función de transición δ se construye con el siguiente algoritmo:

  1. Para cada símbolo terminal a de la gramática, se incluyen las transiciones

     (i) $\delta(q,a,\varepsilon) = (q_a,\varepsilon)$

     (ii) $\delta(q_a,\varepsilon,a) = (q,\varepsilon)$

  2. Para todos los estados q_x, asociados a un símbolo terminal x, que pertenece al conjunto de símbolos directores de la regla $x \rightarrow \alpha$, se incluyen las transiciones:

     (iii) $\delta(q_x,\varepsilon,X) = (q_x,\alpha)$

Para demostrar que este autómata es determinista demostraremos que no existen dos transiciones compatibles en el conjunto de transiciones (i)+(ii)+(iii).

Prueba 1.1. En el subconjunto (i) no existen dos transiciones compatibles, ya que si $a \neq b$, las siguientes transiciones no son compatibles:

$$\delta(q,a,\varepsilon) = (q_a,\varepsilon)$$
$$\delta(q,b,\varepsilon) = (q_b,\varepsilon)$$

Prueba 1.2. Ninguna transición del subconjunto (i) es compatible con otra del subconjunto(ii), ya que para todo x, $q \neq q_x$.

Prueba 1.3. Análogamente al razonamiento anterior, ninguna transición del subconjunto (i) es compatible con otra del subconjunto (iii).

Prueba 2.2. En el subconjunto (ii) no existen dos transiciones compatibles, ya que si $a \neq b$, las siguientes transiciones no son compatibles:

$$\delta(q_a, \varepsilon, a) = (q, \varepsilon)$$
$$\delta(q_b, \varepsilon, b) = (q, \varepsilon)$$

Prueba 2.3. Ninguna transición del subconjunto (ii) es compatible con otra del subconjunto (iii), ya que los conjuntos N y T son disjuntos. Para todo x se cumple la incompatibilidad entre:

$$\delta(q_x,\varepsilon,x) = (q,\varepsilon)$$
$$\delta(q_x,\varepsilon,X) = (q,\alpha)$$

Prueba 3.3. Por último, supongamos que existen dos transiciones compatibles dentro del subconjunto (iii). Éstas han de ser de la forma:

$$\delta(q_x, \varepsilon, X) = (q_x, \gamma_1)$$
$$\delta(q_x, \varepsilon, X) = (q_x, \gamma_2)$$

La primera de estas transiciones corresponde a la inclusión de las transiciones asociadas a la regla $x \rightarrow \gamma_1$, y por tanto, ha de cumplirse que el símbolo terminal x pertenece al conjunto de símbolos directores asociado a dicha regla.

x pertenece a $SD(x \rightarrow \gamma_1)$

Razonando de igual modo, se establece que x pertenece al conjunto de símbolos directores de la regla $x \rightarrow \gamma_2$.

x pertenece a $SD(x \rightarrow \gamma_2)$.

Pero ésto contradice la hipótesis de que la gramática es **LL(1)**, ya que:

$$SD(X \rightarrow \gamma_1) \cap SD(X \rightarrow \gamma_2) = \{x\} \neq \emptyset.$$

Como se indicó en un principio, gran parte de los lenguajes de programación pueden ser descrito mediante una gramática de contexto libre. Desgraciadamente, no todos los lenguajes de contexto libre pueden ser descritos mediante una gramática **LL(1)**. A los lenguajes que pueden ser descritos mediante una gramática **LL(1)** los llamaremos *lenguajes* **LL(1)**.

A diferencia de las gramáticas, no existe ningún algoritmo que nos conteste a la pregunta de si un lenguaje cualquiera es o no **LL(1)**, ya que esta propiedad del lenguaje, como muchas otras, es indecidible.

No obstante, en la práctica, la mayoría de los lenguajes de programación admiten este tipo de análisis. El problema radica en encontrar la gramática susceptible de tal análisis.

El proceso normal que se sigue para obtener un analizador **LL(1)** para un determinado lenguaje consiste en encontrar primero una gramática de contexto libre que genere dicho lenguaje, comprobar si verifica la condición **LL(1)**, y en caso negativo aplicarle una serie de transformaciones, de forma que se obtenga otra gramática equivalente a la anterior en la que se hayan eliminado los posibles conflictos en los conjuntos de símbolos directores.

Las reglas heurísticas de transformación de las gramáticas son:

1) Eliminación de recursividad izquierda.

Si la gramática contiene reglas recursivas por la izquierda asociadas a un símbolo no terminal A, y todas las reglas asociadas a dicho símbolo son las siguientes:

$A \rightarrow A\alpha_1 \quad A \rightarrow \beta_1 A \rightarrow A\alpha_2 \quad A \rightarrow \beta_2 : \quad : \quad : \quad : A \rightarrow A\alpha_n \quad A \rightarrow \beta_m$

entonces todas ellas deben sustituirse por las siguientes reglas:

$A \rightarrow \beta_1 A'A \rightarrow \beta_2 A' : \quad : A \rightarrow \beta_m A'A' \rightarrow \alpha_1 A'A' \rightarrow \alpha_2 A' : \quad : A' \rightarrow \alpha_n A'A' \rightarrow \varepsilon$

donde A' es un nuevo símbolo no terminal distinto de los ya existentes en N.

2) Factorización.

Si la gramática incluye dos reglas de la forma:

$A \rightarrow \alpha\beta \quad \alpha, \beta, \gamma$ pertenecen a $\{N \cup T\}^* A \rightarrow \alpha\gamma \quad \alpha \neq \varepsilon$

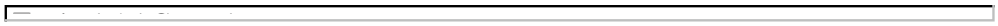
Éstas deben sustituirse por las reglas

$A \rightarrow \alpha A'$

A' -> β
A' -> "(

en donde `A'` es un nuevo símbolo no terminal distinto de los ya existentes en `N`.

La utilización de estas reglas no garantiza la obtención de una gramática **LL(1)**, pues si bien se eliminan ciertos conflictos en la tabla, no se puede garantizar que no aparezcan nuevas colisiones.



Análisis sintáctico

Procedimientos de análisis LR(0)

Para la construcción de tablas de análisis **LR(0)** es preciso realizar unas definiciones previas:

Se denomina *ítem* **LR(0)** de una gramática a una terna $(A,\alpha,\beta) \in N \times (N \cup T)^* \times (N \cup T)^*$, siendo $A \to \alpha\beta$ una regla gramatical de P. En general, los items se escriben como una regla de producción separada en dos partes por un punto:

$$[A \to \alpha_1 . \alpha_2]$$

Para hacer mayor énfasis, y distinguir los items de las reglas gramaticales colocaremos unos corchetes al referirnos a los items.

Cada regla de la gramática dará origen a uno o más items. De hecho, la única regla que da origen a un solo ítem es aquella cuyo consecuente es la tira nula ($X \to \varepsilon$), que genera el ítem $[X \to \cdot]$

Llamamos *ítem completo* a aquéllos de la forma $[A \to \alpha \cdot]$

Al conjunto de todos los items de una gramática lo denotaremos como Φ.

Ejemplo: Sea una gramática con las reglas

S -> E$
E -> E+T
E -> E-T
E -> T
T -> (E)
T -> a

A partir de la misma podemos hallar los siguientes items:

Φ = {[S -> ·E$], [E -> ·E+T], [E -> ·E-T], [S -> E·$], [E -> E·+T], [E -> E·-T], [S -> E$·], [E -> E+·T], [E -> E-·T], [E -> ·T], [E -> E+T·], [E -> E-T·], [E -> T·], [T -> ·(E)], [T -> ·a], [E -> T·], [T -> (·E)], [T -> a·], [T -> (E·)], [T -> (E)·]}

Función de cierre

Sobre un conjunto I de items se define la función $cierre(I)$ como el conjunto de items resultante de aplicar las siguientes reglas:

$$cierre(I) : P(\Phi) \to P(\Phi)$$

Regla 1. Todos los items pertenecientes a `I` pertenecen también a `cierre(I)`.

Regla 2. Si `[A -> α·Bβ]` es un item perteneciente a `cierre(I)`, y existe una regla de producción de la forma `B -> γ`, entonces el item `[B -> · γ]` pertenece a `cierre(I)`.

Regla 3. Repetir la regla anterior hasta que no se añada ningún nuevo ítem al conjunto `cierre(I)`.

Ejemplo: Sea el conjunto `I` compuesto por el ítem

$$I = \{[S \rightarrow \cdot E\$]\}$$

Inicialmente se calcula `cierre(I)` por la primera de las reglas, obteniéndose:

$$cierre(I) = \{[S \rightarrow \cdot E\$]\}$$

Según la segunda de las reglas se añade a `cierre(I)` todos aquellos items de la forma `[E-> γ]` , ya que el símbolo no terminal `E` aparece justo a la derecha del punto del primer ítem. Queda, por tanto:

$$cierre(I) = \{[S \rightarrow \cdot E\$],[E \rightarrow \cdot E+T],[E \rightarrow \cdot E-T],[E \rightarrow \cdot T]\}$$

Se vuelve a aplicar la regla 2 al conjunto obtenido anteriormente. En este caso tendremos que incorporar al mismo todos aquellos ítems de la forma `[E -> · γ]` (que ya están) y los de la forma `[T -> · γ]`, con lo que resulta:

```
cierre(I) = {[S -> ·E$],[E -> ·E+T],[E -> ·E-T], [E -> ·T],[T -> ·(E)],
[T -> ·a]}
```

Nuevamente se intenta la regla 2, pero esta vez no se añade ningún nuevo elemento a `cierre(I)`, ya que tenemos todos los items de la forma `[E -> · γ]` y de la forma `[T -> ·γ]`, y los nuevos items que han aparecido tienen un símbolo no terminal a la derecha del punto separador, por lo que no generan ningún nuevo ítem por la operación de cierre.

Función de transición

Se define la función de transición δ, que se aplica a un conjunto de items y a un símbolo (terminal o no terminal) de la gramática, y da como resultado un nuevo conjunto de items.

$$\delta : P(\Phi) \times \{N \cup T\} \text{ ------> } P\{\Phi\}$$

$$(I,X) \text{ ----> } \delta(I,X)$$

$\delta(I,X)$ es igual al cierre del conjunto de todos los items de la forma `[A -> αX·β]`, tales que `[A -> α·Xβ]` pertenece a `I`.

Ejemplo: Sea el conjunto de items

```
I  = {[S -> ·E$],[E -> ·E+T],[E -> ·E-T],[E -> ·T], [T -> ·(E)],[T -> ·a]}
```

Y sea el símbolo `X = E`

El conjunto δ`(I,E)` es:

cierre({[S -> E·$],[E -> E·+T],[E -> E·-T]})

En este caso el cierre coincide con el conjunto, y por tanto:

δ(I,E) = {[S -> E·$],[E -> E·+T],[E -> E·-T]}

Para el mismo conjunto de items `I`, y para el símbolo `(` se obtendría:

δ(I,() = cierre({[T -> (·E)]}) ={[T -> (·E)],[E -> ·E+T],[E -> ·E-T],[E -> ·T],[T -> ·(E)],[T ->·a]}

Para el mismo conjunto de items `I`, y para el símbolo `a`, se obtendría:

δ(I,a) = {[T -> a·]}

Construcción de la colección de conjuntos de items LR(0)

Sea `C` un conjunto de items **LR(0)** compuesto por los conjuntos de items

$$C = \{I_0,I_1,I_2,...\}$$

Supondremos que la gramática de partida tiene una sola regla asociada al axioma `S`. En caso de que esto no sea así, se añade a la gramática la regla `S'->S`, y se toma como axioma el símbolo `S'`. Esta nueva gramática se denomina *gramática aumentada*.

El algoritmo empleado es el siguiente:

Paso 1. Se construye el conjunto inicial de items I_0, compuesto por todos los items pertenecientes al cierre del primer item asociado al axioma. Es decir:

I_0 = cierre([S'->·S$])

La colección `C` de conjuntos de items tiene ya un elemento.

Paso 2. Para cada conjunto de items `I` perteneciente a la colección `C`, y para cada símbolo `X` (terminal o no terminal) de la gramática se halla el conjunto δ`(I,X)`. Si este conjunto es no vacío, y no pertenece ya a la colección C, se añade a la misma.

Paso 3. Se repite el paso 2 hasta que no se incorpore ningún conjunto nuevo a la colección `C`.

Este proceso es finito. Dado que el número de reglas de una gramática es finito, y que el número de símbolos en el consecuente de cada una de las reglas también lo es, el número de posibles items también lo es. Siendo finito el número de items es también finito el número de conjuntos de items que podemos formar a partir de ellos, por lo que el proceso descrito anteriormente tiene fin en un número finito de pasos.

Ejemplo: Para la gramática anterior

S -> E$
E -> E+T
E -> E-T
E -> T
T -> (E)
T -> a

La colección de items estará compuesta por los doce conjuntos siguientes:

I_0 = {[S -> ·E$],[E -> ·E+T],[E -> ·E-T],[E -> ·T], [T -> ·(E)],[T -> ·a]}
I_1 = {[S -> E·$],[E -> E·+T],[E -> E·-T]}
I_2 = {[E -> T·]}
I_3 = {[T -> a·]}
I_4 = {[T -> (·E)],[E -> ·E+T],[E -> ·E-T],[E -> ·T], [T -> ·(E)],[T -> ·a]}
I_5 = {[E -> E$·]}
I_6 = {[E -> E+·T],[T -> ·(E)],[T -> ·a]}
I_7 = {[E -> E-·T],[T -> ·(E)],[T -> ·a]}
I_8 = {[E -> E+T·]}
I_9 = {[E -> E-T·]}
I_{10} = {[T -> (E·)],[E -> E·+T],[E -> E·-T]}
I_{11} = {[T -> (E)·]}

* * *

Se dice que un ítem [A -> $\beta_1 \cdot \beta_2$] es *válido* para una subcadena viable $\alpha\beta_1$ (es decir, una subcadena que puede ser la parte izquierda de alguna forma sentencial del lenguaje) si existe una secuencia de derivaciones de la forma:

$$S ==>^* \alpha A \gamma ==>^* \alpha\beta_1\beta_2\gamma$$

En general, un ítem puede ser válido para muchas subcadenas. El hecho de que un ítem sea válido para una determinada subcadena proporciona información acerca de qué es lo que se debe hacer al encontrar una subcadena de este tipo en la pila, ya que si el ítem no es completo (es decir, $\beta_2 <> \varepsilon$), esto indica que aún no se ha encontrado un pivote que pueda ser reducido. Por el contrario, cuando existe un ítem completo válido asociado a una determinada subcadena, (es decir, $\beta_2 = \varepsilon$) es posible efectuar en la misma una reducción mediante la aplicación de la regla gramatical correspondiente.

Para construir la máquina de análisis **LR(0)** se utiliza el siguiente procedimiento:

Paso 1. Los estados de la máquina **LR(0)** corresponden a cada uno de los conjuntos de items I de la colección C. El estado inicial será el asociado al conjunto de items que contenga al ítem asociado al axioma, [S' -> ·S]

Paso 2. La tabla de transiciones es la definida por la función de transición δ entre conjuntos de items. Como se ve, la tabla de transiciones está definida sobre el producto cartesiano de la colección C, o *conjunto de estados de la máquina*, por el conjunto de símbolos terminales y no terminales ($N \cup T$) de la gramática.

Paso 3. La tabla de acciones para la máquina **LR(0)** depende exclusivamente del estado en el que se encuentre la máquina, ya que el análisis debe efectuarse sin inspeccionar ningún símbolo de la entrada. La acción a realizar en cada estado es la siguiente:

- Si el conjunto de items asociado al estado contiene un solo ítem, y éste es completo (es decir, si el único ítem del conjunto es de la forma `[A -> Υ ·]`), la acción asociada a dicho estado es la reducción mediante la regla `A -> Υ` (`Acción = Reducción`), salvo en el caso en que esta regla está asociada al axioma, en el que el análisis concluye con la aceptación de la cadena de entrada (`Acción = Aceptar`).
- En caso de que el conjunto de items asociado a un estado no contenga ningún ítem completo, la acción a realizar es desplazamiento. (`Acción = Desplazamiento`) .
- Si existe algún estado cuyo conjunto de items asociado contiene más de un ítem, siendo alguno de ellos completo, se dice que la gramática no cumple las restricciones **LR(0)**, o simplemente que la gramática no es **LR(0)**, y el análisis por este método no es posible.

Ejemplo: Para la gramática anterior

(0) S -> E$
(1) E -> E+T
(2) E -> E-T
(3) E -> T
(4) T -> (E)
(5) T -> a

El conjunto de estados de la máquina **LR(0)** es:

{0,1,2,3,4,5,6,7,8,9,10,11}

siendo el estado inicial el `0`.

Las tablas de análisis son:

TABLA DE ACCIONES

Estado	Acción
0	D
1	D
2	R_3
3	R_5
4	D
5	A
6	D
7	D

8	R_1
9	R_2
10	D
11	R_4

TABLA DE TRANSICIONES

	a	**+**	**-**	**(**	**)**	**$**	**E**	**T**
0	3			4			1	2
1		6	7			5		
2								
3								
4	3			4			10	2
5								
6	3			4				8
7	3			4				9
8								
9								
10		6	7		11			
11								

El reconocimiento de la cadena `a-a+a$` sería el siguiente:

	Pila	**Entrada**	**Acción**
(1)	0	a-a+a$	Desplazamiento
(2)	0a3	-a+a$	Reducción T -> a
(3)	0T2	-a+a$	Reducción E -> T
(4)	0E1	-a+a$	Desplazamiento
(5)	0E1-7	a+a$	Desplazamiento
(6)	0E1-7a3	+a$	Reducción T -> a
(7)	0E2-7T9	+a$	Reducción E -> E-T
(8)	0E1	+a$	Desplazamiento
(9)	0E1+6	a$	Desplazamiento

(10)	0E1+6a3	$	Reducción T -> a
(11)	0E1+6T8	$	Reducción E -> E+T
(12)	0E1	$	Desplazamiento
(13)	0E1$5		Aceptar

* * *

Un método alternativo de plantear la construcción de la colección de conjuntos de items que formarán los estados de la máquina **LR(0)** es la aplicación de la teoría de autómatas finitos de la siguiente forma:

Paso 1: Se construye un autómata finito no determinista AFN de la siguiente forma:

$$AFN=(Q^N,T_e^N,\delta^N,q_0^N,F^N)$$

- $Q^N=\Phi$ = Conjunto de todos los items.
- $T_e^N=N \cup T$
- q_0^N=[S' -> ·S]. Primer ítem asociado al axioma de la gramática.
- F={[A -> Υ ·] pertenecientes a Φ}. Items completos.
- δ^N. La función de transición entre estados de la máquina no determinista.

Se define así:

- Por cada ítem de la forma [A -> α·bβ], donde b pertenece a T (conjunto de símbolos terminales de la gramática) se tiene que:

  $$[A \rightarrow \alpha b \cdot \beta] \text{ pertenece a } \delta^N([A \rightarrow \alpha \cdot b\beta],b)$$

  Si [A -> αB·β] pertenece a δ^N([A -> α·Bβ],B), entonces [B -> ·τ] pertenece a δ^N([A -> α·Bβ],ε), para todos los posibles valores de τ, donde B pertenece a N (conjunto de símbolos no terminales de la gramática).

Paso 2: Una vez construido el AFN se aplica el algoritmo de construcción de subconjuntos para hallar el *Autámata Finito Determinista Mínimo* (AFDM), obteniendo así el mismo resultado que con el procedimiento descrito anteriormente en este epígrafe.

Análisis sintáctico

Construcción de tablas SLR(1)

La condición que impone el análisis**LR(0)**, como ya hemos dicho, es muy restrictiva, ya que los items completos han de aparecer en conjuntos separados. Esto no se cumple en la mayoría de las gramáticas que describen lenguajes de programación. Sin embargo existe una forma sencilla de ampliar la aplicación de ésta a una gama más amplia de gramáticas, mediante la lectura de un símbolo en la entrada para determinar la función a realizar.

Cuando una máquina de reconocimiento **LR** está en un determinado estado, el conjunto de items válidos para la subcadena que se encuentra en la pila en ese momento es precisamente el conjunto de la colección asociado a dicho estado. Si en este conjunto aparece un ítem completo y otro ítem más (ya sea completo o no), se produce un conflicto en el análisis **LR(0)**, ya que no podemos decidir que acción realizar.

Supongamos el caso del análisis de la cadena `tas`. (`t`,`a`,`s` pertenecen a `T`). Supongamos que en un momento determinado del análisis, y tras leer la subcadena `t`, tendremos en la pila una subcadena viable $\alpha\tau$, quedando por leer aún la subcadena `as`. Esto nos indica que $\alpha\tau$`as` es una forma sentencial del lenguaje. Supongamos también que existe un ítem completo válido asociado a esta subcadena viable, de la forma `[A -> τ·]`. Tras efectuar la reducción correspondiente obtendríamos la tira α`Aas`, que debe ser una forma sentencial de nuestro lenguaje. Es evidente que una condición necesaria para que esto suceda es que el símbolo `a` pueda aparecer a la derecha de `A` en alguna forma sentencial. Es decir, que `a` pertenezca al conjunto `SIGUIENTE(A)`, tal y como se definió en el apartado dedicado al análisis **LL(1)**.

Esto proporciona un método para eliminar algunas de las duplicidades que pueden aparecer en la tabla de acciones del análisis **LR(0)**. El nuevo método se denomina **SLR(1)**. El algoritmo de construcción de las tablas de análisis **SLR(1)** queda de la siguiente forma:

1. Construimos la colección de conjuntos de items. Este paso es igual que en el análisis **LR(0)**.
2. Los estados de la máquina **SLR(1)** corresponden a cada uno de los conjuntos de items I_j de la colección `C`. El estado inicial será el asociado al conjunto de items que contenga al primer ítem asociado al axioma, `[S' -> ·S]`. También es igual que en el análisis **LR(0)**.
3. La tabla de transiciones es la definida por la función de transición entre conjuntos de items δ. Ídem.
4. La tabla de acciones para la máquina **SLR(1)** depende del estado en que se encuentra la máquina y del primer símbolo de la entrada:
   - Si el ítem `[A -> α·aβ]` pertenece al conjunto asociado al estado `q`, entonces:

     ACCIÓN(q,a)=Desplazamiento
   - Si el ítem `[A -> τ·]`, `A<>S`, pertenece al conjunto asociado al estado `q`, entonces para todo `a` que pertenezca a `SIGUIENTE(A)`:

     `ACCIÓN(q,a)=Reducción A -> τ`
   - Si el ítem `[S -> τ·]` pertenece al conjunto asociado al estado `q`, entonces:

     ACCIÓN(q,$)=Aceptar.

Nota: Esto equivale a suponer que el único símbolo siguiente al axioma es el de final de cadena (`$`), que ocurre siempre en las gramáticas aumentadas.

Si tras efectuar las asignaciones correspondientes a todos los items de la gramática no obtenemos ninguna duplicidad en la tabla de acciones, entonces la gramática se dice que es **SLR(1)**. Todas las gramáticas **LR(0)** son **SLR(1)**.

Ejemplo: Sea la gramática aumentada

(0) S -> E
(1) E -> E+T
(2) E -> T
(3) T -> T*F
(4) T -> F
(5) F -> (E)
(6) F -> a

La colección de items estará compuesta por los conjuntos siguientes:

I_0 = {[S -> ·E],[E -> ·E+T],[E -> ·T],[T -> ·T*F], [T -> ·F],[F -> ·(E)],[F -> ·a]}

I_1 = {[S -> E·],[E -> E·+T]}

I_2 = {[E -> T·],[T -> T·*F]}

I_3 = {[T -> F·]}

I_4 = {[F -> (·E)],[E -> ·E+T],[E -> ·T],[T -> ·T*F], [T -> ·F],[F -> ·(E)],[F -> ·a]}

I_5 = {[E -> a·]}

I_6 = {[E -> E+·T],[T -> ·T*F],[T -> ·F],[F -> ·(E)], [F -> ·a]}

I_7 = {[E -> T*·F],[F -> ·(E)],[F -> ·a]}

I_8 = {[F -> (E·)],[E -> E·+T]}

I_9 = {[E -> E+T·],[T -> T·*F]}

I_{10} = {[T -> T*F·]}

I_{11} = {[F -> (E)·]}

Como se ve, esta gramática no es **LR(0)**, ya que en los conjuntos I_1, I_2 e I_9 hay dos estados, y uno de ellos es completo. Se produce un conflicto del tipo `reducción/desplazamiento` en la casilla correspondiente a dichos estados. Sin embargo, aplicando el procedimiento **SLR** pueden construirse las siguientes tablas de análisis:

Para el estado 2, por ejemplo, existen dos items:

[E -> T·] [E -> T·*F]

Según el primero de ellos, se puede aplicar la reducción `E -> T`, pero ésta se realizará solamente para aquellos elementos pertenecientes a `SIGUIENTE(E)`.

`SIGUIENTE(E) = {+,),$}`, por tanto:

ACCIÓN (2,"+") = Reducción E -> T
ACCIÓN (2,")") = Reducción E -> T
ACCIÓN (2,"$") = Reducción E -> T

El segundo de los items del conjunto I_2 nos indica que la acción a realizar cuando el símbolo siguiente es *

es:

ACCIÓN(2,*) = Desplazamiento.

De igual forma se analizarían el resto de los estados, atendiendo a los items de su correspondiente conjunto.

TABLA DE ACCIONES

	a	**+**	*****	**(**	**)**	**$**
0	D			D		
1		D				A
2		R2	D		R2	R2
3		R4	R4		R4	R4
4	D			D		
5		R6	R6		R6	R6
6	D			D		
7	D			D		
8		D			D	
9		R1	D		R1	R1
10		R2	R3		R3	R3
11		R3	R5		R5	R5

TABLA DE TRANSICIONES

	a	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
0	5			4			1	2	3
1		6							
2			7						
3									

4	5			4			8	2	3
5									
6	5			4				9	3
7	5			4					10
8		6			11				
9			7						
10									
11									

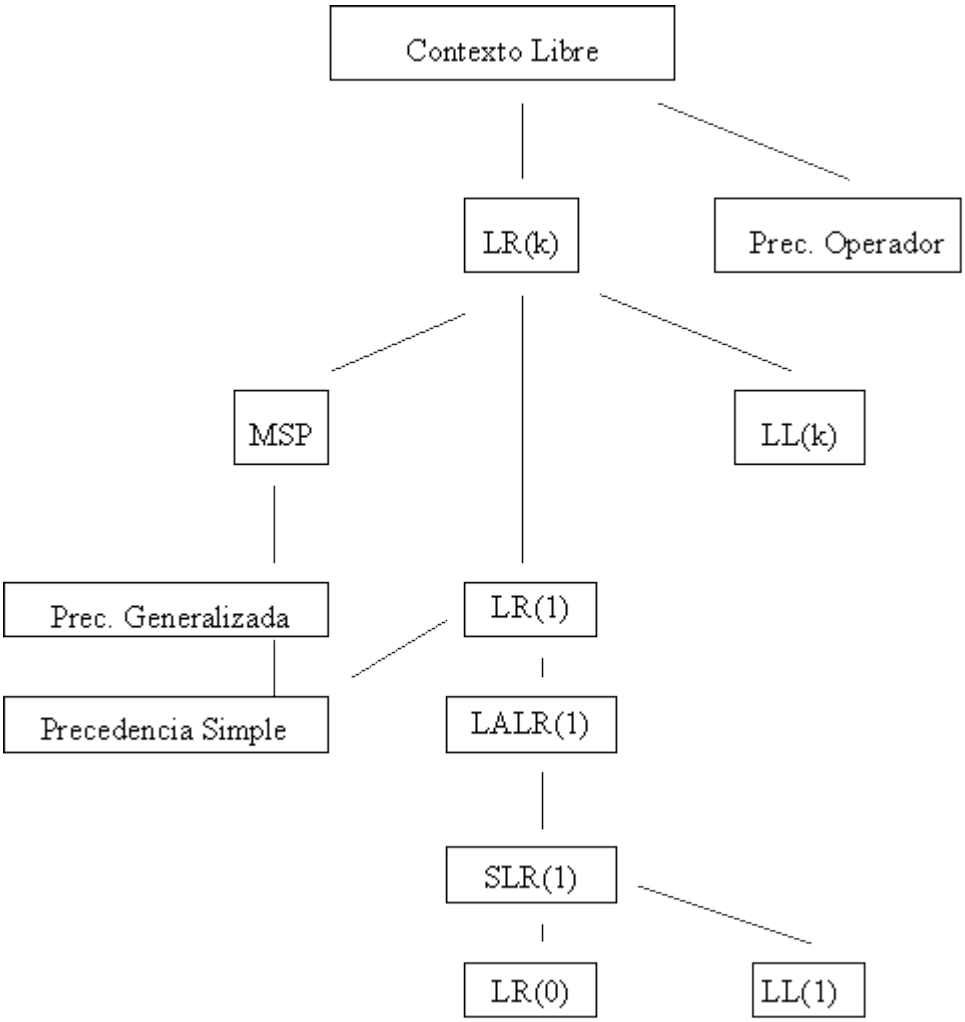
Análisis sintáctico.

Comparación entre los métodos de análisis

Para finalizar este capítulo, vamos a hacer una comparación no exhaustiva entre los distintos métodos de análisis sintáctico que se ha visto, atendiendo a distintos aspectos de los mismos. Algunas de las características que resumimos aquí han sido ya explicadas convenientemente dentro de su contexto.

Generalidades

Como se ha visto, no todos los analizadores sirven para todas las gramáticas, ya que para poder construir las tablas de análisis correspondientes hay que imponer ciertas restricciones a la gramática. En este sentido, podemos establecer una clasificación entre las gramáticas que pueden ser analizadas según cada uno de los métodos:



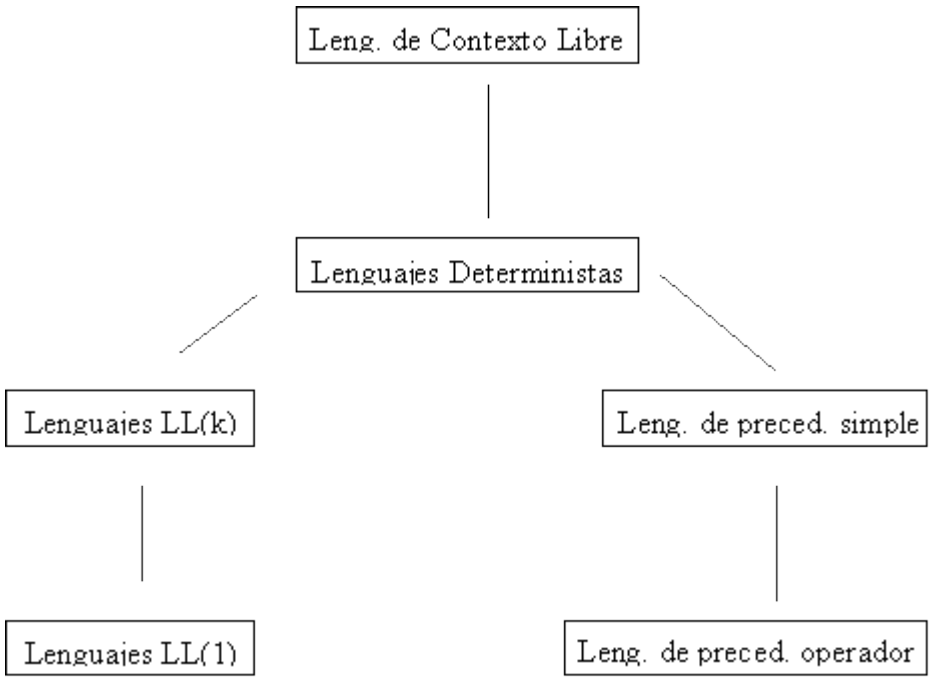
Este grafo indica la inclusión de un tipo de gramática en otras. Sin embargo, no es en sí indicativo de la amplitud de la clase de gramáticas que abarca cada nodo en comparación con otro que le sea disjunto. Entre todos los métodos existentes, el que cubre un mayor espectro de gramáticas en la práctica es sin duda el método LR. Los métodos basados en precedencia, por

lo general, imponen unas condiciones muy estrictas a la gramática, por lo que en la actualidad son menos usados que los LR o LL.

En cuanto a los lenguajes, se ha demostrado que todos aquellos que pueden ser reconocidos mediante análisis **LR(k),** también pueden ser reconocidos mediante análisis **LR(1),** o mediante un análisis **LALR(1)** o **SLR(1).** Es decir, que para toda gramática que acepta un análisis **LR(k),** existe una gramática equivalente que acepta un análisis **SLR(1).** De hecho, existe un algoritmo de transformación que realiza dicha conversión (a estos lenguajes se les llama lenguajes deterministas, y son un subconjunto propio de aquellos que pueden ser generados por gramáticas de contexto libre). Del mismo modo se ha demostrado que los lenguajes que admiten un análisis mediante precedencia generalizada pertenecen a esta clase, pero los lenguajes analizables mediante precedencia simple son un subconjunto propio de ella, y éstos a su vez incluyen a los lenguajes para los que existe una gramática que cumple las condiciones necesarias para un análisis de precedencia de operador.

Por otra parte, puede demostrarse que los lenguajes **LL(k)** son un subconjunto propio de los lenguajes **LR(k),** y que para todo k existen lenguajes que son **LL(k+1)** pero no **LL(k).**

El siguiente árbol resume la estructuración de los lenguajes:



Facilidad de construcción

La dificultad de construcción de las tablas de análisis ha quedado ya plasmada con suficiente claridad al explicar cada uno de los métodos. La construcción más laboriosa es sin duda la de las tablas **LALR**, **LR** y **SLR**, por este orden, seguidos de los análisis **LL**. Los métodos de precedencia suelen ser más simples de construir (especialmente el método de precedencia de operador) ya que la gramática es más sencilla, y no contiene producciones nulas. Sin embargo, tal como se indicó en el punto anterior, puede ser más difícil (si no imposible) la tarea de hallar la gramática que cumpla las restricciones necesarias. Este es el principal problema del método **LL**.

Facilidad de depuración

Debido a su complejidad, en las tablas **LR** se hace más difícil la resolución de las ambigüedades locales o de los pequeños ajustes finales necesarios para el correcto funcionamiento del analizador. Esta tarea es más fácil en los análisis basados en precedencia, y más aún en los análisis **LL**, dada la claridad con que resultan los programas escritos siguiendo este tipo de análisis.

Detección de errores

En cuanto a la detección y recuperación de los errores, los mejores métodos son los **LR** y los **LL**, ya que detectan el error tan pronto como es posible, y posibilitan un correcto diagnóstico de los mismos. De entre ellos el mejor es el **LR(1)** canónico. Los métodos basados en relaciones de precedencia dan un diagnóstico muy pobre, y en algunos casos no detectan el error más que tras haber leído una serie de símbolos adicionales, con la dificultad que esto entraña para la adecuada recuperación del proceso de análisis.

Utilización de la memoria

Todos los analizadores sintácticos pueden ser representados mediante tablas. Las más pequeñas son las de los métodos basados en precedencia de operador (más pequeñas aún si existen funciones de precedencia) que tienen tamaño t^2, siendo t el número de símbolos terminales de la gramática. Los análisis de precedencia simple y el método mixto (**MSP**) requieren unas tablas de tamaño proporcional a $(n+t)^2$, siendo n el número de símbolos no terminales de la gramática. Las tablas de precedencia generalizada necesitan, por lo general, tablas mayores, aún en los casos en los que sólo se almacenen los elementos no nulos de las mismas.

Los análisis **SLR(1)** y **LALR(1)** utilizan tablas del mismo tamaño, mucho menores que las necesarias para el análisis **LR(1)** canónico de la misma gramática. En cuanto a los análisis **LL(1)**, suelen generar tablas de longitud menor que las correspondientes **LALR(1)** o **SLR(1)**. Sin embargo, por lo general, las gramáticas que se emplean para el análisis **LL(1)** contienen mayor número de producciones que las que definen el mismo lenguaje y puedan ser analizadas según **LALR(1)**. Esto se debe a que la eliminación de la recursividad izquierda, o la factorización necesaria para encajar la gramática en la condición LL(1), introducen reglas adicionales, por lo que en la práctica se equipara la necesidad de memoria para un análisis **LL(1)** o **LALR(1)**.

Velocidad del análisis

En lo que se refiere a la rapidez del análisis los métodos **LR** y **LL** aventajan considerablemente a los métodos basados en precedencia, quedando entre medias de ambos el método mixto **MSP**. El método **LL(1)** es algo más lento que el **LR(1)** debido al mayor número de reglas de producción, y el consiguiente mayor número de movimientos necesarios para alcanzar la cadena final a partir del axioma.

Dentro de cada uno de los métodos estudiados existen diferentes "trucos" para agilizar la ejecución de los programas, por lo que la comparación no debe hacerse tanto entre los métodos como entre determinadas aplicaciones de los mismos. Existen varios estudios prácticos en este sentido.