Compiladores

Ingeniería Informática, 4º curso
Master en Informática, 1er curso

Primer cuatrimestre:

- Introducción
- Lenguajes y gramáticas
- Análisis Léxico
- Análisis Sintáctico

Profesores: **Bernardino Arcay**
Carlos Dafonte
Departamento de Tecnoloxías da Información e as Comunicacións.
Universidade da Coruña
Curso 2008/2009 Rev.081008

ÍNDICE

BIBLIOGRAFÍA.

Aho, A.V.; Lam M.; *Sethi, R. ; Ullman, J.D.*
"Compiladores: Principios, técnicas y herramientas"
Addison-Wesley, Reading, Massachussetts (2008).

Louden D. K. [2004], Construcción de compiladores. Principios y Práctica, Paraninfo Thomson Learning.

Garrido, A. ; Iñesta J.M. ; Moreno F. ; Pérez J.A. [2004] Diseño de compiladores, Publicaciones Universidad de Alicante.

Sanchis, F.J.; Galán, J.A.
"Compiladores, teoría y construcción"
Ed. Paraninfo (1987).

Aho, A.V.; Ullman, J.D.
"The theory of parsing, translation and compiling", I y II
Prentice-Hall (1972).

Aho, A.V.; Ullman J.D.
"Principles of compiler design"
Addison-Wesley, Reading, Massachussetts.

Hopcroff, J.E. ; Motwani R. ; Ullman, J. D. [2002] Introducción a la teoría de autómatas, lenguajes y computación, Addison-Wesley, Madrid.

Allen I.; Holub
"Compiler design in C"
Prentice-Hall (1991).

Sánchez, G.; Valverde J.A.
"Compiladores e Intérpretes"
Ed. Díaz de Santos (1984).

Sudkamp T.A.
“Languages and machines”
Addison-Wesley.

1 INTRODUCCIÓN.

Un **lenguaje** es un conjunto de oraciones finito o infinito, cada una de ellas de longitud finita (es decir, constituídas cada una de ellas por un número finito de elementos).

Compilación: Proceso de traducción en el que se convierte un programa fuente (en un lenguaje de alto nivel) en un lenguaje objeto, generalmente código máquina (en general un lenguaje de bajo nivel).

La **Teoría de compiladores** se usa para:

- Realizar compiladores.
- Diseñar procesadores de texto.
- Manejo de datos estructurados (XML).
- Inteligencia Artificial (en el diseño de interfaces hombre-máquina).
- Traductores de lenguajes naturales.
- Etc.

Se denominan **tokens** a los **lexemas**, las unidades mínimas de información (por ejemplo, una instrucción FOR). La identificación de estos elementos es la finalidad del **análisis léxico**.

Sintaxis de un lenguaje: Conjunto de reglas formales que nos permiten construir las oraciones del lenguaje a partir de los elementos mínimos.

Semántica: Es el conjunto de reglas que nos permiten analizar el significado de las frases del lenguaje para su interpretación.

Intérprete: Conjunto de programas que realizan la traducción de lenguaje fuente a objeto, "paso a paso", no de todo el programa (aparecieron por problemas de memoria).

Ensamblador: Compilador sencillo donde el lenguaje es simple, con una relación uno-a-uno entre la sentencia y la instrucción máquina.

Compilación cruzada: La compilación se realiza en una máquina A y la ejecución se realiza en otra máquina B.

Link (encadenar, enlazar): Es el proceso por el cual un programa dividido en varios módulos, compilados por separado, se unen en un solo.

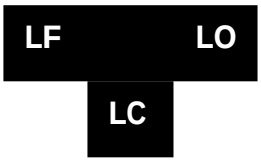
Pasadas en compilación: Recorridos de todo el programa fuente que realiza el compilador. Cuantas más pasadas, el proceso de compilación es más completo, aunque más lento.

Traductor o compilador incremental, interactivo o conversacional: Es un tipo de compilador que, al detectar un error, intenta compilar el código del entorno en donde está el error, no todo el programa de nuevo.

El **programa fuente** es el conjunto de oraciones escritas en el lenguaje fuente, y que normalmente estará en un fichero.

Bootstraping: Mediante esta técnica se construye un lenguaje L utilizando el mismo lenguaje L, realizando mejoras en el propio compilador como puede ser, por ejemplo, la inclusión de nuevas sentencias y estructuras.

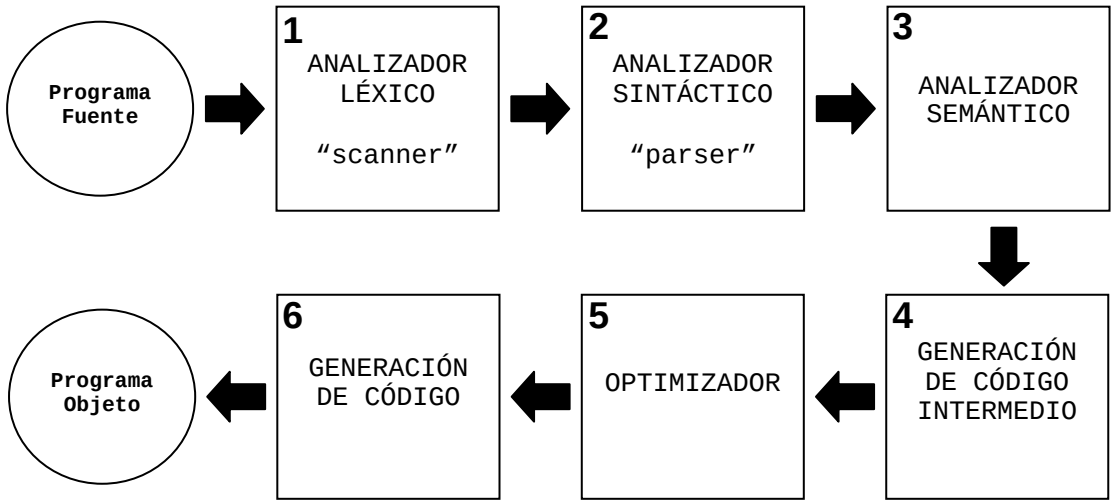
Existe una notación gráfica, denominada **Notación de Bratman**, que muestra como se realiza un compilador:



LF -> Lenguaje fuente
LO -> Lenguaje objeto
LC -> Lenguaje del compilador

Decompilador: Conjunto de programas que a partir de un código de bajo nivel obtiene código de alto nivel.

1.1 Estructura de un compilador.



Es importante comentar que aunque aquí se especifica un módulo de análisis semántico, éste también está implicado en la generación de código.

Es cada una de esas etapas pueden aparecer errores que han de ser detectados. Las etapas 1, 2 y 3 se denominan *etapas de análisis* y las 4, 5 y *6 etapas de síntesis*.

Etapa 1. Analizador léxico o "scanner".

En esta etapa, es preciso mantener separados cada uno de los tokens y almacenarlos en una *tabla de símbolos*. Aquí ya se podría detectar algún error, por ejemplo, poner FAR no correspondería a una palabra del lenguaje. No sólo en esta etapa se utiliza la tabla de símbolos ya que la información ahí

contenida se va completando durante las siguientes fases; por ejemplo, también almacenaremos el tipo establecido en su definición.

Etapa 2. Analizador sintáctico o "parser".

Se pretende ver la estructura de la frase, ver si los elementos tienen estructura de frase del lenguaje.

Etapa 3. Analizador semántico.

Se analiza si la frase encontrada tiene significado. Utilizará la tabla de símbolos para conocer los tipos de las variables y poder estudiar el significado semántico de la oración. Por ejemplo, si tenemos:

a = b + c

y sabemos que *b* es de tipo "carácter" y *c* es de tipo "entero" entonces, dependiendo del lenguaje, puede significar un error.

Etapa 4. Generación de código intermedio.
Se traduce el programa fuente a otro lenguaje más sencillo. Esto servirá para:

- Facilitar el proceso de optimización.
- Facilitar la traducción al lenguaje de la máquina.
- Compatibilización (el análisis será independiente de la computadora física, con el consecuente ahorro económico).

Etapa 5. Optimizador.

Intenta optimizar el programa en cuanto a variables utilizadas, bucles, saltos en memoria, etc.

Etapa 6. Generación de código.
Construye del programa objeto, esto es, se genera el código en ensamblador, propio de la plataforma en la que se ejecutará el programa. La tabla de símbolos contendrá normalmente información detallada sobre la memoria utilizada por las variables.

1.2 Ejemplo de las fases de un compilador.

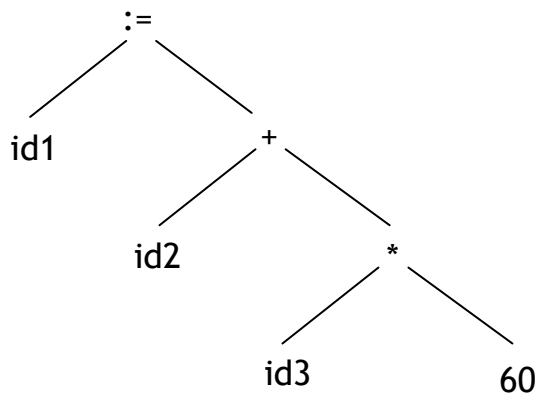
Supongamos una instrucción de la siguiente forma:

posicion := inicial + velocidad * 60

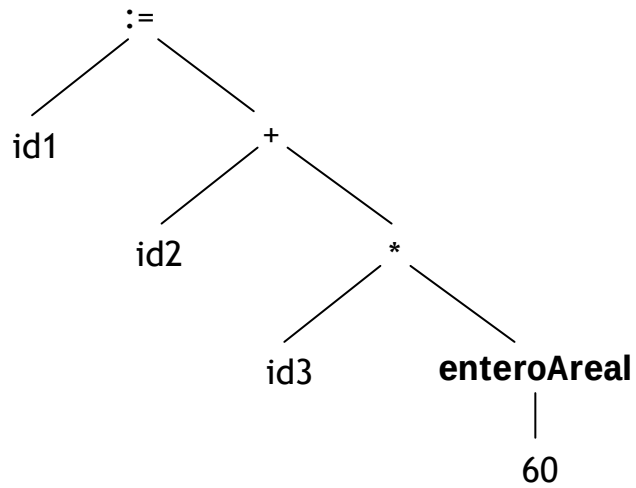
En la tabla de símbolos tendremos estas tres variables registradas de tipo real. Veamos las fases por las que pasaría dentro del compilador:
ANALIZADOR LÉXICO

Id1 = id2 + id3 * 60

ANALIZADOR SINTÁCTICO



ANALIZADOR SEMÁNTICO



GENERADOR DE CÓDIGO INTERMEDIO

temp1 := enteroAreal (60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3

OPTIMIZADOR DE CÓDIGO

temp1 := id3 * 60.0
id1 := id2 + temp1

GENERADOR DE CÓDIGO

MOVF id3, R2
MULT #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1

2 LENGUAJES Y GRAMÁTICAS.

Definiciones:

Alfabeto: Conjunto de símbolos que nos permiten construir las sentencias del lenguaje.

Tira de caracteres: Yustaposición o concatenación de los símbolos del alfabeto.

Tira nula (λ ó ε): Tira de longitud 0, es la tira mínima del lenguaje

Longitud de una tira: El número de caracteres que tiene la tira. Si tenemos una tira x, su logitud se representa como |x|.

Ejemplo.- x = abc |x| = 3
|λ| = 0

Potencia de una tira: Concatenación de una tira consigo misma tantas veces como indique el exponente.

Ejemplo.- x = abc
x^2 = abcabc
x^0 = λ
x^1 = abc

Cabeza de una tira: El conjunto de subtiras que podemos formar tomando caracteres desde la izquierda de la tira.

Ejemplo.- x = abc
Head (x) o Cabeza (x) = λ, a, ab, abc

Cola de una tira: Conjunto de subtiras que podemos formar tomando caracteres desde la derecha de la tira.

Ejemplo.- x = abc
Tail (x) o Cola (x) = λ, c, bc, abc

Producto Cartesiano: Sean A y B dos conjuntos de caracteres, el producto cartesiano entre ellos (AB) se define como:

$AB = \{ xy / x \in A, y \in B \}$

Definimos **Potencia** $A^n = A.A^{n-1}$; $A^0=\{\lambda\}$; $A^1 = A$
Definimos **Cierre Transitivo** $A^+ = U_{i>0} A^i$
Definimos **Cierre Transitivo y Reflexivo** $A^* = U_{i\geq 0} A^i$
A partir de estas dos últimas tenemos que: $A^* = A^+ \cup A^0 = A^+ \cup \{\lambda\}$

Definición: Un lenguaje estará formado por dos elementos: un diccionario y un conjunto de reglas.

Diccionario: significado de las palabras del lenguaje.
Conjunto de reglas (gramática): son las que indican si las sentencias construidas a partir de las palabras pertenecen o no al lenguaje.

Tenemos distitas **formas de definir un lenguaje**:

a) Enumerando sus elementos.
   Ejemplo.- $L_1 = \{ a, abc, d , ef, g \}$
b) Notación matemática.
   Ejemplo 1.- $L_2 = \{ a^n / n \in N\} = \{ \lambda, a, aa, ... \}$
   Ejemplo 2.- $L_3 = \{ x / |x| = 3 \text{ y } x \in V \}$
   $V = \{ a, b, c, d \}$
   ¿Cuántos elementos tiene el lenguaje?
   $VR_4^3 = 4^3 = 64$ elementos

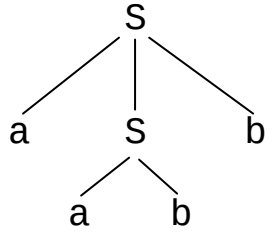
2.1 Notación de Chomsky (1959).

Esta notación define una gramática como una tupla (**N**, **T**, **P**, **S**)
N es el conjunto de los no terminales (conjunto de símbolos que introducimos en la gramática como elementos auxiliares que no forman parte de las sentencias del lenguaje).
T es el conjunto de los terminales (son los símbolos o caracteres que forman las sentencias del lenguaje).
P es el conjunto de reglas de producción (reglas de derivación de las tiras).
S es el axioma de la gramática, S está incluido en N (es el símbolo inicial o metanoción).

Ejemplo.-

$G1 = \{ \{S\}, \{a,b\}, P, S\}$
P: $S \rightarrow ab$
$S \rightarrow aSb$

Si aplicamos las reglas a la cadena "aabb", su árbol de derivación sería:



Lenguaje L(G): Es el conjunto de las tiras con símbolos terminales que podemos formar a partir del axioma de la gramática aplicando las reglas de producción P que pertenecen a la gramática.

2.2 Clasificación de Chomsky.

Esta clasificación define cuatro tipos dependiento de las reglas de producción de la gramática.

a) Gramáticas tipo 0, gramáticas con estructura de frase o reconocibles por una Máquina de Turing.

G = (N, T, P, S) $\quad \alpha \rightarrow \beta$
$\alpha \in (N \cup T)^+$ (NOTA: α nunca puede ser λ)
$\beta \in (N \cup T)^*$ (NOTA: β puede ser λ)

Con esta definición podemos observar que podría admitir cualquier estructura de frase.

b) Gramáticas tipo 1, gramáticas sensibles al contexto o dependientes del contexto.

G = (N, T, P, S) $\quad \alpha A \beta \rightarrow \alpha \nu \beta$
$\alpha, \beta \in (N \cup T)^*$
$\nu \in (N \cup T)^+$
$A \in N$

Se puede ver que, si llamamos $\alpha A \beta = x1$ y $\alpha \nu \beta = x2$, entonces $|x2| \geq |x1|$

Basándonos en esta propiedad podemos definir este tipo de gramáticas también de la siguiente forma (tiene una peculiaridad que veremos al final):

G = (N, T, P, S) $\quad \omega \rightarrow \nu$
$\omega, \nu (N \cup T)^+$
$|\omega| \leq |\nu|$

c) Gramáticas tipo 2, gramáticas de contexto libre (GCL) o independientes del contexto (GIC).

G = (N, T, P, S) $\quad A \rightarrow \alpha$
$A \in N$
$\alpha \in (N \cup T)^*$

Este tipo de gramática es la de los lenguajes de alto nivel, será la que usaremos durante el curso.

d) Gramáticas tipo 3, gramáticas regulares.

G = (N, T, P, S) $\quad A \rightarrow aB \;;\; A \rightarrow a \;;\; A \rightarrow \lambda$
$a \in T$
$A, B \in N$

Las gramáticas tipo 3 son las más restrictivas y las tipo 0 las menos pero existe una peculiaridad porque, de forma teórica extricta, un lenguaje independiente del contexto es también un lenguaje dependiente del contexto, pero una gramática independiente del contexto puede no ser una gramática dependiente del contexto. Esto ocurrirá si la gramática contiene reglas λ.

También existe una pequeña diferencia entre las dos definiciones mostradas de gramáticas tipo 1. Ejemplo.-

G = (N, T, P, S)
$S \rightarrow ASBc$
$bB \rightarrow bb$
$S \rightarrow aBc$
$BC \rightarrow bc$
$CB \rightarrow BC$
$CC \rightarrow cc$
$AB \rightarrow ab$

Es tipo 1 según la 2ª definición de tipo 1, pero es tipo 0 si usamos la 1ª definición de tipo 1 pues la regla $CB \rightarrow BC$ no la cumple.

2.3 Gramáticas de contexto libre (GCL).

Recordemos la definición:

G = (N, T, P, S) $A \rightarrow \alpha$
$A \in N$
$\alpha \in (N \cup T)^*$

Derivación: $\alpha \Rightarrow \beta$ (se lee α deriva β), α y β son dos tiras, si se puede escribir:

$\alpha = \delta A \mu$, $\beta = \delta \nu \mu$, δ y μ son dos tiras y aplicamos $A \rightarrow \nu$

δ y μ pueden ser tiras nulas, con lo cual sería una derivación directa.

Una derivación de longitud n, de la forma:

$\alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow .... \Rightarrow \alpha_n$

se representará como:

$\alpha_0 \Rightarrow^+ \alpha_n$
$\alpha_0 \Rightarrow^* \alpha_n$ (incluyendo la derivación nula, sin cambio)

Ejemplo.- Gramática $S \rightarrow aSb \mid \lambda$
$S \rightarrow aSb \rightarrow aaSbb \rightarrow aaaSbbb \rightarrow aaaaSbbbb \rightarrow aaaaaSbbbbb$
Es una derivación de longitud 5.

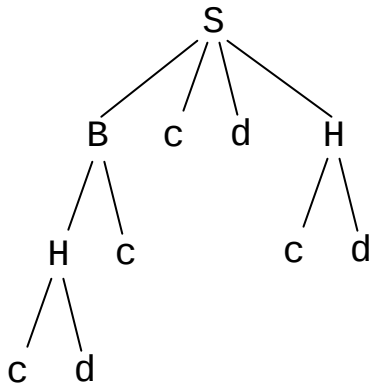
Sentencias, con G = (N, T, P, S), son el conjunto de tiras de la siguiente forma, $L(G) = \{ \omega / S \Rightarrow^* \omega, \omega \in T^* \}$.

Formas sentenciales, con G = (N, T, P, S), son el conjunto de combinaciones de terminales y no terminales que se pueden generar aplicando las reglas de derivación de la siguiente forma, $D(G) = \{ \alpha / S \Rightarrow^* \alpha, \alpha \in (N \cup T)^* \}$

Un **árbol sintáctico** es una representación gráfica de una regla que se realiza mediante un árbol. Cada nodo es la parte izquierda de una regla aplicada (siendo la raíz del árbol la metanoción); a continuación se colocan tantas ramas como nodos terminales o no terminales tengamos en la regla en su parte derecha, incluyendo todas las derivaciones que se hayan hecho.

Ejemplo.-

$S \rightarrow BcdH$
$B \rightarrow Hc$
$H \rightarrow cd$

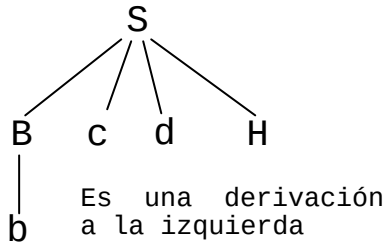


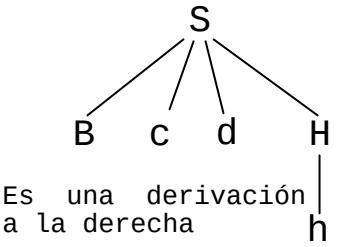
Derivación a la izquierda: es la derivación que se realiza sustituyendo las metanociones que se encuentran más a la izquierda de la forma sentencial.

Derivación a la derecha: es la derivacion que se realiza sustituyendo las metanociones que se encuentran más a la derecha de la forma sentencial.

Ejemplo.-

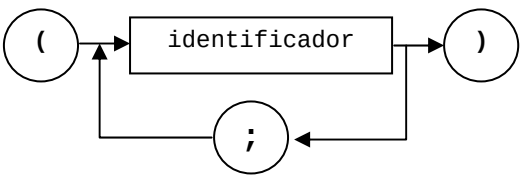
$S \rightarrow BcdH$
$B \rightarrow b$
$H \rightarrow h$





2.4 Diagramas de Conway.

Este tipo de diagrama se utilizó por primera vez para representar el lenguaje PASCAL. Es un grafo dirigido formado por rectángulos y círculos, el rectángulo corresponde a los no terminales y el círculo indica un símbolo terminal. Servirá para representar gráficamente las reglas de producción de la gramática de un lenguaje.



2.5 Reglas BNF.

La notación Backus-Naur (más comunmente conocida como BNF o *Backus-Naur Form*) es una forma matemática de describir un lenguaje. Esta notación fue originalmente desarrollada por John Backus basándose en los trabajos del matemático Emil Post y mejorada por Peter Naur para describir la sintaxis del lenguaje Algol 60. Por ello Naur la denominaba BNF por *Backus Normal Form*, mientras en la bibliografía aparece como *Backus-Naur Form*.

Esta notación nos permite definir formalmente la gramática de un lenguaje evitando la ambigüedad, definiendo claramente qué está permitido y qué no lo está. La gramática ha de ser, al menos, una **gramática de contexto libre**, de tipo 2. De hecho, existe una gran cantidad de teoría matemática alrededor de este tipo de gramáticas, pudiéndose construir mecánicamente un reconocedor sintáctico de un lenguaje dado en forma de gramática BNF (en realidad en algunos tipos de gramáticas esto es imposible, aunque pueden ser modificadas manualmente para que sí lo sea, de ello hablaremos a continuación).

La forma de una regla BNF es la que sigue:

símbolo → alternativa1 | alternativa2 ...

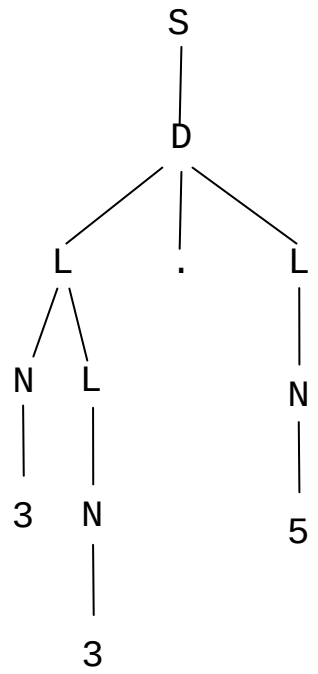
NOTA: También se puede utilizar, en lugar de →, el símbolo :=

Ejemplo.-

S → - D | D
D → L | L . L
L → N | N L
N → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

S es la metanoción
D es un número decimal
L es una lista de dígitos
N es un número, un dígito

Por ejemplo, el árbol para el número "33.5" sería el que se muestra a la derecha.



2.6 Problemas en las GCL.

En este apartado describiremos los distintos problemas que nos vamos a encontrar en las GCL así como los mecanismos de los que disponemos para resolverlos.

2.6.1 Recursividad.

Tenemos diferentes tipos de recursividad. Supongamos que tenemos una regla de la forma $A \rightarrow \alpha A \beta$

a) Si $\alpha = \lambda$, $A \rightarrow A\beta$, $A \in N$, $\beta \in (N \cup T)^*$ diremos que la regla es recursiva por la izquierda.

b) Si $\beta = \lambda$, $A \rightarrow \alpha A$, $\alpha \in (N \cup T)^*$, diremos que la regla es recursiva por la derecha.

c) Si $\alpha \neq \lambda$ y $\beta \neq \lambda$ diremos que se trata de una recursividad autoimbricada.

PROBLEMA: Recursividad por la izquierda.

La recursividad por la izquierda nos creará problemas pues en el desarrollo del árbol, que habitualmente analizaremos de izquierda a derecha. Solucionar este problema será especialmente interesante en el caso del análisis sintáctico (parsing).

Supongamos que tenemos:

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_q \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_p$

Esto podemos ponerlo de la siguiente forma:

$A \rightarrow A\alpha_i \qquad 1 \leq i \leq q$
$A \rightarrow \beta_j \qquad 1 \leq j \leq p$

Estas reglas podemos transformarlas de la siguiente forma sin variar el lenguaje generado:

$A \rightarrow \beta_j A' \qquad 1 \leq j \leq p$
$A' \rightarrow \alpha_i A' \text{ , } A' \rightarrow \lambda \qquad 1 \leq i \leq q$

Por ejemplo, si tenemos las reglas:

$A \rightarrow Aa \mid Ab \mid c \mid d$

La transformación podría ser la siguiente:

$A \rightarrow cA' \mid dA'$

$A' \rightarrow aA' \mid bA' \mid \lambda$

Si no queremos introducir reglas λ en la gramática la transformación podría ser de la siguiente forma:

$A \rightarrow \beta_j, \quad A \rightarrow \beta_j A' \qquad 1 \leq j \leq p$
$A' \rightarrow \alpha_i, \quad A' \rightarrow \alpha_i A' \qquad 1 \leq i \leq q$

Siguiendo el ejemplo anterior, la transformación sería:

$A \rightarrow c \mid d \mid cA' \mid dA'$
$A' \rightarrow a \mid b \mid aA' \mid bA'$

Ejemplo 1.-

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid x$

Vamos a resolver la recursividad por la izquierda paso a paso:

En el caso de $E \rightarrow E + T \mid T$. Esto es de la forma $A \rightarrow A\,\alpha \mid \beta$ con α = "+ T" y β = "T". Cambiándolo como $A \rightarrow \beta\ A'$, $A' \rightarrow \alpha\ A' \mid \lambda$, tenemos:

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

En el caso de $T \rightarrow T * F \mid F$ hacemos lo mismo:

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

La gramática resultante es:

$E \rightarrow T\ E'$
$E' \rightarrow + T\ E' \mid \lambda$
$T \rightarrow F\ T'$
$T' \rightarrow * F\ T' \mid \lambda$
$F \rightarrow (E) \mid x$

Ejemplo 2.-

$S \rightarrow uBDz$		$S \rightarrow uBDz$
$B \rightarrow Bv \mid w$	===> transformamos así ===>	$B \rightarrow wB'$
	=>	$B' \rightarrow vB' \mid \lambda$
$D \rightarrow EF$		$D \rightarrow EF$
$E \rightarrow y \mid \lambda$		$E \rightarrow y \mid \lambda$
$F \rightarrow x \mid \lambda$		$F \rightarrow x \mid \lambda$

2.6.2 Reglas con factor repetido por la izquierda.

Si tenemos una regla de la forma:

$A \rightarrow \alpha B \mid \alpha C$

Cuando vamos a realizar el reconocimiento de izquierda a derecha, al encontrar un α, algunos analizadores no podrán decidir cual de las dos reglas aplicar. Para solucionar este problema realizamos una factorización de la siguiente forma, sin variar el lenguaje:

$A \rightarrow \alpha A'$
$A' \rightarrow B \mid C$

2.6.3 Ambigüedad.

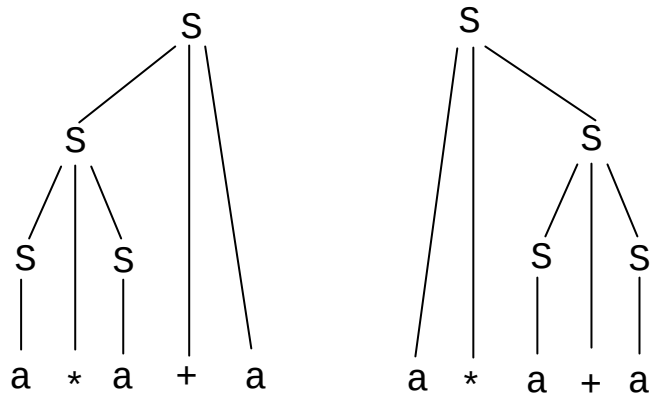
Diremos que una gramática G es ambigua si el lenguaje definido por ella tiene alguna sentencia para la cual existe más de un árbol sintáctico. Hay que tener claro que la que es ambigua es la gramática, no el lenguaje y que con una gramática ambigua no será posible implementar un analizador. Asimismo, siempre es posible modificar las reglas de la gramática para solucionar la ambigüedad y manteniendo el mismo lenguaje.

Ejemplo.-

G = { {s}, {a,+,*}, P, S }

P: $S \rightarrow a$
$S \rightarrow S+S$
$S \rightarrow S*S$

Si dibujamos el árbol de derivación para "a*a+a", nos puede salir de dos formas distintas:



Aquí se ve claramente que la gramática es ambigua, pues podemos aplicar indistintamente la segunda o la tercera regla.

Existen algunas soluciones para este tipo de ambigüedad sin reescribir la gramática y como un "mecanismo externo":

a) Dar mayor prioridad al producto que a la suma.
b) En caso de igualdad en la prioridad obligar al uso de paréntesis.

2.7 Simplificación de gramáticas.

Nos permiten eliminar símbolos no terminales o reglas que no aportan nada al lenguaje.

Sabemos que un lenguaje podemos describirlo como:

$L(G) = \{ \omega \ / \ S \Rightarrow^* \omega, \omega \in T\}$

2.7.1 Detección de un lenguaje vacío.

Vamos a ver un algoritmo para saber si una determinada gramática genera el lenguaje vacío.

Algoritmo $L(G) \neq \varnothing$?

```
BEGIN
        viejo = {∅}
        nuevo = {A / (A → t) ∈ P, t ∈ T*}
        WHILE nuevo ≠ viejo do BEGIN
                viejo := nuevo
                nuevo := viejo U { B / (B → α) ∈ P, α ∈ (T U viejo)*}
        END
        IF S ∈ nuevo THEN vacio := "NO" ELSE vacio := "SI"
END.
```

Ejemplo.-

$S \rightarrow Ac$
$A \rightarrow a \mid b$

Aplicando el algoritmo, primero tendríamos el conjunto {A}, pues es el único que contiene en su parte derecha únicamente terminales. En la segunda pasada ya se incluiría S pues $S \rightarrow Ac$, una combinación de un terminal y un elemento del conjunto de la pasada anterior, es decir {A}.

2.7.2 Eliminación de reglas lambda (λ).

Una regla-λ es de la forma $A \rightarrow \lambda$. Se dice una **gramática sin lambda** si verifica que no existen reglas de la forma $A \rightarrow \lambda$ en P excepto en el caso de la metanoción, para la cual puede existir la regla de la forma $S \rightarrow \lambda$ siempre y

cuando S no esté en la parte derecha de una regla de P. Es decir, λ no puede ser un elemento del lenguaje, si no no se puede eliminar completamente (debemos de mantener $S \rightarrow \lambda$).

Un no terminal A es **anulable** si existe una regla de la forma $A \Rightarrow^* \lambda$.

TEOREMA

Sea el lenguaje L=L(G), con una gramática de contexto libre G=(N, T, P, S) entonces L - {λ} es L(G´) con G´sin símbolos inútiles ni reglas-λ.

Algoritmo de detección de símbolos anulables

```
BEGIN
        viejo := {∅}
        nuevo := {A / A → λ ∈ P}
        WHILE viejo ≠ nuevo DO
                BEGIN
                        viejo := nuevo
                        nuevo := viejo U {B / B → α, α ∈ viejo*}
                END
        ANULABLES := nuevo
END
```

Una vez que tenemos ANULABLES, P´se construye de la siguiente forma:

i) Eliminamos todas las reglas-λ excepto la $S \rightarrow \lambda$, si existe.
ii) Con reglas de la forma $A \rightarrow \alpha_0 B_0 \alpha_1 B_1 \alpha_2 B_2 \ldots \alpha_k B_k$ siendo $k \geq 0$, estando las B_i en el conjunto ANULABLES pero las α_i no, entonces añadimos en P´ las reglas de la siguiente forma $A \rightarrow \alpha_0 X_0 \alpha_1 X_1 \alpha_2 X_2 \ldots \alpha_k X_k$, en donde las X_i serán B_i o bien λ, realizando todas las combinaciones posibles.
iii) Si S está en ANULABLES añadimos a P´ la regla $S' \rightarrow S \mid \lambda$.

Ejercicio.-

$S \rightarrow aS \mid AB \mid AC$
$A \rightarrow aA \mid \lambda$
$B \rightarrow bB \mid bS$
$C \rightarrow cC \mid \lambda$

ANULABLES = { A, C, S}

$S' \rightarrow S \mid \lambda$
$S \rightarrow aS \mid a \mid AB \mid B \mid AC \mid A \mid C$
$A \rightarrow aA \mid a$
$B \rightarrow bB \mid bS \mid b$
$C \rightarrow cC \mid c$

2.7.3 Eliminación de reglas unitarias o reglas cadena.

Denominamos reglas unitarias o reglas cadena a las reglas del tipo $A \rightarrow B$, con $A, B \in N$. En cualquier caso, nos interesará eliminarlas pues no añaden nada a la gramática.
A continuación veremos un algoritmo para la eliminación de este tipo de reglas, partimos de una gramática G = (N, T, P, S) y la convertimos en una gramática G´ = (N´, T, P´, S). Como condición necesaria tenemos que las gramáticas no tengan reglas λ.

En el algoritmo primero construiremos una serie de conjuntos (NA, NB, ..), tantos como nodos no terminales tenga la gramática, utilizando este algoritmo para cada no terminal.

Algoritmo

```
BEGIN
        viejo := {∅}
        nuevo := {A}
        WHILE viejo ≠ nuevo DO
                BEGIN
                        viejo := nuevo
                        nuevo := viejo U {C / B →C, B ∈ viejo}
                END
        NA := nuevo
END
```

Una vez construidos los conjuntos NX, generamos el conjunto P´ de la siguiente forma:

Si $B \rightarrow \alpha \in P$, y no es una regla unitaria, entonces hacemos que $A \rightarrow \alpha \in P' \ \forall$ A en los cuales $B \in NA$.

NOTA: con este algoritmo también eliminaremos las derivaciones de la forma: $A \Rightarrow^* X$, es decir $A \rightarrow B \rightarrow C \rightarrow \ldots \rightarrow X$, y no solamente las cadenas directas.

Ejemplo.-

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

Aplicando el algoritmo, algo muy sencillo, vemos que los conjuntos NX son:

NE = {E, T, F}
NT = {T, F}
NF = {F}

Y la gramática sin reglas cadena resultante sería:

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

2.7.4 Eliminación de símbolos inútiles.

Dada una gramática G= (N, T, P, S), decimos que un símbolo X es un **símbolo útil** si tenemos una cadena de derivaciones:

$S \Rightarrow^* \alpha X \beta \Rightarrow^* t$, $t \in T^*$, α y $\beta \in (N \cup T)^*$

La primera parte ($S \Rightarrow^* \alpha X \beta$) indica que X es un "símbolo accesible", es decir, puedo conseguir una cadena que la contenga, por derivación desde el axioma de la gramática.

La segunda parte ($\alpha X \beta \Rightarrow^* t$) indica que X es un "símbolo terminable", es decir, podemos dar lugar, partiendo de él, a una cadena en la que todos los elementos estén en T^*.

Entonces, por negación de cualquiera de estas dos características tendríamos un **símbolo inútil**.

Algoritmo para la eliminación de símbolos no terminables

Veremos a continuación un algoritmo para, partiendo de una gramática G = (N, T, P, S), llegar a otra gramática G´= (N´, T, P´, S) en la que no existen símbolos no terminables, expresado en forma matemática:

$\forall A \in N' \; \exists t \in T^* / A \Rightarrow^* t$

Lo que haremos será incluir A en N´ si existe una regla de la forma $A \rightarrow t$ o si tenemos $A \rightarrow x_1x_2...x_n$ con $x_i \in (N' \cup T)$.

Algoritmo

```
BEGIN
        viejo = {∅}
        nuevo = { A / A → t ∈ P, t ∈ T*}
        WHILE viejo ≠ nuevo DO
                BEGIN
                        viejo := nuevo
                        nuevo :=  viejo U {B / B → α ∈ P, α ∈ (T U viejo)*}
                END
                N´ := nuevo
                P´ := reglas cuyos símbolos estén en (N´ U T)
END
```

Ejemplo.-

$A \rightarrow a$	A es terminable pues genera un terminal
$B \rightarrow c$	B es también terminable por la misma razón
$H \rightarrow Bd$	Como $B \rightarrow c$, H también es terminable

Algoritmo para la eliminación de símbolos no accesibles.

Este algoritmo transformará una gramática G = (N, T, P, S) en otra gramática G´= (N´, T´, P´, S) en la que no existen símbolos no accesibles, expresado en forma matemática:

$\forall\ x_i \in (N' \cup T'), \exists\ \alpha,\beta \in (N' \cup T')^* / S \Rightarrow^* \alpha x_i \beta$

Algoritmo

```
BEGIN
        viejo := {S}
        nuevo := {x / S → αxβ ∈ P}
        WHILE viejo ≠ nuevo DO
                BEGIN
                        viejo := nuevo
                        nuevo := viejo U {y / A → αyβ, A ∈ viejo}
                END
        N´:= nuevo ∩ N
        T´:= nuevo ∩ T
        P´:= Las reglas que hacen referencia a N´ U T´
END
```

Es importante hacer notar que para la eliminación de símbolos inútiles es preciso aplicar los dos algoritmos anteriores y en este mismo orden.

Ejercicio.- Eliminar los símbolos inútiles en la siguiente gramática.
G = (N, T, P, S)
N = {S, A, B, C, D}
T = {a, b, c}
$S \rightarrow aAA$
$A \rightarrow aAb \mid aC$
$B \rightarrow BD \mid Ac$
$C \rightarrow b$

Aplicamos primero el algoritmo de eliminación de símbolos no terminables:

	viejo	**nuevo**	
Pasada 1	∅	C	Regla que genera un terminal
Pasada 2	C	C, A	C es generado por A
Pasada 3	C, A	C, A, S, B	A es generado por S y por B
Pasada 4	C, A, S, B	C, A, S, B	Aquí paramos

La nueva gramática es la siguiente:

N´= {C, A, S, B}
S → aAA
A → aAb | aC
B → Ac
C → b

Ahora es cuando podemos aplicar el segundo algoritmo, el de eliminación de símbolos no accesibles:

	viejo	nuevo	
Pasada 1	S	S, a, A	
Pasada 2	S, a, A	S, a, A, b, C	
Pasada 3	S, a, A, b, C	S, a, A, b, C	

T´= { a, b }
N´= { S, A, C }
S → aAA
A → aAb | aC
C → b

2.8 Gramática limpia.

Decimos que una gramática G = (N, T, P, S) es limpia (en mucha de la bibliografía aparece como "propia" por una mala traducción del francés "propre", que significa limpia) si no posee ciclos, no tiene reglas λ y no tiene símbolos inútiles.

Una gramática sin ciclos es aquella que no tiene reglas de la forma $A \Rightarrow^{+} A$, con $A \in N$. Es importante hacer notar que eliminando reglas-λ, reglas unitarias o cadena y símbolos inútiles ya resolvemos el problema.

2.9 Forma normal de Chomsky (FNC).

Una gramática G=(N, T, P, S) está en forma normal de Chomsky si las reglas son de la forma: A → BC ó A → a, con A,B,C ∈ N y a ∈ T. Como se puede observar, la FNC no admite recursividad en S.

A continuación veremos un algoritmo para convertir una gramática a la forma normal de Chomsky. Para ello, exigimos a la gramática original, G, que sea una gramática propia y sin reglas unitarias

Algoritmo

BEGIN
    P´= {∅}
    Añadimos a P´ las reglas del tipo A → a

Añadimos a P´ las reglas del tipo $A \rightarrow BC$
IF $A \rightarrow x_1\ x_2\ ...x_k \in P$, $k>2$ THEN
BEGIN
Añadimos a P´ $A \rightarrow x_1\,'< x_2\ ...x_k>$
$< x_2\ ...x_k> \rightarrow x_2\,'< x_3\ ...x_k>$
$< x_3\ ...x_k> \rightarrow x_3\,'< x_4\ ...x_k>$
$< x_{k-2} > \rightarrow x_{k-1}x_k$
END
IF $A \rightarrow x_1\ x_2 \in P$, si $x_1 \in T$ ó $x_2 \in T$ THEN
BEGIN
Llamamos x_a al $x_i \in T$
Sustituímos x_a por $x_a\,'$ en esa regla de P´
Añadimos a P´ $x_a\,' \rightarrow x_a$
END
END

Ejercicio.- Convertir la siguiente gramática a la FNC.

$S \rightarrow BA$
$A \rightarrow 01ABO$
$A \rightarrow 0$
$B \rightarrow 1$

El único caso problemático es el de $A \rightarrow 01AB0$, lo resolvemos así.
$A \rightarrow 01AB0$ **$A \rightarrow A_1A_2$** **$A_1 \rightarrow 0$** $A_2 \rightarrow 1AB0$ **$A_2 \rightarrow A_3A_4$**
$A_3 \rightarrow 1$ $A_4 \rightarrow ABO$ **$A_4 \rightarrow AA_6$** **$A_5 \rightarrow B0$** **$A_5 \rightarrow BA_6$**
$A_6 \rightarrow 0$

En negrita están las reglas que definitivamente quedan en la gramática.

2.10 Resumen.

Resumiendo, los problemas que podemos encontrarnos en las gramáticas son:

i) Recursividad por la izquierda.
ii) Factor repetido por la izquierda.
iii) Ambigüedad.

Como resumen de la simplificación de gramáticas, los pasos a seguir son los siguientes:

i) Eliminar reglas-λ.
ii) Eliminar reglas unitarias o cadena.
iii) Eliminar símbolos inútiles (1º no terminables y 2º no accesibles).

2.11 Ejercicios.

Ejercicio1.-

Simplificar la siguiente gramática:

$S \rightarrow aS \mid AB \mid AC$
$A \rightarrow aA \mid \lambda$
$B \rightarrow bB \mid bS$
$C \rightarrow cC \mid \lambda$

Solución 1.-

$S' \rightarrow aS \mid a \mid AB \mid bB \mid bS \mid b \mid AC \mid aA \mid a \mid cC \mid c \mid \lambda$
$S \rightarrow aS \mid a \mid AB \mid bB \mid bS \mid b \mid AC \mid aA \mid a \mid cC \mid c$
$A \rightarrow aA \mid a$
$B \rightarrow bB \mid bS \mid b$
$C \rightarrow cC \mid c$

Ejercicio 2 .-

$S \rightarrow ACA \mid CA \mid AA \mid C \mid A \mid \lambda$
$A \rightarrow aAa \mid aa \mid B \mid C$
$B \rightarrow bB \mid b$
$C \rightarrow cC \mid c$

Solución 2.-

$S \rightarrow ACA \mid CA \mid AA \mid cC \mid c \mid aAa \mid aa \mid bB \mid b \mid \lambda$
$A \rightarrow aAa \mid aa \mid bB \mid b \mid cC \mid c$
$B \rightarrow bB \mid b$
$C \rightarrow cC \mid c$

Ejercicio 3.-

$S \rightarrow AC \mid BS \mid B$
$A \rightarrow aA \mid aF$
$B \rightarrow cC \mid D$
$D \rightarrow aD \mid BD \mid C$
$E \rightarrow aA \mid BSA$
$F \rightarrow bB \mid b$

Solución 3.-

Vacío

3 ANÁLISIS LÉXICO.

En el análisis léxico analizamos token a token el archivo fuente. El analizador léxico se comunica con la tabla de símbolos (TDS) en donde se encuentra la información sobre los tokens, asimismo, también se comunica con el módulo de errores.

El scanner abre el fichero fuente y convierte los datos a un formato más regular, deshaciendo marcos, comprobando los formatos si el lenguaje es de formato fijo, etc. Posteriormente pasa a analizar los tokens (variables, instrucciones, etc).

Una **máquina reconocedora** o **autómata** es un dispositivo formal que nos permite identificar un lenguaje en función de la aceptación o no de las cadenas que lo forman. Esto es, crearemos un autómata que nos reconozca si una cadena determinada pertenece o no al lenguaje.

Esta máquina reconocedora tendrá una cinta de entrada dividida en celdas con unos símbolos que denotaremos como **Te**. También tendremos un símbolo especial que será el "**$**" que nos indicará el fin de la tira de entrada. A veces también dispondremos de otro alfabeto, denominado **Tp**, en el caso de utilizar autómatas con pila.

Vamos a tener movimientos que dan lugar a almacenamientos en memoria o cambios de estado. Para realizar este trabajo dispondremos de una memoria con posibilidad de búsqueda y un sistema de control de estados para manejar la transición entre los estados.

Denominamos **configuración del autómata** al conjunto de elementos que definen la situación del autómata en un momento t. Existen dos configuraciones especiales: inicial y final, un autómata ha reconocido una cadena cuando pasa del estado inicial a alguno de los estados finales. Aquí podemos decir que el lenguaje es el conjunto de tiras que reconoce el autómata.

Hablaremos de dos tipos de autómatas:

i) **Determinista**: Dada una entrada existe una única posibilidad de transición desde un estado determinado con una entrada determinada (si es con pila, usando también el mismo valor de la pila). $S_i \rightarrow S_j$.
ii) **No determinista**: Ante una entrada (si es con pila, teniendo en cuenta el mismo valor en la pila también) existirán varios estados a los que puede transitar el autómata. $S_i \rightarrow S_j$ o bien $S_i \rightarrow S_k$.

3.1 Tipos de máquinas reconocedoras o autómatas.

a) Autómatas finitos, reconocen lenguajes regulares (Gramáticas tipo 3). Podemos representarlo como una expresión regular, a partir de ella obtener un Autómata Finito No Determinista (AFND), para luego convertirlo en un Autómata Finito Determinista (AFD).
b) Autómatas con pila, reconocen lenguajes tipo 2 (Gramáticas de contexto libre, tipo 2)
c) Autómatas bidireccionales o autómatas lineales acotados (LBA). Para Gramáticas tipo 1. La cinta de entrada de la máquina está limitada a la longitud de la entrada a reconocer.
d) Máquinas de Turing para el reconocimiento de lenguajes tipo 0.

3.2 Autómatas Finitos.

Un autómata finito podemos definirlo como una tupla de la forma:

$AF = (Q, Te, \delta, q_0, F)$

Q es el conjunto de estados.
Te es el alfabeto de entrada.
q_0 es el estado inicial.
F es el conjunto de estados finales ($F \subset Q$).
δ es la función:

$$Q \times Te \rightarrow P(Q)$$

La configuración del autómata la representaremos con:
(q, t) q es el estado y t es la tira por ejecutar.

Dos configuraciones especiales serán:
(q_0, ω) estado inicial, tenemos la tira completa.
(q_f, λ) estado final y tira vacía.

Llamamos **movimiento** a la transición de un estado a otro y lo representamos de la siguiente forma:

$(q, a\alpha) \vdash (q', \alpha)$

Entonces decimos que $q' = \delta(q,a)$, en el caso de un AFD q´ sería el único estado al que podría transitar, en el caso de un AFND existirían varios estados, es decir, $\delta(q,a)$ sería un conjunto.

Para representar k pasos pondremos:

$\vdash^k$

El lenguaje definido por un AF podemos expersarlo como:

L(AF) = { t / t $\in$Te*, (q_0,t) |--*-- (q_i,λ), qi $\in$ F}

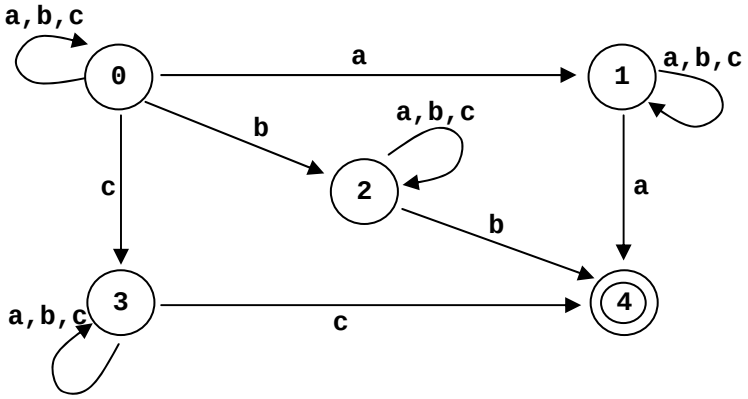
Ejemplo.-

AF = (Q,Te, δ, q_0, F)

δ	**a**	**b**	**c**
0	0, 1	0, 2	0, 3
1	1, 4	1	1
2	2	2, 4	2
3	3	3	3, 4
4	-	-	-

Como se puede observar, se trata de un AFND pues existen varias transiciones para el mismo estado y entrada.

Este sería el diagrama de transiciones:



Para realizar un compilador no es una buena opción utilizar un AFND pues el recorrido del árbol es más lento (habrá muchas ramas inútiles). Normalmente nos interesará convertir el AFND en un AFD, algo, que como nos muestra el teorema que viene a continuación, es siempre posible.

Teorema

Sea L un conjunto aceptado por un AFND, entonces existe un AFD que acepta L.

Teorema

Sea **r** una expresión regular, entonces existe un AFND con transiciones λ que acepta L(r).

Teorema

Si L es aceptado por un AFD, entonces L se donota por una expresión regular.

Estos tres teoremas nos permiten decir lo siguiente:

i) L es un conjunto regular.
ii) L se denota por una expresión regular.
iii) L puede ser definido por un AFND.
iv) Cualquier AFND puede ser transformado en un AFD.

3.3 Conversión de una Gramática Regular en un Autómata finito.

Aquí veremos como podemos convertir una gramática regular en un autómata finito, para ello utilizaremos un ejemplo:

Una gramática regular es de la forma:

$G = (N, T, P, S)$ $\quad A \rightarrow aB \;;\; A \rightarrow a \;;\; A \rightarrow \lambda, a \in T, A, B \in N$

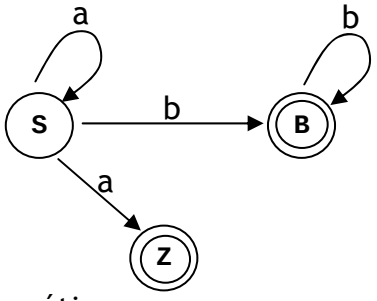
Vamos a convertirla en un autómata AFND (Q, Te, δ, q_0, F) en el cual:

i) $Q = N \cup \{Z\}$, $Z \notin N$ si P contiene alguna regla de la forma $A \rightarrow a$
   $Q = N$ en otro caso (cuando sólo tenemos reglas de la forma $A \rightarrow \lambda$)
ii) $\delta(A, a) = B$ si $A \rightarrow aB \in P$
   $\delta(A, a) = Z$ si $A \rightarrow a \in P$
iii) $F = \{A / A \rightarrow \lambda \in P\} \cup Z$ si $Z \in Q$
   $F = \{A / A \rightarrow \lambda \in P\}$ en otro caso

Ejemplo 1.- Sea la gramática:

$S \rightarrow aS \mid bB \mid a$
$B \rightarrow bB \mid \lambda$

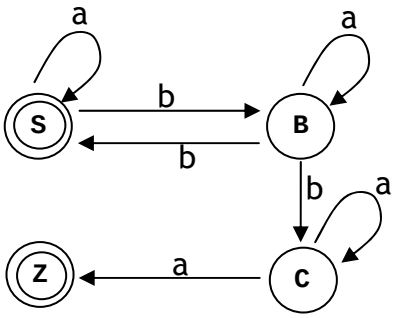
El autómata sería:



Ejemplo 2.- Sea la gramática:

$S \rightarrow aS \mid bB \mid \lambda$
$B \rightarrow aB \mid bS \mid bC$
$C \rightarrow aC \mid a$

El autómata sería:



3.4 Expresión regular.

Es una forma de definir un lenguaje asociado a las gramáticas de tipo 3. Una expresión regular se define a partir de una serie de axiomas:

i) $\varepsilon \equiv \lambda$ $|\varepsilon| = 0$ $\{\varepsilon\}$ es una expresión regular (ER).
ii) Si $a \in T$ (alfabeto), a es una ER.
iii) Si r,s son ER representaremos:
   (r) | (s) $\rightarrow$ L(r) U L(s)
   (r) (s) $\rightarrow$ L(r) L(s)
   (r)* $\rightarrow$ (L(r))*
   (r) $\rightarrow$ NOTA: pueden ponerse paréntesis

Propiedades:

r | s	≡	s | r	Conmutativa "|"
r | (s | t)	≡	(r | s) | t	Asociativa "|"
(r s)t	≡	r (s t)	Asociativa de la concatenación
r (s | t)	≡	r s | r t	Concatenación distributiva sobre "|"
(s | t) r	≡	s r | t r	Concatenación distributiva sobre "|"
r ε	≡	ε r ≡ r	ε elemento identidad para concatenación
r*	≡	(r | ε)*	
r**	≡	r*	Idempotencia

$L \cup M = \{ s / s \in L \text{ ó } s \in M\}$
$LM = \{ st / s \in L \text{ y } t \in M\}$

Además tenemos que:

$L^* = \cup^{\infty}_{i=0} L^i$
$L^+ = \cup^{\infty}_{i=1} L^i$

Lógicamente: $L^i = L^i L$

* tiene mayor precedencia, luego concatenación y por último la "|". Todas ellas son asociativas por la izquierda.

Llamamos **conjunto regular** al lenguaje asociado a una expresión regular.

Ejemplo.- Identificadores en Pascal
Su expresión regular sería: letra (letra | número)*

Ejemplo.-
Σ = T = alfabeto = {a, b}
a | b = {a, b}
a* = {ε, a, aa, ...}
(a | b) (a | b) = {aa, bb, ab, ba}
(a | b)* = (a*b*)*

NOTA: Algunas abreviaturas utilizadas en la bibliografía son las siguientes:

r? = r | ε
r^+ = rr*
r* = r^+ | ε
(r)? = L(r) U {ε}
[a, b, c] = a | b | c
[a-z] = a | b | c | ... | z

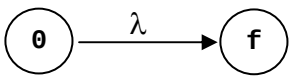
3.5 Algoritmo de Thompson.

Este algoritmo nos va a permitir convertir una expresión regular en un autómata finito no determinista con transiciones λ (AFND-λ).

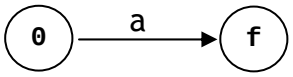
Un AFND-λ es un AFND al que incorporamos a Te un elemento más, el λ, con lo cual: Q x Te U {λ} $\rightarrow$ Q. Es decir, se puede mover de estado si usar entrada.

Los pasos a seguir son los siguientes:

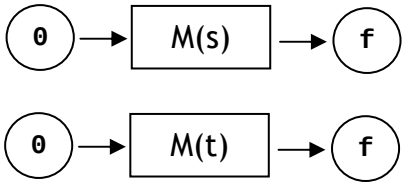
i) Para λ construiríamos:



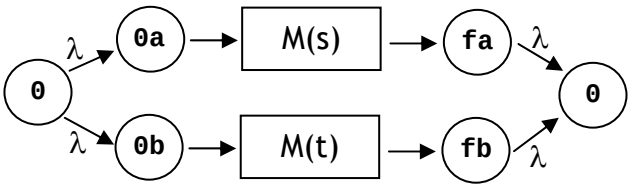
ii) $\forall$ a $\in$ Te construiríamos:



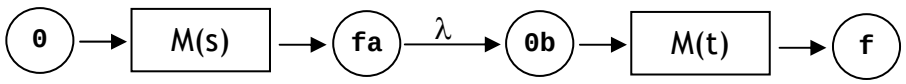
iii) Tenemos **M(s)**, **M(t)**, dos AFND para 2 expresiones regulares **s** y **t**, de la forma:



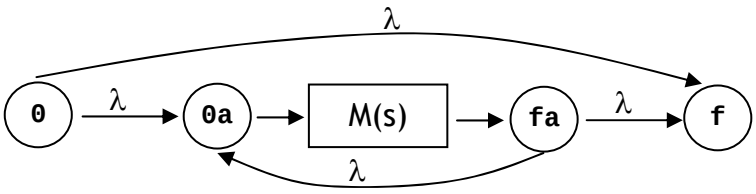
Entonces **s|t** podemos reconocerlo con un AFND-λ de la forma:



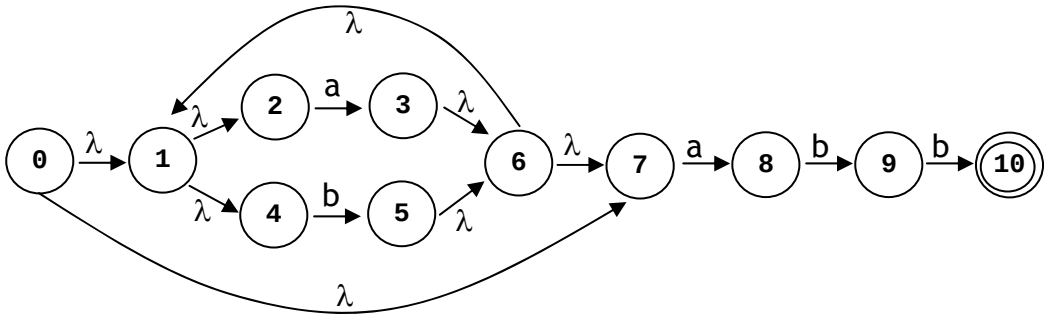
Entonces **st** podemos reconocerlo con un AFND-λ de la forma:



Entonces **s*** podemos reconocerlo con un AFND-λ de la forma:



Ejemplo.- Sea la expresión regular r = (a | b)* abb, crear el AFND-λ.



3.6 Transformación de un AFND-λ en un AFD.

Primeramente definimos:

Si S es un estado:
Cerradura-Σ (S) = { $S_k \in$ AFND / $S \rightarrow^{\lambda} S_k$}

Siendo W un conjunto de estados:
Cerradura-Σ (W) = { $S_k \in$ AFND / $\exists\ S_j \in W$, $S_j \rightarrow^{\lambda} S_k$}

Y la función de movimiento (llamémosle Mueve):
Mueve (W,a) = { $S_k \in$ AFND / $\exists\ S_j \in W$, $S_j \rightarrow^{a} S_k$}

El mecanismo para realizar la transformación consiste en:

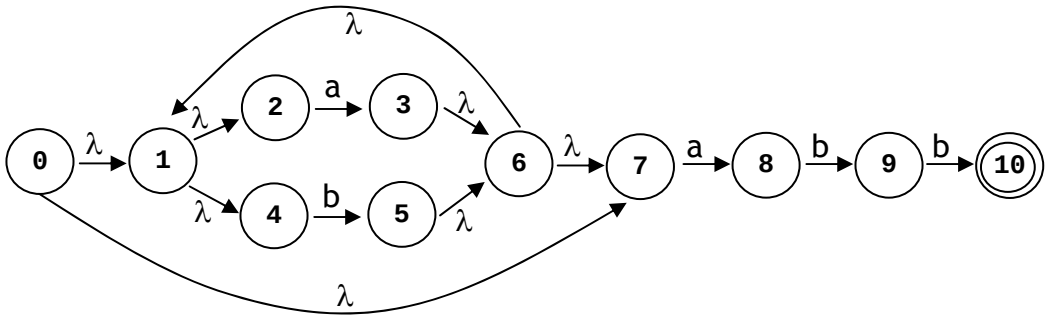
1) Calcular primero el nuevo estado A = Cerradura-Σ(0), es decir, la Cerradura-Σ del estado 0 del AFND-λ.
2) Calculamos Mueve (A, a_i), $\forall$ $a_i \in T$. Luego calculamos Cerradura-Σ (Mueve (A, a_i)), $a_i \in T$, es decir, para todos los terminales, y así vamos generando los estados B, C, D...
3) Calculamos Mueve (B, a_i), $\forall a_i \in T$, y hacemos lo mismo. Esto se repite sucesivamente hasta que no podamos incluír nuevos elementos en los conjuntos Cerradura-Σ.

Algoritmo

```
BEGIN
        Calcular Cerradura-Σ (0) = Estado A;
        Incluír A en NuevosEstados;
        WHILE no están todos los W de NuevosEstados marcados DO BEGIN
                Marcar W;
                FOR cada a_i ∈ Te DO BEGIN
                        X = Cerradura-Σ (Mueve (W, a_i));
                        IF X no está en el conjunto NuevosEstados añadirlo;
                        Transición [W, a] = X;
                END
        END
END
```

Serán estados finales todos aquellos que contengan alguno de los estados finales del autómata original.

Ejemplo.- Si tenemos el siguiente AFND-λ que reconoce la expresión regular r = (a | b)* abb, convertirlo en un AFD.



Cerradura-Σ (0) = {0, 1, 7, 4, 2} = A
Mueve (A, a) = {3, 8}
Mueve (A, b) = {5}

Cerradura-Σ (Mueve(A,a)) = {3, 6, 7, 1, 2, 4, 8} = B (A con a mueve a B)
Cerradura-Σ (Mueve(A,b)) = {5, 6, 7, 1, 2, 4} = C (A con b mueve a C)
Mueve (B, a) = {3, 8} = Mueve (A, a) (B con a mueve a B)
Mueve (B, b) = {5, 9}
Mueve (C, a) = {3, 8} = Mueve (A, a) (C con a mueve a B)
Mueve (C, b) = {5} = Mueve (A, b) (C con b mueve a C)

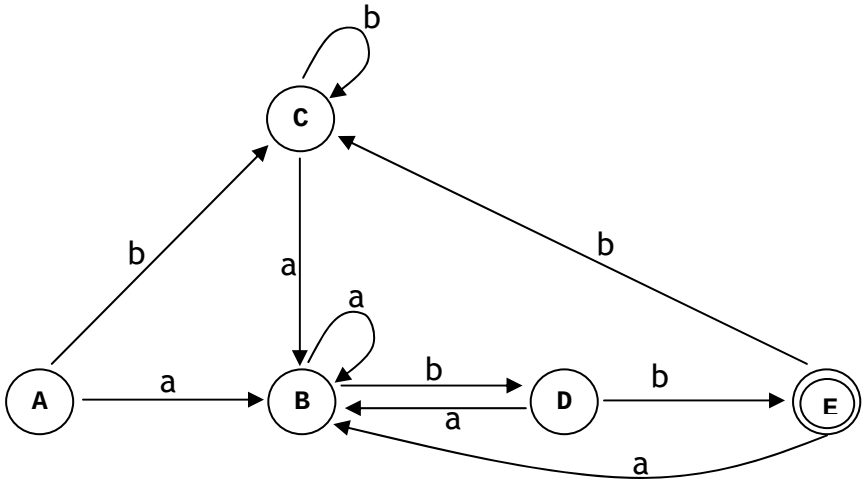
Cerradura-Σ (Mueve(B, b)) = {5, 6, 7, 1, 2, 4, 9} = D (B con b mueve a D)
Mueve (D, a) = {3, 8} = Mueve (A, a) (D con a mueve a B)
Mueve (D, b) = {5, 10}

Cerradura-Σ (Mueve(D, b)) = {5, 6, 7, 1, 2, 4, 10} = E (D con b mueve a E)
Mueve (E, a) = {3, 8} = Mueve (B, a) (E con a mueve a B)
Mueve (E, b) = {5} = Mueve (C, b) (E con b mueve a C)

Tabla de transiciones:

Estado	**a**	**b**
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Vamos a ver el autómata:



3.7 Traductores finitos (TF).

Será un autómata finito de la forma:

TF = (Q, Te, Ts, δ, q_0, F)

Q es el conjunto de estados.
Te es el alfabeto de entrada.
Ts es el alfabeto de salida.
q_0 es el estado inicial.
F son los estados finales (un subconjunto de Q).

δ: Q x {Te U {λ}} $\rightarrow$ Q x Ts*

La **configuración** del TF viene definida por una tupla (q, t, s), donde q es el estado actual, t es la tira de caracteres por analizar ($t \in Te^*$) y s es la tira que ha salido ($s \in Ts^*$).

Un **movimiento** es una transición de la forma:

$(q, a\omega, s)$ |---- (q', ω, sz) cuando $\delta(q, a) = (q', z)$
$q, q' \in Q$
$\omega \in Te^*$
$s \in Ts^*$

Podemos definir el **conjunto de traducción** como:

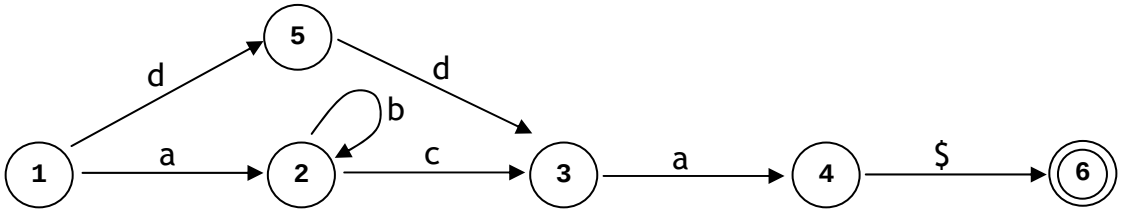
$Tr(TF) = \{(t, s), t \in Te^*, s \in Ts^* / (q_0, t, \lambda)$ |--*-- $(q_f, \lambda, s), q_f \in F\}$

Al igual que en el caso de los AF, pueden ser deterministas y no deterministas, dependiendo de si exíste una única transición posible o varias ante la misma entrada y el mismo estado actual.

3.8 Implementación de autómatas.

3.8.1 Tabla compacta.

Sea el siguiente lenguaje $L = \{dda\$, ab^nca\$ / n \geq 0\}$ y su autómata:



La tabla de transiciones (M) correspondiente es la siguiente:

	a	**b**	**c**	**d**	**$**
1	2			5	
2		2	3		
3	4				
4					6
5				3	
6					

Se precisa mucha cantidad de memoria para almacenar esta matriz al completo, entonces se utiliza la técnica de la tabla compacta, que consiste en definir los siguientes vectores:

iv) Valor.- En él se almacenan los elementos no nulos de la tabla.
v) Columna.- La columna en que se encuentra el elemento del vector valor.
vi) PreFil.- Este vector tiene tantos elementos como filas tenga la tabla y almacenaremos los números de orden en el vector valor que comienzan cada fila no nulos.
vii) NumFil.- Almacenamos el número de elementos no nulos que hay en cada fila.

En nuestro ejemplo sería:

Valor
2
5
2
3
4
6
3

Columna
1
4
2
3
1
5
4

PreFil
1
3
5
6
7
0

NumFil
2
2
1
1
1
0

El espacio ocupado viene dado por: (nº de filas x 2) + (nº no nulos x 2). En nuestro caso sería (6 x 2) + (7 x 2) = 26.

En el caso de este ejemplo no parece un ahorro de espacio muy sustancial, pero supongamos una tabla de tamaño 100 x 100. Si suponemos 110 valores no nulos tenemos que el espacio ocupado sería 10.000 sin comprimir y 420 implementando una tabla compacta.

Vamos a ver a continuación como se realiza el direccionamiento mediante un algoritmo.

Algoritmo

```
Function M (i, j: integer) : integer
VAR
        num, com, k: integer;
        hallado : boolean;
BEGIN
        num := NUMFIL[i];
        IF num = 0 THEN M := 9999;
        ELSE BEGIN
                com := PREFIL[i];
                hallado := FALSE;
                k := 0;
                WHILE (k < num) AND NOT hallado DO
                IF col[com+k] =  j THEN hallado := TRUE
                ELSE k := k + 1;
                IF hallado THEN M := VALOR[com + k]
```

```
            ELSE M := 9999;
        END;
END;
```

En nuestro ejemplo, si queremos localizar M[2, 3] haremos:

PreFil [2] = 3, NumFil[2] = 2 Entonces puede ser Valor[3] o Valor[4]
Como columna es 3 entonces M[2, 3] = 3.

3.8.2 Autómata programado.

Otra opción es hacer un programa para realizar las transiciones, de la siguiente forma:

```
estado = 1
WHILE estado ≠ 6 DO BEGIN
        LeerCarácter
        CASE estado OF
                1 :     IF Car="a" THEN estado = 2
                        ELSE IF Car="d" THEN estado = 5
                        ...
                2 :     IF Car="e" THEN estado = 7
                        ...
...
```

3.9 Ejemplo. Scanner para números reales sin signo en Pascal.

Vamos a reconocer números del tipo: 23E+12, 100E-15, ...

Reglas BNF:

<número> → <entero> | <real>

<entero> → <dígito> {<dígito>} con la { } indicamos 0 ó más veces

<real> → <entero> . <dígito> {<dígito>}
| <entero> . <dígito> {<dígito>} E <escala>
| <entero>
| <entero> E <escala>

<escala> → <entero>
| <signo> <entero>

<signo> → + | -

<dígito → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Esta gramática podemos ponerla en forma de gramática regular, que sería la siguiente:

<número> → 0 <resto_número>
<número> → 1 <resto_número>
...
<número> → 9 <resto_número>

Este tipo de reglas las resumiremos como:

<número> → dígito <resto_número> (10 reglas)

<resto_número> → dígito <resto_número> (10 reglas)
| . <fracción>
| E <exponente>
| λ

<fracción> → dígito <resto_fracción> (10 reglas)

<resto_fracción> →dígito <resto_fracción> (10 reglas)
| E <exponente>
| λ

<exponente> → signo <entero_exponente> (2 reglas, + y -)

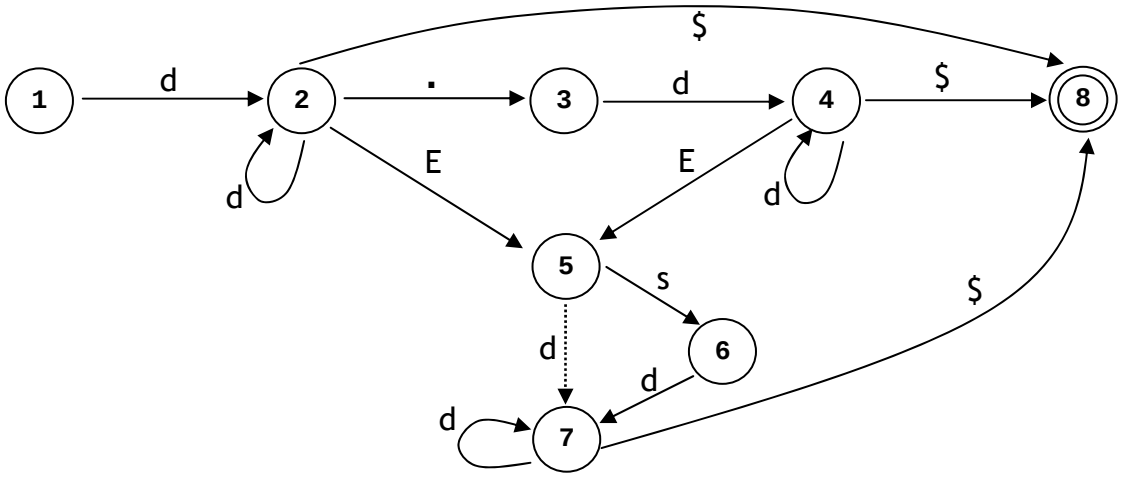
<entero_exponente> → dígito <resto_entero_exponente> (10 reglas)

<resto_entero_exponente> → dígito <resto_entero_exponente> (10 reglas)
| λ

Ahora implementaremos el autómata. Empezamos por definir los estados:

1 número 2 resto_número 3 fracción 4 resto_fracción
5 exponente 6 entero_exponente 7 resto_entero_exponente

El autómata sería:



La transición que marcamos punteada (de 5 a 7 con dígito), nos permite aceptar números sin signo en el exponente (por ejemplo.- 2E32), tal y como estaba originalmente teníamos que poner siempre un + o un -.
La tabla asociada es la siguiente:

ESTADO	Dígito	.	E	Signo	$
1	2				
2	2	3	5		8
3	4				
4	4		5		8
5	7			6	
6	7				
7	7				8

Ejemplo.- Gramática de las expresiones regulares

<expresión> → <literal> | <alternativa> |
<secuencia> | <repetición> | '(' <expresión> ')'
<alternativa> → <expresión> '|' <expresión>
<secuencia> → <expresión> <expresión>
<repetition> → <expresión> ' * '
<literal> → 'a' | 'b' | 'c' | ... ('a' | 'b' | 'c' | ...)*

Ejemplo.- Especificaciones de tiempo en UNIX

timespec = time | time date | time increment
| time date increment | time decrement
| time date decrement | nowspec

nowspec = NOW | NOW increment | NOW decrement

time = hr24clock | NOON | MIDNIGHT | TEATIME

date = month_name day_number
| month_name day_number ',' year_number
| day_of_week | TODAY | TOMORROW
| year_number '-' month_number '-' day_number

increment = '+' inc_number inc_period

decrement = '-' inc_number inc_period

hr24clock = INT 'h' INT

inc_period = MINUTE | HOUR | DAY | WEEK | MONTH | YEAR

day_of_week = SUN | MON | TUE | WED | THU | FRI | SAT

month_name = JAN | FEB | MAR | APR | MAY | JUN | JUL
| AUG | SEP | OCT | NOV | DEC

3.10 Acciones semánticas.

Estas acciones se pueden incluír en la tabla de transiciones. Por ejemplo, en el caso de las casillas en blanco, que son errores, se puede informar acerca del error en concreto.

En el ejemplo del apartado anterior, en la casilla [3, "."] podríamos mostrar el error de "Doble punto decimal" y realizar incluso una corrección automática.

Otra acción semántica interesante en el caso anterior sería ir almacenando en la Tabla de Símbolos el valor del número que se está introduciendo. Por ejemplo, una tabla para acciones semánticas de este tipo podría ser:

ESTADO	Dígito	$
1	T	
2	T	Z
3	U	
4	U	Z
5	V	
6	V	
7	V	Z

T: num = num * 10 + dígito (calculamos número)
U: n = n + 1 (calculamos fracción)
num = 10 * num + dígito
V exp = 10 * exp + dígito (calculamos exponente)

El cálculo final ser realizaría en Z, que sería:

Z: Valor = num * $10^{\text{signo exp - n}}$

3.11 Generador LEX.

Lex es una de las herramientas que pueden ser utilizadas para generar un analizador léxico a partir de una descripción basada en expresiones regulares.

Si tenemos un programa lex en un fuente con nombre *ejemplo.l*, los pasos para convertirlo en un ejecutable son los siguiente:

```
lex ejemplo.l
cc –o miejemplo lex.yy.c –ll
/*A veces puede ser necesario también –lm*/
miejemplo < fichero_prueba.txt
```

Si se utiliza Flex y Gcc (GNU) :

```
flex ejemplo.l
gcc –o miejemplo lex.yy.c –lfl
```

La secuencia es la siguiente:

1. Lex crea un fichero fuente en C, lex.yy.c, desde una descripción del analizador escrita en lenguaje Lex.
2. Lex.yy.c se compila (es un programa C) generándose el ejecutable.
3. Este ejecutable analizará una entrada lexicamente para producir una secuencia de tokens.

3.11.1 Partes de un programa LEX

La estructura de un programa en LEX es la siguiente:

Declaraciones

%%

Reglas de traducción

%%

Procedimientos Auxiliares

```
main()   /* Si se desea ejecutar algo más después del analizador*/
{
        yylex();
}
```

- En la parte de **Declaraciones** se incluyen identificadores, expresiones regulares, constantes, los “includes”, etc

- Las **reglas de traducción** son de la forma (siendo Pi una expresión regular):

```
P1    {Acción 1}
P2    {Acción 2}
...   ...
Pn    {Acción n}
```

Es importante hacer notar que el orden de las relgas es fundamental. Las expresiones regulares más específicas se deben de poner antes que las más generales (ejemplo: “.”, que es cualquier cosa, si se utiliza, debería de ponerse al final ya que en otro caso siempre tomaría esta expresión regular como válida).

Los **procedimientos auxiliares** son todos aquellos procedimientos en **C** que se van a utilizar en las reglas.

3.11.2 Expresiones regulares.

Algunos ejemplos de definición de expresiones regulares son los siguientes:

Si queremos reconocer números del tipo: 100, 111.02, 1, 111.2, etc, la expresión regular sería la siguiente:

`([0-9]*\.)?[0-9]+`

Vamos a revisarla por partes:

`[0-9]+`

Dígitos de 0 a 9 (los [] indican elección de uno de entre el rango), al añadir el + indicamos la repetición 1 o más veces de lo que lo precede.

`([0-9]*\.)`

El * significa repetición al igual que el +, aunque en este caso es de 0 o más veces, es decir lo que precede al * puede no aparecer. El “.” simplemente indicaría cualquier carácter excepto salto de línea, en este caso queremos identificar al punto decimal, por ello añadimos el carácter de escape “\”.

En la siguiente tabla se muestra un resumen de la sintaxis para la definición de expresiones regulares:

```
     Symbol  |  Meaning
-------------+------------------
          x  |  The "x" character
          .  |  Any character except \n
      [xyz]  |  Either x, either y, either z
      [^bz]  |  Any character, EXCEPT b and z
      [a-z]  |  Any character between a and z
     [^a-z]  |  Any character EXCEPT those between a and z
         R*  |  Zero R or more; R can be any regular expression
         R+  |  One R or more
         R?  |  One or zero R (that is an optionnal R)
     R{2,5}  |  Two to 5 R
```

```
      R{2,}  |  Two R or more
       R{2}  |  Exactly two R
"[xyz\"foo"  |  The string "[xyz"foo"
   {NOTION}  |  Expansion of NOTION, that as been defined above in the file
         \X  |  If X is a "a", "b", "f", "n", "r", "t", or
             |  "v", this represent the ANSI-C interpretation of \X
         \0  |  ASCII 0 character
       \123  |  ASCII character which ASCII code is 123 IN OCTAL
       \x2A  |  ASCII character which ASCII code is 2A in hexadecimal
         RS  |  R followed by S
        R|S  |  R or S
        R/S  |  R, only if followed by S
         ^R  |  R, only at the beginning of a line
         R$  |  R, only at the end of a line
     <<EOF>> | End of file
```

3.11.3 Paso de valores en Lex.

Si queremos devolver el valor de un token podemos utilizar **return**. Por ejemplo, si hemos definido el valor de NUMBER a 257 (#define NUMBER 257), podemos incorporar en nuestro scanner lo siguiente:

```
  [0-9]+  {return NUMBER;}
```

Existen dos variables para el paso de valores a funciones C o bien a un parser diseñado mediante la herramienta YACC. La variable externa **yytext** contiene el string de caracteres que son aceptados por la expresión regular. También tenemos la variable externa **yylval**, definida como un INTEGER (aunque puede ser redefinida), pensada inicialmente para el paso del token desde el analizador léxico al parser. Para asignar el valor de **yytext** a **yylval** es necesario realizar una conversión.

Se puede utilizar la función C **atoi** para convertir el número almacenado en **yytext** en forma de cadena de caracteres en su valor numérico en **yylval**:

```
[0-9]+    { yylval = atoi(yytext);
          return NUMBER;

          }
```

3.11.4 Ejemplos.

Ejemplo1. Identificador de números y palabras

```
digit     [0-9]
letter    [A-Za-z]
delim     [ \t\n]
ws        {delim}+

%%
{ws}      { /* ignore white space */ }
{digit}+  { printf("%s", yytext);  printf(" = number\n"); }
{letter}+ { printf("%s", yytext); printf(" = word\n";) }
%%
main()
{
```

```
      yylex();
}
```

Ejemplo 2. Un contador de palabras similar al wc de UNIX

```
%{
unsigned charCount = 0, wordCount = 0, lineCount = 0;
%}

word [^ \t\n]+
eol \n

%%
{word} {wordCount++; charCount += yyleng;}
{eol}  {charCount++; lineCount++; }
.      {charCount++;}
%%
main()
{
      yylex();
      printf("%d %d %d\n", lineCount, wordCount, charCount);
}
```

Ejemplo 3. Reconocedor de comentarios en lenguaje C.

```
%%
(.|\n)? {printf(""); }
"/*""/"*([^*/]|[^*]"/"|"*"[^/])*"*"*"*/" {printf("[comment]%s", yytext);}
%%

NOTA: Probar con el siguiente ejemplo:

/* comentarios en una linea */
/* comentarios sobre multiples
       lineas */
comentarios /*incluidos dentro de una */linea
comentarios /**/ raros
/*/Acepta comentarios que no lo son?
/**_/*/ Es correcto, lo acepta?
```

Ejemplo 4. Calculadora

```
%{
/*
**      lex program for simple dc
**      input of the form:  opd opr opd
**      output of the form: opd opr opd = ans
*/
#define NUM 1
#define PLUS 2
#define MINUS 3
#define MUL 4
#define DIV 5
#define MOD 6
#define DELIM 7
%}

ws    [ \t]+

%%
{ws}   ;
-?[0-9]+       {
```

```
        printf("%s",yytext);
        return (NUM);
        }

"+"     {
        ECHO;
        return (PLUS);
        }
"-"     {
        ECHO;
        return (MINUS);
        }
"*"     {
        ECHO;
        return (MUL);
        }
"/"     {
        ECHO;
        return (DIV);
        }
"%"     {
        ECHO;
        return (MOD);
        }
. |
"\ n"   {
        ECHO;
        return (DELIM);
        }



%%
/* main.c */
# include <stdio.h>

main()
{
int opd1, opd2, opr;

       /* Look for first operand. */
         if (yylex() != NUM)
                 {
                 printf("missing operand\n");
                 exit(1);
                 }
         opd1 = atoi(yytext);

       /* Check for operator. */
         opr = yylex();
       /* Bad cases - no operand - either a delimiter for a number. */
         if (opr == DELIM)
                 {
                 printf("missing operator\n");
                 exit(1);
                 }
       else if (opr == NUM)
              {
              if ((opd2=atoi(yytext)) >= 0)
                     {
                      printf("missing operator\n");
                      exit(1);
                     }
              /* Problem case - distinguish operator from operand. */
              else
                     {
                     opr = MINUS;
                     opd2 = -opd2;
```

```
                }
        }
    /* Check for second operand, if not yet found. */
     else if (yylex() != NUM)
            {
            printf("missing operand\n");
            exit(1);
            }
    else /* Must have found operand 2 */
          opd2 = atoi(yytext);

     switch (opr)
     {
     case PLUS:      {
                     printf(" = %d\n",opd1 + opd2);
                     break;
                     }
     case MINUS:     {
                     printf(" = %d\n",opd1 - opd2);
                     break;
                     }
     case MUL:       {
                     printf(" = %d\n",opd1 * opd2);
                     break;
                     }
     case DIV:       {
                 if (opd2 == 0)
                        {
                        printf("\nERROR: attempt to divide by zero!\n");
                        exit(1);
                        }
                 else
                        {
                        printf(" = %d\n",opd1 / opd2);
                        break;
                        }
                     }
     case MOD:       {
                 if (opd2 == 0)
                        {
                        printf("\nERROR: attempt to divide by zero!\n");
                        exit(1);
                        }
                 else
                        {
                               printf(" = %d\n",opd1 % opd2);
                        break;
                        }
                     }
     }
}
```

4 Análisis sintáctico (Parsing).

Analizar sintácticamente una tira de tokens es encontrar para ella el árbol de derivación (árbol sintáctico) que tenga como raíz el axioma de la gramática.

Si lo encontramos diremos que la cadena pertenece al lenguaje, pasando a la siguiente fase, si no lo encontramos quiere decir que no está bien construída y entrarán en funcionamiento los mensajes de error.

Tenemos dos posibilidades a la hora de realizar el parsing:

a) Parsing a la izquierda, si para obtener el árbol de derivación empleamos derivaciones por la izquierda.
b) Parsing a la derecha, si las derivaciones las realizamos por la derecha.

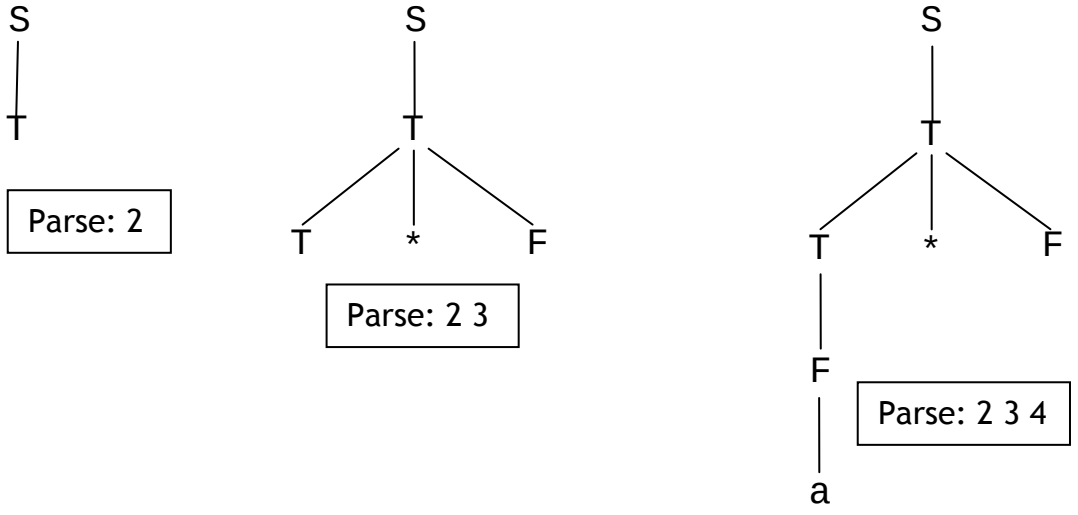
Ejemplo.-

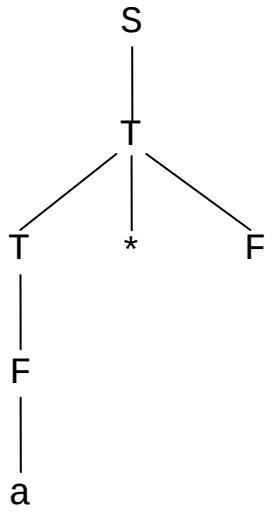
Sea la gramática:

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

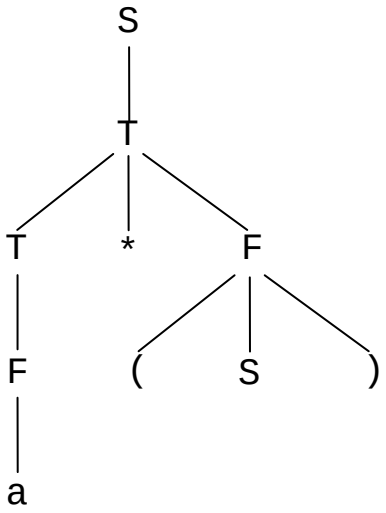
Vamos a realizar el parsing descendente de la cadena "a*(a+b)" por la izquierda y por la derecha:

<u>Parsing descendente por la izquierda</u>

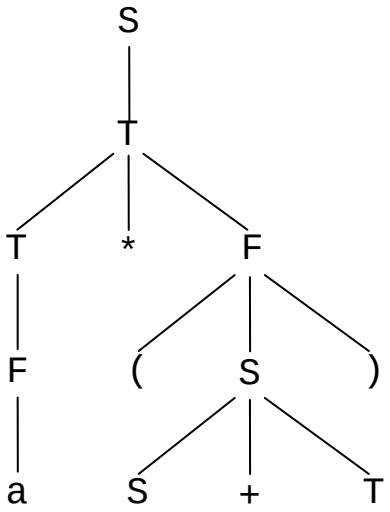




Parse: 2 3 4 6

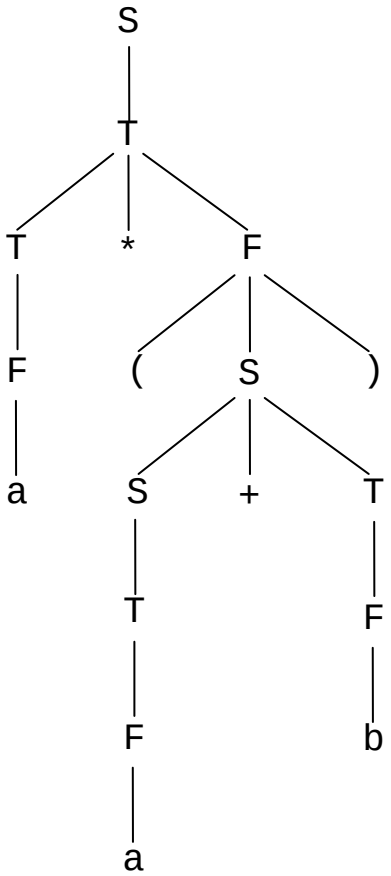


Parse: 2 3 4 6 5



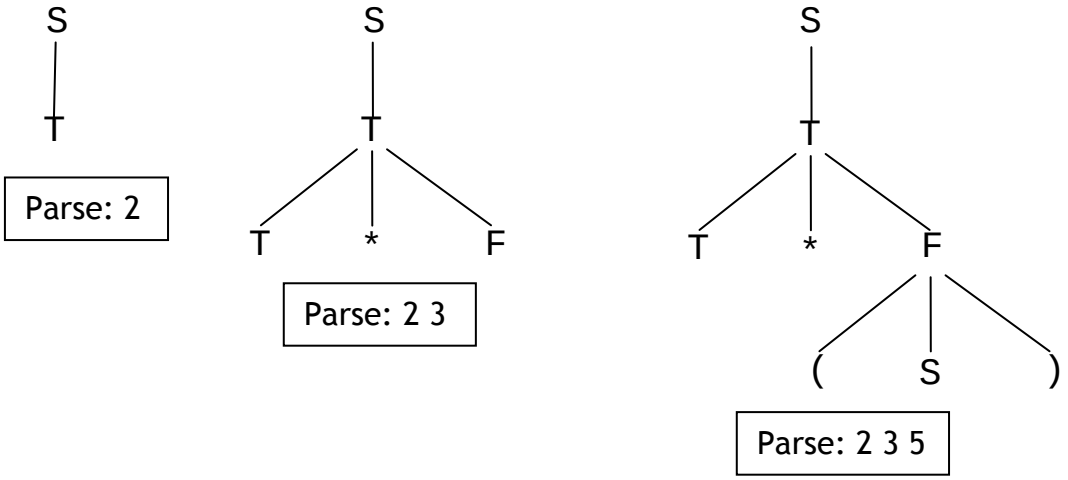
Parse: 2 3 4 6 5 1

...

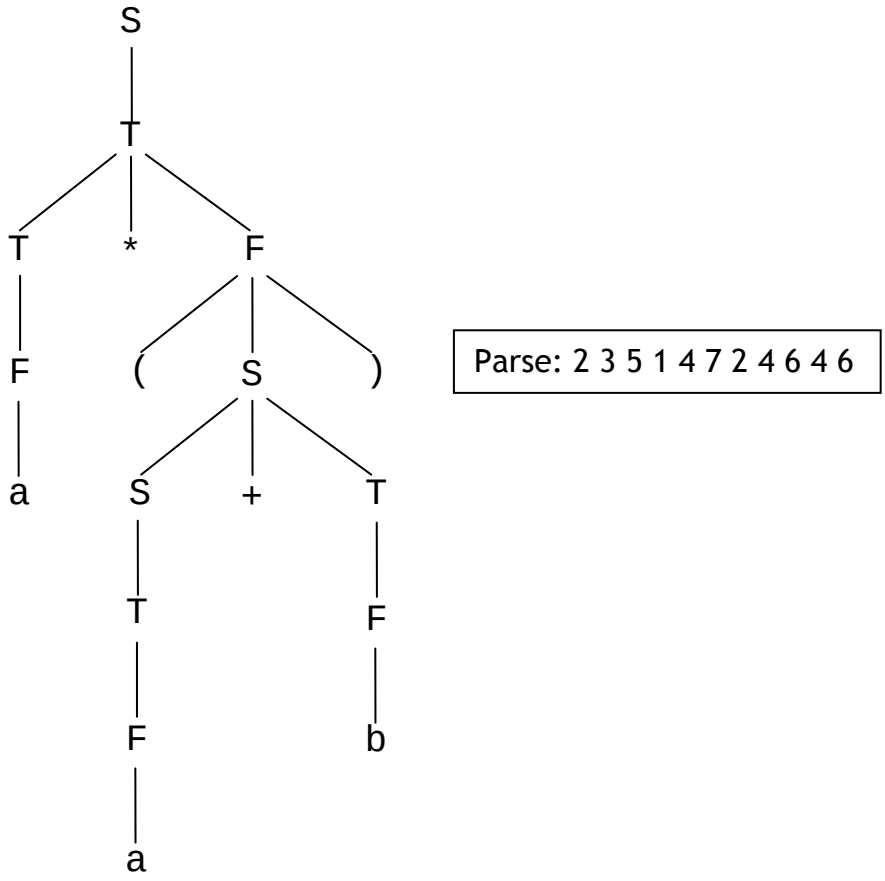


Parse: 2 3 4 6 5 1 2 4 6 4 7

Parsing descendente por la derecha



Así sucesivamente hasta llegar al árbol final (con un parse distinto):

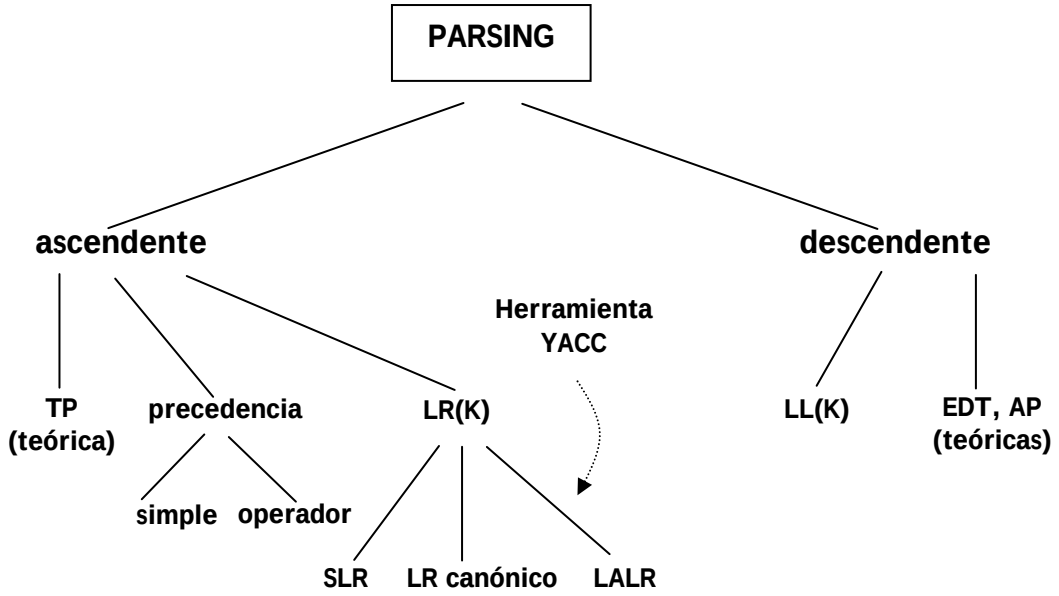


Además, también tenemos dos posibilidades en cuando a si comenzamos desde la metanoción o desde la cadena que queremos reconocer. Esto sería:

a) Parsing descendente, si comenzamos por la metanoción.

a) Parsing ascendente, si comenzamos por la palabra a reconocer.

En general podemos realizar la siguiente clasificación:



4.1 Máquinas teóricas, mecanismos con retroceso.

En este tipo de algoritmos siempre tendremos el problema de buscar la trayectoria correcta en el árbol, si seguimos una trayectoria equivocada, es preciso volver hacia atrás, eso es precisamente el retroceso. Veremos una serie de máquinas teóricas que funcionan mediante este mecanismo.

4.1.1 Autómatas con pila (AP).

Este tipo de autómatas pueden reconocer lenguajes de contexto libre.

Esta máquina teórica podemos definirla como:

$G = (N, T, P, S)$

$P: A \rightarrow \alpha \; ; \; A \in N \; ; \; \alpha \in (N \cup T)^*$

$AP = (Q, Te, Tp, \delta, q_0, z_0, F)$

Q es el conjunto de estados
Te es el alfabeto de entrada
Tp es el alfabeto de la pila
q_0 es el estado inicial
z_0 es el carácter inicial de la pila
F es el conjunto de estados finales
δ función $Q \times \{Te \cup \{\lambda\}\} \times \{Tp \cup \{\lambda\}\} \rightarrow Q \times Tp^*$

Para un estado, una configuración de la pila y una entrada las imágenes son varias en el caso de un APND y una sóla imagen en un APD.

Llamamos **configuración** al estado actual del autómata y podemos definirlo como:

$(q, \omega, \alpha) \in Q \times Te^* \times Tp^*$

q es el estado en que está el autómata ($q \in Q$)
ω es la cadena de caracteres que queda por analizar ($\omega \in Te^*$)
α es la cadena de caracteres que hay en la pila ($\alpha \in Tp^*$)

Un **movimiento** podemos representarlo de la siguiente forma:

$(q, a\omega, z\alpha)$ |---- $(q', \omega, \beta\alpha)$ siendo $(q', \beta) \in \delta(q, a, z)$

Si $\delta(q, a, z) = \{(p_1, \nu_1), (p_2, \nu_2), \ldots (p_n, \nu_n)\}$ será no determinista.
Si $\delta(q, \lambda, z) = \{(p_1, \nu_1), (p_2, \nu_2), \ldots (p_n, \nu_n)\}$ será no determinista de transiciones λ.

Configuración inicial: (q_0, t, z_0), t es la tira que queremos reconocer.

Existen tres posibilidades para el reconocimiento de un AP:

1. Una forma de definir un autómata con pila de forma que aceptará una tira de entrada t si partiendo de una configuración inicial consigue transitar a un **estado final** final empleando movimientos válidos.

(q_0, t, z_0) |--*-- (q_f, λ, α) con $q_f \in F$, $\alpha \in Tp^*$

Con esta definición, la **configuración final** será: (q_f, λ, α), $q_f \in F$

Lenguaje de un autómata con pila definido de esta forma:

$L(AP) = \{ t, t \in Te^* / (q_0, t, z_0)$ |--*-- $(q_f, \lambda, \alpha)\}$

Aquí realizamos el reconocimiento por estado final ($q_f \in F$) pero α no tiene por qué ser λ.

2. Hay otra forma de definirlo, de forma que aceptará una tira de entrada t si partiendo de una configuración inicial consigue dejar la **pila vacía** empleando movimientos válidos:

$L(AP) = \{ t, t \in Te^* / (q_0, t, z_0)$|--*-- $(q', \lambda, \lambda)\}$

Con esta definición, la **configuración final** será: (q' , λ, λ), $q' \in Q$

En este segundo caso reconocemos por pila vacía, q´ no tiene por qué ser un estado final (en ese caso no definimos estados finales ya que no existen).

3. También se puede definir el reconocimiento del autómata por ambas condiciones, **estado final y pila vacía**. Se puede demostrar que todo lenguaje

aceptado por una de las dos cosas puede serlo por ambas. Los ejemplos los veremos con este tipo de reconocimiento.

Con esta definición, la **configuración final** será: (q_f, λ, λ), $q_f \in F$

Ejemplo.- un palíndromo.

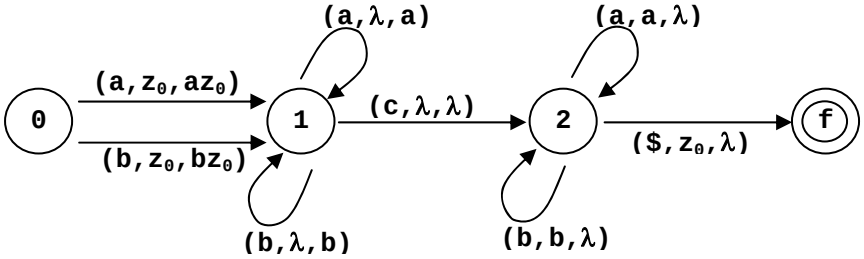
$L = \{ t\ c\ t^r \ / \ t=(a \mid b)^+\}$
$T = \{a, b, c\}$

Por ejemplo, si t = ab, entonces t^r = ba.

Elementos válidos del lenguaje serían: abcba, abbcbba, ababcbaba, etc

1	$\delta(q_0, a, z_0) = (q_1, az_0)$	
2	$\delta(q_0, b, z_0) = (q_1, bz_0)$	
3	$\delta(q_1, a, \lambda) = (q_1, a)$	NOTA : el λ significa "sin mirar la pila"
4	$\delta(q_1, b, \lambda) = (q_1, b)$	
5	$\delta(q_1, c, \lambda) = (q_2, \lambda)$	
6	$\delta(q_2, a, a) = (q_2, \lambda)$	NOTA : aquí "tomamos la a de la pila"
7	$\delta(q_2, b, b) = (q_2, \lambda)$	
8	$\delta(q_1, \$, z_0) = (q_f, \lambda)$	

Con ocho transiciones podemos reconocer el lenguaje. Lo que hacemos es utilizar la pila para comprobar sin coinciden los caracteres de uno y otro lado. Se trata de un autómata con pila determinista. El diagrama de estados es el siguiente:



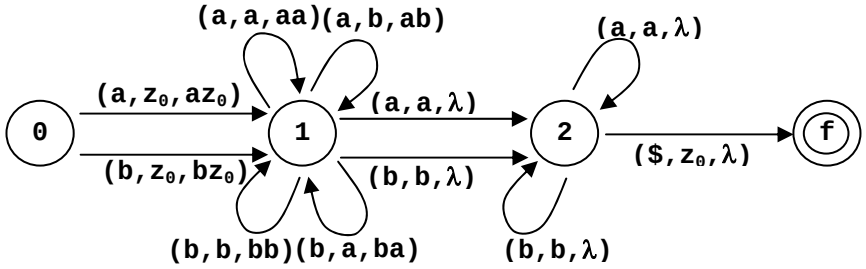
Ejemplo.-

$L = \{t\ t^r \ / \ t = (a \mid b)+\}$
$T=\{a,b\}$

1	$\delta(q_0, a, z_0) = (q_1, az_0)$	También podríamos decir:
2	$\delta(q_0, b, z_0) = (q_1, bz_0)$	
3	$\delta(q_1, a, a) = \{(q_1, aa), (q_2, \lambda)\}$	$\delta(q_1, a, a) = (q_2, \lambda)$
4	$\delta(q_1, a, b) = (q_1, ab)$	$\delta(q_1, a, \lambda) = (q_1, a)$
5	$\delta(q_1, b, b) = \{(q_1, bb), (q_2, \lambda)\}$	$\delta(q_1, b, b) = (q_2, \lambda)$
6	$\delta(q_1, b, a) = (q_1, ba)$	$\delta(q_1, b, \lambda) = (q_1, b)$
7	$\delta(q_2, a, a) = (q_2, \lambda)$	

8 $\delta(q_2, b, b) = (q_2, \lambda)$
9 $\delta(q_2, \$, z_0) = (q_f, \lambda)$

Se trata de un autómata con pila no determinista.



Vamos a intentar reconocer la cadena "abba$":

(0, abba$, z_0) → (1, bba$, az_0) → (1, ba$, baz_0) → (a) y (b)
(a) → (1, a$, $bbaz_0$) → (1, $, $abbaz_0$) No aceptamos.
(b) → (2, a$, az_0) → (2, $, z_0) → (f, λ, λ) Aceptamos.

4.1.1.1 Conversión de una GCL en un Autómata con pila.

El lenguaje expresado en forma de Gramática de Contexto Libre puede también expresarse en forma de un Autómata con pila. Veremos mediante un ejemplo, como, dada una gramática de contexto libre podemos construír un autómata con pila que reconoce el mismo lenguaje. El tipo de análisis que se realiza es un análisis sintáctico descendente.

Es importante hacer notar que aquí daremos por reconocida la cadena siempre que la cadena de entrada se ha terminado y la pila está vacía.

Sea la gramática:

S → S+T | T
T → T*F | F
F → (S) | a | b

Crearemos un autómata con pila de la siguiente forma:

AP = ({q}, {a, *, +, (,)}, Te U {S, T, F}, δ, q, S, q}

Como se puede observar, solamente tenemos un estado que es inicial y final.

$\delta(q, \lambda, S) = \{(q, S+T), (q, T)\}$
$\delta(q, \lambda, T) = \{(q, T*F), (q, F)\}$
$\delta(q, \lambda, F) = \{(q, (S)), (q, a), (q, b)\}$
$\delta(q, a, a) = (q, \lambda)$
$\delta(q, b, b) = (q, \lambda)$
$\delta(q, (, () = (q, \lambda)$
$\delta(q,),)) = (q, \lambda)$
$\delta(q, +, +) = (q, \lambda)$
$\delta(q, *, *) = (q, \lambda)$

A continuación vamos a ver el "camino feliz" para reconocer "a+a*b":

$(q, a+a*b, S) \rightarrow (q, a+a*b, S+T) \rightarrow (q, a+a*b, T+T) \rightarrow (q, a+a*b, F+T) \rightarrow$
$(q, a+a*b, a+T) \rightarrow (q, +a*b, +T) \rightarrow (q, a*b, T) \rightarrow (q, a*b, T*F) \rightarrow$
$(q, a*b, F*F) \rightarrow (q, a*b, a*F) \rightarrow (q, *b, *F), \rightarrow (q, b, F) \rightarrow (q, b, b) \rightarrow (q, \lambda, \lambda)$

Esta es una máquina teórica, con este tipo de máquinas nunca podremos resolver problemas como la ambigüedad; en el tema siguiente hablaremos de implementaciones de AP pero con restricciones para solucionar estos problemas.

4.1.2 Esquemas de traducción (EDT).

Con este tipo de máquinas podemos definir un análisis sintáctico descendente. Un esquema de traducción es una máquina teórica que podemos definir de la siguiente forma:

EDT = (N, Te, Ts, R, S)

N son los no terminales.
Te es el alfabeto de entrada.
Ts es el alfabeto de salida.
S es el axioma.
R son las reglas de la producción de la forma:

$A \rightarrow \alpha, \beta$ con: $\alpha \in (N \cup Te)^*$, $\beta \in (N \cup Ts)^*$ y además $N \cap (Ts \cup Te) = \varnothing$

Lo que hace esta máquina es traducir una tira de caracteres a otro lenguaje, el esquema podría ser el siguiente:

La gramática de salida Gs = (N, Ts, P, S), donde $P = \{ A \rightarrow \beta \ / \ A \rightarrow \alpha,\beta \in R\}$, es decir, nos quedamos con la parte derecha de las reglas R.

NOTA: Es importante hacer notar que aquí la salida es "modificable", es decir, es una memoria. Es una tipo de autómata con pila en el cual tomamos la salida de los valores que quedan en esa memoria.

Una **forma de traducción** es un par (t, s), donde t es una combinación de caracteres entrada y s la salida proporcionada por el EDT, dichos caracteres son una combinación de terminales y no terminales. A la forma (S, S) la llamamos forma inicial de traducción.

Sea un par (t, s), una **derivación** sería:

$(\nu A\mu, \gamma A\sigma) \Rightarrow (\nu\alpha a, \gamma\beta\sigma)$ si existe una regla $A \rightarrow \alpha,\beta$

Representamos con $\Rightarrow^*$ 0 o más derivaciones y con $\Rightarrow^+$ 1 o más derivaciones.

Podemos definir un **conjunto de traducción** como:

$Tr\ (EDT) = \{(t, s) / (S, S) \Rightarrow^+ (t, s), t \in Te^*, s \in Ts^*\}$

Llamamos **traducción regular** a aquella cuyas reglas del esquema de traducción son regulares (es una gramática regular).

Ejemplo1.- Notación polaca inversa.

Ge = ({S}, {a, b, *, /, (,)}, P, S}

Reglas de P:
$S \rightarrow (S)$
$S \rightarrow a \mid b$
$S \rightarrow S*S \mid S/S$

Vamos a crear las reglas del EDT:

Reglas de P:	Reglas de R:
$S \rightarrow (S)$	$S \rightarrow (S), S$
$S \rightarrow a$	$S \rightarrow a, a$
$S \rightarrow b$	$S \rightarrow b, b$
$S \rightarrow S*S$	$S \rightarrow S*S, SS*$
$S \rightarrow S/S$	$S \rightarrow S/S, SS/$

Ya tenemos un EDT = (N, Te, Ts, R, S) en el que:
N = {S}
Te = {a, b, *, /, (,)}
Ts = {a, b, *, /}

Vamos a introducir en el EDT la tira "a/(a*b)" y ver su recorrido:

$(S, S) \Rightarrow (S/S, SS/) \Rightarrow (a/S, aS/) \Rightarrow (a/(S), aS/) \Rightarrow (a/(S*S), aSS*/) \Rightarrow$
$\Rightarrow (a/(a*S), aaS*/) \Rightarrow (a/(a*b), aab*/)$

Ejemplo2.-

T = { a, b, c, (,), *, + }
EDT = (N, T, <1, 2, ..., n>, R, S>

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

Vamos a intentar reconocer la palabra "a*(a+b)":

$(S, S) \Rightarrow (T, 2T) \Rightarrow (T*F, 23TF) \Rightarrow (F*F, 234FF) \Rightarrow (a*F, 2346F) \Rightarrow$
$\Rightarrow (a*(S), 23465S) \Rightarrow ... \Rightarrow (a*(a+b), 23465124647)$

La cadena que obtenemos nos informa del camino tomado por el parsing, es lo que denominamos parse.

4.1.3 Traductores con pila (TP).

Un traductor con pila realiza un análisis sintáctico ascendente y tiene la posibilidad de generar una salida, que será el parse. También se trata de una máquina teórica y podemos definirlo como:

$TP = (Q, Te, Tp, Ts, \delta, q_0, z_0, F)$

Q es el conjunto de estados.
Te es el alfabeto de entrada.
Tp es el alfabeto de la pila.
Ts es el alfabeto de salida.
q_0 es el estado inicial.
z_0 es el elemento inicial de la pila.
F son los estados finales (un subconjunto de Q).

δ: Q x {Te U {λ}} x {Tp U {λ}} $\rightarrow$ P(Q) x Tp* x Ts*

Llamamos **configuración** a la siguiente tupla: (q, t, α, s), en donde q es el estado actual, t es la tira de entrada que queda por analizar, α es el contenido de la pila y s la cadena que está saliendo.

Un **movimiento** es una transición de la forma:

(q, ax, zν, y) |---- (q´, x, $\alpha\nu$, yz´) con q, q´ $\in$ Q, x $\in$ Te*, $\nu \in$ Tp*, y $\in$ Ts*

El **conjunto de traducción** podemos definirlo como:

$Tr(TP) = \{(t,s) / (q_0, t, z_0, \lambda) \text{ |--*-- } (q_f, \lambda, \alpha, s), q_f \in F\}$

si realizamos la traducción hasta que se agote la cadena de entrada y nos encontremos en un estado final, o bien:

$Tr(TP) = \{(t,s) / (q_0, t, z_0, \lambda) \text{ |--*-- } (q', \lambda, \lambda, s)\}$

si realizamos la traducción hasta que se agote la cadena de entrada y la memoria de la pila.

Ejemplo.-
T = { a, b, (,), *, + }

TP = ({q}, T, N U T, <1, 2, ..., 7>, δ, q, z_0, q>

$S \rightarrow S + T$	$\delta(q, \lambda, S+T) = (q, S, 1)$
$S \rightarrow T$	$\delta(q, \lambda, T) = (q, S, 2)$
$T \rightarrow T * F$	$\delta(q, \lambda, T*F) = (q, T, 3)$
$T \rightarrow F$	$\delta(q, \lambda, F) = (q, T, 4)$
$F \rightarrow (S)$	$\delta(q, \lambda, (S)) = (q, F, 5)$
$F \rightarrow a$	$\delta(q, \lambda, a) = (q, F, 6)$
$F \rightarrow b$	$\delta(q, \lambda, b) = (q, F, 7)$

Si queremos vaciar la pila añadimos la siguiente regla:

$\delta(q, \lambda, S) = (q, \lambda, \lambda)$

Siendo t un terminal, para introducirlo en la pila:

$\delta(q, t, \lambda) = (q, t, \lambda)$

Vamos a reconocer la cadena "a*(a+b)" (veremos el “camino feliz”):

ENTRADA	PILA	SALIDA
a*(a+b)	z_0	λ
*(a+b)	z_0a	λ
*(a+b)	z_0F	6
*(a+b)	z_0T	64
(a+b)	z_0T*	64
a+b)	z_0T*(	64
+b)	z_0T*(a	64
+b)	z_0T*(F	646
...	...	...
λ	z_0S	64642741532
λ	z_0	64642741532

4.2 Algoritmos sin retroceso.

En los algoritmos vistos anteriormente tenemos el problema de buscar la trayectoria correcta en el árbol, si es la equivocada, es preciso volver hacia atrás y eso es precisamente el retroceso. Nos interesarán algoritmos predictivos que, viendo los elementos que van a venir a continuación sabrán a que estado transitar. Esto implica que las gramáticas serán más restrictivas, así hablaremos de gramáticas LL1, LR, precedencia simple, etc, estas gramáticas siempre serán subconjuntos de las Gramáticas de Contexto Libre.

4.2.1 Análisis sintáctico ascendente por precedencia simple.

Es un método de parsing ascendente que se basa en la relaciones $<$, $>$, $\pm$, relaciones que no son de equivalencia.

Decimos que una relación **R** es una **relación de equivalencia** cuando:
i) Reflexiva. x **R** x $\forall x \in$ **R**
ii) Simétrica. x **R** y $\Rightarrow$ y **R** x
iii) Transitiva. x **R** y, y **R** z $\Rightarrow$ x **R** z

Ejemplo.-

R = { (x,y) / x > y } x, y $\in$ N
No es de equivalencia

x\y	0	1	2	3	...
0	0	0	0	0	
1	1	0	0	0	
2	1	1	0	0	
3	1	1	1	0	
...					

S = { (x,y) / x = y } x, y $\in$ N
Es de equivalencia

x\y	0	1	2	3	...
0	1	0	0	0	
1	0	1	0	0	
2	0	0	1	0	
3	0	0	0	1	
...					

NOTA: No confundir la relación "=" numérica, que sí es de equivalencia, con la relación "$\pm$" que usaremos aquí, que no es de equivalencia.

Definición: **Relación universal**

U = A x A = { (x, y) / x $\in$ A, y $\in$ A }

Definición: **Relación traspuesta** (**R´** o $\mathbf{R^T}$)

Dada una relación **R**, $\mathbf{R^T}$ = { (y, x) / x **R** y }

Definición: **Complementario de una relación** ($\mathbf{R^{\sim}}$)

$\mathbf{R^{\sim}}$ = { (x, y) / $\sim$(x **R** y) } = U - **R**

Dadas dos relaciones R y S sobre A:

R + **S** = { (x, y) / (x **R** y) $\vee$ (x **S** y) } (OR)
R x **S** = { (x, y) / (x **R** y) $\wedge$ (x **S** y) } (AND)

R + **S** = M_{ij} OR M_{kp}
R x **S** = M_{ij} AND M_{kp}

R . S = { (x, y) / $\exists$ z (x **R** z) $\wedge$ (z **S** y) } denominado producto relativo.

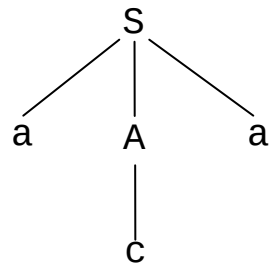
A la hora de realizar la tabla de relaciones de la gramática se puede hacer "a mano", haciendo todas las posibles derivaciones a partir del axioma de la

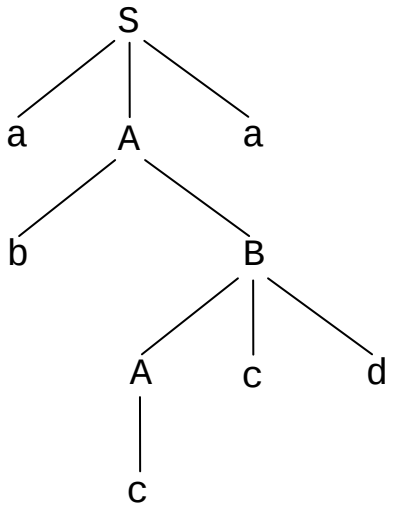
gramática y determinar las relaciones en función del lugar en que nos encontremos en el árbol.

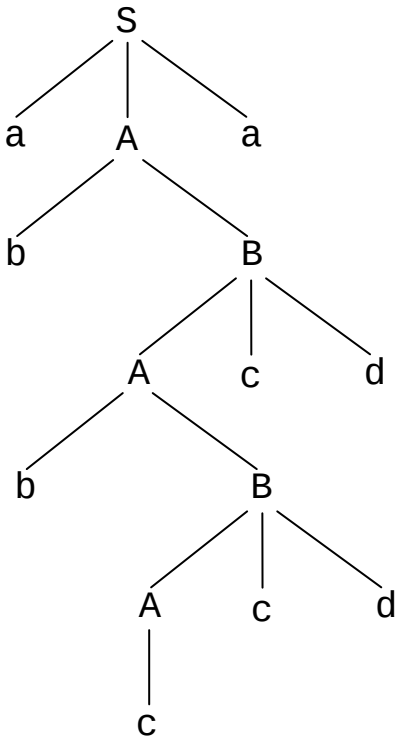
Por ejemplo, supongamos la gramática:

$S \rightarrow aAa$
$A \rightarrow bB$
$A \rightarrow c$
$B \rightarrow Acd$

Algunos árboles serían los siguientes:







Las relaciones que podemos obtener son las siguientes:

b $\pm$ B
a $<$ b
B $>$ a
b $<$ c
A $\pm$ c
Etc...

Si están en la misma regla la relación es $\pm$, si están en el nivel superior por la izquierda $<$ y si están en el nivel superior por la derecha o cualquiera que "cuelgue" de éste $>$.

Finalmente, la tabla sería:

	S	a	A	B	c	b	d
S							
a			±		<	<	
A		±			±		
B		>			>		
c		>			>		±
b			<	±	<	<	
d		>			>		

Para que la gramática sea de precedencia simple solamente ha de existir un símbolo por casilla. Existen más razones para que una gramática no sea de precedencia simple pero siempre que exista la misma parte derecha en dos reglas distintas esta gramática no lo será.

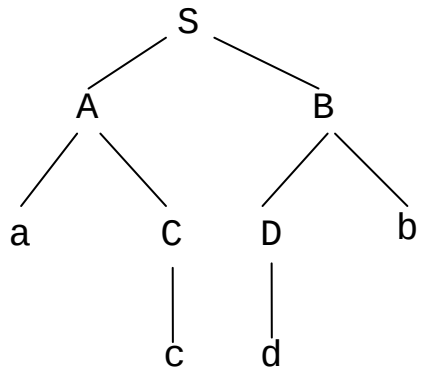
Llamamos **pivote** a un conjunto de símbolos que se pueden sustituir por las metanociones de la regla correspondiente (son símbolos situados entre < y > con los intermedios relacionados con ±). La tira completa la rodeamos por izquierda y derecha pcon < y >. Utilizando el concepto de pivote, vamos a realizar a continuación el parsing de una tira, necesitamos encontrar un pivote, si éste no existe querrá decir que no pertenece al lenguaje:

TIRA	PIVOTE	REDUCCIÓN
<a<b<**c**>c±d>a>	c	A → c
<a<b<**A±c±d**>a>	Acd	B → Acd
<a<**b±B**>a>	bB	A → bB
<a±A±a>	aAa	S → aAa
<S>		ÉXITO, ACEPTADA

4.2.1.1 Cálculo de la tabla por el método matricial.

Dada una gramática de contexto libre G = (N, T, P, S):

i) x < y si y sólo si $A \to \alpha x B \beta \in P$ y $B \to^+ y\nu$
ii) x ± y si y sólo si $A \to \alpha xy\beta \in P$
iii) x > y si y sólo si $A \to \alpha CD\beta$ con $C \to^+ \nu x$ y $D \to^* y\mu$



Solamente **A < D y A < d**
pero:

C > B, C > D, C > d

y

c > B, c > D, c > d

El ejemplo anterior creemos que hará comprender la diferencia entre < y >.

Sea una GCL G = (N, T, P, S), y sea $A \rightarrow \alpha x \beta \in P$, definimos **cabecera** como:
$Cab(A) = \{ x / A \rightarrow^{+} x... \}$

Relación **PRIMERO**: Diremos que A es PRIMERO de X si y sólo si $\exists A \rightarrow X\alpha\beta$
Así podemos decir que $Cab(A) = \{ X / (A, X) \in PRIMERO^{+} \}$

A **PRIMERO^{+}** X si y sólo si $\exists A \rightarrow^{+} X\alpha\beta$
A **ULTIMO** X si y sólo si $\exists A \rightarrow \alpha\beta X$
A **ULTIMO^{+}** X si y sólo si $\exists A \rightarrow^{+} \alpha\beta X$
A **DENTRO** x si y sólo si $\exists A \rightarrow \alpha X\beta$
A **DENTRO^{+}** x si y sólo si $\exists A \rightarrow^{+} \alpha X\beta$

Ejemplo.-

$A \rightarrow Aa \mid B$
$B \rightarrow DbC \mid Dc$
$C \rightarrow c$
$D \rightarrow Ba$

PRIMERO+ = { (A, A), (A, B), (A, D), (B, D), (B, B), (C, c), (D, B), (D, D) }

La matriz PRIMERO+ sería la siguiente:

	A	**B**	**C**	**D**	**a**	**b**	**c**
A	1	1	0	1	0	0	0
B	0	1	0	1	0	0	0
C	0	0	0	0	0	0	1
D	0	1	0	1	0	0	0
a	0	0	0	0	0	0	0
b	0	0	0	0	0	0	0
c	0	0	0	0	0	0	0

Si una vez calculada la matriz PRIMERO la multiplicamos por sigo misma (con AND, es decir 1+1=1, el denominado producto relativo) tantas veces como haga falta hasta que no aparezcan más unos (los que vayan apareciendo los añadiremos), llegaremos a obtener PRIMERO^{+}.

Algoritmo de Warshall

Nos permite hallar A^{+} a partir de A; $A^{+} = U_{i>0} A^{i}$ a partir de la relación R en forma de una matriz.

```
        B = A
        I = 1
3       REPEAT
                IF B(I, J) = 1 THEN
                        FOR K = 1 TO N DO
                                IF A(J, K) = 1 THEN A(I, K) = 1
        UNTIL para todos los valores de J
```

I = I + 1
IF I ≤ N THEN GOTO 3 ELSE STOP

Cálculo de las matrices (±), (<) y (>)

(±) se calcula por la propia definición.

(<) = **(±) (PRIMERO^{+})**

(>) = **(ULTIMO^{+})^{T} (±) (I + PRIMERO^{+})** , donde **I** es la matriz identidad.

Ejemplo 1.- Calcular las relaciones de precedencia simple para la gramática:

$S \rightarrow aSB \mid \lambda$
$B \rightarrow b$

Tenemos que eliminar la regla λ, de forma que la gramática nos quedaría:

$S' \rightarrow S \mid \lambda$
$S \rightarrow aSB \mid aB$
$B \rightarrow b$

La regla $S' \rightarrow S \mid \lambda$ podemos no tenerla en cuenta (y por lo tanto tampoco el símbolo S´), si bien al implementarlo contemplaríamos la "sentencia vacía" como parte del lenguaje. Se indican en negrita y con un asterisco los 1s que serían 0s si no eliminásemos la regla λ existente en la gramática.

Vamos a calcular las matrices **(±)** y **PRIMERO**:

±	S	B	a	b
S	0	1	0	0
B	0	0	0	0
a	1	**1***	0	0
b	0	0	0	0

PRIMERO	S	B	a	b
S	0	0	1	0
B	0	0	0	1
a	0	0	0	0
b	0	0	0	0

ULTIMO	S	B	a	b
S	0	1	0	0
B	0	0	0	1
a	0	0	0	0
b	0	0	0	0

Por lo tanto:

$$\textbf{(±) =} \begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & \mathbf{1^*} & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \qquad \textbf{PRIMERO =} \begin{vmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Vamos a hacer el producto relativo con PRIMERO:

$$\begin{vmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \quad \begin{vmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Entonces $PRIMERO^+ = PRIMERO$

Vamos a hacer el producto relativo con ULTIMO:

$$\begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \quad \begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Como aparece un nuevo "1", volvemos a multiplicarlo por ULTIMO:

$$\begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \quad \begin{vmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Ahora ya tenemos las matrices **$PRIMERO^+$** y **$ULTIMO^+$**:

$$\mathbf{PRIMERO^+} = \begin{vmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \qquad \mathbf{ULTIMO^+} = \begin{vmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Ya podemos calcular **(<)** y **(>)**:

$$\mathbf{(<)} = \begin{vmatrix} \multicolumn{4}{c}{\mathbf{(\pm)}} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & \mathbf{1^*} & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \quad \begin{vmatrix} \multicolumn{4}{c}{\mathbf{(PRIMERO^+)}} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} & & & \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & \mathbf{1^*} \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

$$\mathbf{(ULTIMO^+)^T\ (\pm)\ (\clubsuit)} = \begin{vmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{vmatrix} \quad \begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & \mathbf{1^*} & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{vmatrix}$$

$$\mathbf{(>)} = \begin{vmatrix} \multicolumn{4}{c}{\mathbf{(\clubsuit)}} \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{vmatrix} \quad \begin{vmatrix} \multicolumn{4}{c}{\mathbf{(I+PRIMERO^+)}} \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} & & & \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{vmatrix}$$

La tabla final de parsing será:

	S	B	a	b
S		±		<
B		>		>
a	±	**±***	<	**<***
b		>		>

Si en esta tabla eliminamos los símbolos **±*** y **<*** (que sería la tabla resultante si no eliminámaramos la regla λ) no podríamos resolver elementos del lenguaje como podría ser "aabb"; al mirar las precedencias no seríamos capaces de encontrar un pivote y no aceptaríamos una palabra que sí es del lenguaje. En cambio con la tabla tal cual aparece podemos reconocer todo el lenguaje excepto el elemento "cadena vacía", que habría que tratarlo y reconocerlo específicamente (algo, por otro lado, muy sencillo).

Ejercicio 2.-
$S \rightarrow AB$
$B \rightarrow A$
$A \rightarrow a$

±	S	A	B	a
S	0	0	0	0
A	0	0	1	0
B	0	0	0	0
a	0	0	0	0

PRIMERO	S	A	B	a
S	0	1	0	0
A	0	0	0	1
B	0	1	0	0
a	0	0	0	0

ULTIMO	S	A	B	a
S	0	0	1	0
A	0	0	0	1
B	0	1	0	0
a	0	0	0	0

Por lo tanto:

$$(\pm) = \begin{vmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \qquad \text{PRIMERO} = \begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Vamos a hacer el producto relativo con PRIMERO:

$$\begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Como aparecen dos nuevos “1”, volvemos a multiplicarlo por PRIMERO:

$$\begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \begin{vmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Vamos a hacer el producto relativo con ULTIMO:

$$\begin{vmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \begin{vmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Como aparecen nuevos “1”, volvemos a multiplicarlo por ULTIMO:

$$\begin{vmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Nuevamente aparece un "1", y seguimos:

$$\begin{vmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \begin{vmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Ahora ya tenemos las matrices **PRIMERO$^+$** y **ULTIMO$^+$**:

$$\mathbf{PRIMERO^+} = \begin{vmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{vmatrix} \qquad \mathbf{ULTIMO^+} = \begin{vmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Ya podemos calcular **(<)** y **(>)**:

$$\mathbf{(<)} = \begin{vmatrix} \mathbf{(\pm)} & & & \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \begin{vmatrix} \mathbf{(PRIMERO^+)} & & & \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} & & & \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

$$\mathbf{(ULTIMO^+)^T\ (\pm)}\ (*) = \begin{vmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{vmatrix} \begin{vmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{vmatrix}$$

$$\mathbf{(>)} = \begin{vmatrix} \mathbf{(*)} & & & \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{vmatrix} \begin{vmatrix} \mathbf{(I+PRIMERO^+)} & & & \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} & & & \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{vmatrix}$$

La tabla final de parsing será:

	S	A	B	a
S				
A		<	±	<
B				
a		>	>	>

Ejercicio 3.-

$S \rightarrow CC$
$C \rightarrow cC$
$C \rightarrow d$

±	S	C	c	d
S	0	0	0	0
C	0	1	0	0
c	0	1	0	0
d	0	0	0	0

PRIMERO	S	C	c	d
S	0	1	0	0
C	0	0	1	1
c	0	0	0	0
d	0	0	0	0

ULTIMO	S	C	c	d
S	0	1	0	0
C	0	1	0	1
c	0	0	0	0
d	0	0	0	0

Por lo tanto:

$$(\pm) = \begin{vmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \qquad \textbf{PRIMERO} = \begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Vamos a hacer el producto relativo con PRIMERO:

$$\begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \quad \begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Como aparecen dos nuevos “1”, volvemos a multiplicarlo por PRIMERO:

$$\begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \quad \begin{vmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Vamos a hacer el producto relativo con ULTIMO:

$$\begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \quad \begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Como aparece un nuevo “1”, volvemos a multiplicarlo por ULTIMO:

$$\begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \begin{vmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Ahora ya tenemos las matrices **PRIMERO$^+$** y **ULTIMO$^+$**:

$$\mathbf{PRIMERO^+} = \begin{vmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \qquad \mathbf{ULTIMO^+} = \begin{vmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Ya podemos calcular **(<)** y **(>)**:

$$\mathbf{(<)} = \begin{vmatrix} \mathbf{(\pm)} & & & \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \begin{vmatrix} \mathbf{(PRIMERO^+)} & & & \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} & & & \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

$$\mathbf{(ULTIMO^+)^T\ (\pm)\ (*)} = \begin{vmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{vmatrix} \begin{vmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{vmatrix}$$

$$\mathbf{(>)} = \begin{vmatrix} \mathbf{(*)} & & & \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{vmatrix} \begin{vmatrix} \mathbf{(I+PRIMERO^+)} & & & \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} & & & \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{vmatrix}$$

La tabla final de parsing será:

	S	**C**	**c**	**d**
S				
C		> $\pm$	> <	> <
c		$\pm$	<	<
d		>	>	>

En la fila de B aparecen varias relaciones en la misma casilla, se trata de una gramática no implementable por precedencia simple.

4.2.2 Análisis sintáctico ascendente por precedencia de operadores.

Una gramática de contexto libre G=(N, T, P, S) diremos que es una gramática de operador si no posee reglas λ y si en la parte derecha de sus reglas no aparecen dos no terminales adyacentes. Esto es:

1) no $\exists$ A $\rightarrow \lambda$
2) no $\exists$ A $\rightarrow \alpha BC\beta$ con A, B, C $\in$ N

Sean a y b dos símbolos, trabajaremos con las siguientes precedencias: a > b, b < a y a $\pm$ b.

Algoritmo

1. Obtener la tira de entrada, se ponen las reglas de precedencia entre cada dos elementos y se delimita mediante otro símbolo, por ejemplo $. Por ejemplo: $**<id>**+<**id>**$. NOTA: Sólo existen relaciones entre terminales.
2. Analizamos la cadena de entrada de izquierda a derecha y vamos avanzando hasta encontrar el símbolo de precedencia mayor (>), luego iremos hacia atrás hasta encontrar el símbolo de precedencia menor (<), con lo que todo lo encerrado entre ambos será el pivote y el que nos permitirá realizar la reducción.

De esta forma, los errores que pueden producirse son:

a) Ninguna relación de precedencia entre un símbolo y el siguiente.
b) Una vez encontrada la relación de precedencia, es decir, tenemos pivote, no existe ninguna regla que permita reducirlo.

Algoritmo para la obtención de las relaciones de precedencia operador

i) Si el operador θ_1 tiene mayor precedencia que el $\theta_2 \Rightarrow \theta_1 > \theta_2$ $\theta_2 < \theta_1$
ii) Si el operador θ_1 tiene igual precedencia que el $\theta_2 \Rightarrow$
- Si son asociativos por la izquierda $\theta_1 > \theta_2$ $\theta_2 > \theta_1$
- Si son asociativos por la derecha $\theta_1 < \theta_2$ $\theta_2 < \theta_1$

iii) Hágase:

θ < (< θ >) > θ	(< ($\pm$) >)
$ < (	) > $
θ < id > θ	θ > $ < θ
(< id >)	$ < id > $

Ejemplo1.- Sea la siguiente gramática:

S $\rightarrow$ S + S asumimos que el operador + es asociativo por la izquierda
S $\rightarrow$ S * S asumimos que el operador * es asociativo por la izquierda
S $\rightarrow$ id

La tabla de precedencia operador será la siguiente:

	id	**+**	*****	**$**
id		>	>	>
+	<	>	<	>
*****	<	>	>	>
$	<	<	<	

Vamos a reconocer la tira "$id+id*id$"

TIRA	PIVOTE	REDUCCIÓN
$<**id**>+<id>*<id>$	id	S → id
$<S+<**id**>*<id>$	id	S → id
$<S+<S*<**id**>$	id	S → id
$<S+<S*S>$	S*S	S → S*S
$<S+S>$	S+S	S → S+S
S		ÉXITO, ACEPTADA

Ejemplo 2.- Sea la siguiente gramática:

S → SAS | (S) | id
A → + | - | * | / | ↑

Esta gramática no es de precedencia operador porque en S → SAS tenemos tres no terminales consecutivos, pero si sustituímos las alternativas de A en esa regla la podemos convertir en una gramática de precedencia operador de la siguiente forma:

S → S + S
S → S - S
S → S * S
S → S / S
S → S ↑ S
S → (S)
S → id

Asumimos que:
- El operador ↑ tiene la mayor precedencia y es asociativo por la derecha.
- Los operadores * y / tienen la siguiente mayor precedencia y son asociativos por la izquierda.
- Los operadores + y – son los de menor precedencia y son asociativos por la izquierda.

La tabla de precedencia operador sería la siguiente:

	+	**-**	*****	**/**	**↑**	**id**	**(**	**)**	**$**
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*****	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>

↑	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			>	>
(	<	<	<	<	<	<	<	±	
)	>	>	>	>	>			>	>
$	<	<	<	<	<	<	<		

Funciones de precedencia

Normalmente no se almacena la tabla de precedencias de operador sino que se definen unas funciones de precedencia. Utilizando estas funciones ahorramos memoria pues no es preciso almacenar la tabla. El método se basa en implementar dos funciones **f** y **g** que transformarán símbolos terminales en enteros que compararemos para mirar su prioridad. Esto es:

$\forall$ a, b $\in$ T
f(a) < g(b) sii a < b
f(a) > g(b) sii a $\pm$ b
f(a) > g(b) sii a > b

Para la tabla del ejercicio 2, las funciones **f** y **g** serían las siguientes:

	+	**-**	*****	***/***	↑	**id**	**(**	**)**	**$**
f	2	2	4	4	4	6	0	6	0
g	1	1	3	3	5	5	5	0	0

De esta forma, ante una cadena de entrada como la siguiente:

$id+id*id$

Las precedencias las calcularíamos de la siguiente forma:

f($) = 0	y	g(id) = 5	entonces	$ < id
f(id) = 6	y	g(+) = 1	entonces	id < +
f(+) = 2	y	g(id) = 5	entonces	+ < id
f(id) = 6	y	g(*) = 3	entonces	id > *
f(*) = 4	y	g(id) = 5	entonces	* < id
f(id) = 6	y	g($) = 0	entonces	id > $

Con lo cual:

$<id>+<**id**>*<id>$

Reduciríamos <id> con la regla S $\rightarrow$ id y continuaríamos el proceso.

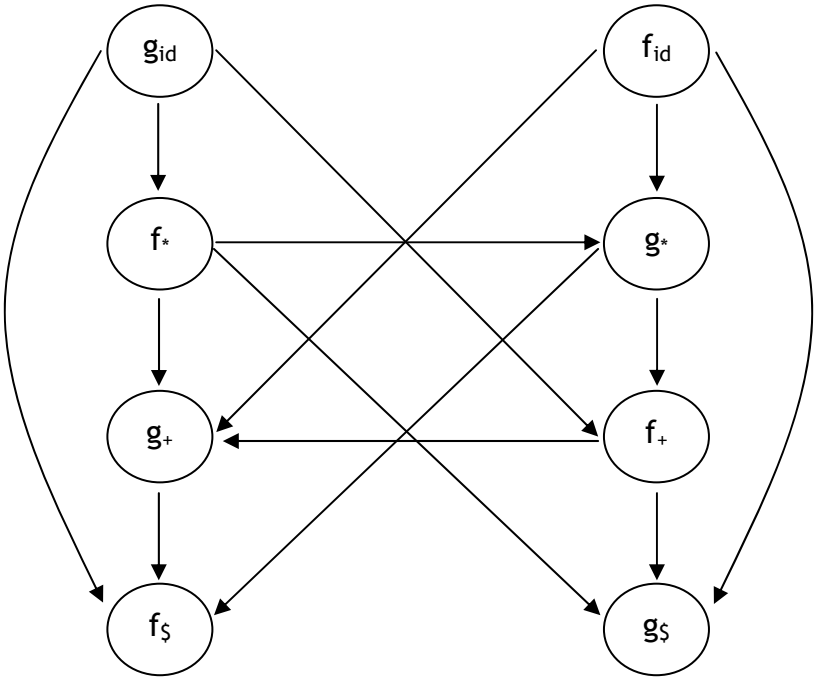
Algoritmo para la construcción de las funciones de precedencia

1. Partimos de la matriz de precedencia.
2. Creamos para un grafo los nodos f_a y g_a $\forall$a $\in$ T incluyendo $.
3. Si tienen igual precedencia los agrupamos.

4. Creamos un grafo dirigido cuyos nodos sean los símbolos creados en el paso anterior.
5. Si a < b la arista va de g_b a f_a.
6. Si a > b la arista va de f_a a g_b.
7. Si el grafo así construido tiene ciclos significa que no se pueden calcular las funciones de precedencia. En otro caso,las funciones las construiremos asignando a f_a la longitud del camino más largo que podemos recorrer por el grafo desde el nodo donde está f_a, al igual que con g_a.

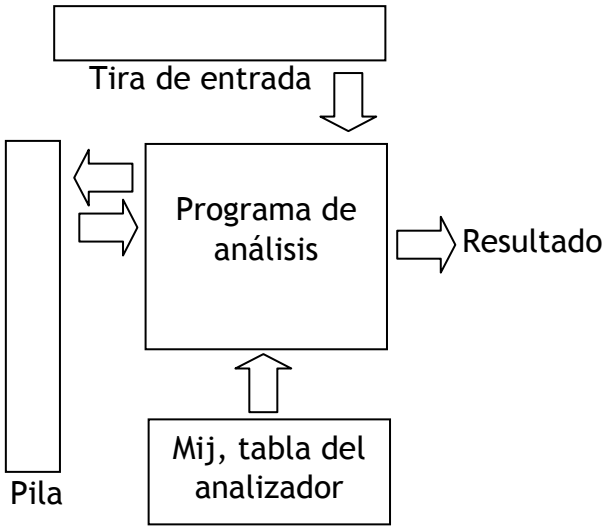
Para la tabla del ejemplo 1 las funciones serían las siguientes (construídas a partir del grafo):

	id	**+**	*****	**$**
f	4	2	4	0
g	5	1	3	0



4.2.3 Analizadores descendentes LL(K).

Este tipo de analizadores consta de los siguientes elementos:



Los analizadores sintácticos LL(k) funcionan de forma descendente, sin retroceso y son predictivos en k símbolos. Nosotros nos quedaremos con los LL(1). La primera "L" significa "Left", izquierda en inglés, indicando que el análisis de la entrada se realiza de izquierda a derecha; la segunda "L" indica también "Left" y representa una derivación por la izquierda; K, en nuestro caso K=1, indica que utilizamos un símbolo de entrada de examen por anticipado a cada paso para tomar decisiones de acción en el análisis sintáctico.

Suponiendo que tenemos la tabla, el funcionamiento será:

Si X = a = \$ $\Rightarrow$ Éxito en el análisis (X elemento de la pila, a es entrada).
Si X = a $\neq$ \$ $\Rightarrow$ El analizador quita X de la pila y va a la siguiente entrada.
Si X $\in$ N $\Rightarrow$ Consulta la tabla en M[X,a], en donde tendremos una producción, por ejemplo X $\rightarrow$ WZT, sustituyendo X en la pila por WZT.

Ejemplo.- Sea la gramática:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

Y la tabla correspondiente:

	id	**+**	*****	**(**	**)**	**\$**
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E´		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T´		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Vamos a realizar un reconocimiento de "id+id*id\$":

PILA	ENTRADA	Reducción
$E	id+id*id$	
$E´T	id+id*id$	$E \rightarrow TE´$
$E´T´F	id+id*id$	$T \rightarrow FT´$
$E´T´id	id+id*id$	$F \rightarrow id$
$E´T´	+id*id$	
$E´	+id*id$	$T´ \rightarrow \varepsilon$
$E´T+	+id*id$	$E´ \rightarrow +TE´$
$E´T	id*id$	
$E´T´F	id*id$	$T \rightarrow FT´$
$E´T´id	id*id$	$F \rightarrow id$
$E´T´	*id$	
$E´T´F*	*id$	$T´ \rightarrow *FT´$
$E´T´F	id$	
$E´T´id	id$	$F \rightarrow id$
$E´T´	$	
$E´	$	$T \rightarrow \varepsilon$
$	$	$E´ \rightarrow \varepsilon$
$	$	**ÉXITO**

Si realizáramos el recorrido con la cadena "id**id" en algún paso nos encontraríamos con una casilla en blanco, es decir, un error.

4.2.3.1 Construcción de la tabla Mij.

Cálculo del conjunto PRIMERO (X):

i) $X \in T$ entonces PRIMERO(X) = {X}
ii) $X \rightarrow \varepsilon$ entonces $\varepsilon \in$ PRIMERO(X)
iii) $X \in N$, $X \rightarrow X_1X_2... X_n$ entonces $\forall a \in T$, $a \in$ PRIMERO(X_j), $a \in$ PRIMERO(X) siempre y cuando: $X_1 \Rightarrow^* \varepsilon$, $X_2 \Rightarrow^* \varepsilon$, ..., $X_{j-1} \Rightarrow^* \varepsilon$. Si ocurriera que $X_1 \Rightarrow^* \varepsilon$, $X_2 \Rightarrow^* \varepsilon$, ..., $X_n \Rightarrow^* \varepsilon$ entonces $\varepsilon \in$ PRIMERO(X).

Cálculo del conjunto SIGUIENTE(X):

i) Si X = S (símbolo inicial o axioma) entonces $ ∈ SIGUIENTE(X).
ii) Si tenemos una producción de la forma $A \rightarrow \alpha B\beta$, con $\beta \neq \varepsilon$, entonces PRIMERO(β) \ {ε} ∈ SIGUIENTE(B) (nota: \ significa "menos").
iii) Si tenemos producciones de la forma: $A \rightarrow \alpha B$ ó bien $A \rightarrow \alpha B\beta$ donde PRIMERO(β) contenga ε (o lo que es lo mismo, $\beta \Rightarrow^* \varepsilon$) entonces hacemos SIGUIENTE(A) $\subset$ SIGUIENTE(B).

Ejemplo.- Sea la gramática del ejemplo anterior:

$E \rightarrow TE´$
$E´ \rightarrow +TE´ \mid \varepsilon$

$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

Si calculamos PRIMERO y SIGUIENTE quedaría:

	PRIMERO	SIGUIENTE
E	(, id	), $
E´	+, ε	), $
T	(, id	+,), $
T´	*, ε	+,), $
F	(, id	*, +,), $

En el caso de E tenemos que $ $\in$ SIGUIENTE(E) por (i), además tenemos la regla $F \rightarrow (E)$, caso (ii), por lo que PRIMERO()) = {) } $\in$ SIGUIENTE(E) (no está ε, en ese caso no la añadiríamos), entonces tenemos ya que SIGUIENTE(E) = {$,)}.

En el caso de E´ tenemos la regla $E \rightarrow TE'$, que es el caso (iii), con lo cual SIGUIENTE(E) $\subset$ SIGUIENTE(E´), con lo cual SIGUIENTE(E´) = {), $}.

En el caso de T tenemos la regla $E \rightarrow TE'$, por la regla (ii) tenemos que PRIMERO (E´)\ε = {+} $\in$ SIGUIENTE(T). Además, en la regla $E' \rightarrow +TE'$ tenemos la situación (iii), con lo cual SIGUIENTE(E´) $\subset$ SIGUIENTE(T), por lo que añadimos {), $}. Finalmente tenemos que SIGUIENTE(T) = {+,), $}.

Algoritmo de confección de la tabla Mij

1) $\forall A \rightarrow \alpha \in P$ aplicamos (2) y (3).
2) $\forall a \in T$, $a \in$ PRIMERO(α), añadir $A \rightarrow \alpha$ en M[A, a].
3) Si $\varepsilon \in$ PRIMERO(α), añadir $A \rightarrow \alpha$ en M[A, b] $\forall b \in$ SIGUIENTE(A). Como caso más especial, incluido en el anterior, tenemos que si ε está en PRIMERO(α) y $ está en SIGUIENTE(A), añádase $A \rightarrow \alpha$ a M[A, $].
4) Cualquier entrada no definida en Mij será un error.

Es importante señalar que una gramática será LL(1) si en cada casilla tenemos, como máximo, una producción. Si la gramática es recursiva por la izquierda o ambigua, entonces M tendrá al menos una entrada con definición múltiple, y por lo tanto no se podrá implementar mediante LL(1).

Ejemplo.- Para la gramática anterior vamos a calcular Mij:

	Id	**+**	*****	**(**	**)**	**$**
E	E → TE´			E → TE´		
E´		E´→ +TE´			E´→ ε	E´→ ε
T	T → FT´			T → FT´		
T´		T´→ ε	T´→ *FT´		T´→ ε	T´→ ε
F	F → id			F → (E)		

Con la regla E → TE´
En este caso PRIMERO (TE´) = PRIMERO(T) porque ε no está incluido en este último. Utilizando la regla (2), {(, id} ∈ PRIMERO(T), entonces hemos añadido E → TE´ en M[E, (] y M[E, id}

Con la regla E´→ +TE´
También tenemos que PRIMERO(+TE´) = PRIMERO(+) = {+}, entonces, también con la regla (2), en M[E´, +] introducimos E´ → +TE´.

Con la regla E´→ ε
Como PRIMERO(ε) = ε y SIGUIENTE(E´) = {), $}, entonces introducimos E´ → ε en M[E´,)] y M[E´, $], aplicando la regla (3).

Y así continuaríamos con el resto de las reglas.

Condiciones a cumplir para que una gramática sea LL(1):

1) Ninguna gramática ambigua o recursiva por la izquierda puede ser LL(1).
2) Puede demostrarse que una gramática G es de tipo LL(1) si, y sólo si, cuando A → α | β sean dos producciones distintas de G se cumplan las siguientes condiciones:
3) Para ningún terminal a tanto α como β derivan a la vez cadenas que comiencen con a.
4) A lo sumo una de α y β pude derivar la cadena vacía.
5) Si β ⇒* ε, α no deriva ninguna cadena que comience con un terminal en SIGUIENTE(A).

Ejemplo.- Impleméntese un analizador LL(1) de la gramática:

E → **E** + **T** | **E** - **T** | **T**
T → **T** * **F** | **T** / **F** | **F**
F → id | id [**E**] | (**E**)

Ejemplo.- Sea la gramática:

S → { A }
A → id = E
E → id

Implementar un analizador LL(1).

Ejemplo.-

S → **S** ; **L**
S → **L**
L → if expr then **S** else **S** fi
L → if expr then **S** fi
L → instr

No es LL(1), hay que transformarlo eliminando recursividad por la izquierda y factorización en:

S → **L S´**
S´ → ; **L S´** | ε
L → if expr then **S X** fi
L → instr
X → else **S** | ε

Primero calculamos los conjuntos PRIMERO y SIGUIENTE:

	PRIMERO	SIGUIENTE
S	if expr then, instr	$, else, fi
L	if expr then, instr	;, $, else, fi
S´	;, ε	$, else, fi
X	ε, else	fi,

La tabla resultante es la que viene a continuación:

	;	if expr then	fi	instr	else	$
S		**S** → **L S´**		**S** → **L S´**		
L		**L** → if expr then **S X** fi		**L** → instr		
S'	**S´** → ; **LS´**		**S´** → ε		**S´** → ε	**S´** → ε
X			**X** → ε		**X** → else **S**	

Reconozcamos "if expr then if expr then instr fi else if expr then instr fi fi$":

PILA	**ENTRADA**	**SALIDA**
$S	if expr then if expr then instr fi else if expr then instr fi fi$	
$S´L	if expr then if expr then instr fi else if expr then instr fi fi$	**S** → **L S´**
$S´fi X S then expr if	if expr then if expr then instr fi else if expr then instr fi fi$	**L** → if expr then **S X** fi
$S´ fi X S	if expr then instr fi else if expr then instr fi fi$	
$S´ fi XS´L	if expr then instr fi else if expr then instr fi fi$	**S** → **L S´**
$S´ fi XS´fi X S then expr if	if expr then instr fi else if expr then instr fi fi$	**L** → if expr then **S X** fi
$S´ fi XS´fi X S	instr fi else if expr then instr fi fi$	
$ S´ fi X S´fi X S´ L	instr fi else if expr then instr fi fi$	**S** → **L S´**
$ S´ fi X S´fi X S´ instr	instr fi else if expr then instr fi fi$	**L** → instr
$ S´ fi X S´fi X S´	fi else if expr then instr fi fi$	

$ S´ fi X S´fi X	fi else if expr then instr fi fi$	**S´** → ε
$ S´ fi X S´fi	fi else if expr then instr fi fi$	**X** → ε
$ S´ fi X S´	else if expr then instr fi fi$	
$ S´ fi X	else if expr then instr fi fi$	**S´** → ε
$ S´ fi S else	else if expr then instr fi fi$	**X** → else **S**
$ S´ fi S	if expr then instr fi fi$	
$ S´ fi S´ L	if expr then instr fi fi$	**S** → **L S´**
$ S´ fi S´ fi X S then expr if	if expr then instr fi fi$	**L** → if expr then **S X** fi
$ S´ fi S´ fi X S	instr fi fi$	
$ S´ fi S´ fi X S´L	instr fi fi$	**S** → **L S´**
$ S´ fi S´ fi X S´ instr	instr fi fi$	**L** → instr
$ S´ fi S´ fi X S´	fi fi$	
$ S´ fi S´ fi X	fi fi$	**S´** → ε
$ S´ fi S´ fi	fi fi$	**X** → ε
$ S´ fi S´	fi$	
$ S´ fi	fi$	**S´** → ε
$ S´	$	
$	$	**S´** → ε
$	$	ÉXITO

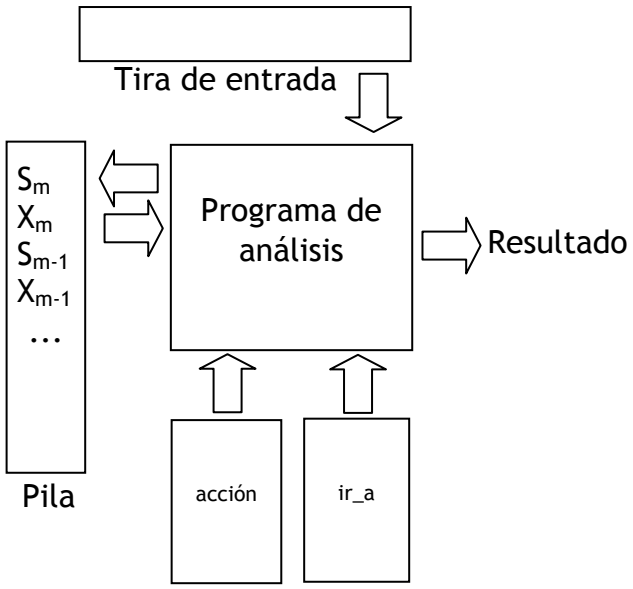
4.2.4 Analizadores ascendentes LR(k).

Los analizadores sintácticos LR(k) funcionan de forma ascendente, sin retroceso y son predictivos en k símbolos. Es un mecanismo que se puede utilizar para analizar una clase más amplia de GCL. La primera "L" significa "Left", izquierda en inglés, indicando que el análisis de la entrada se realiza de izquierda a derecha; la segunda "R" significa "Right" e indica que se realizan las derivaciones por la derecha; k, en nuestro caso k=1, indica que utilizamos un símbolo de entrada de examen por anticipado a cada paso para tomar decisiones de acción en el análisis sintáctico.

A la hora de construir la tabla LR tenemos tres técnicas:

1) SLR.- El más sencillo y menos potente y más restrictivo.
2) LALR.- De potencia intermedia (YACC utiliza este analizador).
3) LR-canónico.- El método más general y por lo tanto el más potente aunque es más difícil de implementar.

Este tipo de analizador consta de los siguientes elementos:



La tabla de "**acción**" nos dirá lo que hacer con cada entrada y la tabla "**ir_a**" nos indicará las transiciones entre estados. El funcionamiento es:

1) Tomamos S_m de la pila y a_i de la tira de entrada.
2) Miramos en accion[S_m, a_i] que nos indicará una de las siguientes acciones:
    a) Desplazar S_p (S_p es un estado).
    b) Reducir $A \rightarrow \beta$. Miramos ir_a[S_{m-r}, A] y obtendremos un nuevo estado al que transitar S_q (r es la longitud de β).
    c) Aceptar.
    d) Error.

Llamamos **configuración** a un par formado por el estado de la pila y la tira de entrada que estamos analizando.
Supongamos una configuración inicial: ($S_0X_1S_1X_2S_2$ ... X_mS_m, a_ia_{i+1}... a_n\$). Respecto a la notación, los S_i son estados y los X_i son símbolos gramaticales (terminales y no terminales).

Vamos a ver las diferentes posibilidades:

a) Si tenemos que la accion[S_k, a_i] = desplazar S_p, entonces la configuración variaría de la siguiente forma:

$$(S_0X_1S_1X_2S_2 \ldots X_mS_ma_iS_p, a_{i+1}\ldots a_n\$)$$

b) Sin embargo, si ocurriera que accion[S_k, a_i] = reducir $A \rightarrow \beta$, la configuración quedaría como:

$$(S_0X_1S_1X_2S_2 \ldots X_{m-r}S_{m-r}AS_q, a_ia_{i+1}\ldots a_n\$)$$

En donde S_q = ir_a[S_{m-r},A] y r es la longitud de β.

c) Si accion[S_k, a_i] = aceptar entonces significa el fin del análisis sintáctico.

d) Si accion[S_k, a_i] = error significará que no es una frase correcta, posteriormente se podría lanzar una rutina que manejara los mensajes de error o de recuperación de errores.

En el siguiente ejemplo veremos como funciona el reconocimiento LR cuando ya disponemos de la tabla.

Ejemplo.-

(1) $E \rightarrow E+T$ (2) $E \rightarrow T$ (3) $T \rightarrow T*F$ (4) $T \rightarrow F$
(5) $F \rightarrow (E)$ (6) $F \rightarrow id$

Suponemos que ya hemos calculado la tabla LR(1), es decir, con un símbolo de anticipación, que sería:

ESTADO	id	+	*	(	)	$	E	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		r3	r3		r5	r5			
	ACCIÓN						IR_A		

Vamos a reconocer "id*id+id":

PILA	ENTRADA	Acción
0	id*id+id$	d5
0id5	*id+id$	r6 ($F \rightarrow id$)
0F3	*id+id$	r4 ($T \rightarrow F$)
0T2	*id+id$	d7
0T2*7	id+id$	d5
0T2*7id5	+id$	r6 ($F \rightarrow id$)
0T2*7F10	+id$	r3 ($T \rightarrow T*F$)
0T2	+id$	r2 ($E \rightarrow T$)
0E1	+id$	d6
0E1+6	id$	d5
0E1+6id5	$	r6 ($F \rightarrow id$)
0E1+6F3	$	r4 ($T \rightarrow F$)
0E1+6T9	$	r1 ($E \rightarrow E+T$)
0E1	$	ACEPTAR

Llamamos gramática LR a aquella gramática de contexto libre para la cual es posible construir la tabla.

4.2.4.1 Método SLR.

Es el método más sencillo pero el menos general, siendo por lo tanto el que tiene más restricciones para la gramática que LALR ó LR-canónico.

Llamamos **elemento de análisis LR(0)** a una producción de la gramática G con un punto en alguna posición del lado derecho de la regla.

Por ejemplo.-

$A \rightarrow XYZ$ los elementos LR(0) son: $A \rightarrow .XYZ$
$A \rightarrow X.YZ$
$A \rightarrow XY.Z$
$A \rightarrow XYZ.$

Un caso especial serán las producciones $A \rightarrow \varepsilon$, en donde $A \rightarrow .$

La base para construir la tabla de análisis LR(0) son una serie de elementos LR(0) asociados a una gramática G se denomina **colección canónica LR(0)**. Para conseguir la colección canónica LR(0) tenemos que definir previamente los conceptos de : gramática aumentada, cerradura y función ir_a.

Dada una gramática G se define G´ como una **gramática aumentada**, equivalente a G, a la que se le añade la producción $S' \rightarrow S$, siendo S la metanoción de la gramática.

Dado un conjunto I de elementos LR(0) de G, **cerradura(I)** será el conjunto de elementos obtenidos de la siguiente manera:

i) Todo elemento de $I \in$ cerradura(I).
ii) $\forall A \rightarrow \alpha.B\beta \in$ cerradura(I), $B \rightarrow \nu \in P$, entonces $B \rightarrow .\nu \in$ cerradura(I).
iii) Se repite (2) hasta que no se puedan añadir más elementos.

Algoritmo para calcular cerradura(I)

```
Función cerradura(I)
BEGIN
        J = I
        REPEAT
                FOR cada elemento A → α.Bβ ∈ J Y cada B → ν ∈ G Y B → .ν ∉ J
                DO añadir B → .ν a J
        UNTIL no se puedan añadir más a J
END.
```

Ejemplo.- En la gramática aumentada:

(0) E´ → E (1) E → E+T (2) E → T (3) T → T*F
(4) T → F (5) F → (E) (6) F → id

Vamos a calcular cerradura(E´ → .E):

1) E´ → .E por la regla (i)
2) E → .E+T por la regla (ii) desde (1)
3) E → .T por la regla (ii) desde (1)
4) T → .T*F por la regla (ii) desde (3)
5) T → .F por la regla (ii) desde (3)
6) F → .(E) por la regla (ii) desde (5)
7) F → .id por la regla (ii) desde (5)

Se llaman **elementos nucleares** a aquellos elementos que no tienen el punto en el extremo izquierdo. Un caso especial es S´ → .S, que se pone en este grupo.

Se llaman **elementos no nucleares** a aquellos que tienen el punto en el extremo izquierdo.

Definimos la **función Ir_a(I, X)** con I un conjunto LR(0) y X $\in$ (N U T):

Ir_a (I, X) = { cerradura(A → αX.β) / A → α.Xβ $\in$ I}

NOTA: Es la transición que nos produce X en el autómata.

Ejemplo.- En la gramática del ejemplo anterior, si tomamos:

I = { E´ → E., E → E.+T }

Ir_a(I, +) se calcula como cerradura(E → E+.T), que sería:

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

Intuitivamente puede interpretarse que, si **A → a.Bb** está en la **cerradura(I)**, en este punto el parsing la secuencia Bb debería ser nuestra entrada. Además, si tenemos que **B → g** es otra regla, un substring prefijo de g debería ser también nuestra entrada. **Ir_a** nos indica que si I es un conjunto de elementos LR(0) válidos para un posible prefijo de **g**, entonces **Ir_a(I, X)** es un conjunto de sucesores válidos si **X** viene después de **g** y **gX** es un prefijo posible.

Obtención de la colección canónica LR(0)

Para construir la **colección canónica LR(0)** para una gramática aumentada G´, utilizamos el siguiente algoritmo:

Algoritmo colección canónica LR(0)

Procedimiento elementos_LR(0)(G´)
BEGIN
    C = {cerradura({[S´ → .S]})}
    REPEAT
        FOR cada I, I ⊂ C Y cada X ∈ G´ donde Ir_a(I, X) ≠ ∅ Y
        Ir_a(I, X) ⊄ C, entonces añadir Ir_a(I, X) a C
    UNTIL no se puedan añadir más conjuntos a C
END.

Ejemplo.- Para la gramática aumentada anterior:

(0) E´ → E (1) E → E+T (2) E → T (3) T → T*F
(4) T → F (5) F → (E) (6) F → id

calculamos la colección canónica LR(0) como:

$\mathbf{I_0}$ = cerradura(E´ → .E) = E´ → .E T → .T*F F → .id
E → .E+T T → .F
E → .T F → .(E)

$\mathbf{I_1}$ = Ir_a(I_0, E) = E´ → E. E → E.+T

$\mathbf{I_2}$ = Ir_a(I_0, T) = E → T. T → T.*F

$\mathbf{I_3}$ = Ir_a(I_0, F) = T → F.

$\mathbf{I_4}$ = Ir_a(I_0, () = cerradura(F → (.E)) = E´ → (.E)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

$\mathbf{I_5}$ = Ir_a(I_0, id) = F → id.

$\mathbf{I_6}$ = Ir_a(I_1, +) = cerradura(E → E+.T) = E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id

$\mathbf{I_7}$ = Ir_a(I_2, *) = T → T*.F
F → .(E)
F → .id

$\mathbf{I_8}$ = Ir_a(I_4, E) = F → (E.)
E → E.+T

Ir_a(I_4, T) = I_2

$Ir\_a(I_4, F) = I_3$
$Ir\_a(I_4, () = I_4$
$Ir\_a(I_4, id) = I_5$

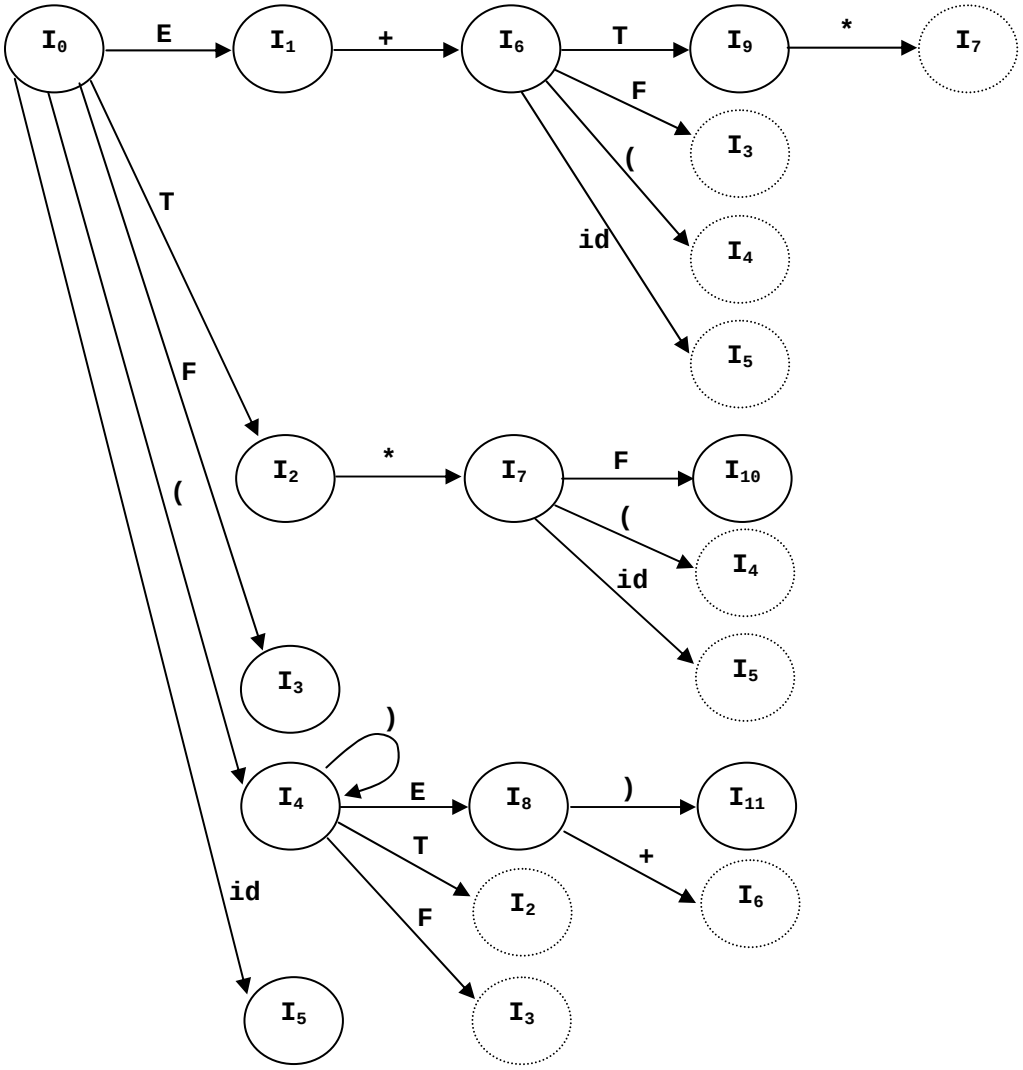
I_9 = $Ir\_a(I_6, T)$ **=** $E \rightarrow E+T.$ $T \rightarrow T.*F$

I_{10} = $Ir\_a(I_7, F)$ **=** $T \rightarrow T*F.$

$Ir\_a(I_7, () = I_4$
$Ir\_a(I_7, id) = I_5$

I_{11} = $Ir\_a(I_8,))$ **=** $F \rightarrow (E).$

$Ir\_a(I_8, +) = I_6$



Cuando tenemos el autómata LR(0), tendremos diversas posibilidades en cuanto a las reducciones posibles, es un método con retroceso (si nos equivocamos de rama tenemos que volver hacia atrás). En realidad se trata de un Autómata finito no determinista pues existen transiciones de la forma A →

$\alpha.X\beta$ a $A \rightarrow \alpha X.\beta$ etiquetadas con "X" y transiciones ε (dentro de los propios I_i) entre elementos de la forma $A \rightarrow \alpha.B\beta$ a $B \rightarrow .\gamma$.

Decimos que el elemento $A \rightarrow \beta_1.\beta_2$ es un **elemento válido** para un prefijo variable $\alpha\beta_1$ si existe una derivación $S' \Rightarrow^* \alpha Aw \Rightarrow^* \alpha\beta_1\beta_2 w$. En general, un elemento será válido para muchos prefijos variables. El hecho de que tengamos un elemento $A \rightarrow \beta_1.\beta_2$ válido para $\alpha\beta_1$ informa sobre si desplazar ($\beta_2 \neq \varepsilon$) o reducir ($\beta_2 = \varepsilon$) cuando se encuentre $\alpha\beta_1$ en la pila del analizador.

Obtención de la tabla de análisis SLR

Para la construcción de la tabla seguimos los siguientes pasos:

i) Construir el conjunto $C = \{ I_0, I_1, ..., I_n \}$
ii) Los estados se construyen a partir de los conjuntos I_i. El estado k es I_k, para rellenar las casillas hacemos (siendo a un terminal):
   a) Si $A \rightarrow \alpha.a\beta \in I_i$, $Ir\_a(I_i, a) = I_j \Rightarrow$ accion[i, a] = desplazar j
   b) Si $A \rightarrow \alpha. \in I_i \Rightarrow$ accion[i, a] = reducir "$A \rightarrow \alpha$" $\forall a \in$ SIGUIENTE(A).
   c) Si $S' \rightarrow S. \in I_i \Rightarrow$ accion[i, $] = ACEPTAR
iii) Si $Ir\_a[I_i, A] = I_j$, siendo A un no terminal $\Rightarrow$ IR_A[i, A] = j
iv) Todas las entradas no definidas constituyen los errores.

Ejemplo.- Para la gramática aumentada ya vista:

(0) $E' \rightarrow E$ (1) $E \rightarrow E+T$ (2) $E \rightarrow T$ (3) $T \rightarrow T*F$
(4) $T \rightarrow F$ (5) $F \rightarrow (E)$ (6) $F \rightarrow id$

Veremos como con los terminales hallamos la tabla de acción y con los no terminales la IR_A:

$I_0 =$ $E' \rightarrow .E$
$E \rightarrow .E+T$
$E \rightarrow .T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$ $F \rightarrow .(E) \in I_0$, $Ir\_a(I_0, () = I_4 \Rightarrow$ accion[0, (] = d4
$F \rightarrow .id$ $F \rightarrow .id \in I_0$, $Ir\_a(I_0, id) = I_5 \Rightarrow$ accion[0, id] = d5

Además, como:

$Ir\_a(I_0, E) = I_1 \Rightarrow IR\_A[0, E] = 1$
$Ir\_a(I_0, T) = I_2 \Rightarrow IR\_A[0, T] = 2$
$Ir\_a(I_0, F) = I_3 \Rightarrow IR\_A[0, F] = 3$

$E' \rightarrow E. \in I_1 \Rightarrow$ accion[1, $] = ACEPTAR

$E \rightarrow E.+T \in I_1$, $Ir\_a(I_1, +) = I_6 \Rightarrow$ accion[1, +] = d6

$E \rightarrow T. \in I_2 \Rightarrow$ accion[2, a] = r2 (regla $E \rightarrow T$) $\forall a \in$ SIGUIENTE(E) = {), +, $}, con lo cual:

accion[2,)] = r2
accion[2, +] = r2
accion[2, $] = r2

$T \rightarrow T.*F \in I_2$, Ir_a($I_2$, *) = $I_7 \Rightarrow$ accion[2, *] = d7

$T \rightarrow T*F. \in I_{10} \Rightarrow$ accion[10, a] = r3 (regla $T \rightarrow T*F$) $\forall a \in$ SIGUIENTE(T) = { *,), +, $}, con lo cual:

accion[10, *] = r3
accion[10,)] = r3
accion[10, +] = r3
accion[10, $] = r3

$E \rightarrow (E.) \in I_8$, Ir_a($I_8$,)) = $I_{11} \Rightarrow$ accion[8,)] = d11
$E \rightarrow E.+T \in I_1$, Ir_a($I_8$, +) = $I_6 \Rightarrow$ accion[8, +] = d6

Así continuaríamos para todos los I_i. Al final nos quedaría la tabla ya vista anteriormente:

ESTADO	**id**	**+**	*****	**(**	**)**	**$**	**E**	**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		r3	r3		r5	r5			
	ACCIÓN						**IR_A**		

Veamos ahora el caso de una gramática que no es SLR, si consideramos la siguiente gramática:

(0) $S' \rightarrow S$ (1) $S \rightarrow L=E$ (2) $S \rightarrow E$ (3) $E \rightarrow L$
(4) $L \rightarrow *E$ (5) $L \rightarrow$ id

Si realizamos el ejercicio completo nos quedaría:

	PRIMERO	SIGUIENTE
S	*, id	$
L	*, id	=, $

E	*, id	=, $

Calculemos los elementos LR(0):

$\mathbf{I_0}$ = {cerradura({[S´ → .S]})} = S´→ .S
S → .L=E
S → .E
L → .*E
E → .L
L → .id

$\mathbf{I_1}$ = Ir_a(I_0, S) = S´ → S.

Al generar la tabla el problema estaría en I_2 porque obtendríamos:

$\mathbf{I_2}$ = Ir_a(I_0, L) = S → L.=E ⇒ desplazar con "="
E → L. ⇒ reducir E→L con "="
porque "=" ∈ SIG(E)

Entre I_0 e I_2 la transición se realizaría por "L".

Con lo cual, mediante SLR no podríamos generar la tabla, este método no sería lo suficientemente potente para esta gramática.

$\mathbf{I_3}$ = Ir_a(I_0, E) = S → E.

$\mathbf{I_4}$ = Ir_a(I_0, *) = L → *.E
E → .L
L → .*E
L → .id

$\mathbf{I_5}$ = Ir_a(I_0, id) = L → id.

$\mathbf{I_6}$ = Ir_a(I_2, =) = S → L=.E
E → .L
L → .*E
L → .id

$\mathbf{I_7}$ = Ir_a(I_4, E) = L → *E.

$\mathbf{I_8}$ = Ir_a(I_4, L) = E → L.

Ir_a(I_4, *) = $\mathbf{I_4}$
Ir_a(I_4, id) = $\mathbf{I_5}$

$\mathbf{I_9}$ = Ir_a(I_6, E) = S → L=E.

Ir_a(I_6, L) = $\mathbf{I_8}$
Ir_a(I_6, *) = $\mathbf{I_4}$

Ir_a(I_6, id) = **I_5**

La tabla nos quedaría:

ESTADO	id	=	*	$	S	L	E
0	d5		d4		1	2	3
1				ACEP.			
2		**d6/r3**		r3			
3				r2			
4	d5		d4			8	7
5		r5		r5			
6	d5		d4			8	9
7		r4		r4			
8		r3		r3			
9				r1			
	ACCIÓN				**IR_A**		

Como se puede ver, tenemos el conflicto en [2,=] tal y como habíamos anunciado.

Ejemplo.- Calcular la tabla de análisis LR de la siguiente gramática mediante el método SLR:

E → E+T
E → T
T → TF
T → F
F → F*
F → a
F → b

Ejemplo.- Calcular la tabla de análisis LR de la siguiente gramática mediante el método SLR:

S → E $
E → T + E
E → T
T → (E)
T → x

Ejemplo.- Calcular la tabla de análisis LR de la siguiente gramática mediante el método SLR:

S´ → S $
S → (L)
S → x
L → S
L → L , S

4.2.4.2 Método LR-canónico.

Es el método LR más potente, funciona para casi todas las gramáticas.

Llamamos **elemento de análisis LR(1)** a una producción con un punto y un terminal, esto es: $[A \rightarrow \alpha.\beta, a]$

Si tenemos un elemento de la forma $[A \rightarrow \alpha.\beta, a]$, no tiene efecto, en cambio si es $[A \rightarrow \alpha., a]$ lo que nos indica es que ese es el elemento que hay que mirar cuando tenemos varios.

Al igual que en SLR, partimos de la gramática aumentada G´ generada a partir de G.

Algoritmo para calcular cerradura(I)

```
Función cerradura(I)
BEGIN
        REPEAT
                FOR cada elemento [A → α.Bβ, a] ∈ I
                Y cada B → ν ∈ G´
                Y b ∈ PRIMERO(βa)
                Y [B → .ν, b] ∉ I
                DO añadir [B → .ν, b] en I
        UNTIL no se puedan añadir más a I
END.
```

Algoritmo para calcular Ir_a(I, X)

```
Función Ir_a(I, X)
BEGIN
        Sea J el conjunto [A → αX.β, a] tal que [A → α.Xβ, a] ∈ I
        Return cerradura(J)
END.
```

Algoritmos de colección de elementos LR(1)

```
Procedimiento elementos_LR(1)(G´)
BEGIN
        C = {cerradura({[S´ → .S, $]})}
        REPEAT
                FOR cada I, I ⊂ C Y cada X ∈ G´ donde Ir_a(I, X) ≠ ∅ Y
                Ir_a(I, X) ⊄ C, entonces añadir Ir_a(I, X) a C
        UNTIL no se puedan añadir más conjuntos a C
END.
```

Ejemplo.- Sea la gramática:

$S \rightarrow CC$
$C \rightarrow cC \mid d$

Calcular los elementos de análisis por el método LR-canónico.

Primero creamos la gramática extendida:

$S' \rightarrow S$
$S \rightarrow CC$
$C \rightarrow cC \mid d$

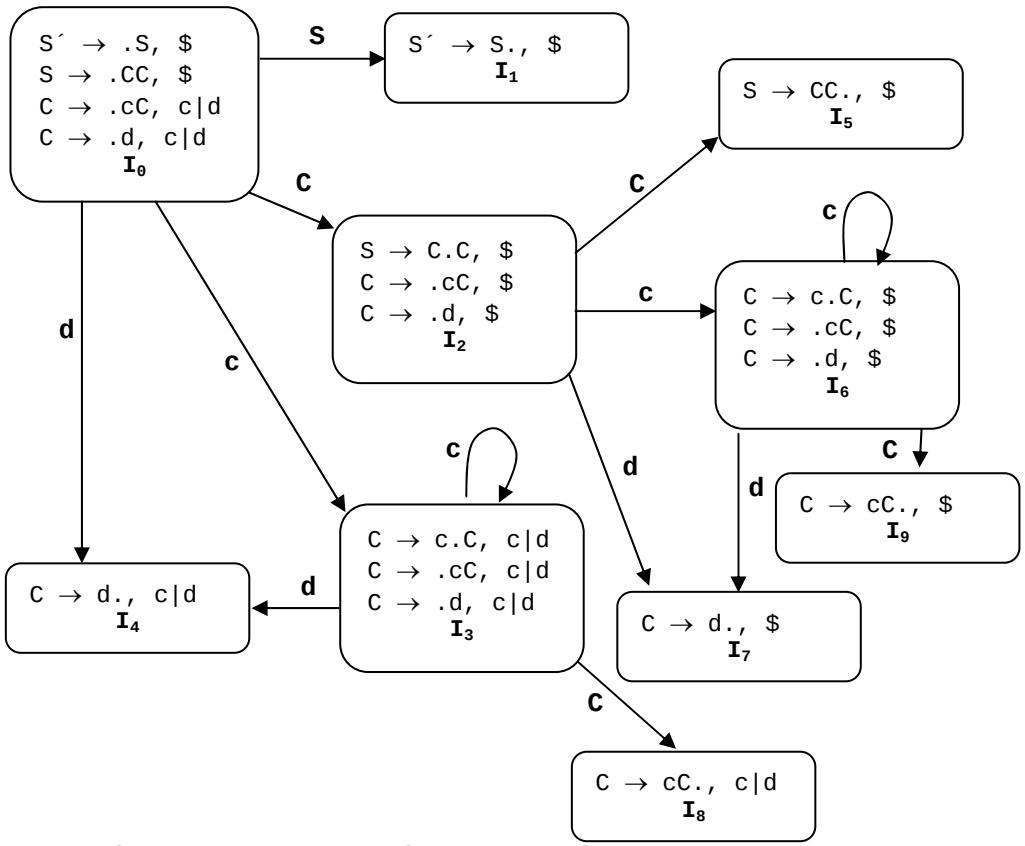
$I_0 = \{cerradura(\{[S' \rightarrow .S, \$]\})\} = S' \rightarrow .S, \$$
$S \rightarrow .CC, \$$ porque PRIMERO($) = {$}
$C \rightarrow .cC, c$ porque PRIMERO(C$} = {c, d}
$C \rightarrow .cC, d$ porque PRIMERO(C$} = {c, d}
$C \rightarrow .d, c$ porque PRIMERO(C$} = {c, d}
$C \rightarrow .d, d$ porque PRIMERO(C$} = {c, d}

$Ir\_a(I_0, S) = cerradura([S' \rightarrow S., \$]) = I_1 =$ $S \rightarrow S. , \$$

$Ir\_a(I_0, C) = cerradura([S \rightarrow C.C, \$]) = I_2 =$ $S \rightarrow C.C, \$$
$C \rightarrow .cC, \$$
$C \rightarrow .d, \$$

$Ir\_a(I_0, c) = cerradura([C \rightarrow c.C, c], [C \rightarrow c.C, d]) = I_3 =$ $C \rightarrow c.C, c$
$C \rightarrow c.C, d$
$C \rightarrow .cC, c$
$C \rightarrow .cC, d$
$C \rightarrow .d, c$
$C \rightarrow .d, d$

Así continuaríamos hasta llegar a conseguir el autómata siguiente:



Obtención de la tabla de análisis LR-canónico

Para la construcción de la tabla seguimos los siguientes pasos:

i) Construir el conjunto C = { I_0, I_1, ..., I_n }

ii) Los estados se construyen a partir de los conjuntos I_i. El estado k es I_k, para rellenar las casillas hacemos:
   a) Si $[A \rightarrow \alpha.a\beta, b] \in I_i$, $Ir\_a(I_i, a) = I_j \Rightarrow$ accion[i, a] = desplazar j
   b) Si $[A \rightarrow \alpha., a] \in I_i$, $A \neq S' \Rightarrow$ accion[i, a] = reducir "$A \rightarrow \alpha$"
   c) Si $[S' \rightarrow S., \$] \in I_i \Rightarrow$ accion[i, $] = ACEPTAR

iii) Si $Ir\_a[I_i, A] = I_j$, siendo A un no terminal $\Rightarrow$ IR_A[i, A] = j

iv) Todas las entradas no definidas constituyen los errores.

Ejemplo.- Para la gramática anterior la tabla resultante sería:

ESTADO	c	d	$	S	C
0	d3	d4		1	2
1			ACEP.		
2	d6	d7			5
3	d3	d4			8
4	r3	r3			
5			r1		
6	d6	d7			9
7			r3		
8	r2	r2			
9			r2		
	ACCIÓN			IR_A	

4.2.4.3 Método LALR.

El número de estados que genera es equiparable al SLR, pero es más potente, más general. Además la herramienta YACC utiliza este tipo de análisis.

Existen bastantes algoritmos para realizarlo pero vamos a ver como calcularlo a partir del LR-canónico.

En LR-canónico tenemos, por ejemplo, un estado en el que nos encontramos:

$[A \rightarrow \alpha.a\beta, e]$
$[A \rightarrow \alpha.a\beta, f]$

En LALR, llamamos a "$\alpha.a\beta$" **corazón** y definimos un único estado para ambos. Tendremos menos estados, menos errores y también es menos potente.

En el ejercicio realizado en LR-canónico teníamos:

$I_4 = [C \rightarrow d., c|d]$
$I_7 = [C \rightarrow d., \$]$

Entonces definimos un estado único $I_{47} = [C \rightarrow d., c|d|\$]$.

Ocurre lo mismo con los estados 3 y 6 y con los estados 8 y 9.
Para realizar la tabla LALR hacemos los siguientes pasos:

1. Construimos el conjunto LR(1) como en el caso de LR-canónico.
2. Remplazamos los conjuntos con el mismo corazón por su unión.
3. Encontramos las acciones de la misma forma que en LR-canónico, si aparecen conflictos es que no puede implementarse como LALR.
4. Construimos la tabla IR_A usando que: el valor del corazón de Ir_a (I_i, X) es el mismo para todos los I_i con el mismo corazón.

La tabla resultante para la gramática anterior es la siguiente:

ESTADO	c	D	$	S	C
0	d36	d47		1	2
1			ACEP.		
2	d36	d47			5
36	d36	d47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		
	ACCIÓN			IR_A	

Ejemplo.- Sea la siguiente gramática, ya extendida:

$S' \rightarrow S$
$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$
$A \rightarrow c$
$B \rightarrow c$

$I_0 = \{cerradura(\{[S' \rightarrow .S, \$]\})\} = S' \rightarrow .S, \$$
$S \rightarrow .aAd, \$$
$S \rightarrow .bBd, \$$
$S \rightarrow .aBe, \$$
$S \rightarrow .bAe, \$$

$Ir\_a(I_0, S) = cerradura([S' \rightarrow S., \$]) = S \rightarrow S. , \$ = I_1$

$Ir\_a(I_0, a) = cerradura([S \rightarrow a.Ad, \$], [S \rightarrow a.Be, \$]) =$ $S \rightarrow a.Ad, \$ = I_2$
$S \rightarrow a.Be, \$$
$A \rightarrow .c, d$
$B \rightarrow .c, e$

$Ir\_a(I_0, b) = cerradura([S \rightarrow b.Bd, \$], [S \rightarrow b.Ae, \$]) =$ $S \rightarrow b.Bd, \$ = I_3$
$S \rightarrow b.Ae, \$$
$A \rightarrow .c, e$
$B \rightarrow .c, d$

$Ir\_a(I_2, A) = S \rightarrow aA.d, \$ = I_4$
$Ir\_a(I_2, B) = S \rightarrow aB.e, \$ = I_5$

$Ir\_a(I_2, c) = A \rightarrow c., d = I_6$
$B \rightarrow c., e$

$Ir\_a(I_3, A) = S \rightarrow bA.e, \$ = I_7$
$Ir\_a(I_3, B) = S \rightarrow bB.d, \$ = I_8$

$Ir\_a(I_3, c) = A \rightarrow c., e = I_9$
$B \rightarrow c., d$

$Ir\_a(I_4, d) = S \rightarrow aAd., \$ = I_{10}$
$Ir\_a(I_5, e) = S \rightarrow aBe., \$ = I_{11}$
$Ir\_a(I_7, e) = S \rightarrow bAe., \$ = I_{12}$
$Ir\_a(I_8, d) = S \rightarrow bBd., \$ = I_{13}$

Si construimos los conjuntos LR(1), dos de ellos serán:

$I_6 = \{ [A \rightarrow c., d], [B \rightarrow c., e] \}$
$I_9 = \{ [A \rightarrow c., e], [B \rightarrow c., d] \}$

que tienen el mismo corazón y darían lugar a la unión de ambos:

$I_{69} = \{ [A \rightarrow c., d|e], [B \rightarrow c., d|e] \}$

en donde, al generar la tabla de análisis LR, producirá un conflicto, en dos casillas aparecerán dos reducciones simultáneamente A → c y B → c. Este sería un ejemplo de gramática que sí es LR-canónico pero no LALR.

4.2.5 Generador de analizadores sintácticos YACC.

YACC significa "Yet Another Compiler-Compiler", es decir "otro compilador de compiladores más"; su primera versión fue creada por S. C. Johnson. Este generador se encuentra disponible como una orden del sistema UNIX y se ha utilizado para facilitar la implementación de gran cantidad de analizadores sintácticos, tanto para compiladores de lenguajes de programación como con otros fines.

Si tenemos un programa yacc en un fuente con nombre *ejemplo.y*, los pasos para convertirlo en un ejecutable son los siguiente:

```
yacc ejemplo.y
cc –o miejemplo y.tab.c –ly
/*A veces puede ser necesario también –lm*/
miejemplo < fichero_prueba.txt
```

La secuencia es la siguiente:

4. Yacc crea un fichero fuente en C, y.tab.c, desde una descripción del analizador sintáctico en YACC.
5. y.tab.c se compila (es un programa C) generándose el ejecutable.
6. Este ejecutable realizará la traducción especificada.

4.2.5.1 Partes de un programa YACC

La estructura de un programa en YACC es la siguiente:

Declaraciones
%%
Reglas de traducción
%%
Rutinas de apoyo en C

- En la parte de **Declaraciones** se incluyen declaraciones ordinarias de C, delimitadas por %{ y %}, declaraciones de variables temporales usadas por las reglas de traducción o procedimientos auxiliares de la segunda y tercera secciones y componentes léxicos.

- Las **reglas de traducción** constan de una producción de la gramática y una acción semántica asociada, de la siguiente forma:

```
<lado izquierdo>   :    <alt 1>    { acción semántica 1}
                   |    <alt 2>    { acción semántica 2}
```

```
...
|       <alt n>        { acción semántica n
;
```

En las reglas de producción, un carácter simple entrecomillado ´c´ se considera como el símbolo terminal c, y las cadenas sin comillas de letras y dígitos no declarados como componentes léxicos se consideran no terminales. Las acciones semánticas son instrucciones C, dentro de ellas, el símbolo $$ se refiere al valor del atributo asociado con el no terminal del lado izquierdo, mientras que $i se refiere al valor asociado con el i-ésimo símbolo gramatical, terminal o no terminal, del lado derecho.

- Las **rutinas de apoyo en C** es la tercera parte de un programa en YACC. Es necesario proporcionar un análisis léxico con nombre *yylex()*. También pueden añadirse rutinas e recuperación de apoyo.

El analizador léxico *yylex()* produce pares formados por un componente léxico y su valor de atributo asociado. Si se devuelve un componente léxico como CIFRA, éste debe de declararse en la primera sección del programa. El valor del atributo asociado a un componente léxico se comunica al analizador sintáctico mediante la variable *yylval*.

Vamos a ver un ejemplo de implementación sobre YACC de una calculadora sencilla. La gramática para las expresiones aritméticas será la siguiente:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$R \rightarrow (E) \mid$ dígito

El componente léxico **dígito** es un solo dígito entre 0 y 9. El código YACC es el siguiente:

```
%{
#include <ctype.h>
%}

%token CIFRA

%%

linea        :     expr ´\n'          { printf("%d\n", $1); }

expr         :     expr ´+´ termino   { $$ = $1 + $3; }
             |     termino
             ;

termino      :     termino ´*´ factor {$$ = $1 * $3; }
             |     factor
             ;

factor       :     ´(´expr ´)´        { $$ = $2; }
             |     CIFRA
             ;
%%

yylex() {
```

```
        int c;
        c = getchar();
        if (isdigit(c)) {
                yylval = c-´0´;
                return CIFRA;
        }
        return c;
}
```

En la parte de declaraciones se ha introducido el include <ctype.h>, que nos permitirá utilizar el yylex() la función “isdigit”. La producción inicial (linea) se introduce para obligar al programa a aceptar una entrada seguida de un carácter de nueva línea.