





Procesamiento de Lenguajes

Grado en Ingeniería Informática
4º curso

Profesores: Bernardino Arcay
Carlos Dafonte
Ángel Gómez
María Martínez

*Departamento de Tecnoloxías da Información e as Comunicacóns.
Universidade da Coruña
Curso 2014/2015 Rev. 140923*

ÍNDICE

1	INTRODUCCIÓN.....	1
1.1	Estructura de un compilador.....	2
1.2	Ejemplo de las fases de un compilador.....	3
2	LENGUAJES Y GRAMÁTICAS.....	5
2.1	Notación de Chomsky (1959).....	6
2.2	Clasificación de Chomsky.....	7
2.3	Gramáticas de contexto libre (GCL).....	8
2.4	Reglas BNF.....	9
2.5	Problemas en las GCL.....	10
2.5.1	Recursividad. 	10
2.5.2	Reglas con factor repetido por la izquierda. 	12
2.5.3	Ambigüedad. 	13
2.6	Simplificación de gramáticas.....	14
2.6.1	Detección de un lenguaje vacío.....	14
2.6.2	Eliminación de reglas lambda (λ).....	15
2.6.3	Eliminación de reglas unitarias o reglas cadena.....	16
2.6.4	Eliminación de símbolos inútiles.....	17
2.7	Gramática limpia.....	19
2.8	Forma normal de Chomsky (FNC).....	20
2.9	Resumen.....	20
2.10	Ejercicios.....	21
3	ANÁLISIS LÉXICO.....	23
3.1	Tipos de máquinas reconocedoras o autómatas.....	23
3.2	Autómatas Finitos.....	24
3.3	Conversión de una Gramática Regular en un Autómata finito.....	26
3.4	Expresión regular.....	27
3.5	Algoritmo de Thompson.....	28
3.6	Transformación de un AFND- λ en un AFD.....	29
3.7	Traductores finitos (TF).....	31
3.8	Implementación de autómatas.....	32
3.8.1	Tabla compacta.....	32
3.8.2	Autómata programado.....	33
3.9	Ejemplo. Scanner para números reales sin signo en Pascal.....	34
3.10	Acciones semánticas. 	36
4	Análisis sintáctico (Parsing).....	38

4.1	Máquinas teóricas, mecanismos con retroceso.....	41
4.1.1	Autómatas con pila (AP).....	41
4.1.2	Esquemas de traducción (EDT).....	45
4.1.3	Traductores con pila (TP).....	47
4.2	Algoritmos sin retroceso.....	48
4.2.1	Análisis sintáctico ascendente por precedencia simple.....	49
4.2.2	Análisis sintáctico ascendente por precedencia de operadores.....	59
4.2.3	Analizadores descendentes LL(K).....	62
4.2.4	Analizadores ascendentes LR(k).....	68
5	ANÁLISIS SEMÁNTICO.....	84
5.1	Definiciones dirigidas por la sintáxis.	84
5.1.1	Grafos de dependencias.....	86
5.2	Esquema de traducción	87
5.3	Comprobaciones en tiempo de compilación.	88
5.3.1	Sistemas de tipos.....	89
5.3.2	Una gramática y sus comprobaciones de tipos.....	89
5.3.3	Equivalencia de expresiones de tipos.....	91
5.3.4	Codificación de tipos.....	91
5.3.5	Conversión de tipos.....	92
5.3.6	Sobrecarga de funciones y operadores.....	93
6	GENERACIÓN DE CÓDIGO.....	96
6.1	Lenguajes intermedios.....	96
6.1.1	Notación Polaca Inversa (RPN).....	96
6.1.2	Cuartetos.....	96
6.1.3	Tercetos.....	98
6.2	Generación de código intermedio.....	99
6.2.1	Generación de RPN desde expresiones aritméticas.....	99
6.2.2	Generación de cuartetos.....	100
6.3	Generación de código desde lenguaje intermedio.....	101
6.3.1	Definición de la máquina objeto.....	101
6.3.2	Generación de código desde RPN.....	102
6.3.3	Generación de código desde cuartetos.....	103
7	OPTIMIZACIÓN DE CÓDIGO.....	105
7.1	Algoritmo de Nakata.	107
7.2	Un ejemplo de optimización manual.....	111
7.3	Lazos en los grafos de flujo.....	111
7.4	Análisis global del flujo de datos.....	112
7.4.1	Alcance de definiciones en estructuras de control.....	114
7.4.2	Notación vectorial para representar genera y desactiva.....	117
7.5	Solución iterativa de las ecuaciones de flujo de datos.....	120
7.5.1	Análisis de alcance de definiciones.....	120
7.5.2	Análisis de expresiones disponibles.....	123
7.5.3	Análisis de variables activas.....	126
8	INTÉRPRETES.....	129
8.1	Estructura de un intérprete actual.....	129

BIBLIOGRAFÍA

Aho, A.V.; Lam M.; Sethi, R. ; Ullman, J.D.
Compiladores: Principios, técnicas y herramientas (*)
Addison-Wesley, Reading, 2008 (2ª edición)

Garrido, A. ; Iñesta J.M. ; Moreno F. ; Pérez J.A.
Diseño de compiladores (*)
Publicaciones Universidad de Alicante, 2004

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

Alfonseca, M.; de la Cruz, M.; Ortega, A.; Pulido, E.
Compiladores e intérpretes: teoría y práctica
Pearson Prentice-Hall, 2006

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

Allen I.; Holub
Compiler design in C
Prentice-Hall, 1991

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

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

Sudkamp T.A.
Languages and machines: An Introduction to the Theory of Computer Science
Addison-Wesley, 2005 (3ª edición)

() Libros especialmente recomendables y con un enfoque muy práctico. El primero de ellos es la base de la asignatura.*

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.
- Seguridad.
- 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.

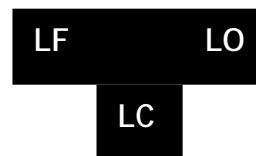
Bootstrapping: Mediante esta técnica se construye un lenguaje L utilizando el mismo lenguaje L, realizando mejoras en el propio compilador como puede ser, por ejemplo, la inclusión de nuevas sentencias y estructuras.

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

LF -> Lenguaje fuente

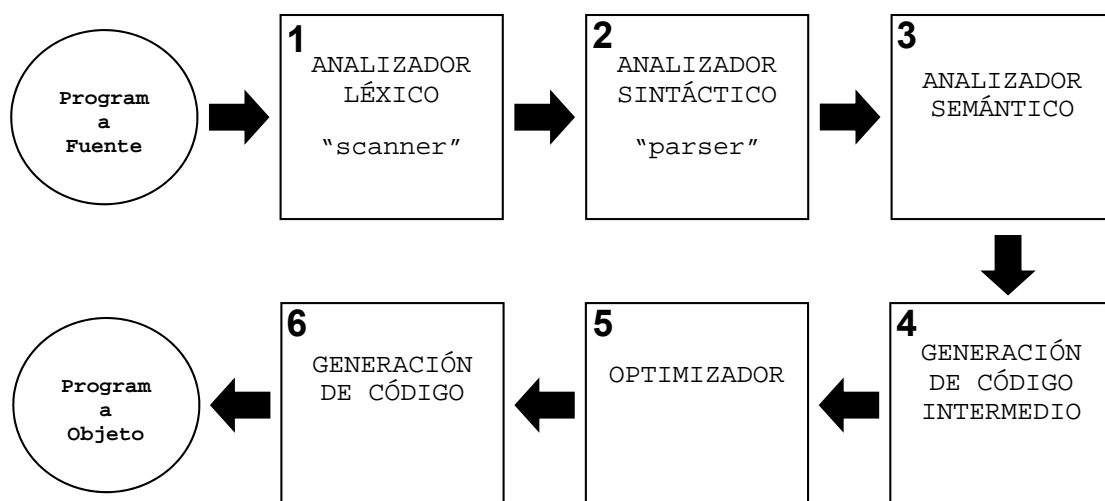
LO -> Lenguaje objeto

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*.

Etapas 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.

Etapas 2. Analizador sintáctico o “parser”.

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

Etapas 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.

Etapas 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).

Etapas 5. Optimizador.

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

Etapas 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:

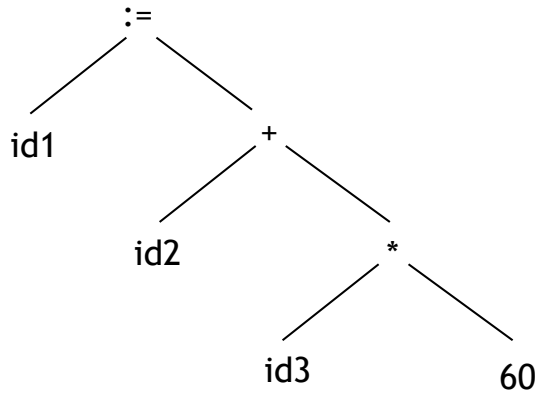
posicion := inicial + velocidad * 60

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

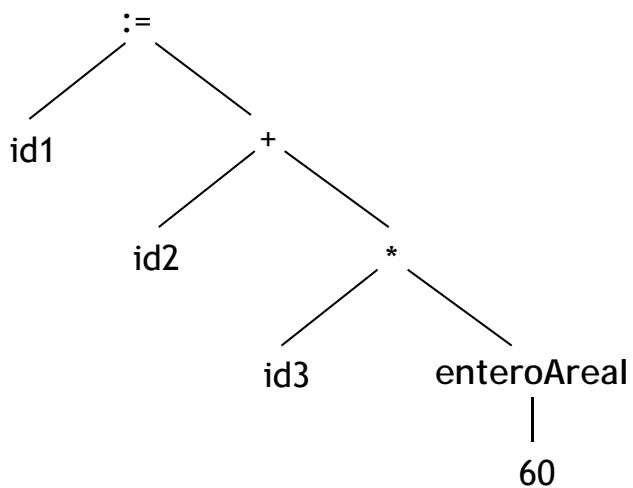
ANALIZADOR LÉXICO

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


2 LENGUAJES Y GRAMÁTICAS.

Definiciones:

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

Tira de caracteres: Yuxtaposición 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 longitud se representa como $|x|$.

Ejemplo.- $x = abc \quad |x| = 3$
 $|\lambda| = 0$

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

Ejemplo.- $x = abc$
 $x^2 = abcabc$
 $x^0 = \lambda$
 $x^1 = abc$

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:

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

Definimos Potencia

Definimos Cierre Transitivo

Definimos Cierre Transitivo y Reflexivo

A partir de estas dos últimas tenemos que:

$$A^n = A.A^{n-1}; A^0 = \{\lambda\}; A^1 = A$$

$$A^+ = \bigcup_{i>0} A^i$$

$$A^* = \bigcup_{i \geq 0} A^i$$

$$A^* = A^+ \cup A^0 = A^+ \cup \{\lambda\}$$

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 distintas formas de definir un lenguaje:

a) Enumerando sus elementos.

Ejemplo.- $L_1 = \{ a, abc, d, ef, g \}$

b) Notación matemática.

Ejemplo 1.- $L_2 = \{ a^n / n \in \mathbb{N} \} = \{ \lambda, a, aa, \dots \}$

Ejemplo 2.- $L_3 = \{ x / |x| = 3 \text{ y } x \in V \}$

$V = \{ a, b, c, d \}$

¿Cuántos elementos tiene el lenguaje?

$VR_4^3 = 4^3 = 64 \text{ elementos}$

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).

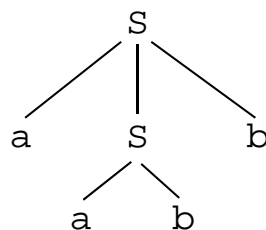
Ejemplo.-

$G1 = \{ \{S\}, \{a,b\}, P, S \}$

$P:$ $S \rightarrow ab$

$S \rightarrow aSb$

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 dependiente de las reglas de producción de la gramática.

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

$$\begin{aligned} G = (N, T, P, S) \quad & \alpha \rightarrow \beta \\ & \alpha \in (N \cup T)^+ \text{ (NOTA: } \alpha \text{ nunca puede ser } \lambda) \\ & \beta \in (N \cup T)^* \text{ (NOTA: } \beta \text{ puede ser } \lambda) \end{aligned}$$

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

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

$$\begin{aligned} G = (N, T, P, S) \quad & \alpha A \beta \rightarrow \alpha v \beta \\ & \alpha, \beta \in (N \cup T)^* \\ & v \in (N \cup T)^+ \\ & A \in N \end{aligned}$$

Se puede ver que, si llamamos $\alpha A \beta = x_1$ y $\alpha v \beta = x_2$, entonces $|x_2| \geq |x_1|$

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

$$\begin{aligned} G = (N, T, P, S) \quad & \omega \rightarrow v \\ & \omega, v \in (N \cup T)^+ \\ & |\omega| \leq |v| \end{aligned}$$

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

$$\begin{aligned} G = (N, T, P, S) \quad & A \rightarrow \alpha \\ & A \in N \\ & \alpha \in (N \cup T)^* \end{aligned}$$

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.

$$\begin{aligned} G = (N, T, P, S) \quad & A \rightarrow aB ; A \rightarrow a ; A \rightarrow \lambda \\ & a \in T \\ & A, B \in N \end{aligned}$$

Las gramáticas tipo 3 son las más restrictivas y las tipo 0 las menos pero existe una peculiaridad porque, de forma teórica estricta, 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:

$$G = (N, T, P, S) \quad \begin{array}{l} A \rightarrow \alpha \\ A \in N \\ \alpha \in (N \cup T)^* \end{array}$$

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

$$\alpha = \delta A \mu, \beta = \delta \nu \mu, \delta \text{ y } \mu \text{ son dos tiras y aplicamos } A \rightarrow \nu$$

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

Una derivación de longitud n , de la forma:

$$\alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$$

se representará como:

$$\alpha_0 \Rightarrow^+ \alpha_n$$

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

Ejemplo.- Gramática $S \rightarrow aSb \mid \lambda$

$$S \rightarrow aSb \rightarrow aaSbb \rightarrow aaaSbbb \rightarrow aaaaSbbbb \rightarrow aaaaaSbbbbb$$

Es una derivación de longitud 5.

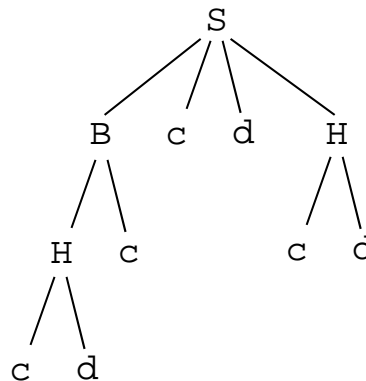
Sentencias, con $G = (N, T, P, S)$, son el conjunto de tiras de la siguiente forma, $L(G) = \{ \omega / S \Rightarrow^* \omega, \omega \in T^* \}$.

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.-

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

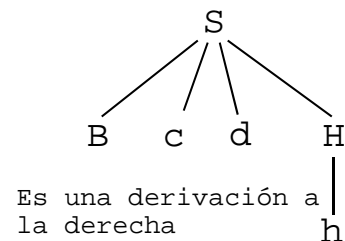
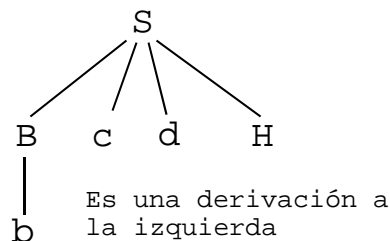


Derivación a la izquierda: es la derivación que se realiza sustituyendo las metanociones que se encuentran más a la izquierda de la forma sentencial.

Derivación a la derecha: es la derivación 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 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 \rightarrow alternativa1 | alternativa2 ...

NOTA: También se puede utilizar, en lugar de \rightarrow , el símbolo $:=$

Ejemplo.-

$S \rightarrow - D \mid D$

$D \rightarrow L \mid L . L$

$L \rightarrow N \mid N L$

$N \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

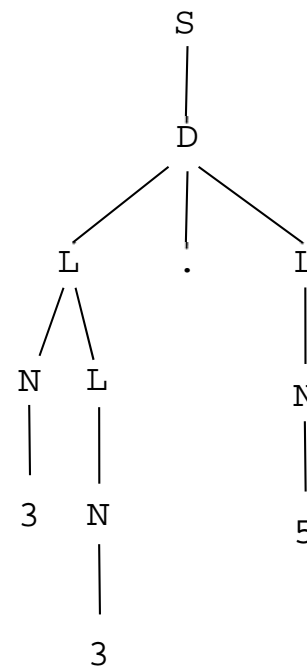
S es la metanoción

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.



2.5 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.

2.5.1 Recursividad. ⚠️

Tenemos diferentes tipos de recursividad. Supongamos que tenemos una regla de la forma $A \rightarrow \alpha A \beta$, dependiendo de lo que sean α y β tendremos:

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

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

Esto podemos ponerlo de la siguiente forma:

$$\begin{aligned} A &\rightarrow A\alpha_i & 1 \leq i \leq q \\ A &\rightarrow \beta_j & 1 \leq j \leq p \end{aligned}$$

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

$$\begin{aligned} A &\rightarrow \beta_j A' & 1 \leq j \leq p \\ A' &\rightarrow \alpha_i A', A' \rightarrow \lambda & 1 \leq i \leq q \end{aligned}$$

Por ejemplo, si tenemos las reglas:

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

La transformación podría ser la siguiente:

$$\begin{aligned} A &\rightarrow cA' \mid dA' \\ A' &\rightarrow aA' \mid bA' \mid \lambda \end{aligned}$$

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

$$\begin{aligned} A &\rightarrow \beta_j, A \rightarrow \beta_j A' & 1 \leq j \leq p \\ A' &\rightarrow \alpha_i, A' \rightarrow \alpha_i A' & 1 \leq i \leq q \end{aligned}$$

Siguiendo el ejemplo anterior, la transformación sería:

$$\begin{aligned} A &\rightarrow c \mid d \mid cA' \mid dA' \\ A' &\rightarrow a \mid b \mid aA' \mid bA' \end{aligned}$$

Ejemplo 1.-

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid x \end{aligned}$$

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 $\alpha = "+ T"$ y $\beta = "T"$. Cambiándolo como $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \lambda$, tenemos:

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \lambda \end{aligned}$$

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

$$\begin{aligned} T &\rightarrow F T' \\ T' &\rightarrow * F T' \mid \lambda \end{aligned}$$

La gramática resultante es:

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \lambda \\ T &\rightarrow F T' \\ T' &\rightarrow * F T' \mid \lambda \\ F &\rightarrow (E) \mid x \end{aligned}$$

Ejemplo 2.-

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

2.5.2 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:

$$A \rightarrow \alpha A'$$

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

Ejemplo 1.- Eliminación del factor repetido por la izquierda y recursividad también por la izquierda:

$S \rightarrow S ; L$	$S \rightarrow L S'$
$S \rightarrow L$	$S' \rightarrow ; S \mid \lambda$
$L \rightarrow \text{if expr then } S \text{ else } S \text{ fi}$	$L \rightarrow \text{if expr then } S X \text{ fi}$
$L \rightarrow \text{if expr then } S \text{ fi}$	$L \rightarrow \text{instr}$
$L \rightarrow \text{instr}$	$L \rightarrow \text{else } S \mid \lambda$

Ejemplo 2.- Eliminación del factor repetido por la izquierda y recursividad también por la izquierda:

$X \rightarrow XW$	$X \rightarrow bX' \mid X'$
$X \rightarrow b \mid \lambda$	$X \rightarrow X'$
$W \rightarrow aW \mid z$	$X' \rightarrow WX' \mid \lambda$
	$W \rightarrow Aw \mid z$

2.5.3 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.-

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

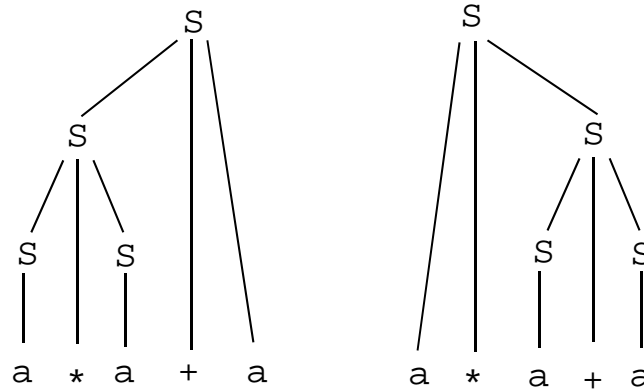
P:

$$S \rightarrow a$$

$$S \rightarrow S+S$$

$$S \rightarrow S*S$$

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.

2.6 Simplificación de gramáticas.

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

Sabemos que un lenguaje podemos describirlo como:

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

2.6.1 Detección de un lenguaje vacío.

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

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

```

BEGIN
  viejo = { $\emptyset$ }
  nuevo = {A / (A  $\rightarrow$  t)  $\in$  P, t  $\in$  T*}
  WHILE nuevo  $\neq$  viejo do BEGIN
    viejo := nuevo
    nuevo := viejo U { B / (B  $\rightarrow$   $\alpha$ )  $\in$  P,  $\alpha \in$  (T U viejo)* }
  END
  IF S  $\in$  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 $\{A\}$.

2.6.2 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 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 $L - \{\lambda\}$ es $L(G')$ con G' sin símbolos inútiles ni reglas- λ .

Algoritmo de detección de símbolos anulables

```

BEGIN
    viejo :=  $\{\emptyset\}$ 
    nuevo :=  $\{A \mid A \rightarrow \lambda \in P\}$ 
    WHILE viejo  $\neq$  nuevo DO
        BEGIN
            viejo := nuevo
            nuevo := viejo  $\cup \{B \mid B \rightarrow \alpha, \alpha \in \text{viejo}^*\}$ 
        END
    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 $A \rightarrow \alpha_0 B_0 \alpha_1 B_1 \alpha_2 B_2 \dots \alpha_k B_k$ 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 $A \rightarrow \alpha_0 X_0 \alpha_1 X_1 \alpha_2 X_2 \dots \alpha_k X_k$, 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 $S' \rightarrow S \mid \lambda$ (si S no está involucrada en la parte derecha de ninguna regla, no es necesario).

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 }

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

2.6.3 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 (N_A, N_B, \dots), tantos como nodos no terminales tenga la gramática, utilizando este algoritmo para cada no terminal.

Algoritmo

```
BEGIN
  viejo := { $\emptyset$ }
  nuevo := {A}
  WHILE viejo  $\neq$  nuevo DO
    BEGIN
      viejo := nuevo
      nuevo := viejo U {C /  $B \rightarrow C, B \in$  viejo}
    END
  NA := nuevo
END
```

Una vez contruidos los conjuntos N_X , 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 $B \in N_A$.

NOTA: con este algoritmo también eliminaremos las derivaciones de la forma: $A \Rightarrow^* X$, es decir $A \rightarrow B \rightarrow C \rightarrow \dots \rightarrow X$, 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:

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

2.6.4 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:

$$S \Rightarrow^* \alpha X \beta \Rightarrow^* t, \quad t \in T^*, \alpha \text{ y } \beta \in (N \cup T)^*$$

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 $A \rightarrow x_1 x_2 \dots x_n$ con $x_i \in (N' \cup T)$.

Algoritmo

```
BEGIN
    viejo = { $\emptyset$ }
    nuevo = { A / A  $\rightarrow$  t  $\in$  P, t  $\in$  T* }
    WHILE viejo  $\neq$  nuevo DO
        BEGIN
            viejo := nuevo
            nuevo := viejo U { B / B  $\rightarrow$   $\alpha \in$  P,  $\alpha \in$  (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  $\rightarrow$   $\alpha x \beta \in$  P}
    WHILE viejo  $\neq$  nuevo DO
        BEGIN
            viejo := nuevo
            nuevo := viejo U {y / A  $\rightarrow$   $\alpha y \beta$ , A  $\in$  viejo}
        END
        N' := nuevo  $\cap$  N
        T' := nuevo  $\cap$  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	\emptyset	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:

$N' = \{C, A, S, B\}$
 $S \rightarrow aAA$
 $A \rightarrow aAb \mid aC$
 $B \rightarrow Ac$
 $C \rightarrow b$

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	

$T' = \{a, b\}$
 $N' = \{S, A, C\}$
 $S \rightarrow aAA$
 $A \rightarrow aAb \mid aC$
 $C \rightarrow b$

2.7 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.

2.8 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: $A \rightarrow BC$ ó $A \rightarrow a$, con $A,B,C \in N$ y $a \in T$. 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

$P' = \{\emptyset\}$

Añadimos a P' las reglas del tipo $A \rightarrow a$

Añadimos a P' las reglas del tipo $A \rightarrow BC$

IF $A \rightarrow x_1 x_2 \dots x_k \in P$, $k > 2$ THEN

BEGIN

Añadimos a P' $A \rightarrow x_1' < x_2 \dots x_k >$

$< x_2 \dots x_k > \rightarrow x_2' < x_3 \dots x_k >$

$< x_3 \dots x_k > \rightarrow x_3' < x_4 \dots x_k >$

$< x_{k-2} > \rightarrow x_{k-1} x_k$

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 $x_i \in T$

Sustituimos 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.

$S \rightarrow BA$

$A \rightarrow 01AB0$

$A \rightarrow 0$

$B \rightarrow 1$

El único caso problemático es el de $A \rightarrow 01AB0$, lo resolvemos así.

$A \rightarrow 01AB0$ $A \rightarrow A_1 A_2$ $A_1 \rightarrow 0$ $A_2 \rightarrow 1AB0$ $A_2 \rightarrow A_3 A_4$

$A_3 \rightarrow 1$ $A_4 \rightarrow ABO$ $A_4 \rightarrow A A_6$ $A_5 \rightarrow BO$ $A_5 \rightarrow B A_6$

$A_6 \rightarrow 0$

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

2.9 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).

2.10 Ejercicios.

Ejercicio1.-

Simplificar la siguiente gramática:

$$\begin{aligned} S &\rightarrow aS \mid AB \mid AC \\ A &\rightarrow aA \mid \lambda \\ B &\rightarrow bB \mid bS \\ C &\rightarrow cC \mid \lambda \end{aligned}$$

Solución 1.-

$$\begin{aligned} S' &\rightarrow aS \mid a \mid AB \mid bB \mid bS \mid b \mid AC \mid aA \mid cC \mid c \mid \lambda \\ S &\rightarrow aS \mid a \mid AB \mid bB \mid bS \mid b \mid AC \mid aA \mid cC \mid c \\ A &\rightarrow aA \mid a \\ B &\rightarrow bB \mid bS \mid b \\ C &\rightarrow cC \mid c \end{aligned}$$

Ejercicio 2 .-

$$\begin{aligned} S &\rightarrow ACA \mid CA \mid AA \mid C \mid A \mid \lambda \\ A &\rightarrow aAa \mid aa \mid B \mid C \\ B &\rightarrow bB \mid b \\ C &\rightarrow cC \mid c \end{aligned}$$

Solución 2.-

$$\begin{aligned} S &\rightarrow ACA \mid CA \mid AA \mid cC \mid c \mid aAa \mid aa \mid bB \mid b \mid \lambda \\ A &\rightarrow aAa \mid aa \mid bB \mid b \mid cC \mid c \\ B &\rightarrow bB \mid b \\ C &\rightarrow cC \mid c \end{aligned}$$

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 autómatas 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 autómatas 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 T_e . 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 T_p , 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 autómatas al conjunto de elementos que definen la situación del autómatas en un momento t . Existen dos configuraciones especiales: inicial y final, un autómatas 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 autómatas.

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 autómatas. $S_i \rightarrow S_j$ 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 Autómatas Finito No Determinista (AFND), para luego convertirlo en un Autómatas 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:

$$Q \times \{Te \cup \{\lambda\}\} \rightarrow P(Q)$$

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:

$$(q, a\alpha) \vdash \text{---} (q', \alpha)$$

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:

$$\vdash \text{---}^k \text{---}$$

El lenguaje definido por un AF podemos expresarlo como:

$$L(AF) = \{ t / t \in Te^*, (q_0, t) \vdash \text{---}^* (q_i, \lambda), q_i \in F \}$$

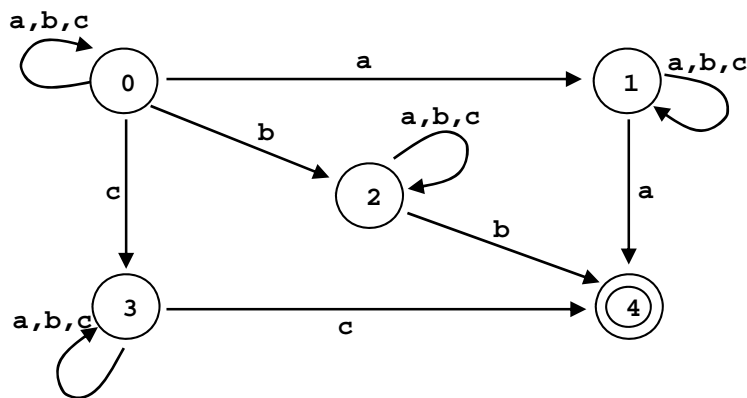
Ejemplo.-

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

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). 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 denota por una expresión regular.

Estos tres teoremas los podemos resumir de la siguiente manera:

- 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:

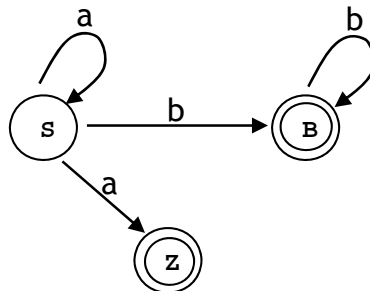
- i) $Q = N \cup \{Z\}$, $Z \notin N$ si P contiene alguna regla de la forma $A \rightarrow a$
 $Q = N$ en otro caso (cuando sólo tenemos reglas de la forma $A \rightarrow \lambda$)
- ii) $\delta(A, a) = B$ si $A \rightarrow aB \in P$
 $\delta(A, a) = Z$ si $A \rightarrow a \in P$
- iii) $F = \{A / A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



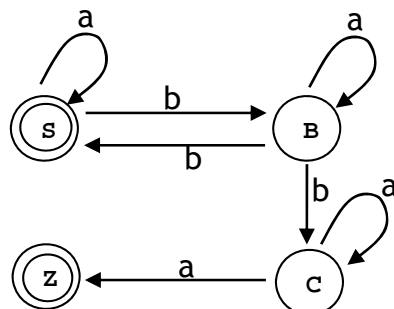
Ejemplo 2.- Sea la gramática:

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

$$B \rightarrow aB \mid bS \mid bC$$

$$C \rightarrow aC \mid a$$

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

Propiedades:

$r \mid s$	\equiv	$s \mid r$	Conmutativa “ ”
$r \mid (s \mid t)$	\equiv	$(r \mid s) \mid t$	Asociativa “ ”
$(r s) t$	\equiv	$r (s t)$	Asociativa de la concatenación
$r (s \mid t)$	\equiv	$r s \mid r t$	Concatenación distributiva sobre “ ”
$(s \mid t) r$	\equiv	$s r \mid t r$	Concatenación distributiva sobre “ ”
$r \varepsilon$	\equiv	$\varepsilon r \equiv r$	ε elemento identidad para concatenación
r^*	\equiv	$(r \mid \varepsilon)^*$	
r^{**}	\equiv	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:

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

$$\text{Lógicamente: } L^i = L^{(i-1)} L$$

* 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.-

$$\Sigma = T = \text{alfabeto} = \{a, b\}$$

$$a \mid b = \{a, b\}$$

$$a^* = \{\varepsilon, a, aa, \dots\}$$

$$(a \mid b) (a \mid b) = \{aa, bb, ab, ba\}$$

$$(a \mid b)^* = (a^* b^*)^*$$

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

$$r? = r \mid \varepsilon$$

$$r^+ = rr^*$$

$$r^* = r^+ \mid \varepsilon$$

$$(r)? = L(r) \cup \{\varepsilon\}$$

$$[a, b, c] = a \mid b \mid c$$

$$[a-z] = a \mid b \mid c \mid \dots \mid z$$

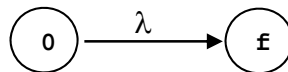
3.5 Algoritmo de Thompson.

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

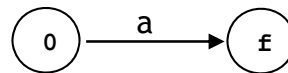
Un AFND- λ es un AFND al que incorporamos a Te un elemento más, el λ , con lo cual: $Q \times Te \cup \{\lambda\} \rightarrow Q$. Es decir, se puede mover de estado si usar entrada.

Los pasos a seguir son los siguientes:

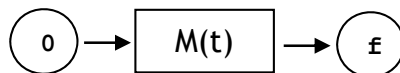
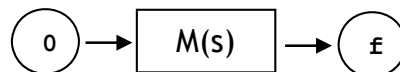
i) Para λ construiríamos:



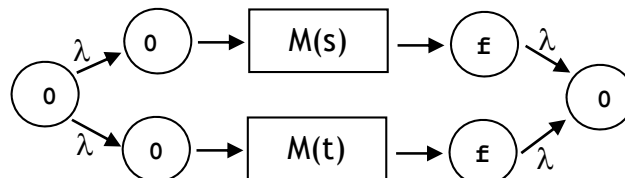
ii) $\forall a \in Te$ construiríamos:



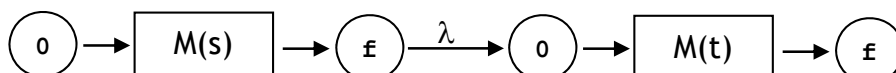
iii) Tenemos $M(s)$, $M(t)$, dos AFND para 2 expresiones regulares s y t , de la forma:



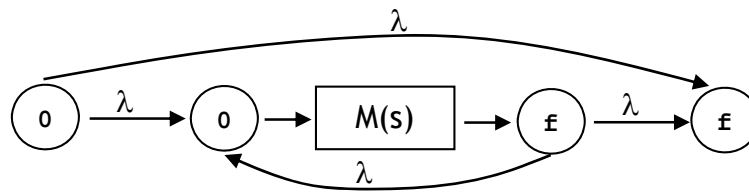
Entonces $s \mid t$ podemos reconocerlo con un AFND- λ de la forma:



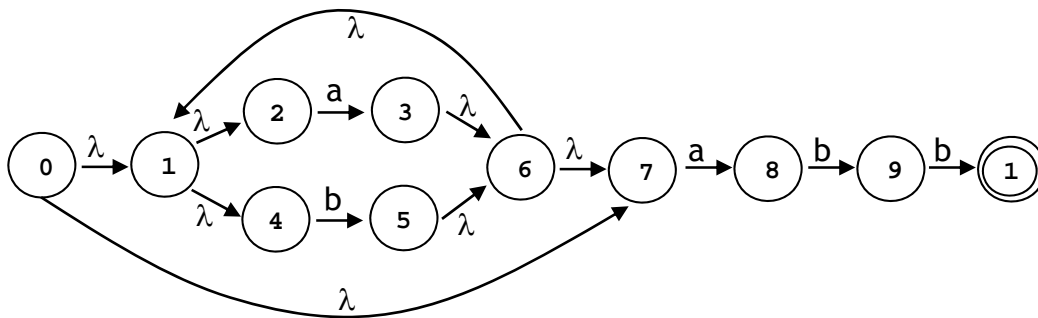
Entonces st podemos reconocerlo con un AFND- λ de la forma:



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



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



3.6 Transformación de un AFND- λ en un AFD.

Primeramente definimos:

Si S es un estado:

$$\text{Cerradura-}\Sigma(S) = \{ S_k \in \text{AFND} / S \xrightarrow{\lambda} S_k \}$$

Siendo W un conjunto de estados:

$$\text{Cerradura-}\Sigma(W) = \{ S_k \in \text{AFND} / \exists S_j \in W, S_j \xrightarrow{\lambda} S_k \}$$

Y la función de movimiento (llamémosle Mueve):

$$\text{Mueve}(W, a) = \{ S_k \in \text{AFND} / \exists S_j \in W, S_j \xrightarrow{a} S_k \}$$

El mecanismo para realizar la transformación consiste en:

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

Algoritmo

BEGIN

 Calcular Cerradura- $\Sigma(0) = \text{Estado A};$

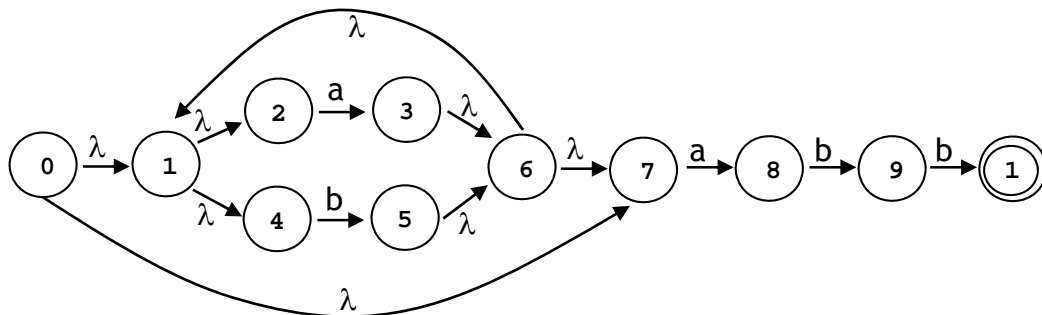
```

Incluir A en NuevosEstados;
WHILE no están todos los W de NuevosEstados marcados DO BEGIN
  Marcar W;
  FOR cada  $a_i \in T_e$  DO BEGIN
     $X = \text{Cerradura-}\Sigma (\text{Mueve}(W, a_i));$ 
    IF X no está en el conjunto NuevosEstados añadirlo;
    Transición  $[W, a] = X;$ 
  END
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 $r = (a \mid b)^* abb$, convertirlo en un AFD.



$\text{Cerradura-}\Sigma (0) = \{0, 1, 7, 4, 2\} = A$

$\text{Mueve}(A, a) = \{3, 8\}$

$\text{Mueve}(A, b) = \{5\}$

$\text{Cerradura-}\Sigma (\text{Mueve}(A, a)) = \{3, 6, 7, 1, 2, 4, 8\} = B$ (A con a mueve a B)

$\text{Cerradura-}\Sigma (\text{Mueve}(A, b)) = \{5, 6, 7, 1, 2, 4\} = C$ (A con b mueve a C)

$\text{Mueve}(B, a) = \{3, 8\} = \text{Mueve}(A, a)$ (B con a mueve a B)

$\text{Mueve}(B, b) = \{5, 9\}$

$\text{Mueve}(C, a) = \{3, 8\} = \text{Mueve}(A, a)$ (C con a mueve a B)

$\text{Mueve}(C, b) = \{5\} = \text{Mueve}(A, b)$ (C con b mueve a C)

$\text{Cerradura-}\Sigma (\text{Mueve}(B, b)) = \{5, 6, 7, 1, 2, 4, 9\} = D$ (B con b mueve a D)

$\text{Mueve}(D, a) = \{3, 8\} = \text{Mueve}(A, a)$ (D con a mueve a B)

$\text{Mueve}(D, b) = \{5, 10\}$

$\text{Cerradura-}\Sigma (\text{Mueve}(D, b)) = \{5, 6, 7, 1, 2, 4, 10\} = E$ (D con b mueve a E)

$\text{Mueve}(E, a) = \{3, 8\} = \text{Mueve}(B, a)$ (E con a mueve a B)

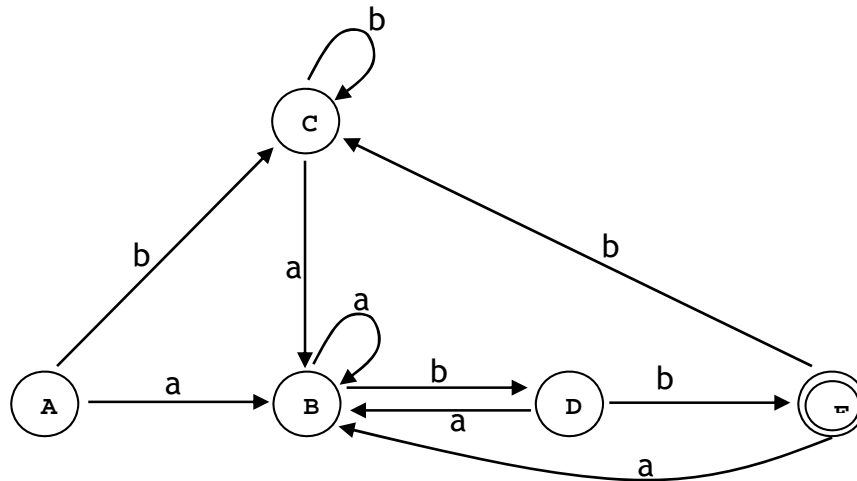
$\text{Mueve}(E, b) = \{5\} = \text{Mueve}(C, b)$ (E con b mueve a C)

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:

$$TF = (Q, Te, Ts, \delta, q_0, F)$$

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).

$$\delta: Q \times \{Te \cup \{\lambda\}\} \rightarrow P(Q) \times Ts^*$$

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:

$$(q, a\omega, s) \vdash \dots (q', \omega, sz) \text{ cuando } \delta(q, a) = (q', z)$$

$$q, q' \in Q$$

$$\omega \in Te^*$$

$$s \in Ts^*$$

Podemos definir el conjunto de traducción como:

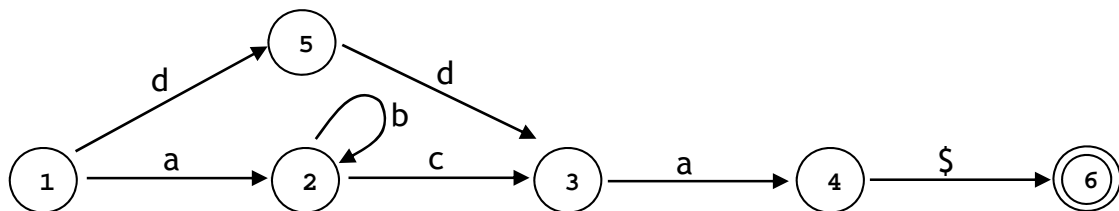
$$\text{Tr}(\text{TF}) = \{(t, s), t \in \text{Te}^*, s \in \text{Ts}^* / (q_0, t, \lambda) \vdash^* (q_f, \lambda, s), q_f \in F\}$$

Al igual que en el caso de los AF, pueden ser deterministas y no deterministas, dependiendo de si existe 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:

- i) Valor.- En él se almacenan los elementos no nulos de la tabla.
- ii) Columna.- La columna en que se encuentra el elemento del vector valor.
- iii) 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.
- iv) NumFil.- Almacenamos el número de elementos no nulos que hay en cada fila.

En nuestro ejemplo sería:

Valor
2
5

Columna
1
4

PreFil
1
3

NumFil
2
2

2
3
4
6
3

2
3
1
5
4

5
6
7
0

1
1
1
0

El espacio ocupado viene dado por: $(n^\circ \text{ de filas} \times 2) + (n^\circ \text{ no nulos} \times 2)$. En nuestro caso sería $(6 \times 2) + (7 \times 2) = 26$.

En el caso de este ejemplo no parece un ahorro de espacio muy sustancial, pero supongamos una tabla de tamaño 100×100 . 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;
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 -)

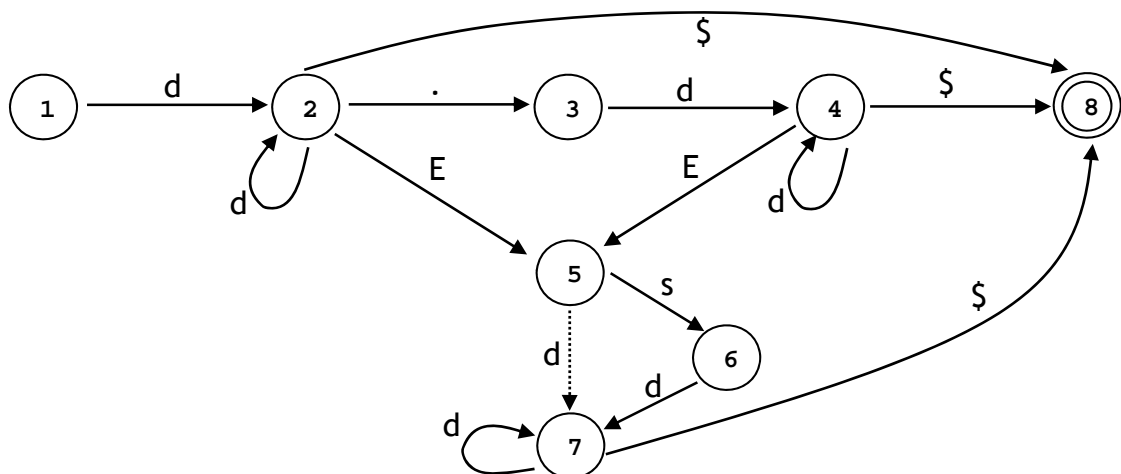
<entero_exponente> → dígito <resto_entero_exponente> (10 reglas)

<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 incluir 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: $\text{num} = 10 * \text{num} + \text{dígito}$ (calculamos número)
 U: $n = n + 1$ (calculamos fracción)
 $\text{num} = 10 * \text{num} + \text{dígito}$
 V: $\text{exp} = 10 * \text{exp} + \text{dígito}$ (calculamos exponente)

El cálculo final se realizaría en Z, que sería:

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

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 tira pertenece al lenguaje, pasando a la siguiente fase; si no lo encontramos quiere decir que la tira 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.

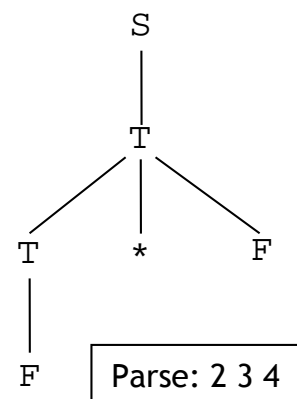
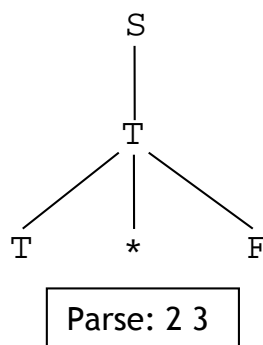
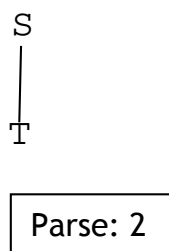
Ejemplo.-

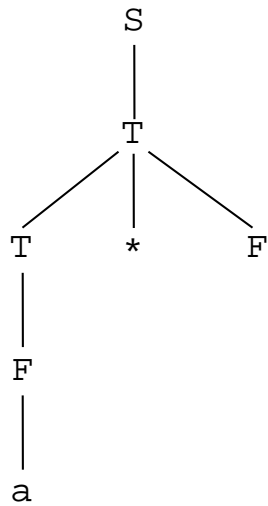
Sea la gramática:

- 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. $F \rightarrow b$

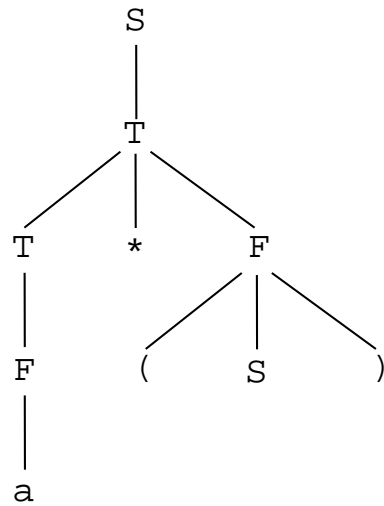
Vamos a realizar el parsing descendente de la cadena "a*(a+b)" por la izquierda y por la derecha:

Parsing descendente por la izquierda

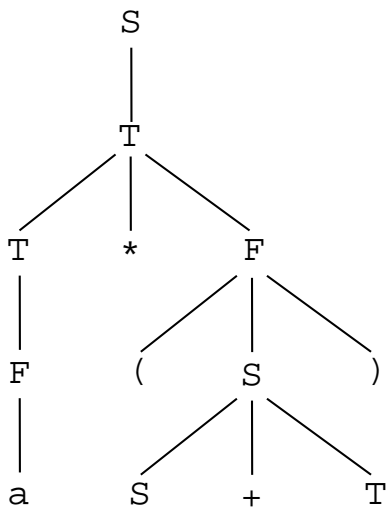




Parse: 2 3 4 6

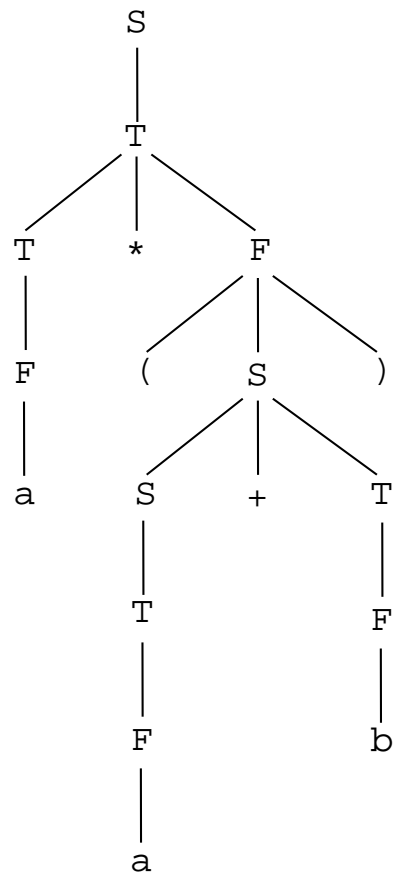


Parse: 2 3 4 6 5



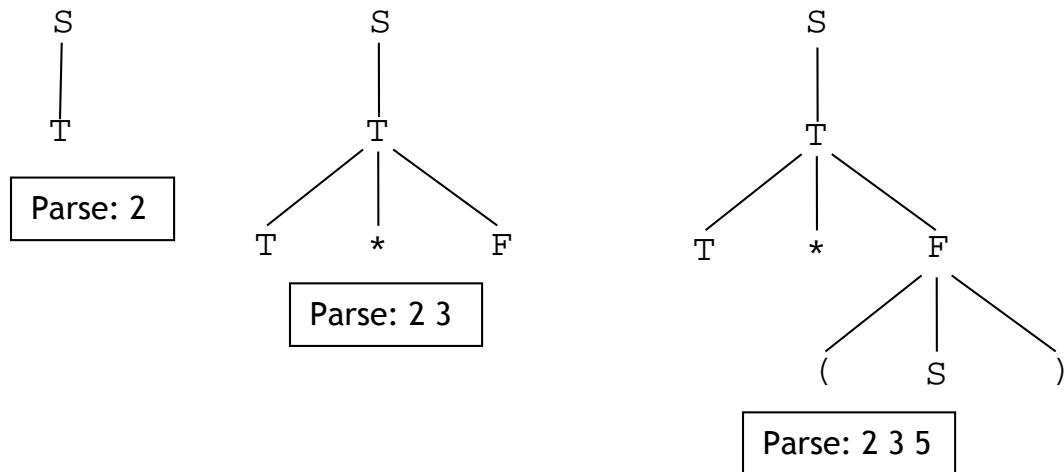
Parse: 2 3 4 6 5 1

...

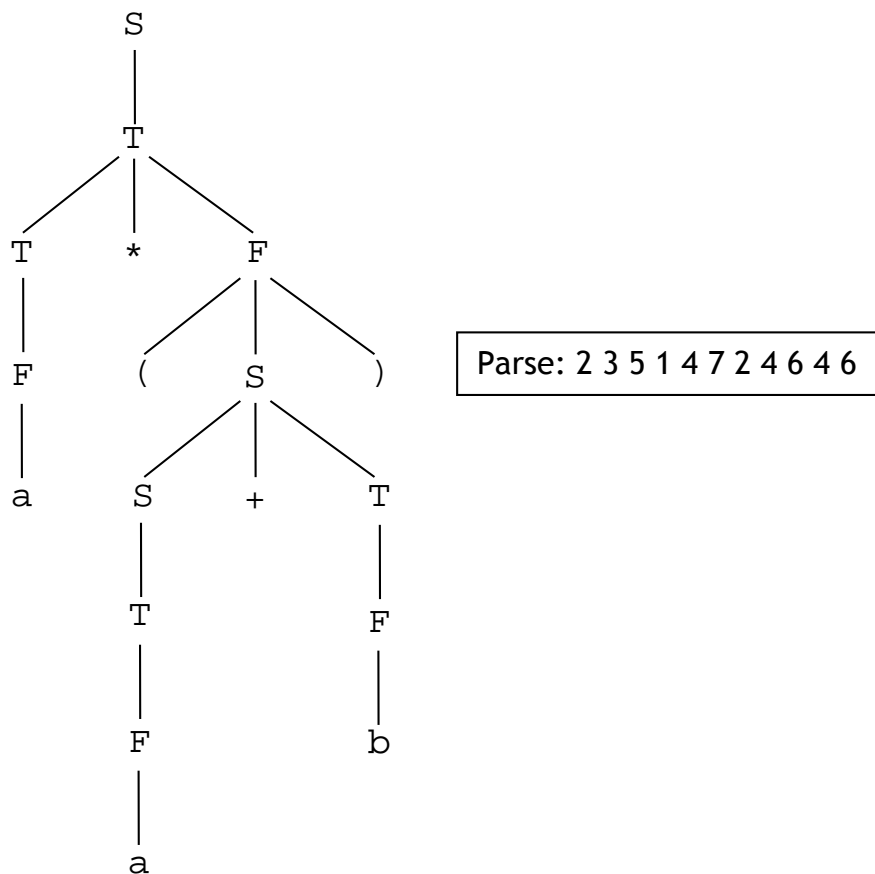


Parse: 2 3 4 6 5 1 2 4 6 4 7

Parsing descendente por la derecha



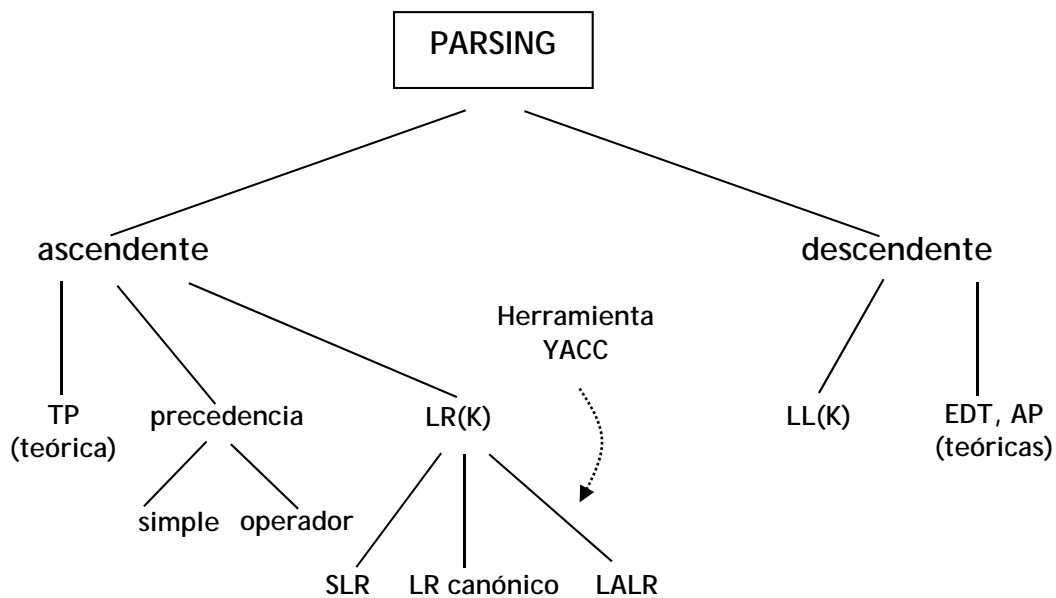
Así sucesivamente hasta llegar al árbol final (con un parse distinto):



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 sola imagen en un APD.

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

$$(q, \omega, \alpha) \in Q \times T_e^* \times T_p^*$$

q es el estado en que está el autómata ($q \in Q$)

ω es la cadena de caracteres que queda por analizar ($\omega \in T_e^*$)

α es la cadena de caracteres que hay en la pila ($\alpha \in T_p^*$)

Un movimiento podemos representarlo de la siguiente forma:

$$(q, a\omega, z\alpha) \vdash \text{----} (q', \omega, \beta\alpha) \text{ siendo } (q', \beta) \in \delta(q, a, z)$$

Si $\delta(q, a, z) = \{(p_1, v_1), (p_2, v_2), \dots (p_n, v_n)\}$ será no determinista.

Si $\delta(q, \lambda, z) = \{(p_1, v_1), (p_2, v_2), \dots (p_n, v_n)\}$ 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 empleando movimientos válidos.

$$(q_0, t, z_0) \vdash \text{--*--} (q_f, \lambda, \alpha) \text{ con } q_f \in F, \alpha \in T_p^*$$

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:

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

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:

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

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.

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

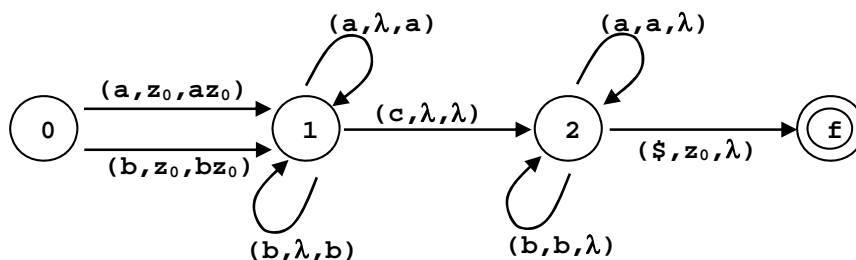
$$T = \{a, b, c\}$$

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

Elementos válidos del lenguaje serían: $abcba$, $abbcbbba$, $ababcbaba$, 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 | $\delta(q_2, \$, z_0) = (q_f, \lambda)$ | |

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



Ejemplo.-

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

$$T = \{a, b\}$$

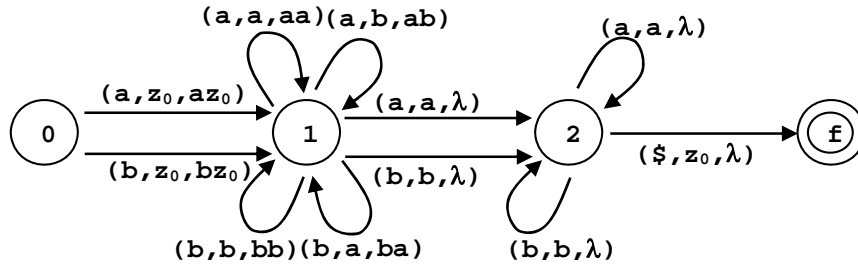
- | | |
|---|---|
| 1 | $\delta(q_0, a, z_0) = (q_1, az_0)$ |
| 2 | $\delta(q_0, b, z_0) = (q_1, bz_0)$ |
| 3 | $\delta(q_1, a, a) = \{(q_1, aa), (q_2, \lambda)\}$ |
| 4 | $\delta(q_1, a, b) = (q_1, ab)$ |
| 5 | $\delta(q_1, b, b) = \{(q_1, bb), (q_2, \lambda)\}$ |
| 6 | $\delta(q_1, b, a) = (q_1, ba)$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

También podríamos decir:

$$\begin{aligned} \delta(q_1, a, a) &= (q_2, \lambda) \\ \delta(q_1, a, \lambda) &= (q_1, a) \\ \delta(q_1, b, b) &= (q_2, \lambda) \\ \delta(q_1, b, \lambda) &= (q_1, b) \end{aligned}$$

- 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) \rightarrow $(1, bba$, az_0) \rightarrow $(1, ba$, baz_0) \rightarrow (a) y (b)
 (a) \rightarrow $(1, a$, $bbaz_0$) \rightarrow $(1, $, $abbaz_0$)$ No aceptamos.
 (b) \rightarrow $(2, a$, az_0) \rightarrow $(2, $, z_0) \rightarrow (f, λ, λ) Aceptamos.$$$$$$

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 construir 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:

$$\begin{aligned} S &\rightarrow S+T \mid T \\ T &\rightarrow T*F \mid F \\ F &\rightarrow (S) \mid a \mid b \end{aligned}$$

Crearemos un autómata con pila de la siguiente forma:

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

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

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

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 $N \cap (Ts \cup Te) = \emptyset$

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



La gramática de salida $G_s = (N, Ts, P, S)$, 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:

$(\forall A\mu, \gamma A\sigma) \Rightarrow (\forall \alpha\mu, \gamma\beta\sigma)$ 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:

$$\text{Tr (EDT)} = \{(t, s) / (S, S) \Rightarrow^+ (t, s), t \in T_e^*, s \in T_s^*\}$$

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

Ejemplo1.- Notación polaca inversa.

$$G_e = (\{S\}, \{a, b, *, /, (,)\}, P, S)$$

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:

$$S \rightarrow (S)$$

$$S \rightarrow a$$

$$S \rightarrow b$$

$$S \rightarrow S^*S$$

$$S \rightarrow S/S$$

Reglas de R:

$$S \rightarrow (S), S$$

$$S \rightarrow a, a$$

$$S \rightarrow b, b$$

$$S \rightarrow S^*S, SS^*$$

$$S \rightarrow S/S, SS/$$

Ya tenemos un EDT = (N, T_e , T_s , R, S) en el que:

$$N = \{S\}$$

$$T_e = \{a, b, *, /, (,)\}$$

$$T_s = \{a, b, *, /\}$$

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

$$\begin{aligned} (S, S) &\Rightarrow (S/S, SS/) \Rightarrow (a/S, aS/) \Rightarrow (a/(S), aS/) \Rightarrow (a/(S^*S), aSS^*/) \Rightarrow \\ &\Rightarrow (a/(a^*S), aaS^*/) \Rightarrow (a/(a^*b), aab^*/) \end{aligned}$$

Ejemplo2.-

$$T = \{a, b, c, (,), *, +\}$$

$$\text{EDT} = (N, T, \langle 1, 2, \dots, n \rangle, R, S)$$

$$S \rightarrow S + T, 1ST$$

$$S \rightarrow T, 2T$$

$$T \rightarrow T * F, 3TF$$

$$T \rightarrow F, 4F$$

$$F \rightarrow (S), 5S$$

$$F \rightarrow a, 6$$

$F \rightarrow b, 7$

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

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

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).

$\delta: Q \times \{Te \cup \{\lambda\}\} \times \{Tp \cup \{\lambda\}\} \rightarrow P(Q) \times Tp^* \times Ts^*$

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:

$(q, ax, zv, y) \vdash \dots (q', x, \alpha v, yz')$ con $q, q' \in Q, x \in Te^*, v \in Tp^*, y \in Ts^*$

El conjunto de traducción podemos definirlo como:

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

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

$Tr(TP) = \{(t, s) / (q_0, t, z_0, \lambda) \vdash^* (q', \lambda, \lambda, s)\}$

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

Ejemplo.-

$T = \{ a, b, (,), *, + \}$

$TP = (\{q\}, T, N \cup T, <1, 2, \dots, 7>, \delta, q, z_0, q)$

$S \rightarrow S + T \quad \delta(q, \lambda, S+T) = (q, S, 1)$

$S \rightarrow T \quad \delta(q, \lambda, T) = (q, S, 2)$

$T \rightarrow T * F \quad \delta(q, \lambda, T*F) = (q, T, 3)$

$T \rightarrow F \quad \delta(q, \lambda, F) = (q, T, 4)$

$F \rightarrow (S) \quad \delta(q, \lambda, (S)) = (q, F, 5)$

$F \rightarrow a \quad \delta(q, \lambda, a) = (q, F, 6)$

$F \rightarrow b \quad \delta(q, \lambda, b) = (q, F, 7)$

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 las relaciones $<$, $>$, \pm , relaciones que no son de equivalencia.

Decimos que una relación R es una relación de equivalencia cuando:

- i) Reflexiva. $x R x \quad \forall x \in R$
- ii) Simétrica. $x R y \Rightarrow y R x$
- iii) Transitiva. $x R y, y R z \Rightarrow x R z$

Ejemplo.-

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

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

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

$x \backslash y$	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

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

Definición: Relación traspuesta (R^{\sim} o R^T)

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

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

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

Dadas dos relaciones R y S sobre A :

$$R + S = \{ (x, y) / (x R y) \vee (x S y) \} \quad (\text{OR})$$

$$R \times S = \{ (x, y) / (x R y) \wedge (x S y) \} \quad (\text{AND})$$

$$R + S = M_{ij} \text{ OR } M_{kp}$$

$$R \times S = M_{ij} \text{ AND } M_{kp}$$

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

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.

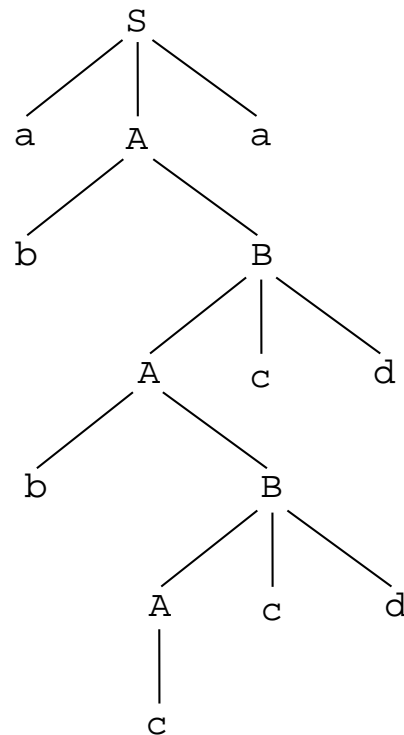
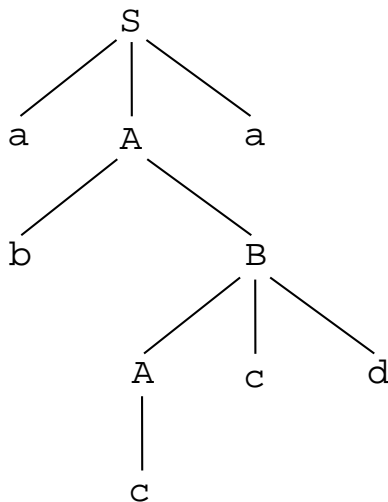
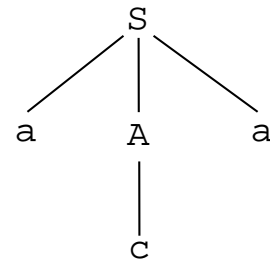
Por ejemplo, supongamos la gramática:

$S \rightarrow aAa$ Algunos árboles serían los siguientes:

$A \rightarrow bB$

$A \rightarrow c$

$B \rightarrow Acd$



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

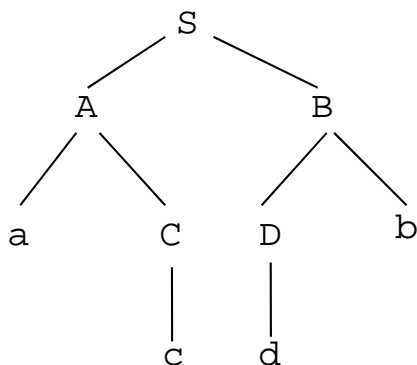
Llamamos pivote a un conjunto de símbolos que se pueden sustituir por las metanociones de la regla correspondiente (son símbolos situados entre $<$ y $>$ con los intermedios relacionados con \pm). La tira completa la rodeamos por izquierda y derecha pcon $<$ y $>$. Utilizando el concepto de pivote, vamos 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
$\langle a < b < c > c \pm d \rangle a \rangle$	c	$A \rightarrow c$
$\langle a < b < A \pm c \pm d \rangle a \rangle$	Acd	$B \rightarrow Acd$
$\langle a < b \pm B \rangle a \rangle$	bB	$A \rightarrow bB$
$\langle a \pm A \pm a \rangle$	aAa	$S \rightarrow aAa$
$\langle S \rangle$		ÉXITO, ACEPTADA

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 $A \rightarrow \alpha x B \beta \in P$ y $B \rightarrow^+ y \nu$
- ii) $x \pm y$ si y sólo si $A \rightarrow \alpha x y \beta \in P$
- iii) $x > y$ si y sólo si $A \rightarrow \alpha C D \beta$ con $C \rightarrow^+ \nu x$ y $D \rightarrow^* y \mu$



En el centro, 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 $>$.

Relación PRIMERO: Diremos que $X \in \text{PRIMERO}(A)$ si y sólo si $\exists A \rightarrow X\alpha\beta$

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

En relación a PRIMERO^+ , se puede decir que $\text{Cab}(A) = \{ X / X \in \text{PRIMERO}^+(A) \}$

$X \in \text{PRIMERO}^+(A)$ si y sólo si $\exists A \rightarrow^+ X\alpha\beta$

$X \in \text{ULTIMO}(A)$ si y sólo si $\exists A \rightarrow \alpha\beta X$

$X \in \text{ULTIMO}^+(A)$ si y sólo si $\exists A \rightarrow^+ \alpha\beta X$

$X \in \text{DENTRO}(A)$ si y sólo si $\exists A \rightarrow \alpha X \beta$

$X \in \text{DENTRO}^+(A)$ si y sólo si $\exists A \rightarrow^+ \alpha X \beta$

Representándolo en forma de matriz, “los A” serían las filas y “los X” las columnas.

Ejemplo.-

$A \rightarrow Aa \mid B$

$B \rightarrow DbC \mid Dc$

$C \rightarrow c$

$D \rightarrow Ba$

$\text{PRIMERO}^+ = \{ (A, A), (A, B), (A, D), (B, D), (B, B), (C, c), (D, B), (D, D) \}$

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 sí 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 ; $A^+ = \bigcup_{i>0} A^i$ a partir de la relación R en forma de una matriz.

$$B = A$$


```

3      I = 1
      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 (\pm) , $(<)$ y $(>)$

(\pm) se calcula por la propia definición.

$(<) = (\pm) (\text{PRIMERO}^+)$

$(>) = (\text{ULTIMO}^+)^T (\pm) (I + \text{PRIMERO}^+)$, 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 elimináramos la regla λ existente en la gramática.

Vamos a calcular las matrices (\pm) y PRIMERO:

\pm	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:

$$(\pm) = \begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & \mathbf{1^*} & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \quad \text{PRIMERO} = \begin{vmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Vamos a hacer el producto relativo con PRIMERO:

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

Entonces PRIMERO⁺ = PRIMERO

Vamos a hacer el producto relativo con ULTIMO:

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

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

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

Ahora ya tenemos las matrices PRIMERO⁺ y ULTIMO⁺:

$$\text{PRIMERO}^+ = \begin{vmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \quad \text{ULTIMO}^+ = \begin{vmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Ya podemos calcular (<) y (>):

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

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

$$(>) = \begin{vmatrix} \begin{matrix} (\clubsuit) \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix} & \begin{matrix} (\text{I}+\text{PRIMERO}^+) \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{vmatrix}$$

La tabla final de parsing será:

	S	B	a	b
S		\pm		<
B		>		>
a	\pm	\pm^*	<	<*
b		>		>

Si en esta tabla eliminamos los símbolos \pm^* y $<^*$ (que sería la tabla resultante si no elimináramos 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.-

$S \rightarrow AB$

$B \rightarrow A$

$A \rightarrow a$

\pm	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:

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

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⁺:

$$\text{PRIMERO}^+ = \begin{vmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{vmatrix} \quad \text{ULTIMO}^+ = \begin{vmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Ya podemos calcular (<) y (>):

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

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

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

La tabla final de parsing será:

	S	A	B	a
S				
A		<	\pm	<
B				
a		>	>	>

Ejercicio 3.-

$S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d$

\pm	S	C	c	d	PRIMERO	S	C	c	d	ULTIMO	S	C	c	d
S	0	0	0	0	S	0	1	0	0	S	0	1	0	0
C	0	1	0	0	C	0	0	1	1	C	0	1	0	1
c	0	1	0	0	c	0	0	0	0	c	0	0	0	0
d	0	0	0	0	d	0	0	0	0	d	0	0	0	0

Por lo tanto:

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

Vamos a hacer el producto relativo con PRIMERO:

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

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

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

Vamos a hacer el producto relativo con ULTIMO:

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

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⁺:

$$\text{PRIMERO}^+ = \begin{vmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} \quad \text{ULTIMO}^+ = \begin{vmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Ya podemos calcular (<) y (>):

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

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

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

La tabla final de parsing será:

	S	C	c	d
S				
C		>	>	>
		±	<	<
c		±	<	<
d		>	>	>

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