Procesamiento de Lenguajes

Grado en Ingeniería Informática
4º curso

Profesores: **Carlos Dafonte**
Bernardino Arcay
Ángel Gómez

Departamento de Computación e Tecnoloxías da Información
Universidade da Coruña
Curso 2024/2025 Rev.20242311

ÍNDICE

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

Sanchís, 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

Hopcroff, 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, con un enfoque muy práctico. El primero de ellos es la base de la asignatura aunque hay conceptos que no están explicados de manera sencilla. Los dos últimos están dedicados a la parte de teoría de la computación (automátas y gramáticas).

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).

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

- Desarrollo de traductores y compiladores.
- Crear procesadores de texto estructurado.
- Manejar datos estructurados (ej.- XML, HTML, JSON).
- Diseño de interfaces hombre-máquina.
- Recuperación de información (ej.- herramientas de *web scraping).*
- Asegurar comunicaciones (ej.- analizar URLs).
- Etc.

Tokens: son 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.

Traductor: Programa que convierte un un texto escrito en un lenguaje determinado (fuente) en otro diferente (objetivo). Puede tratarse, en ambos casos, de un lenguaje de programación.

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).

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).

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

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

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

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

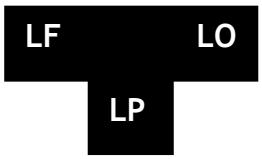
Traductor o compilador incremental, interactivo o conversacional: Es un tipo de compilador que, al detectar un error, intenta compilar el código del entorno en donde está el error, no todo el programa de nuevo.

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

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

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

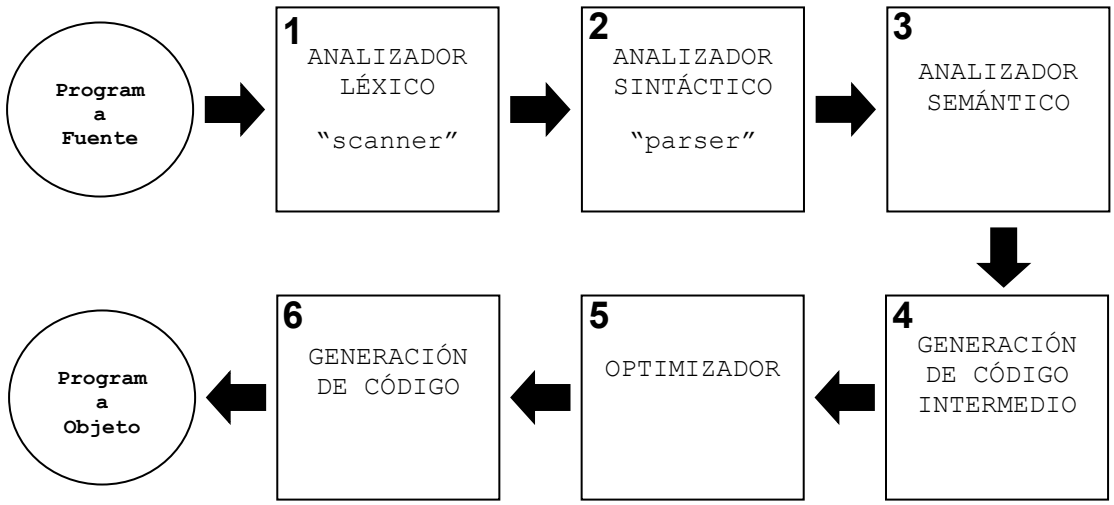
LF -> Lenguaje fuente
LO -> Lenguaje objeto
LP -> Lenguaje del procesador de lenguaje



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

1.1 Estructura de un procesador del lenguaje.

De manera general, aquí presentamos las fases de un compilador. Dependiendo del objetivo del procesador de lenguaje que pretendemos desarrollar y del lenguaje que se quiere analizar, no siempre son necesarias todas las fases. Por ejemplo, si nuestro objetivo es extraer ciertos datos de un archivo XML hacia un formato más sencillo como CSV, se implementará una fase de síntesis mucho menos compleja, y lo mismo ocurre con el analizador semántico o el manejo de errores.



Es importante comentar que contaremos además con la ***Tabla de Símbolos***, dónde recopilaremos la información sobre los elementos del programa fuente.

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

Etapa 1. Analizador léxico o "scanner".
En esta etapa, es preciso mantener separados cada uno de los tokens y almacenarlos en una *tabla de símbolos* (objetos o variables). Aquí ya se podría detectar algún error, por ejemplo, poner FAR no correspondería a una palabra del lenguaje. No sólo en esta etapa se utiliza la tabla de símbolos ya que la información ahí contenida se va completando durante las siguientes fases; por ejemplo, también almacenaremos el tipo establecido en su definición.

Etapa 2. Analizador sintáctico o "parser".
Se pretende ver la estructura de la frase, ver si los elementos tienen estructura de frase del lenguaje.

Etapa 3. Analizador semántico.
Se analiza si la frase encontrada tiene significado. Utilizará la *Tabla de Símbolos* para conocer los tipos de las variables y poder estudiar el significado semántico de la oración. Por ejemplo, si tenemos:

```
a = b + c
```

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

Etapa 4. Generación de código intermedio.
Traducimos 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 y puede ser válido para varios compiladores distintos, con el consecuente ahorro económico).

Etapa 5. Optimizador.
Intenta optimizar el funcionamiento del programa compilado en cuanto a variables utilizadas, bucles, saltos en memoria, tamaño, etc.

Etapa 6. Generación de código.
Construcción 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. Puede ser código de una MV, como ocurre en Java.

Ejemplo de las fases de un procesador del lenguaje:

Supongamos un compilador que procesa una instrucción de la siguiente forma:

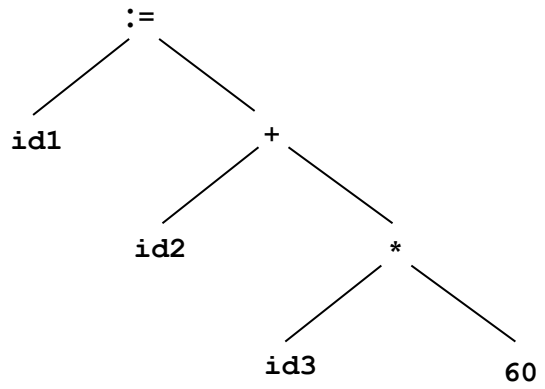
```
posicion := inicial + velocidad * 60
```

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

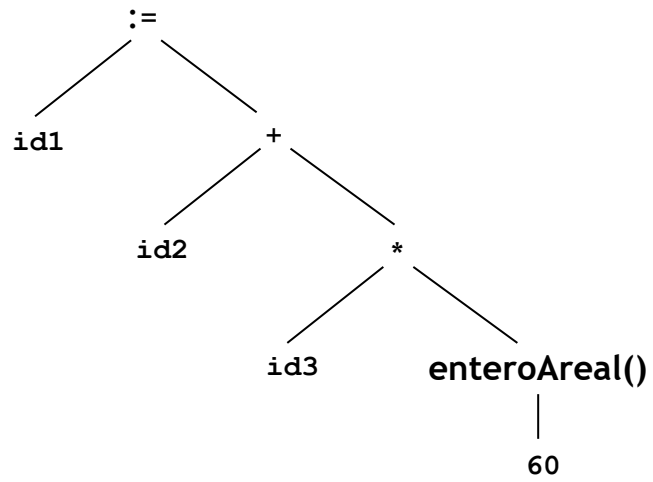
ANALIZADOR LÉXICO:

```
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:

```
MOV id3, R2          ADD R2, R1
MULT #60.0, R2       MOV R1, id1
MOV id2, R1
```

2 LENGUAJES Y GRAMÁTICAS.

Definiciones:

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

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

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

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

Ejemplo.- x = abc |x| = 3
|λ| = 0

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

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

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

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

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

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

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

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

Definimos **Potencia** $A^n = A.A^{n-1}$; $A^0=\{\lambda\}$; $A^1 = A$
Definimos **Cierre Transitivo** $A^+ = U_{i>0} A^i$
Definimos **Cierre Transitivo y Reflexivo** $A^* = U_{i\geq 0} A^i$
A partir de estas dos últimas tenemos que: $A^* = A^+ \cup A^0 = A^+ \cup \{\lambda\}$
Definición: Un lenguaje estará formado por dos elementos: un conjunto de reglas y un diccionario (que en nuestro caso será la *tabla de símbolos*).

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

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

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

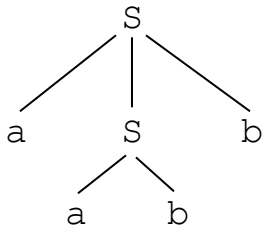
2.1 Notación de Chomsky.

Esta notación de 1959 define una gramática como una tupla (**N**, **T**, **P**, **S**)
N es el conjunto de los no terminales (conjunto de símbolos que introducimos en la gramática como elementos auxiliares que no forman parte de las sentencias del lenguaje).
T es el conjunto de los terminales (son los símbolos o caracteres que forman las sentencias del lenguaje).
P es el conjunto de reglas de producción (reglas de derivación de las tiras).
S es el axioma de la gramática, S está incluido en N (es el símbolo inicial o metanoción).

Ejemplo.-

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

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



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

2.2 Clasificación de Chomsky.

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

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

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

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

b) Gramáticas tipo 1, gramáticas sensibles al contexto o dependientes del contexto. Reconocibles con un *Autómata Linealmente Acotado* (LBA).

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

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

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

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

c) Gramáticas tipo 2, gramáticas de contexto libre (GCL) o independientes del contexto (GIC). Implementables con un *Autómata con Pila*.

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

Es el tipo utilizado, a nivel sintáctico, por los lenguajes de alto nivel.

d) Gramáticas tipo 3, gramáticas regulares. Implementables con un *Autómata Finito* que, como veremos, es equivalente también a una *Expresión Regular*.

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

Las gramáticas tipo 3 son las más restrictivas y las tipo 0 las menos, pero existe una peculiaridad porque, de forma teórica 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 (aunque luego veremos que se pueden eliminar de la gramática, siendo el mismo lenguaje).

También existe otra diferencia entre las dos definiciones mostradas de gramáticas tipo 1, por ejemplo.-

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

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

2.3 Gramáticas de contexto libre (GCL).

Recordemos la definición:

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

Derivación: $\alpha \Rightarrow \beta$ (se lee α deriva β), α y β son dos tiras, si se puede escribir:
$\alpha = \delta A \mu$, $\beta = \delta \nu \mu$, δ y μ son dos tiras y aplicamos $A \rightarrow \nu$
δ y μ pueden ser tiras nulas, con lo cual sería una derivación directa.

Una derivación de longitud n, de la forma:

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

se representará como:

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

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

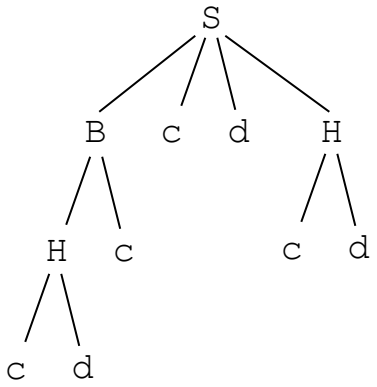
Sentencias, con G = (N, T, P, S), son el conjunto de tiras de la siguiente forma, $L(G) = \{ \omega / S \Rightarrow^* \omega, \omega \in T^* \}$.

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

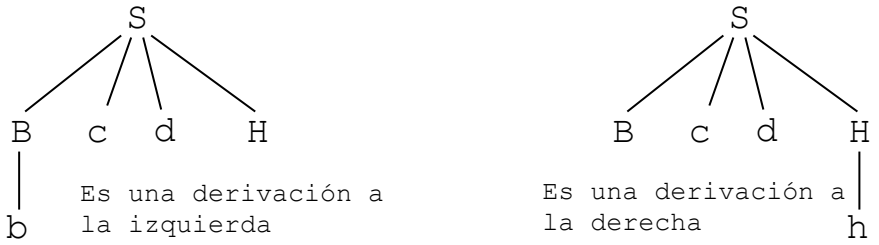


Derivación a la izquierda: es la derivación que se realiza sustituyendo las metanociones que se encuentran más a la izquierda de la forma sentencial.

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

Ejemplo.-

S → BcdH
B → b
H → h



2.4 Reglas mediante notación 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. Naur la bautizó como BNF con el significado de *Backus Normal Form*, pero en la bibliografía consta habitualmente como *Backus-Naur Form*.

Esta notación nos permite definir formalmente la gramática de un lenguaje, definiendo claramente qué está permitido y qué no lo está (aunque no soluciona la ambigüedad). 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, siendo muchas veces posible construir

mecánicamente un reconocedor sintáctico de un lenguaje dado en forma de gramática BNF (con algunos tipos de gramáticas esto es imposible aunque existen modificaciones que podemos hacer sobre ellas, de ello hablaremos a continuación).

La forma de una regla BNF es la que sigue:

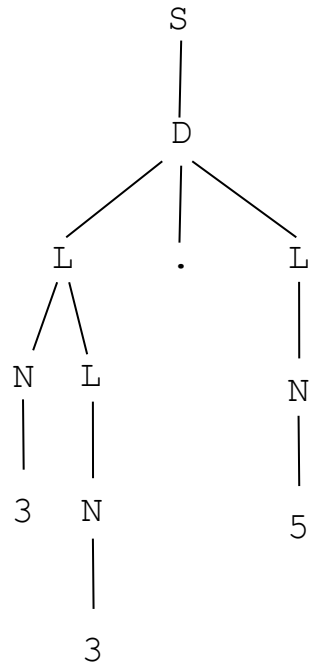
Símbolo no terminal → alternativa1 | alternativa2 ...

Ejemplo.-

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

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

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



Aquí utilizaremos una notación más simplificada (terminales en minúsculas y no terminales en mayúsculas). Estrictamente en BNF los no terminales deberían ir rodeados por <>. En Extended (EBNF), los terminales se rodean con “” y los no terminales se escriben sin utilizar <>. Tanto en BNF como en EBNF, se especificaba := en lugar de la flecha → que hoy en día más habitualmente encontraremos en gran parte de la bibliografía.

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:

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

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

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

PROBLEMA: Recursividad por la izquierda.

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

Supongamos que tenemos:

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

Esto podemos ponerlo de la siguiente forma:

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

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

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

Por ejemplo, si tenemos las reglas:

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

La transformación podría ser la siguiente:

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

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

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

Siguiendo el ejemplo anterior, la transformación sería:

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

Ejemplo 1.-

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

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

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

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

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

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

La gramática resultante es:

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

Ejemplo 2.-

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

2.5.2 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´ → B | C

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

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

S → L S´
S´→ ; LS´ | λ
L → if expr then S X fi
L → instr
X → else S | λ

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

X → XW
X → b | λ
W → aW | z
W → a

X → bX´ | X´
X´ → WX´ | λ
W → aW´ | z
W´→ W | λ

2.5.3 Ambigüedad.

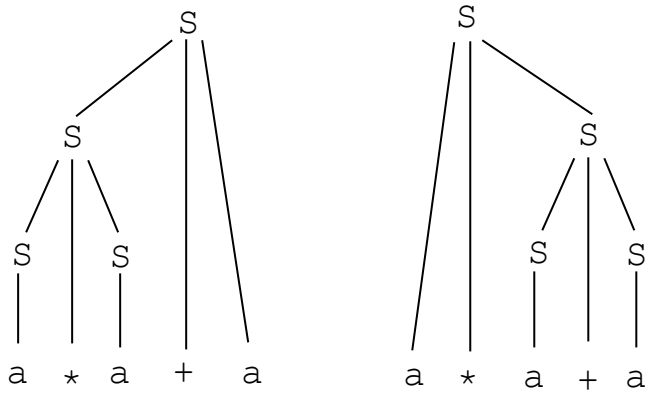
Diremos que una gramática G es ambigua si el lenguaje definido por ella tiene alguna sentencia para la cual existe más de un árbol sintáctico. Hay que tener claro que la que es ambigua es la gramática, no el lenguaje y que una gramática ambigua nos creará dificultades para implementar un analizador, aunque se puede hacer. Lo deseable es modificar las reglas de la gramática para solucionar la ambigüedad y de forma que mantengamos el mismo lenguaje.

Ejemplo.-

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

P: S → a
S → S+S
S → 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 pero nada elegante":

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 $L(G) \neq \varnothing$?

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

Ejemplo.-

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

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

2.6.2 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 no se puede eliminar completamente (debemos de mantener $S \rightarrow \lambda$).

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

TEOREMA

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

Algoritmo de detección de símbolos anulables

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

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

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

Algoritmo

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

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

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

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

Ejemplo.-

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

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

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

Y la gramática sin reglas cadena resultante sería:

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

2.6.4 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:

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

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

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

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

Algoritmo para la eliminación de símbolos no terminables

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

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

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

Algoritmo

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

Ejemplo.-

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

Algoritmo para la eliminación de símbolos no accesibles.

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

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

Algoritmo

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

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

Ejercicio.- Eliminar los símbolos inútiles en la siguiente gramática.
G = (N, T, P, S)
N = {S, A, B, C, D}

T = {a, b, c}
S → aAA
A → aAb | aC
B → BD | Ac
C → b

Aplicamos primero el algoritmo de eliminación de símbolos no terminables:

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

La nueva gramática es la siguiente:

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

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

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

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

2.7 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´ = {∅}
Añadimos a P´ las reglas del tipo $A \rightarrow a$
Añadimos a P´ las reglas del tipo $A \rightarrow BC$
IF $A \rightarrow x_1 x_2 \ldots x_k \in P$, $k>2$ THEN
BEGIN
Añadimos a P´ $A \rightarrow x_1 \text{´} < x_2 \ldots x_k >$
$< x_2 \ldots x_k > \rightarrow x_2 \text{´} < x_3 \ldots x_k >$
$< x_3 \ldots x_k > \rightarrow x_3 \text{´} < x_4 \ldots x_k >$
$< x_{k-2} > \rightarrow x_{k-1} x_k$
END
IF $A \rightarrow x_1 x_2 \in P$, si $x_1 \in T$ ó $x_2 \in T$ THEN
BEGIN
Llamamos x_a al $x_i \in T$
Sustituímos x_a por $x_a \text{´}$ en esa regla de P´
Añadimos a P´ $x_a \text{'} \rightarrow x_a$
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:

$\underline{A \rightarrow 01AB0}$	$\mathbf{A \rightarrow A_1A_2}$	$\mathbf{A_1 \rightarrow 0}$	$\underline{A_2 \rightarrow 1AB0}$	$\mathbf{A_2 \rightarrow A_3A_4}$	
$\mathbf{A_3 \rightarrow 1}$	$\underline{A_4 \rightarrow AB0}$	$\mathbf{A_4 \rightarrow AA_5}$	$\underline{A_5 \rightarrow B0}$	$\mathbf{A_5 \rightarrow BA_6}$	$\mathbf{A_6 \rightarrow 0}$

Se muestran subrayadas las reglas que vamos remplazando por no cumplir la FNC y en negrita las reglas que definitivamente quedan en la gramática.

2.9 Resumen.

A modo de resumen, 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.- Elimina reglas lambda, reglas cadena y símbolos inútiles.

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

Solución:

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

Ejercicio 2.- Elimina reglas lambda, reglas cadena y símbolos inútiles.

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

Solución:

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

Ejercicio 3.- Elimina símbolos inútiles.

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

Solución: Vacío (no quedan reglas)

3 ANÁLISIS LÉXICO (*Scanners*).

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

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

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

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

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

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

Hablaremos de dos tipos de autómatas:

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

3.1 Tipos de máquinas reconocedoras o autómatas.

a) Autómatas Finitos (AF), reconocen lenguajes regulares (Gramáticas tipo 3). Podemos representarlo como una expresión regular, a partir de ella obtener un Autómata Finito No Determinista (AFND), para luego convertirlo en un Autómata Finito Determinista (AFD).

b) Autómatas con Pila (AP), reconocen lenguajes tipo 2 (Gramáticas de contexto libre, tipo 2)
c) Autómatas bidireccionales o autómatas lineales acotados (LBA, de su denominación en inglés). 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 (MT) 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:

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

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

Para representar k pasos pondremos:

$\vdash^k$

El lenguaje definido por un AF podemos expersarlo como:

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

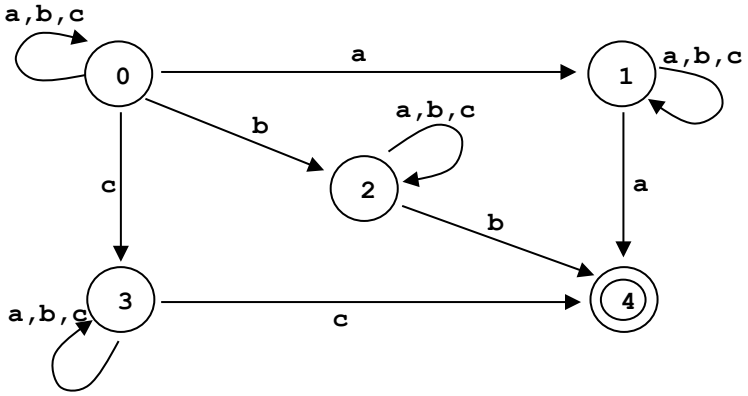
Ejemplo.-

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

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) $A \rightarrow aB$; $A \rightarrow a$; $A \rightarrow \lambda$, $a \in T$, $A, B \in N$

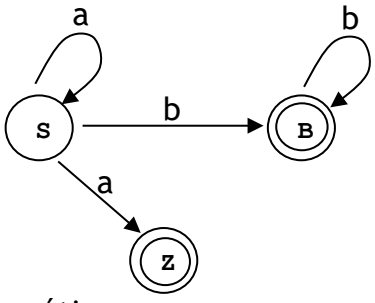
Vamos a convertirla en un autómata AFND (Q, Te, δ, q_0, F) en el cual:

i) Q = N U {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) $F = \{A / A \rightarrow \lambda \in P\}$ U Z si $Z \in Q$
   $F = \{A / A \rightarrow \lambda \in P\}$ en otro caso
iv) El estado inicial, q0, es S

Ejemplo 1.- Sea la gramática:

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

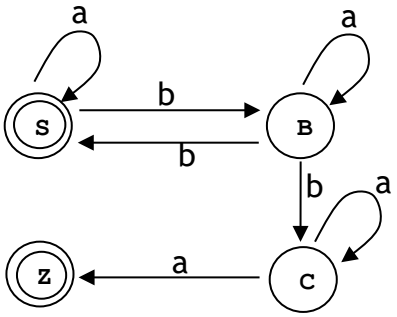
El autómata sería:



Ejemplo 2.- Sea la gramática:

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

El autómata sería:



3.4 Expresión regular.

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

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

Propiedades:

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

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

Además tenemos que:

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

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

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

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

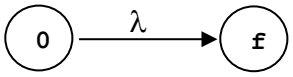
3.5 Algoritmo de Thompson.

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

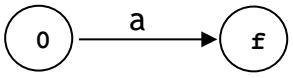
Un AFND-λ es un AFND al que incorporamos a Te un elemento más, el λ, con lo cual: Q x Te U {λ} $\rightarrow$ Q. Es decir, se puede mover de estado si usar entrada.

Los pasos a seguir son los siguientes:

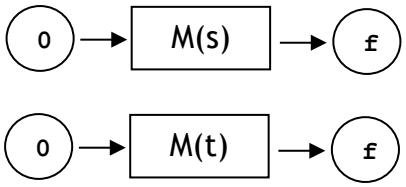
i) Para λ construiríamos:



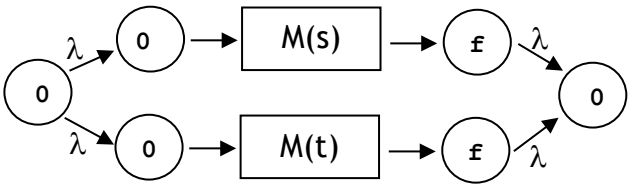
ii) $\forall$ a $\in$ Te construiríamos:



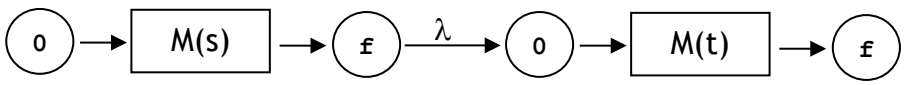
iii) Tenemos **M(s)**, **M(t)**, dos AFND para 2 expresiones regulares **s** y **t**, de la forma:



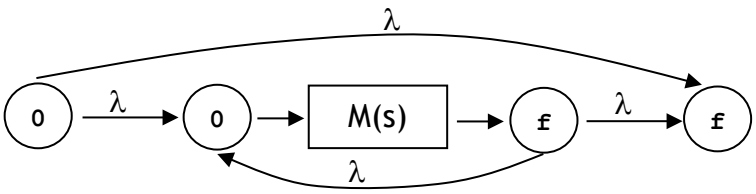
Entonces **s|t** podemos reconocerlo con un AFND-λ de la forma:



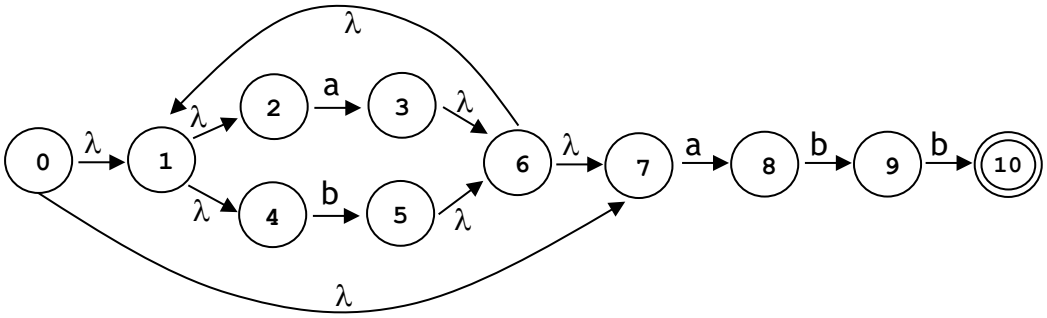
Entonces **st** podemos reconocerlo con un AFND-λ de la forma:



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



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



3.6 Transformación de un AFND-λ en un AFD.

Primeramente definimos:

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

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

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

El mecanismo para realizar la transformación consiste en:

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

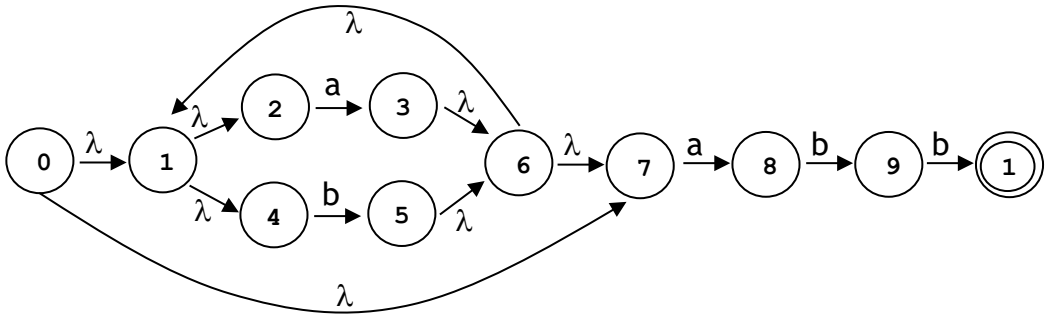
Algoritmo

BEGIN
    Calcular Cerradura-Σ (0) = Estado A;

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

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

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



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

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

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

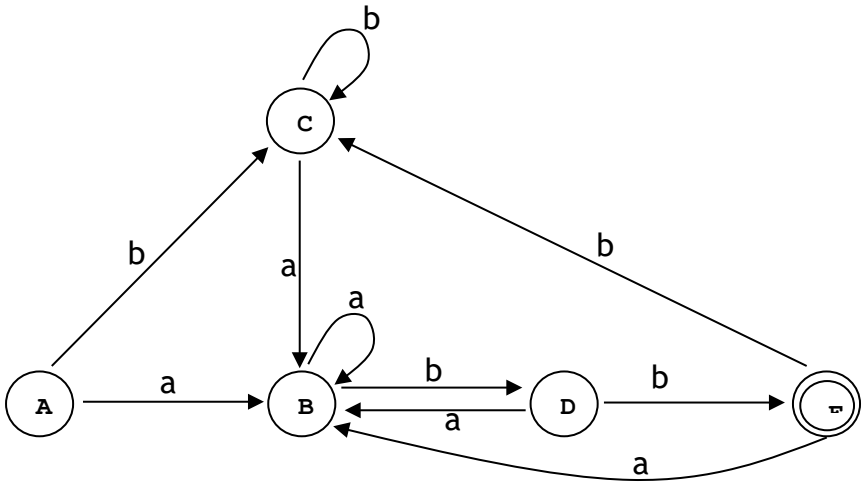
Cerradura-Σ (Mueve(D, b)) = {5, 6, 7, 1, 2, 4, 10} = E (D con b mueve a E)
Mueve (E, a) = {3, 8} = Mueve (B, a) (E con a mueve a B)
Mueve (E, b) = {5} = Mueve (C, b) (E con b mueve a C)

Tabla de transiciones:

Estado	**a**	**b**
A	B	C

B	B	D
C	B	C
D	B	E
E	B	C

Vamos a ver el autómata:



3.7 Traductores finitos (TF).

Se trata de un autómata finito que "habla" en cada transición (no confundir con una pila/memoria, pues no es modificable esa salida). Tiene la forma:

TF = (Q, Te, Ts, δ, q_0, F)

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

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

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

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

(q, aω, s) |---- (q´, ω, sz) cuando δ(q, a) = (q´, z)
q, q´ $\in$ Q
ω $\in$ Te*
s $\in$ Ts*

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

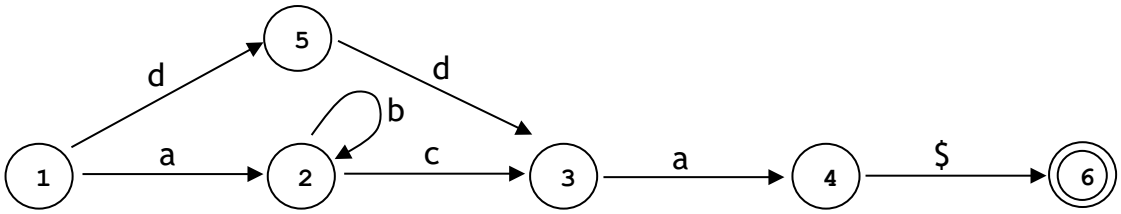
$Tr(TF) = \{(t, s), t \in Te^*, s \in Ts^* / (q_0, t, \lambda) \text{ |--*-- } (q_f, \lambda, s), q_f \in F\}$

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

3.8 Implementación de autómatas.

3.8.1 Tabla compacta.

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



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

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

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

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. Podríamos incluso no utilizarlo y calcular diferencias en PreFil.

En nuestro ejemplo sería:

Valor
2
5
2
3
4
6
3

Columna
1
4
2
3
1
5
4

PreFil
1
3
5
6
7
0

NumFil
2
2
1
1
1
0

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

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

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

Algoritmo

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

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

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

3.8.2 Autómata programado.

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

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

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

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

Reglas BNF:

<número> → <entero> | <real>

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

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

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

<signo> → + | -

<dígito → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Esta gramática podemos ponerla en forma de gramática regular, que sería la siguiente:

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

Este tipo de reglas las resumiremos como:

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

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

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

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

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

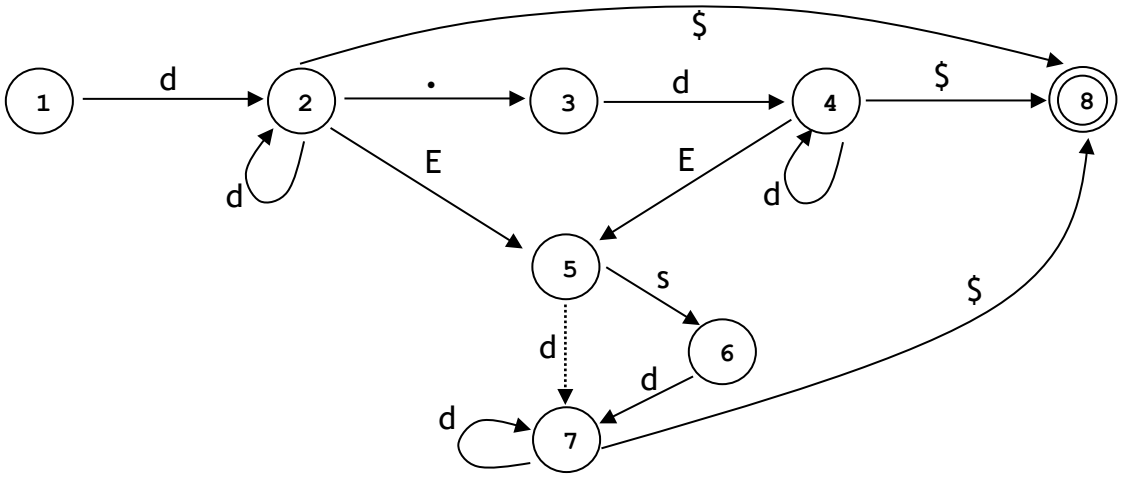
<entero_exponente> → dígito <resto_entero_exponente> (10 reglas)

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

Ahora implementaremos el autómata. Empezamos por definir los estados:

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

El autómata sería:



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

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

5	7			6	
6	7				
7	7				8

Ejemplo.- Gramática de las expresiones regulares

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

Ejemplo.- Especificaciones de tiempo en UNIX

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

nowspec = NOW | NOW increment | NOW decrement

time = hr24clock | NOON | MIDNIGHT | TEATIME

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

increment = '+' inc_number inc_period

decrement = '-' inc_number inc_period

hr24clock = INT 'h' INT

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

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

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

3.10 Acciones semánticas.

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

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

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

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

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

El cálculo final ser realizaría en Z, que sería:

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

4 ANÁLISIS SINTÁCTICO (*Parsers*).

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.

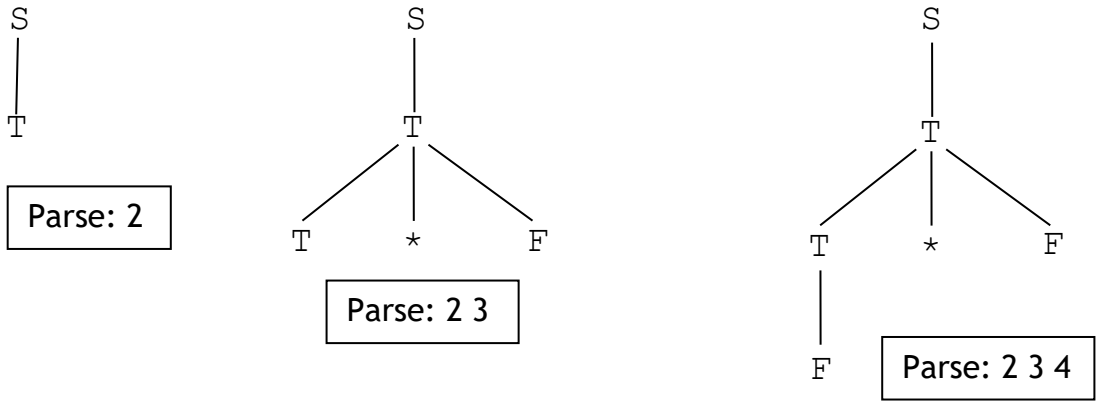
Ejemplo.-

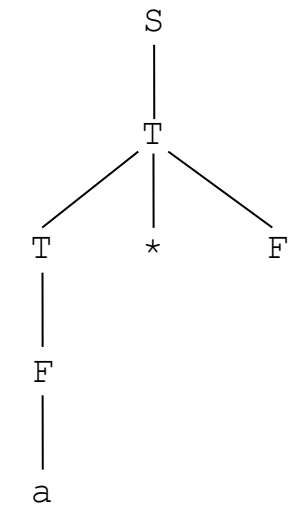
Sea la gramática:

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. $F \rightarrow b$

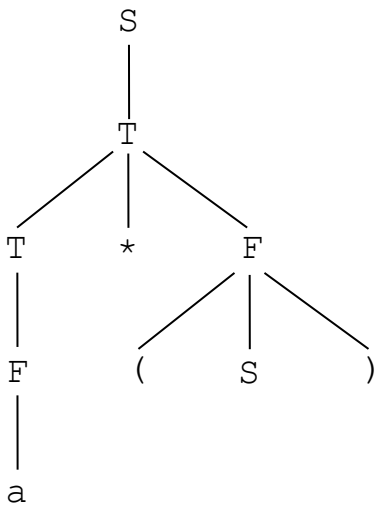
Vamos a realizar el parsing descendente de la cadena "a*(a+b)" por la izquierda y por la derecha:

Parsing descendente por la izquierda

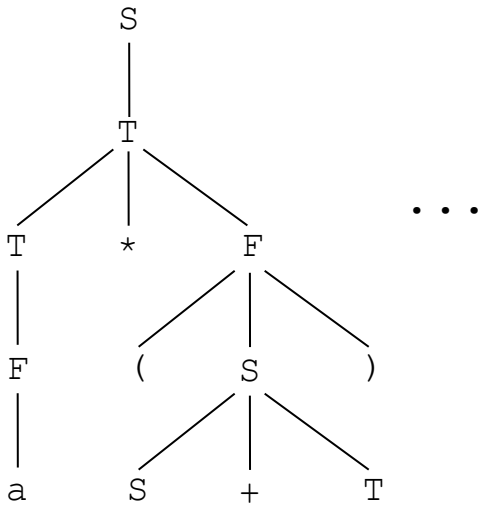




Parse: 2 3 4 6

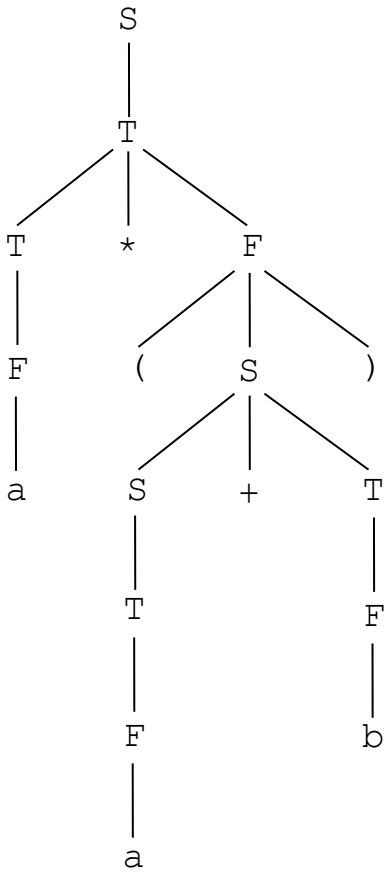


Parse: 2 3 4 6 5



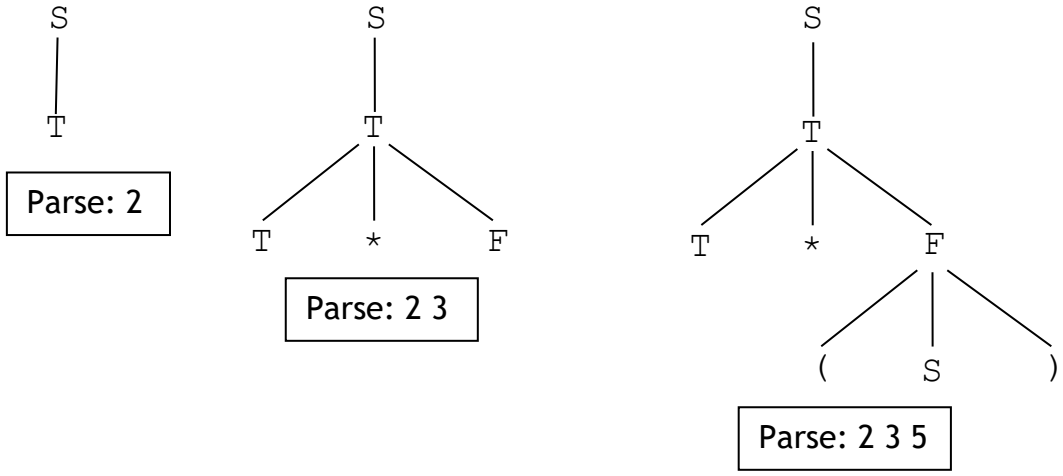
Parse: 2 3 4 6 5 1

. . .

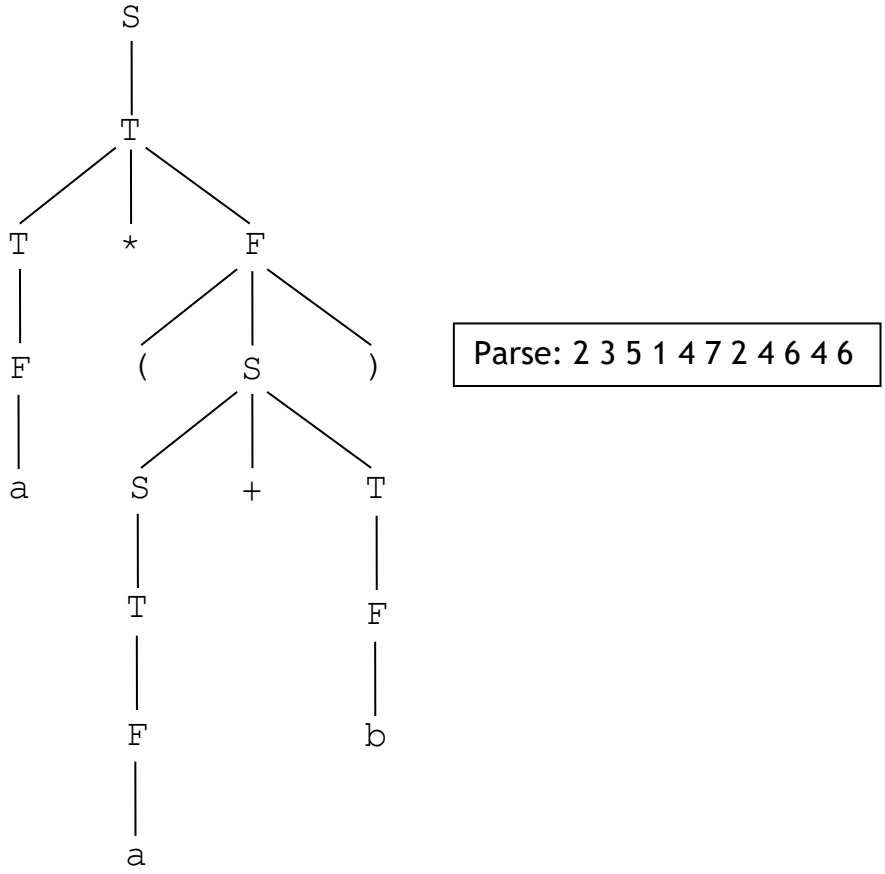


Parse: 2 3 4 6 5 1 2 4 6 4 7

Parsing descendente por la derecha



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

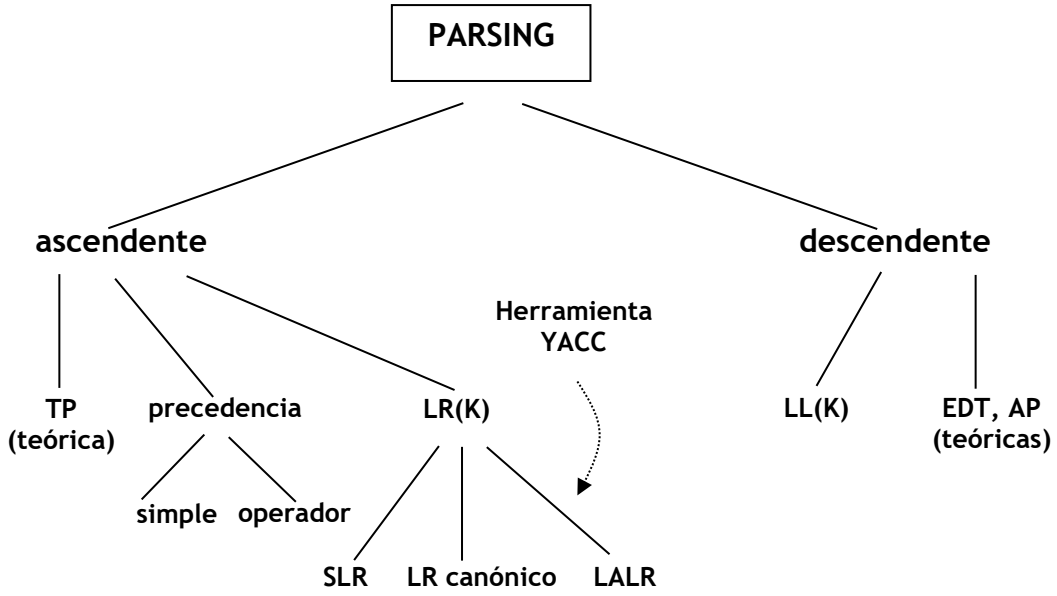


Además, también tenemos dos posibilidades en cuando a si comenzamos desde la metanoción o desde la cadena que queremos reconocer. Esto sería:

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 sóla imagen en un APD.

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

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

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

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

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

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

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

Existen tres posibilidades para el reconocimiento de un AP:

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

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

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

Lenguaje de un autómata con pila definido de esta forma:

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

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

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

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

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

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

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

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

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

Ejemplo.- un palíndromo.

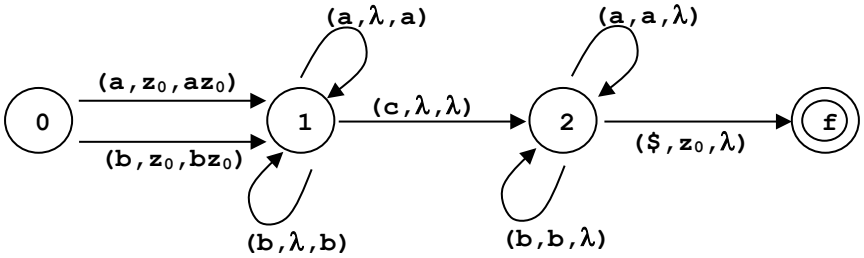
$L = \{ t\ c\ t^r / t=(a \mid b)^+\}$
$T = \{a, b, c\}$

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

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

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

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



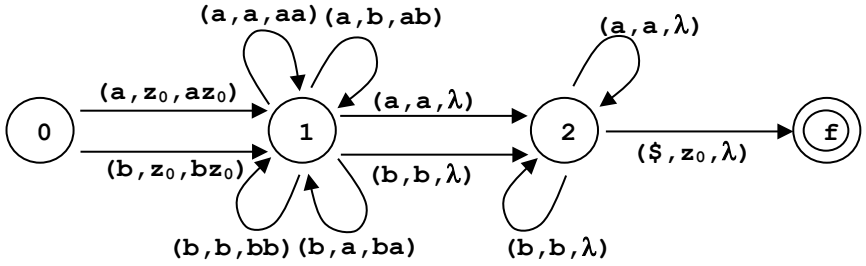
Ejemplo.-

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

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

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

Se trata de un autómata con pila no determinista.



Vamos a intentar reconocer la cadena "abba$":

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

4.1.1.1 Conversión de una GCL en un Autómata con pila.

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

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

Sea la gramática:

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

Crearemos un autómata con pila de la siguiente forma:

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

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

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

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

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

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

4.1.2 Esquemas de traducción (EDT).

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

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

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

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

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

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

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

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

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

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

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

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

Ejemplo1.- Notación polaca inversa.

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

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

Vamos a crear las reglas del EDT:

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

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

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

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

Ejemplo2.-

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

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

$F \rightarrow b, 7$

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

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

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

4.1.3 Traductores con pila (TP).

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

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

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

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

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

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

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

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

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

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

$Tr(TP) = \{(t,s) / (q_0, t, z_0, \lambda) \text{ |--*-- } (q´, \lambda, \lambda, s)\}$

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

Ejemplo.-
T = { a, b, (,), *, + }
TP = ({q}, T, N U T, <1, 2, ..., 7>, δ, q, z_0, q)

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

Si queremos vaciar la pila añadimos la siguiente regla:

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

Siendo t un terminal, para introducirlo en la pila:

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

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

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

4.2 Algoritmos sin retroceso.

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

4.2.1 Análisis sintáctico ascendente por precedencia simple.

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

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

i) Reflexiva. x **R** x $\forall x \in$ **R**

ii) Simétrica. x **R** y $\Rightarrow$ y **R** x

iii) Transitiva. x **R** y, y **R** z $\Rightarrow$ x **R** z

Ejemplo.-

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

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

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

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

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

Definición: **Relación universal**

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

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

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

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

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

Dadas dos relaciones R y S sobre A:

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

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

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

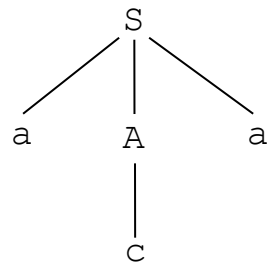
A la hora de realizar la tabla de relaciones de la gramática se puede hacer "a mano", haciendo todas las posibles derivaciones a partir del axioma de la

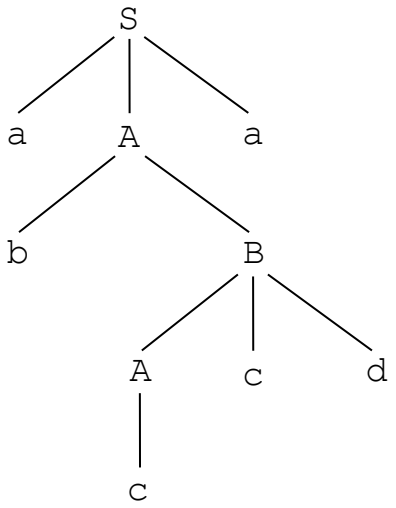
gramática y determinar las relaciones en función del lugar en que nos encontremos en el árbol.

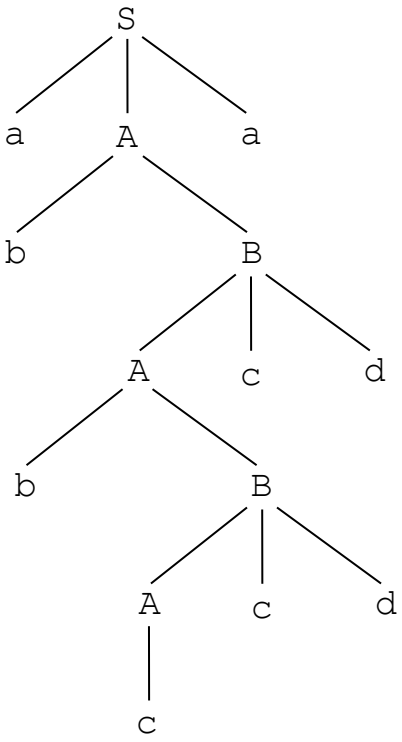
Por ejemplo, supongamos la gramática:

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

Algunos árboles serían los siguientes:







Las relaciones que podemos obtener son las siguientes:

b ± B
a < b
B > a
b < c
A ± c
Etc...

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

Finalmente, la tabla sería:

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

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

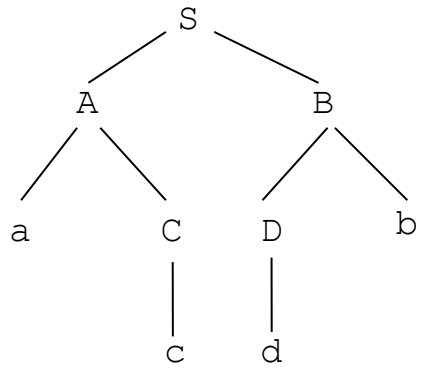
Llamamos **pivote** a un conjunto de símbolos que se pueden sustituir por las metanociones de la regla correspondiente (son símbolos situados entre < y > con los intermedios relacionados con ±). La tira completa la rodeamos por izquierda y derecha pcon < y >. Utilizando el concepto de pivote, vamos a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

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

4.2.1.1 Cálculo de la tabla de precedencia simple 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 ± 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 $X \in$ 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: Cab(A) = { x / $A \rightarrow^+ x...$}

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

$X \in$ **PRIMERO**$^+$(A) X si y sólo si $\exists\ A \rightarrow^+ X\alpha\beta$
$X \in$ **ULTIMO** (A) si y sólo si $\exists\ A \rightarrow \alpha\beta X$
$X \in$ **ULTIMO**$^+$ (A) si y sólo si $\exists\ A \rightarrow^+ \alpha\beta X$
$X \in$ **DENTRO** (A) si y sólo si $\exists\ A \rightarrow \alpha X\beta$
$X \in$ **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.-

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

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

La matriz PRIMERO+ sería la siguiente:

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

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

Algoritmo de Warshall

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

B = A

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

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

(±) se calcula por la propia definición.

(<) = **(±) (PRIMERO$^+$)**

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

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

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

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

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

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

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

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

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

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

Por lo tanto:

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

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 **$ULTIMO^{+}$**:

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

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

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

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

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

La tabla final de parsing será:

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

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

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

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

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

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

Por lo tanto:

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

Vamos a hacer el producto relativo con PRIMERO:

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

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

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

Vamos a hacer el producto relativo con ULTIMO:

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

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

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

Nuevamente aparece un "1", y seguimos:

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

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

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

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

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

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

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

La tabla final de parsing será:

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

Ejercicio 3.-

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

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

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

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

Por lo tanto:

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

Vamos a hacer el producto relativo con PRIMERO:

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

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⁺**:

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

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

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

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

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

La tabla final de parsing será:

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

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

4.2.2 Análisis sintáctico ascendente por precedencia de operadores.

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

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

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

Algoritmo

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

De esta forma, los errores que pueden producirse son:

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

Algoritmo para la obtención de las relaciones de precedencia operador

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

iii) Hágase:

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

Ejemplo1.- Sea la siguiente gramática:

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

La tabla de precedencia operador será la siguiente:

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

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

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

Ejemplo 2.- Sea la siguiente gramática:

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

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

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

Asumimos que:

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

La tabla de precedencia operador sería la siguiente:

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

Funciones de precedencia

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

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

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

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

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

$id+id*id$

Las precedencias las calcularíamos de la siguiente forma:

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

Con lo cual:

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

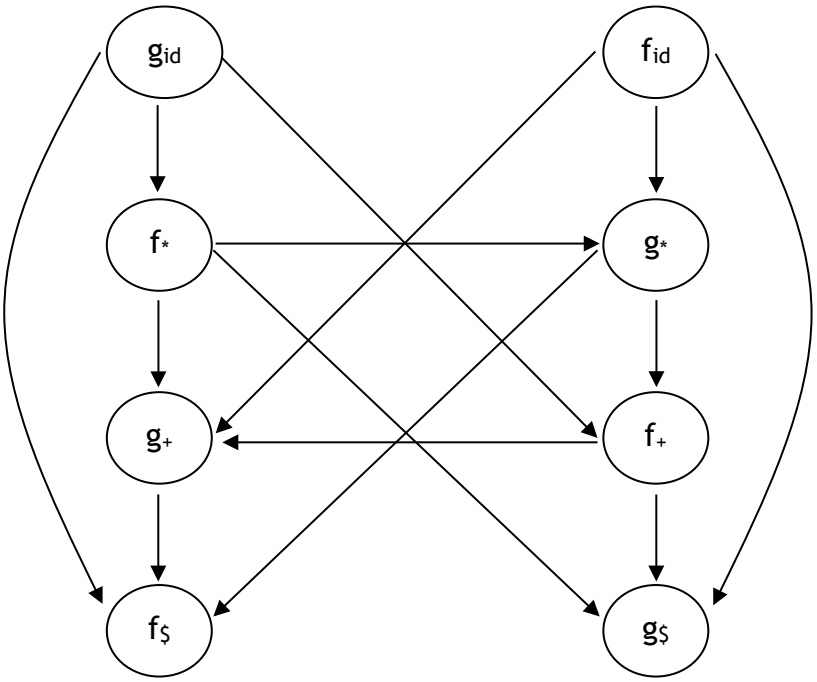
Reduciríamos <id> con la regla S → id y continuaríamos el proceso.

Algoritmo para la construcción de las funciones de precedencia

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

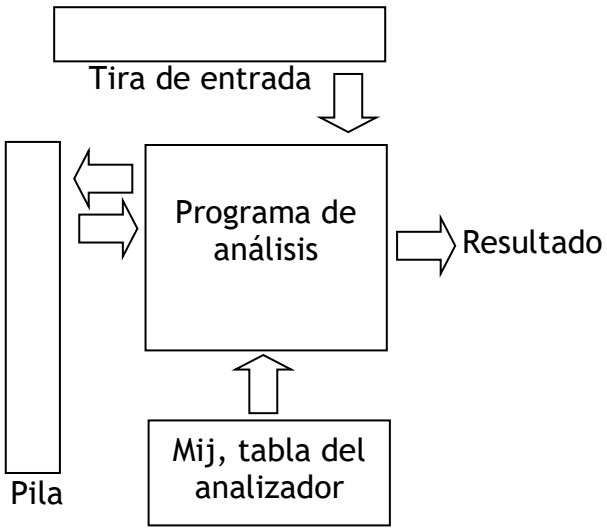
Para la tabla del ejemplo 1 las funciones serían las siguientes (construídas a partir del grafo):

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



4.2.3 Analizadores descendentes LL(k).

Este tipo de analizadores consta de los siguientes elementos:



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

Suponiendo que tenemos la tabla, el funcionamiento será:

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

Ejemplo.- Sea la gramática:

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

Y la tabla correspondiente:

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

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

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

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

4.2.3.1 Construcción de la tabla Mij.

A continuación veremos, como construir una tabla LL(k) en su versión LL(1).

Cálculo del conjunto PRIMERO (X):

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

Cálculo del conjunto SIGUIENTE(X):

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

Ejemplo.- Sea la gramática del ejemplo anterior:

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

Si calculamos PRIMERO y SIGUIENTE quedaría:

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

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

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

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

Algoritmo de confección de la tabla Mij

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

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

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

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

Con la regla $E \rightarrow TE'$
En este caso PRIMERO (TE´) = PRIMERO(T) porque ε no está incluido en este último. Utilizando la regla (2), {(, id} $\in$ PRIMERO(T), entonces hemos añadido E $\rightarrow$ TE´ en M[E, (] y M[E, id}

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

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

Y así continuaríamos con el resto de las reglas.

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

1) Ninguna gramática ambigua o recursiva por la izquierda puede ser LL(1).
2) Puede demostrarse que una gramática G es de tipo LL(1) si, y sólo si, cuando $A \rightarrow \alpha \mid \beta$ sean dos producciones distintas de G se cumplan las siguientes condiciones:
   a) Para ningún terminal a tanto α como β derivan a la vez cadenas que comiencen con a.
   b) A lo sumo una de α y β pude derivar la cadena vacía.
   c) Si $\beta \Rightarrow^* \varepsilon$, α no deriva ninguna cadena que comience con un terminal en SIGUIENTE(A).

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

E $\rightarrow$ **E** + **T** | **E** - **T** | **T**
T $\rightarrow$ **T** * **F** | **T** / **F** | **F**
F $\rightarrow$ id | id **[** **E** **]** | **(** **E** **)**

Ejemplo.- Sea la gramática:

S $\rightarrow$ { A }
A $\rightarrow$ id = E
E $\rightarrow$ id

Implementar un analizador LL(1).

Ejemplo.-

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

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

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

Primero calculamos los conjuntos PRIMERO y SIGUIENTE:

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

La tabla resultante es la que viene a continuación:

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

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

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

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

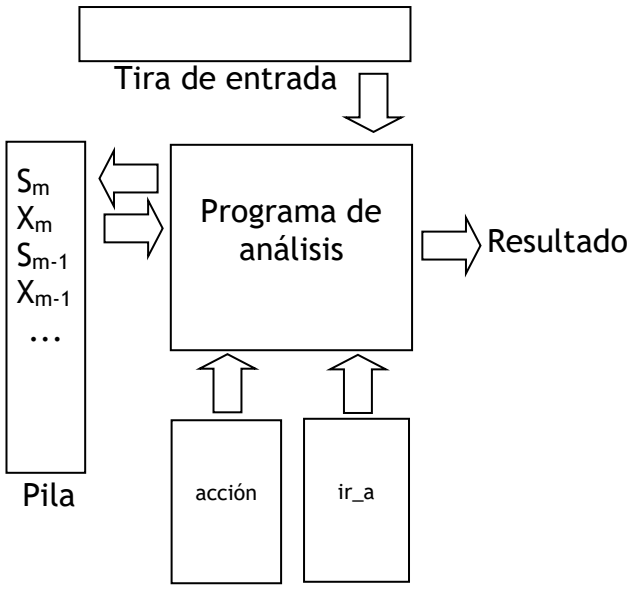
4.2.4 Analizadores ascendentes LR(k).

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

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

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

Este tipo de analizador consta de los siguientes elementos:



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

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

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

Vamos a ver las diferentes posibilidades:

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

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

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

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

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

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

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

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

Ejemplo.-

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

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

ESTADO	id	+	*	(	)	$	E	T	F
0	d5			d4			1	2	3
1		d6				ACEP.			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	r3		r3	r3			
11		r3	r3		r5	r5			
	ACCIÓN						IR_A		

Vamos a reconocer "id*id+id":

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

4.2.4.1 Método SLR.

Llamamos gramática LR a aquella gramática de contexto libre para la cual es posible construir la tabla (SLR, LALR ó LR-canónico).

SLR es el método más sencillo pero el menos general, siendo por lo tanto el que tiene más restricciones para la gramática en comparación con los otros dos.

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

Por ejemplo.-

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

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

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

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

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

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

Algoritmo para calcular cerradura(I)

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

Ejemplo.- En la gramática aumentada:

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

Vamos a calcular cerradura(E´ → .E):

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

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

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

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

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

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

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

I = { E´ → E., E → E.+T }

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

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

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

Obtención de la colección canónica LR(0)

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

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

Ejemplo.- Para la gramática aumentada anterior:

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

podemos obtener PRIMERO y SIGUIENTE

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

y calculamos la colección canónica LR(0) como:

I_0 = cerradura(E´ → .E) = E´ → .E T → .T*F F → .id
E → .E+T T → .F
E → .T F → .(E)

I_1 = Ir_a(I_0, E) = E´ → E.
E → E.+T

I_2 = Ir_a(I_0, T) = E → T.
T → T.*F

I_3 = Ir_a(I_0, F) = T → F.

I_4 = Ir_a(I_0, () = cerradura(F → (.E)) = F → (.E)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_5 = Ir_a(I_0, id) = F $\rightarrow$ id. I_6 = Ir_a(I_1, +) = cerradura(E $\rightarrow$ E+.T) = E $\rightarrow$ E+.T
T $\rightarrow$ .T*F
T $\rightarrow$ .F
F $\rightarrow$.(E)
F $\rightarrow$ .id

I_7 = Ir_a(I_2, *) = T $\rightarrow$ T*.F
F $\rightarrow$.(E)
F $\rightarrow$ .id

I_8 = Ir_a(I_4, E) = F $\rightarrow$ (E.)
E $\rightarrow$ E.+T

Ir_a(I_4, T) = I_2
Ir_a(I_4, F) = I_3
Ir_a(I_4, () = I_4
Ir_a(I_4, id) = I_5

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

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

Ir_a(I_6, () = I_4
Ir_a(I_6, id) = I_5

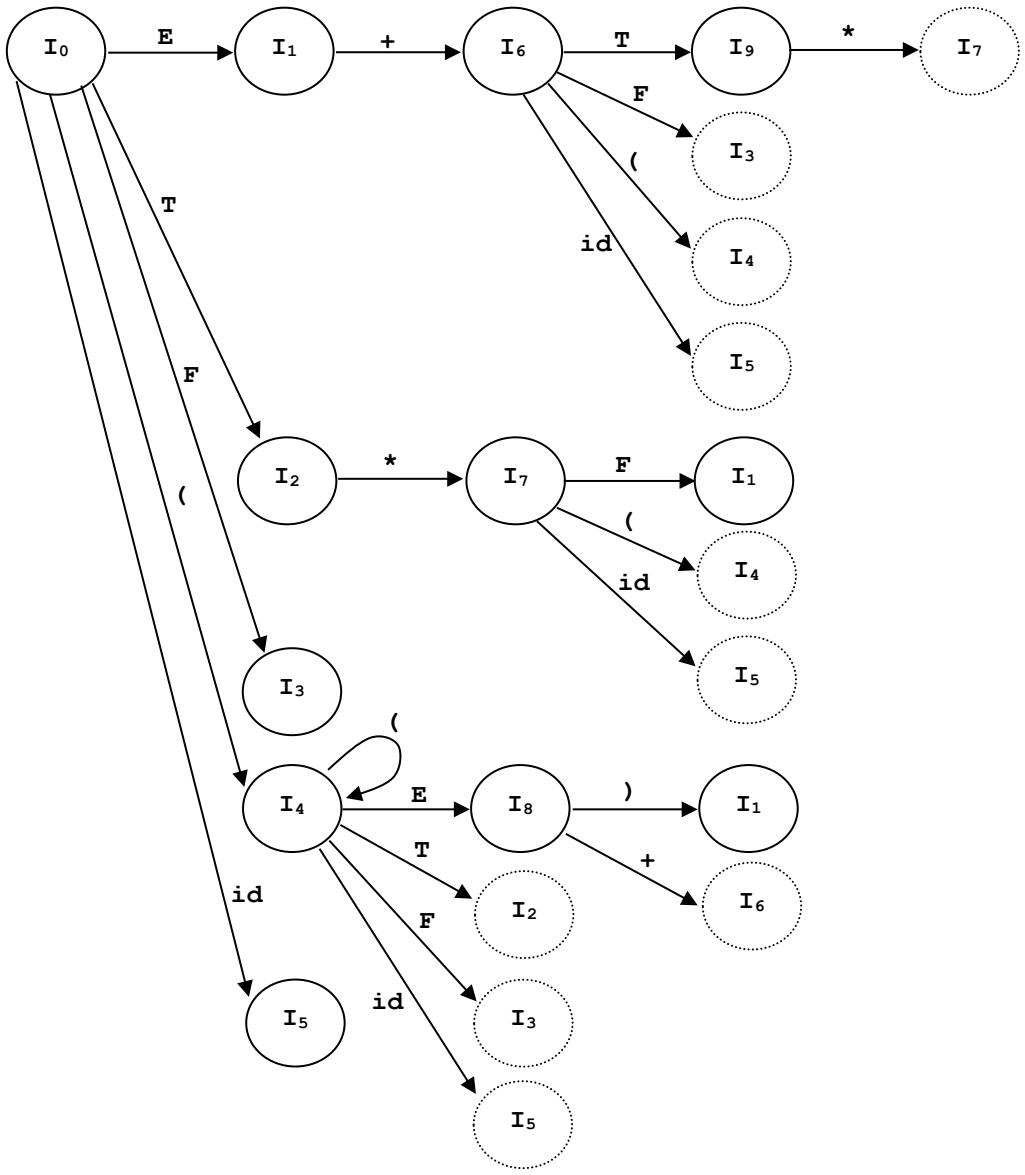
Ir_a(I_7, () = I_4
Ir_a(I_7, id) = I_5

I_{11} = Ir_a(I_8,)) = F $\rightarrow$ (E).

Ir_a(I_8, +) = I_6

Cuando tenemos el autómata LR(0), tendremos diversas posibilidades en cuanto a las reducciones posibles, es un método con retroceso (si nos equivocamos de rama tenemos que volver hacia atrás). En realidad se trata de un Autómata finito no determinista pues existen transiciones de la forma A $\rightarrow \alpha.X\beta$ a A $\rightarrow \alpha X.\beta$ etiquetadas con "X" y transiciones ε (dentro de los propios I_i) entre elementos de la forma A $\rightarrow \alpha.B\beta$ a B $\rightarrow .\gamma$.

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



<u>Obtención de la tabla de análisis SLR(1)</u>

Para la construcción de la tabla seguimos los siguientes pasos:

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

Ejemplo.- Para la gramática aumentada ya vista:

(0) $E' \rightarrow E$ (1) $E \rightarrow E+T$ (2) $E \rightarrow T$ (3) $T \rightarrow T*F$

(4) $T \rightarrow F$ (5) $F \rightarrow (E)$ (6) $F \rightarrow id$

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

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

Además, como:

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

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

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

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

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

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

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

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

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

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

ESTADO	id	+	*	(	)	$		E	T	F
0	d5			d4				1	2	3
1		d6				ACEP.				
2		r2	d7		r2	r2				
3		r4	r4		r4	r4				
4	d5			d4				8	2	3
5		r6	r6		r6	r6				
6	d5			d4					9	3
7	d5			d4						10
8		d6			d11					
9		r1	d7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				
	ACCIÓN							IR_A		

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

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

Si realizamos el ejercicio completo nos quedaría:

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

Calculemos los elementos LR(0):

I_0 = {cerradura({[$S' \rightarrow .S$]})} = $S' \rightarrow .S$
$S \rightarrow .L=E$
$S \rightarrow .E$
$L \rightarrow .*E$
$E \rightarrow .L$
$L \rightarrow .id$

I_1 = Ir_a(I_0, S) = $S' \rightarrow S.$

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

I_2 = Ir_a(I_0, L) = $S \rightarrow L.=E$ $\Rightarrow$ desplazar con "="
$E \rightarrow L.$ $\Rightarrow$ reducir $E \rightarrow L$ con "="
porque "=" $\in$ SIG(E)

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

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

I_3 = Ir_a(I_0, E) = $S \rightarrow E.$

I_4 = Ir_a(I_0, *) = $L \rightarrow *.E$
$E \rightarrow .L$
$L \rightarrow .*E$
$L \rightarrow .id$

I_5 = Ir_a(I_0, id) = $L \rightarrow id.$

I_6 = Ir_a(I_2, =) = $S \rightarrow L=.E$
$E \rightarrow .L$
$L \rightarrow .*E$
$L \rightarrow .id$

I_7 = Ir_a(I_4, E) = $L \rightarrow *E.$

I_8 = Ir_a(I_4, L) = $E \rightarrow L.$

Ir_a(I_4, *) = I_4
Ir_a(I_4, id) = I_5

I_9 = Ir_a(I_6, E) = $S \rightarrow L=E.$

Ir_a(I_6, L) = I_8
Ir_a(I_6, *) = I_4
Ir_a(I_6, id) = I_5

La tabla nos quedaría:

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

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

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

$E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow TF$
$T \rightarrow F$
$F \rightarrow F^*$
$F \rightarrow a$
$F \rightarrow b$

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

$S \rightarrow E$;
$E \rightarrow T + E$
$E \rightarrow T$
$T \rightarrow (E)$
$T \rightarrow x$

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

$S' \rightarrow S$;
$S \rightarrow (L)$
$S \rightarrow x$
$L \rightarrow S$
$L \rightarrow L , S$

4.2.4.2 Método LR-canónico.

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

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

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

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

Algoritmo para calcular cerradura(I)

```
Función cerradura(I)
BEGIN
        REPEAT
                FOR cada elemento [A → α.Bβ, a] ∈ I
```

Y cada $B \to \nu \in G'$
Y $b \in$ PRIMERO(βa)
Y $[B \to .\nu, b] \notin I$
DO añadir $[B \to .\nu, b]$ en I
UNTIL no se puedan añadir más a I
END.

Algoritmo para calcular Ir_a(I, X)

Función Ir_a(I, X)
BEGIN
Sea J el conjunto $[A \to \alpha X.\beta, a]$ tal que $[A \to \alpha.X\beta, a] \in I$
Return cerradura(J)
END.

Algoritmos de colección de elementos LR(1)

Procedimiento elementos_LR(1)(G´)
BEGIN
C = {cerradura({[S´ → .S, $]})}
REPEAT
FOR cada I, $I \subset C$ Y cada $X \in G'$ donde Ir_a(I, X) $\neq \varnothing$ Y
Ir_a(I, X) $\not\subset$ C, entonces añadir Ir_a(I, X) a C
UNTIL no se puedan añadir más conjuntos a C
END.

Ejemplo.- Sea la gramática:

S → CC
C → cC | d

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

Primero creamos la gramática extendida:

S´ → S
S → CC
C → cC | d

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

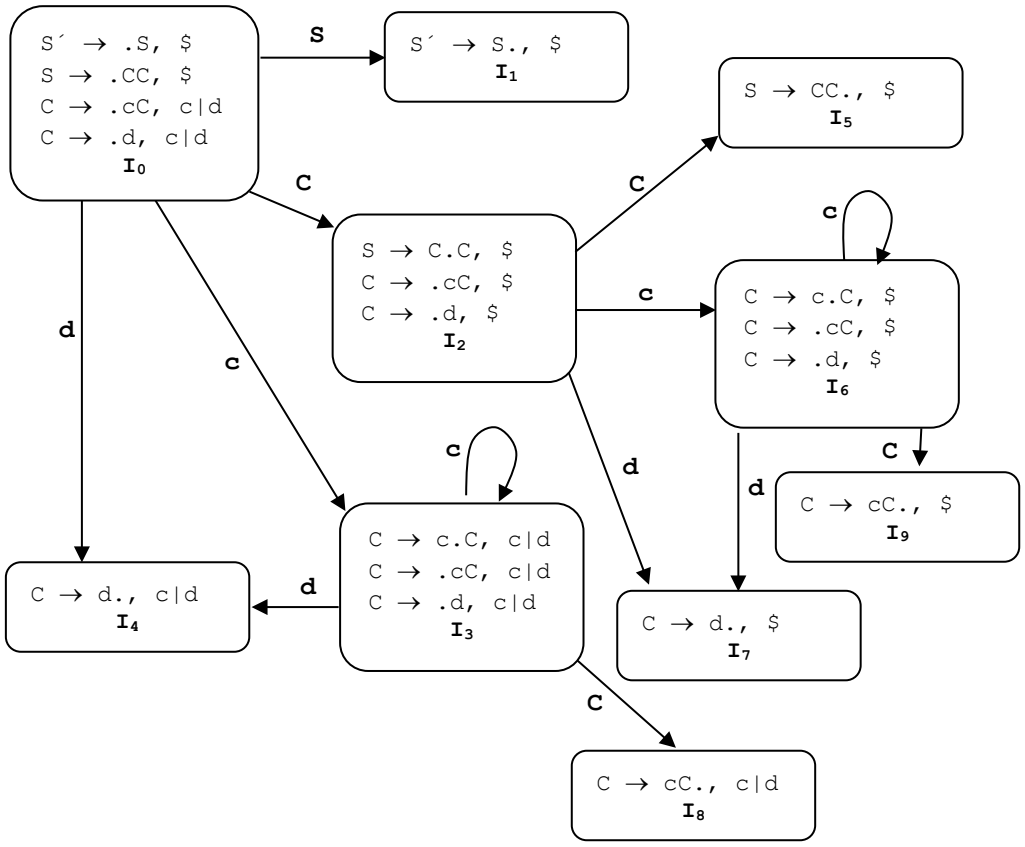
Ir_a(I_0 ,S) = cerradura([S´ → S., $]) = I_1 = S → S. , $

Ir_a(I_0 ,C) = cerradura([S → C.C, $]) = I_2 = S → C.C, $

$C \rightarrow .cC, \$$
$C \rightarrow .d, \$$

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

Así continuaríamos hasta llegar a conseguir el autómata siguiente:



<u>Obtención de la tabla de análisis LR-canónico(1)</u>

Para la construcción de la tabla seguimos los siguientes pasos:

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

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

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

iv) Todas las entradas no definidas constituyen los errores.

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

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

4.2.4.3 Método LALR.

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

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

En LR-canónico(1) tenemos, por ejemplo, un estado en el que nos encontramos:

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

En LALR(1), llamamos a "$\alpha.a\beta$" **corazón** y definimos un único estado para ambos. Tendremos menos estados, menos errores y también es menos potente.

En el ejercicio realizado en LR-canónico(1) teníamos:

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

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

Ocurre lo mismo con los estados 3 y 6 y con los estados 8 y 9.

Para crear la tabla LALR(1) seguimos los siguientes pasos:

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

La tabla resultante para la gramática anterior es la siguiente:

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

Ejemplo.- Sea la siguiente gramática, ya extendida:

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

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

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

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

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

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

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

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

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

$Ir\_a(I_4, d) = S \rightarrow aAd., \$ = I_{10}$

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

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

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

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

$I_{69} = \{ [A \rightarrow c., d|e], [B \rightarrow c., d|e] \}$
en donde, al generar la tabla de análisis LR, producirá un conflicto, en dos casillas aparecerán dos reducciones simultáneamente $A \rightarrow c$ y $B \rightarrow c$. Este sería un ejemplo de gramática que sí es LR-canónico pero no LALR.

5 ANÁLISIS SEMÁNTICO.

En esta fase, el objetivo es encontrar errores semánticos; en este capítulo nos centraremos principalmente en la recopilación de información sobre los tipos para posteriormente realizar la generación de código.

Para poder realizar el análisis semántico, se utiliza la estructura jerárquica definida durante el análisis sintáctico. Esto nos permitirá reconocer los operadores, operandos de expresiones y proposiciones.

Una de las funciones principales de este análisis es el verificar si los operadores tienen los operandos del tipo correcto según el lenguaje. Por ejemplo, puede significar un error sumar un número real y un número entero en un lenguaje y sin embargo, en otro, es correcto y se precisa realizar una conversión.

Las **acciones semánticas** nos van a permitir asociar información a las producciones gramaticales, de forma que incorporamos reglas semánticas e introducimos atributos a los símbolos de la gramática.

Para trabajar con reglas o acciones semánticas se utilizan dos notaciones:

1) Definiciones dirigidas por la sintáxis.
2) Esquemas de traducción.

En general, el metodo consistirá en construir el árbol sintáctico, crear el grafo de dependencias y evaluar las reglas semánticas. Los pasos serían:

CADENA $\rightarrow$ ÁRBOL $\rightarrow$ GRAFO DE DEPEN. $\rightarrow$ EVALUACIÓN REGLAS SEMÁNTICAS

5.1 Definiciones dirigidas por la sintáxis.

Una definición dirigida por la sintáxis es una generalización de una gramática independiente del contexto en la que cada símbolo gramatical tiene un conjunto de atributos asociados (pueden ser sintetizados o heredados).

Forma de una definición dirigida por la sintáxis.

Sea una GIC, G = (N, T, P, S), cada producción $A \rightarrow \alpha$ tiene asociado un conjunto de reglas semánticas de la forma $b = f(c_1, c_2, ..., c_k)$, donde f es una función, $c_1, c_2, ..., c_k$ son atributos de los símbolos gramaticales de la producción y b puede ser:

a) Un atributo sintetizado de A (a partir de atributos de la parte derecha).
b) Un atributo heredado perteneciente uno de los símbolos gramaticales que está en el lado derecho de la producción (calculado a partir de atributos de la parte izquierda y/o de sus hermanos en la parte derecha).

Se dice que un atributo es **sintetizado** si su valor en un nodo del árbol de análisis sintáctico se determina a partir de los valores de los atributos de los hijos del nodo.

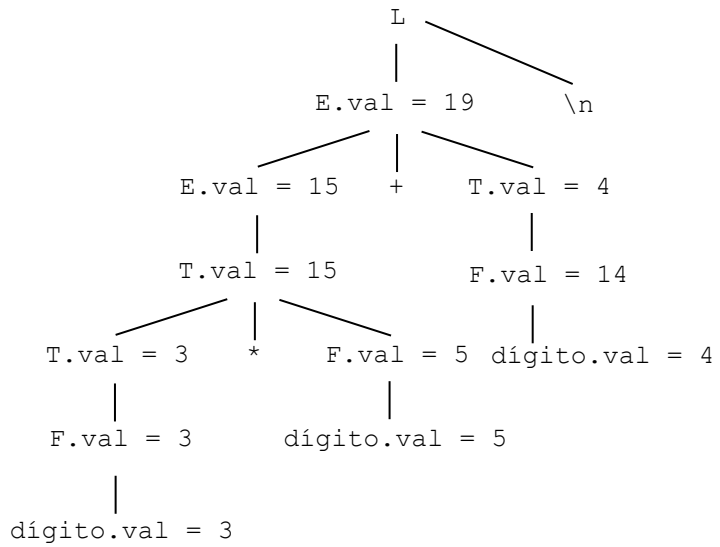
Se dice que un atributo es **heredado** si su valor en un nodo de un árbol de análisis sintáctico está definido a partir de los atributos en el padre y/o de los hermanos de dicho nodo.

Una definición dirigida por la sintáxis que usa exclusivamente atributos sintetizados se llama *definición con atributos sintetizados*.

Ejemplo.- Definición con atributos sintetizados.

$L \rightarrow E$ \n	Print (E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	E.val = T.val
$T \rightarrow T_1 * F$	$T.val = T_1.val$ x F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow$ dígito	F.val = dígito.val
$T \rightarrow F$	T.val = F.val

Para la entrada "3*5+4\n ", el árbol es:

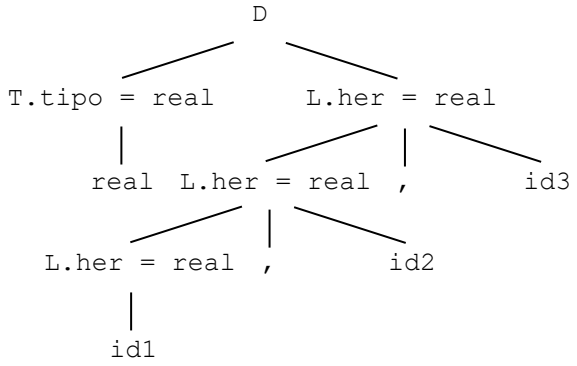


Definición de atributos heredados. Veámoslo con el siguiente ejemplo:

$D \rightarrow TL$	L.her = T.tipo
$T \rightarrow$ **int**	T.tipo = integer
$T \rightarrow$ **real**	T.tipo = real
$L \rightarrow L_1$ **, id**	L_1.her = L.her
	Añadetipo (**id**.entrada, L.her)
$L \rightarrow$ **id**	Añadetipo (**id**.entrada, L.her)

El procedimiento *añadetipo* añadiría el tipo de cada identificador en su zona de la tabla de símbolos.

A continuación mostramos el árbol sintáctico con anotaciones para la entrada "real id1, id2, id3". Los atributos L.her se obtienen a partir del valor del atributo T.tipo en el hijo izquierdo de la raíz y evaluando después L.her de forma descendente en los tres nodos de L en el subárbol derecho de la raíz. En los tres nodos de L, además, se llama al procedimiento *añadetipo*.



5.1.1 Grafos de dependencias.

Si un atrituto d de un nodo del árbol sintáctico es función de c_1, c_2, ..., c_k ahí tenemos una dependencia, pues tendremos que calcular antes los c_1, c_2, ..., c_k que d. Estas interdependencias entre los atributos sintetizados y/o heredados se pueden representar por un grafo que llamaremos "grafo de dependencias". Es decir, nos va a permitir determinar el orden de evaluación de las acciones semánticas.

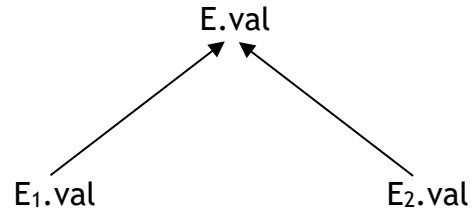
El algoritmo de construcción de este grafo es el siguiente:

for cada nodo *n* en el árbol de análisis sintáctico **do**
    For cada atributo *a* del símbolo gramatical en el nodo *n* **do**
        Construir un nodo en el grafo de dependencias para *a*;
for cada nodo *n* en el árbol de análisis sintáctico **do**
    for cada regla semántica b = f(c_1, c_2, ..., c_k)
    asociada con la producción utilizada en *n* **do**
        for i =1 **to** k **do**
            Construir una arista desde el nodo para c_i hasta *b*;

Ejemplo.-

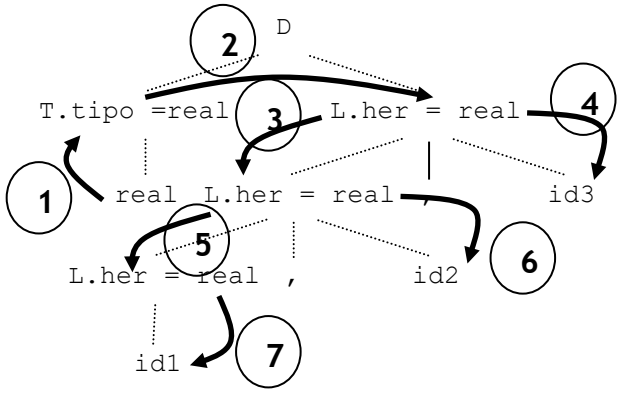
Producción	Regla semántica
$E \rightarrow E_1 + E_2$	$E.val = E_1.val + E_2.val$

Los tres nodos del grafo de dependencias representan los atributos sintetizados E.val, E_1.val y E_2.val en los nodos correspondientes del árbol de análisis.



Llamamos **ordenamiento topológico** de un grafo dirigido a todo ordenamiento $m_1, m_2, ..., m_k$ de los nodos del grafo tal que las aristas vayan desde los nodos que aparecen primero en el ordenamiento a los que aparecen más tarde.

Si tenemos $m_i \rightarrow m_j$, m_i ha de aparecer antes que m_j en el ordenamiento topológico.



5.2 Esquema de traducción

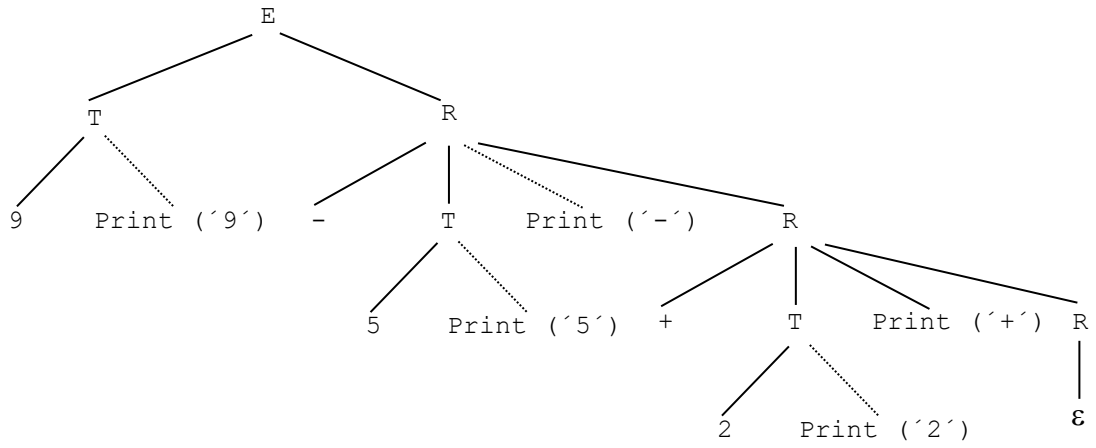
Es otra forma de representar las acciones semánticas, en ella insertamos las acciones semánticas en las reglas entre llaves. También se añaden atributos a los símbolos de la gramática.

Ejemplo.-

$E \rightarrow T\ R$
$R \rightarrow$ **Operador** T {print (**Operador**.lexema)} $R_1 \mid \varepsilon$
$T \rightarrow$ **num** {print (**num**.val)}

Cuando hablamos de **Operador** nos estamos refiriendo a un token que puede ser "+" o "-".

Vamos a ver el árbol de análisis sintáctico para la entrada "9-5+2", con cada acción semántica asociada como el hijo de su parte izquierda correspondiente:



La salida resultante es: "95-2+", es decir, la operación en notación postfija.

5.3 Comprobaciones en tiempo de compilación.

Ya hemos comentado que una tarea fundamental de un compilador, en esta etapa, es comprobar que los tipos de datos son los correctos. En relación a esto, algunas de las comprobaciones a realizar en el llamado *análisis estático* (en tiempo de compilación) son las siguientes:

a) **Comprobaciones de tipos**.- El compilador ha de ser capaz de detectar un error cuando se aplica a un operador un operando incompatible (sumar una matriz con un entero, por ejemplo).

b) **Comprobaciones de unicidad**.- Dependerá en gran medida del lenguaje, pero tiene que ver con donde se definen los objetos (por ejemplo, puede ser correcto la definición de variables globales con el mismo nombre que una variable local en un procedimiento o una función, como ocurre en **C**).

c) **Comprobación de operadores sobrecargados**.- Por ejemplo, el operador + significa algo distinto en el caso de 2+3 (operador de suma) que en el caso de "a" + "b" (concatenación).

d) **Polimorfismos en funciones**.- Una función polimórfica es aquella que se puede ejecutar con distintos tipos de argumentos.

Además de éstas, también tenemos comprobaciones del flujo de control; es importante controlar proposiciones que cambian el flujo de control en el programa. Por ejemplo, la proposición *break* en **C** hace que el control salte fuera de la proposición que la engloba (un *switch-case*, *for* o *while*).

5.3.1 Sistemas de tipos.

El comprobador de tipos se basará en las construcciones sintácticas del lenguaje, los tipos definidos y las reglas definidas para asignar los tipos a las construcciones del lenguaje. Todo ello definirá el sistema de tipos.

Por ejemplo, si tenemos la siguiente construcción:

if a > b then a = 25

tendremos que determinar si a y b son comparables por su tipo, tenemos que asignarle a a > b el tipo *boolean*, debemos de saber si a la variable "a" le podemos asignar el valor 25 por su tipo y, finalmente, si a > b es *true*, para lo cual hay que manejar el flujo de control (esto último ya en la ejecución).

Expresiones de tipos: es la forma de denotar el tipo de una construcción de un lenguaje. Una expresión de tipo puede ser un "tipo básico" o formarse aplicando un operador llamado "constructor de tipos" a otras expresiones de tipos. Nosotros utilizaremos la siguiente definición de *expresiones de tipos*:

1) **Tipos básicos**: algunos de ellos son los *integer, boolean, real, char, error* (identificación de un error), *null* (ausencia de valor).

2) **Constructores de tipos**:
   a) Matrices: *array(I,T)*, siendo I el índice y T una expresión de tipo, indica un tipo matriz con índices I y del tipo T.
   b) Productos: T_1 *x* T_2, producto cartesiano del tipo T_1 y T_2.
   c) Registros: Como un producto pero con campos con nombre.
   d) Punteros: *pointer (T)*, es un apuntador a un objeto de tipo T.
   e) Funciones: Las funciones se pueden definir como transformaciones desde un dominio tipo D a un rango tipo R, de la forma D $\rightarrow$ R. Por ejemplo int x int $\rightarrow$ int.

3) **Nombres de tipos**.- Por ejemplo, si en Pascal definimos:
   type enlace = $\uparrow$ nodo;
   type liga = $\uparrow$ nodo;
   var p : $\uparrow$ nodo;
   q : enlace;
   r : liga;

   El problema es saber si son del mismo tipo p, q y r... depende de la implementación pues en la definición original de Pascal no se definió el concepto "tipo idéntico".

5.3.2 Una gramática y sus comprobaciones de tipos.

Para mostar el funcionamiento de los sistemas de comprobación de tipos veamos a continuación una gramática que iremos ampliando paulatinamente a lo largo del presente tema, tanto en número de producciones como en acciones semánticas:

$P \rightarrow D$; E
$D \rightarrow D$; D | **id** : T
$T \rightarrow$ **char** | **integer** | **array [num] of** T | $\uparrow T$
$E \rightarrow$ **literal** | **num** | **id** | E **mod** E | E **[** E **]** | $E\uparrow$

La parte del esquema de traducción que almacena el tipo de un identificador sería la siguiente:

$P \rightarrow D$; E	
$D \rightarrow D$; D	
$D \rightarrow$ **id** : T	{AñadeTipo (**id**.entrada, T.tipo)}
$T \rightarrow$ **char**	{T.tipo := char}
$T \rightarrow$ **integer**	{T.tipo := integer}
$T \rightarrow \uparrow T_1$	{T.tipo := pointer (T_1.tipo)}
$T \rightarrow$ **array [num] of** T_1	{T.tipo := array (1...**num**.val, T_1.tipo)}

Reglas de comprobación de tipos en expresiones**:**

$E \rightarrow$ **literal**	{E.tipo := char}
$E \rightarrow$ **num**	{E.tipo := integer}
$E \rightarrow$ **id**	{E.tipo := BuscaTipo(**id**)}
$E \rightarrow E_1$ **mod** E_2	{E.tipo := IF E_1.tipo = integer AND E_2.tipo = integer THEN integer ELSE error}
$E \rightarrow E_1$ **[** E_2 **]**	{E.tipo := IF E_2.tipo = integer AND E_1.tipo = array (I, T) THEN T ELSE error}
$E \rightarrow E_1\uparrow$	{E.tipo := IF E_1.tipo = pointer(T) THEN T ELSE error}

Reglas de comprobación de tipos en proposiciones**:**

$S \rightarrow$ **id** := E	{S.tipo := IF BuscaTipo(**id**) = E.tipo THEN null ELSE error}
$S \rightarrow$ **if** E **then** S_1	{S.tipo := IF E.tipo = boolean THEN S_1.tipo ELSE error}
$S \rightarrow$ **while** E **do** S_1	{S.tipo := IF E.tipo = boolean THEN S_1.tipo ELSE error}
$S \rightarrow S_1$; S_2	{S.tipo := IF S_1.tipo = null AND S_2.tipo = null THEN null ELSE error}

Reglas de comprobación de tipos en funciones**:**

$E \rightarrow E_1(E_2)$	{E.tipo := IF E_2.tipo = s AND E_1.tipo = s $\rightarrow$ t THEN t ELSE error}

5.3.3 Equivalencia de expresiones de tipos.

Es importante tener una definición precisa sobre cuando dos expresiones de tipo son equivalentes. Surgen ambigüedades especialmente cuando se asigna un nombre a estas expresiones de tipo.

Para poder implementar en un ordenador la equivalencia de tipos se utiliza la denominada "equivalencia de nombres" y la "equivalencia estructural", es esta última en la que nos vamos a centrar.

Equivalencia estructural de tipos:

Como hemos visto las expresiones de tipos se construyen a partir de tipos básicos y constructores de tipos. Partiendo de estos conceptos decimos que existe una equivalencia estructural de dos expresiones si son el mismo tipo básico o están formadas aplicando el mismo constructor a tipos estructuralmente equivalentes.

Una función válida para saber si dos tipos son estructuralmente equivalentes puede ser la siguiente:

```
Function equival (s, t) : boolean
BEGIN
        IF s y t son del mismo tipo básico
          THEN return TRUE
          ELSE IF s = array (s1, t1) AND t = array (s2, t2)
            THEN return equival (s1, s2) AND equival (t1, t2)
            ELSE IF s = pointer (s1) AND t = pointer (t1)
              THEN return equival (s1, t1)
              ELSE IF s = s1 → s2 and t = t1 → t2
                THEN return  equival (s1, t1) and equival (s2, t2)
                ELSE ...
                  ...
                   ELSE return FALSE
END.
```

5.3.4 Codificación de tipos.

Consideremos la siguiente codificación para los tipos básicos:

TIPO BÁSICO	CODIFICACIÓN
Bolean	0000
Char	0001
Integer	0010
Real	0011

Si además, tenemos expresiones de tipos con los siguientes constructores:

a) pointer (t), un apuntador a tipo t.

b) freturns (t), que señala una función con argumentos y que devuelve un objeto de tipo t.
c) array (t) que indica una matriz.

Las expresiones de tipos formadas mediante la aplicación de dichos constructores tienen una estructura muy uniforme. Algunos ejemplos son los siguientes:

char
freturns (char)
pointer (freturns(char))
array (pointer(freturns(char)))

Si codificamos los constructores de la siguiente forma:

CONSTRUCTOR	CODIFICACIÓN
pointer	01
array	10
freturns	11

La codificación de las expresiones anteriores será:

CONSTRUCTOR	CODIFICACIÓN
char	...000000 0001
freturns (char)	...000011 0001
pointer (freturns(char))	...000111 0001
array (pointer(freturns(char)))	...100111 0001

Como se puede observar los cuatro bits a la derecha codifican el tipo básico y los colocados inmediatamente a su izquierda el constructor aplicado, más a la izquierda el constructor aplicado a este y así sucesivamente.

5.3.5 Conversión de tipos.

Si tenemos la expresión "a+b" tenemos que comprobar los tipos de a y b, bien para comprobar si existe una incompatibilidad de tipos o bien para realizar una conversión (la forma de sumar un entero y un real no es la misma que la forma de sumar dos enteros).

Las conversiones de tipos pueden ser:

a) **Implícitas o coerciones**.- son las que realiza el propio compilador sin intervención del programador.
b) **Explícitas**.- cuando el programador ha de intervenir para que la conversión se realice. Por ejemplo, las conversiones *(tipo) variable* de **C**.

Las conversiones es preferible realizarlas en tiempo de compilación que en tiempo de ejecución pues se ahorra tiempo, algo que es especialmente sencillo en el caso de las constantes.

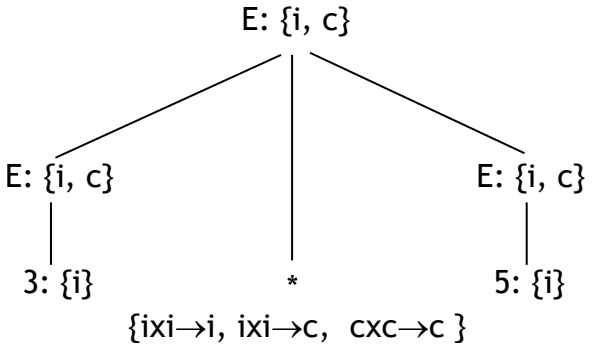
Un ejemplo de gramática para la conversión de tipos es la siguiente:

$E \rightarrow$ **num**	{E.tipo := integer}
$E \rightarrow$ **num . num**	{E.tipo := real}
$E \rightarrow$ **id**	{E.tipo := BuscaTipo(**id**)}
$E \rightarrow E_1$ **op** E_2	{E.tipo := IF E_1.tipo = integer AND E_2.tipo = integer THEN integer ELSE IF E_1.tipo = integer AND E_2.tipo = real THEN real ELSE IF E_1.tipo = real AND E_2.tipo = integer THEN real ELSE IF E_1.tipo = real AND E_2.tipo = real THEN real ELSE error}

5.3.6 Sobrecarga de funciones y operadores.

Un operador está sobrecargado cuando tiene distintos significados y acepta distintos tipos de datos. Es decir, podemos tener más de un tipo posible en la operación. Un ejemplo típico es la operación "+" (puede significar sumar o concatenar).

Veamos por ejemplo un árbol para la operación de multiplicación "*", con la expresión "3 * 5":



Como se ve en el árbol, si el único tipo posible para 3 y 5 es integer, entonces el operador * se aplica a un par de enteros (sería integer x integer). Sin embargo, existen dos posibilidades para el operador *, una devuelve un entero y la otra un complejo, ambos serán los tipos posibles para la raíz E.

Veamos un ejemplo con una función, cuando no existe sobrecarga:

$E \rightarrow E_1(E_2)$	{E.tipo := IF E_2.tipo = s AND E_1.tipo = s $\rightarrow$ t THEN t ELSE error}

Sin embargo, si existe sobrecarga en la función, tenemos que buscar si hay alguna combinación de tipos que encaje, localizando t / $\exists$ s $\rightarrow$ t $\in$ E_1.tipos, s $\in$ E_2.tipos.

Para reducir el tipo de E a un único tipo (algo obligatorio en algunos lenguajes) se puede implantar una definición dirigida por la sintaxis realizando dos recorridos en profundidad de un árbol sintáctico para una expresión. Durante el primer recorrido, el atributo "tipos" se sintetiza de manera ascendente. Durante el segundo recorrido, el atributo "unico" se propaga de forma descendente (*en cursiva*). Las reglas son:

S $\rightarrow$ **id** = E { *E.unico := IF BuscaTipo(id) nos devuelve un solo tipo AND BuscaTipo(id)* $\subset$ *E.tipos THEN BuscaTipo(id) ELSE error*
S.tipo := IF E.unico $\neq$ *error THEN null ELSE error*}

E $\rightarrow$ **id** {E.tipos := BuscaTipo(**id**)}

E $\rightarrow$ E_1(**E_2**) {E.tipos := t / $\exists$ s $\rightarrow$ t $\in$ E_1.tipos, $\exists$s $\in$ E_2.tipos
temp := {s / s $\in$ *E_2.tipos AND s* $\rightarrow$ *E.unico* $\in$ *E_1.tipos}*
E_2.unico := IF temp contiene un solo tipo {s} THEN s ELSE error
E_1.unico := IF temp contiene un solo tipo {s} THEN s $\rightarrow$ *E.unico ELSE error*}

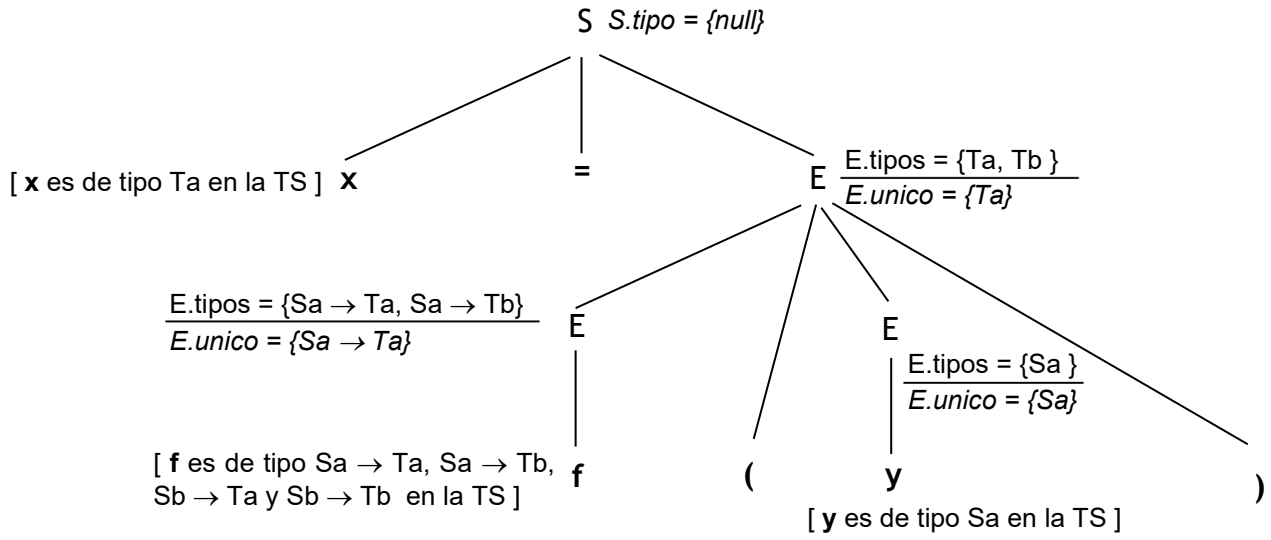
Si tenemos ***c = mifuncion(b)***. Nos quedamos con el tipo de ***c*** (que será el que correspondería a E´ en el análisis descendente de los atributos .unico) y buscamos en el árbol para encontrar una combinación de los tipos de ***mifuncion*** y de ***b*** que nos den el tipo que tenemos predefinido para ***c***. Si hay más de una combinación, la comprobación no se puede hacer y tendremos un error, y si no hay ninguna tambien se produce un error.

Por ejemplo, si tenemos una frase del lenguaje de la forma: x = f(y), siendo:
x de tipo Ta
f de tipo Sa $\rightarrow$ Ta, Sa $\rightarrow$ Tb, Sb $\rightarrow$ Ta, Sb $\rightarrow$ Tb
y de tipo Sa

De forma ascendente tenemos que:
En E $\rightarrow$ **id** (de la "f") E.tipos = {Sa $\rightarrow$ Ta, Sa $\rightarrow$ Tb, Sb $\rightarrow$ Ta, Sb $\rightarrow$ Tb}
En E $\rightarrow$ **id** (de la "y") E.tipos = {Sa}
En E $\rightarrow$ E_1(E_2), E_1.tipos = {Sa $\rightarrow$ Ta, Sa $\rightarrow$ Tb} y E.tipos={Ta, Tb}
En S $\rightarrow$ **id** = E (**id** aquí es "x", de tipo Ta), E.unico ={Ta} => S.tipo=null

De forma descendente tenemos que E.unico = {Ta}, y con ello podemos reducir a un único tipo cada nodo del árbol (por ejemplo, en nuestro caso, la función será de tipo Sa $\rightarrow$ Ta).

Estos recorridos se pueden ver en la siguiente figura:


S S.tipo = {null}
[x es de tipo Ta en la TS] x
=
E E.tipos = {Ta, Tb }
E.unico = {Ta}
E.tipos = {Sa → Ta, Sa → Tb}
E.unico = {Sa → Ta}
E
E
E.tipos = {Sa }
E.unico = {Sa}
[f es de tipo Sa → Ta, Sa → Tb,
Sb → Ta y Sb → Tb en la TS]
f
(
y
)
[y es de tipo Sa en la TS]

6 GENERACIÓN DE CÓDIGO.

La generación de código es una etapa muy relacionada con el procesador (tecnologías CISC y RISC, existencia o no de coprocesador matemático, procesadores vectoriales, sistemas multiprocesador, etc), y precisamente su arquitectura va a influir profundamente en el rendimiento del programa objeto generado.

En esta etapa partimos del árbol resultante del análisis sintáctico y se generará:

1º código intermedio para facilitar la optimización, diseño de fases, etc. Además, si distintos lenguajes general el mismo código intermedio las fases dirigidas a la generación de código máquina pueden unificarse reduciendo los costes de desarrollo.

2º código, normalmente será el código objeto que entiende el procesador aunque podría tratarse también de otro lenguaje distinto (por ejemplo, el GNU Fortran es un traductor de FORTRAN a C).

6.1 Lenguajes intermedios.

6.1.1 Notación Polaca Inversa (RPN)

Esta notación es muy apropiada para analizar y evaluar expresiones aritméticas pero no es demasiado cómoda para utilizarla como lenguaje intermedio de un lenguaje de programación, especialmente para manejar las estructuras de salto.

Una gramática simple para operaciones aritméticas, que genera mediante acciones semánticas código intermedio es la siguiente:

$S \rightarrow S^1 * S^2, S^1 S^2 *$
$S \rightarrow S^1 + S^2, S^1 S^2 +$
$S \rightarrow (S), S$
$S \rightarrow S^1 / S^2, S^1 S^2 /$

6.1.2 Cuartetos

Es la notación más adecuada para representar operaciones binarias. Consta de los siguientes elementos:

(<operador>, <op1>, <op2>, <resultado>)

A / B se escribiría como (/, A, B, t_i), siendo t_i una variable temporal.

Por ejemplo, la siguiente operación:

A + B – C * D ↑ F

Se escribiría con cuartetos de la siguiente forma:

1. (+, A, B, T_1)
2. (↑, D, F, T_2)
3. (*, C, T_2, T_3)
4. (-,T_1, T_3, T_4)

Veremos técnicas para ordenar los cuartetos optimizando las operaciones, incluso llegando a eliminar algunos.

La notación de cuartetos que usaremos será la siguiente:

a) +, -, *, /, ↑.
b) mod (módulo), abs (valor absoluto) y sqr (raíz cuadrada).
c) sin, cos.
d) Asignación (:=, E, , X) que representa X:=E.
e) Entrada / Salida: (READ, , ,X), (WRITE, , , X)
f) Salto absoluto: (JP, n, ,) salta al cuarteto número n.
g) Salto relativo: (JR, n, ,) salta a la posición actual + n.
h) Saltos condicionales:
   - (JZ, n, E,) Si E = 0 salta a n.
   - (JGZ, n, E,) Si E > 0 salta a n.
   - (JLZ, n, E,) Si E < 0 salta a n.
   - (JE, n, X_1, X_2) Si X_1=X_2 salta a n.
   - (JG, n, X_1, X_2) Si X_1>X_2 salta a n.
   - (JL, n, X_1, X_2) Si X_1<X_2 salta a n.

Ejemplo.- Veamos a continuación un bloque de código en Pascal y su correspondiente código en forma de cuartetos:

```
BEGIN
        s:=0;
        i:=0;
        read (n);
100:    i:=i+1;
        read (a);
        s:=s+sqr(a);
        if i<n then goto 100;
        write (s);
END.
```

Código en cuartetos:

1. (:=, 0, , s)

2. (:=, 0, , i)
3. (READ, , , n)
4. (+, i, 1, t_1)
5. (:=,t_1, , i)
6. (READ, , , a)
7. (sqr, a, , t_2)
8. (+, s, t_2, t_3)
9. (:=, t_3, , s)
10. (JL, 4, i, n)
11. (WRITE, , , s)

6.1.3 Tercetos.

La notación de cuartetos es una de las más utilizadas como lenguaje intermedio, sin embargo tiene el inconveniente de ocupar mucho espacio en memoria debido a la utilización de gran cantidad de variables auxiliares.

Los tercetos ahorran espacio eliminando el campo de resultado, quedando implícito, asociado a dicho terceto. El formato de un terceto es el siguiente:

(<operador>, <op1>, <op2>)

La expresión "A+B/C", expresada en forma de tercetos, se representa como:

1. (/, B, C)
2. (+, A, [1])

Con el [1] hacemos referencia al resultado obtenido del primer terceto.

Debido a que el resultado de un terceto está implícito, al realizar una reordenación de los tercetos, por ejemplo en una optimización, hace que estos sean difíciles de manejar (hay que cambiar las referencias). Por ello, se ha definido una modificación denominada "tercetos Indirectos".

En el caso de tercetos indirectos disponemos de dos estructuras: una son los tercetos tal cual teníamos anteriormente y otra es el denominado "vector secuencia", que indica la secuencia de ejecución de los tercetos.

Con este esquema, al reordenar la ejecución no es preciso cambiar en los propios tercetos las referencias a resultados de otros tercetos, y además si tenemos tercetos repetidos en la ejecución, éstos se indicarán solamente en el vector de secuencia.

Ejemplo.- Vamos a escribir en formato de tercetos el mismo código del apartado anterior.

1. (:=, 0, s)
2. (:=, 0, i)
3. (READ, , n)
4. (+, i, 1)

5. (:=, [4], i)
6. (READ, , a)
7. (sqr, , a)
8. (+, s, [7])
9. (:=, (8), s)
10. (-, i, n)
11. (JLZ, 4, [10])
12. (WRITE, , s)

Nótese que aquí no tenemos instrucciones JE, JG y JL, por el número de operandos de los tercetos.

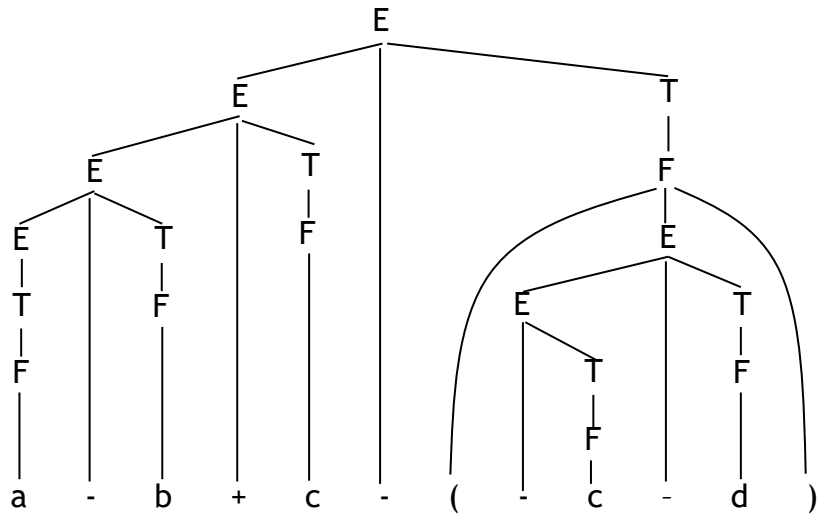
6.2 Generación de código intermedio.

6.2.1 Generación de RPN desde expresiones aritméticas.

Para la generación de la pila con la expresión en formato RPN, utilizaremos las acciones semánticas durante el parsing. A continuación vemos un ejemplo de gramática con reglas semánticas para realizar esta generación (con P(p) indicamos la cabeza de la pila):

REGLAS BNF	ACCIONES SEMÁNTICAS
$S \rightarrow E$	
$E \rightarrow T$	
$E \rightarrow E + T$	P(p) = ´+´, p = p+1
$E \rightarrow E - T$	P(p) = ´-´, p = p+1
$E \rightarrow - T$	P(p) =´@´, p = p+1
$T \rightarrow F$	
$T \rightarrow T * F$	P(p) = ´*´, p = p+1
$T \rightarrow T / F$	P(p) = ´/´, p = p+1
$F \rightarrow id$	P(p) = id, p = p+1
$F \rightarrow (E)$	

Ejemplo.- Vamos a ver el árbol para “a-b+c-(-c-d)”



Haciendo un recorrido ascendente "izquierda-derecha" obtenemos el estado de la pila, que será: "ab-c+c@d--".

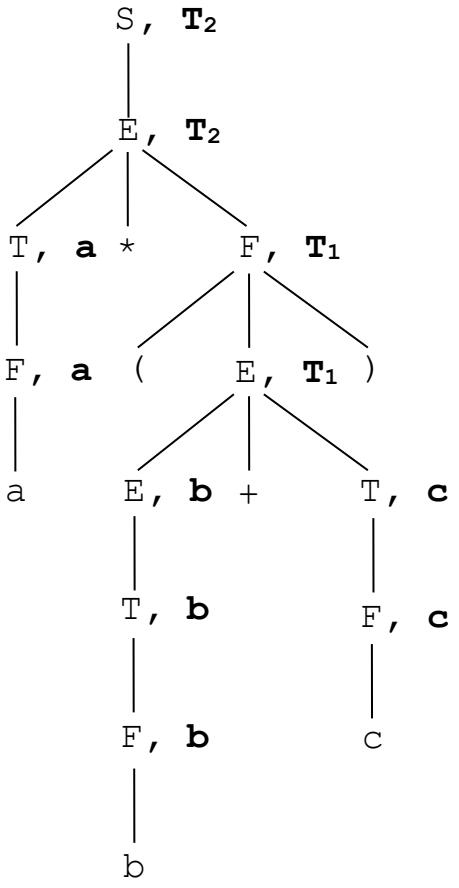
6.2.2 Generación de cuartetos.

A la hora de generar los cuartetos a partir del árbol sintáctico, hay que tener en cuenta el problema del arrastre de la información semántica, para eso utilizamos la notación ".sem". Es decir, tendremos reglas semánticas de arrastre de atributos, así como reglas de generación de código. Además, crearemos variables temporales Ti cuando tengamos resultados de operaciones.

Veamos ahora una gramática y sus reglas asociadas:

$S \rightarrow E$	S.sem = E.sem
$E \rightarrow T$	E.sem = T.sem
$E_1 \rightarrow E_2 + T$	i=i+1; E_1.sem=Ti; Genera ["(´+´, E_2.sem, T.sem, E_1.sem)"]
$E_1 \rightarrow E_2 - T$	i=i+1; E_1.sem=T_i; Genera ["(´-´, E_2.sem, T.sem, E_1.sem)"]
$E \rightarrow -T$	i=i+1; E_1.sem=T_i; Genera ["(´@´, ∅, T.sem, E_1.sem)"]
$T \rightarrow F$	T.sem = F.sem
$T_1 \rightarrow T_2 * F$	i=i+1; T_1.sem=T_i; Genera["(´*´, T_2.sem, F.sem, T_1.sem)"]
$T_1 \rightarrow T_2 / F$	i=i+1; T_1.sem=T_i; Genera["(´/´, T_2.sem, F.sem, T_1.sem)"]
$F \rightarrow id$	F.sem = id.sem
$F \rightarrow (E)$	F.sem = E.sem

Vamos a ver el árbol para “a*(b+c)”



6.3 Generación de código desde lenguaje intermedio.

6.3.1 Definición de la máquina objeto.

En este apartado vamos a definir una máquina objeto para todos los ejemplos que veamos.

Nuestra máquina dispondrá de n registros: R0, R1, ..., Rn. Hay que tener en cuenta que cualquier operación realizada sobre los registros es mucho más rápida que si se realiza sobre datos en memoria.

Las operaciones máquina serán de la forma: OP fuente, destino. Vamos a suponer que las palabras son de 4 bytes.

Las intrucciones válidas para nuestra máquina serán:

MOV fuente, destino	Llevar la información de la fuente al destino.
ADD fuente, destino	Hace fuente + destino y el resultado en destino.
SUB fuente, destino	Hace destino – fuente y el resultado en destino.
MUL fuente, destino	Hace fuente * destino y el resultado en destino.
DIV fuente, destino	Hace destino / fuente y el resultado en destino.

GOTO posición	Saltos en el programa.
HALT	Parada de programa.
CALL	Llamada a subrutina.
RETURN	Retorno de subrutina.
CMP fuente, destino	Comparar fuente con destino.
CJE posición	Salto a posición si tras CMP fuente = destino.
CJL posición	Salto a posición si tras CMP fuente < destino.
CJG posición	Salto a posición si tras CMP fuente > destino.

Modos de direccionamiento.

MODO	FORMA	DIRECCIÓN	COSTO
Absoluto	M	M	1
Registro	Ri	No dirección, valor Ri	0
Indexado	C(R)	C + Contenido(R)	1
Registro indirecto	*R	Contenido(R)	0
Indexado indirecto	*C(R)	Contenido(C + Contenido(R))	1

Ejemplo.-

MOV R0, M	Copia el contenido de R0 a la posición de memoria M.
MOV 4(R0), M	Copia el contenido de la dirección (4 + Contenido(R0)) a la dirección M.
MOV *4(R0), M	Copia el contenido de la dirección de memoria almacenada en la dirección (4 + Contenido(R0)) a la dirección M.
MOV #325h, M	Copia el valor 325h a la dirección de memoria M.

6.3.2 Generación de código desde RPN.

El mecanismo es el siguiente:

a) Si llega un símbolo se almacena en la pila.
b) Si llega un operador se realiza la operación, se sacan los operandos de la pila, se genera el código almacenando el resultado de la operación dentro de la pila.

Veamos el ejemplo con la sentencia "A*B+C-D*E", que en RPN es "AB*C+DE*-", suponiendo que sólo tenemos un registro o acumulador, el R0:

Pila antes	Símbolo actual	Resto sentencia	Acción	Código generado
$	A	B*C+DE*-$	Apilar A	
$A	B	C+DE*-$	Apilar B	

$AB	*	C+DE*-$	Generar	MOV A, R0
				MUL B, R0
				MOV R0, T_1

T_1	C	+DE*-$	Apilar C	

T_1C	+	DE*-$	Generar	MOV T_1, R0
				ADD C, R0
				MOV R0, T_2

$\$T_2$	D	E*-$	Apilar D	
$\$T_2$D	E	*-$	Apilar E	

$\$T_2$DE	*	-$	Generar	MOV D, R0
				MUL E, R0
				MOV R0, T_3

$\$T_2T_3$	-	$	Generar	MOV T_2, R0 SUB T_3, R0
$	$		Fin	

Como se puede observar muy fácilmente, el código generado no es óptimo pues el registro se vacía y se vuelve a llenar con el mismo valor repetidas veces. Para solucionar este problema vamos a utilizar un símbolo (**Acc**) que introduciremos en la pila cuando esta está usada; si en algún momento se intenta introducir en la pila dos **Acc**, significará que tenemos que descargarlo en memoria. Veamos el funcionamiento con el ejemplo anterior.

Pila antes	Símbolo actual	Resto sentencia	Acción	Código generado
$	A	B*C+DE*-$	Apilar A	
$A	B	C+DE*-$	Apilar B	
$AB	*	C+DE*-$	Generar	MOV A, R0 MUL B, R0
$**Acc**	C	+DE*-$	Apilar C	
$**Acc**C	+	DE*-$	Generar	ADD C, R0
$**Acc**	D	E*-$	Apilar D	
$**Acc**D	E	*-$	Apilar E	

$**Acc**DE	*	-$	Generar	MOV R0, T_1
				MOV D, R0
				MUL E, R0

$ T_1**Acc**	-	$	Generar	MOV R0, T_2
				MOV T_1, R0
				SUB T_2, R0

$	$		Fin	

6.3.3 Generación de código desde cuartetos.

En general, a partir de un cuarteto genérico:

(OP, Op1, Op2, Resultado)

obtendremos el siguiente código genérico:

MOV Op1, R0
OP Op2, R0
MOV R0, Resultado (nota: este Resultado será un T_i)

Para manejar el acumulador, y previamente a ejecutar la operación OP, actuaremos de la siguiente forma, según las posibilidades:

a) Acumulador vacío: Genera MOV Op1, R0 y luego OP Op2, R0
b) Acumulador lleno con el primer operando: Genera OP Op2, R0
c) Acumulador lleno con el segundo operando:
   - Si la operación es conmutativa: Genera OP Op1, R0
   - Si la operación no es conmutativa: Genera MOV R0, Ti
     MOV Op1, R0
     OP Ti, R0
d) Acumulador lleno con algo útil, que no es el primero ni el segundo operando: Genera MOV R0, Ti
   MOV Op1, R0
   OP Op2, R0

7 OPTIMIZACIÓN DE CÓDIGO.

En este capítulo trataremos de optimizar tanto velocidad de ejecución como memoria ocupada por el programa. En cualquier caso, los grandes cambios en el hardware que está apareciendo en al actualidad pueden hacer que muchos de los conceptos que veamos cambien.

En este tema trataremos distintos puntos de vista de la optimización. Trataremos de analizar el flujo de datos. Por ejemplo, si tenemos las siguientes instrucciones en nuestro programa:

```
FOR I=1 TO 100
        {
        ...
        a = b + c
        ...
        }
```

La operación a = b + c podría realizarse fuera del bucle en el caso de que "a" no cambie. De esta forma aceleraríamos la ejecución del programa.

También veremos algoritmos para optimizar la ejecución de operaciones aritméticas.

Para los ejemplos de este capítulo utilizaremos código de tres direcciones.

Resumiendo, en este capítulo trataremos los temas:

1. **Reducción de operaciones**:

Si tenemos la operación A=4x3+j dentro de un bucle, y j no depende del bucle, entonces lo más lógico será realizar la operación fuera de él, insertando en el código objeto el resultado de la parte del cálculo basada en constantes.

2. **Reacondicionamiento de instrucciones**:

La operación: A = B*C*(D+E), podríamos generar el código de las siguientes formas:

Forma 1	Forma 2
MOV B, R0	MOV D, R0
MUL C, R0	ADD E, R0
MOV R0, T1	MUL B, R0
MOV D, R0	MUL C, R0
ADD E, R0	
MUL T1, R0	

Si lo hacemos de la forma 2 obtenemos un código más pequeño y que ocupa menos memoria (no utilizamos ningún Ti).

En general veremos que reordenando las operaciones se optimiza el código. Esto lo haremos con el algoritmo de Nakata.

3. **Eliminación de redundancias**:

Veremos el problema de las redundancias aplicado principalmente a las matrices. Veamos el siguiente ejemplo.-

Definición de la matriz:
a : array [1..4; 1..6; 1..8] of integer

Instrucción dentro de un programa:
a[i, j, 5] := a[i ,j ,6] + a[i ,j ,4] + a[i ,j ,3]

Hay que tener en cuenta que las matrices, al almacenarse en memoria, utilizan direcciones de memoria consecutivas. Vamos a suponer que nuestro compilador almacena los datos en memoria en el mismo orden que aparecen los índices.

Vamos a ver la dirección de memoria de cada uno de los componentes "matriz" de la instrucción se calcularán sumando a la dirección del primero [1, 1, 1] diferentes valores:

a[i, j, 5] dirección: [1, 1, 1] + (i-1) * 6 * 8 + (j-1) * 8 + 5 – 1
a[i, j, 6] dirección: [1, 1, 1] + (i-1) * 6 * 8 + (j-1) * 8 + 6 – 1
a[i, j, 4] dirección: [1, 1, 1] + (i-1) * 6 * 8 + (j-1) * 8 + 4 – 1
a[i, j, 3] dirección: [1, 1, 1] + (i-1) * 6 * 8 + (j-1) * 8 + 3 – 1

Por lo tanto, vemos que hay una parte común en el cálculo de las direcciones de los elementos de la matriz utilizada en la instrucción (llamémosle K), podríamos calcularlo primero y luego añadir la parte no común. Sería de la siguente forma:

K = [1, 1, 1] + (i-1) * 6 * 8 + (j-1) * 8

a[i, j, 5] dirección: K + 5 – 1
a[i, j, 6] dirección: K + 6 – 1
a[i, j, 4] dirección: K + 4 – 1
a[i, j, 6] dirección: K + 3 – 1

Realizar esto es especialmente importante en el caso de que la instrucción se encontrara en un bucle pues ahorraríamos mucho tiempo de ejecución.

4. **Reducción de potencia**.

El cálculo a = b^2 puede implementarse de dos formas distintas:

a) a = b ^ 2
b) a = b * b

Para la máquina, realizar el cálculo es distinto pues normalmente ejecutar la opción (a) es un trabajo "software" y la opción (b) habitualmente se realiza por "hardware", y por lo tanto es más rápido.

7.1 Algoritmo de Nakata.

Fue diseñado originalmente en el año 1964 (Anderson) y fue evolucionando hasta 1967 por los trabajos de Nakata. El objetivo es la optimización del uso de los registros minimizando el uso de variables temporales.

Dada una expresión aritmética, primero construímos el árbol (mediante el análisis sintáctico) donde los nodos son los operadores y las hojas son las variables o identificadores. Luego se etiqueta cada arista del árbol con un factor de peso, que inicialmente era el nº de acumuladores que necesitaba esa expresión, aunque el que veremos aquí llevará un factor relativo.

Partiendo de la raíz se toma aquella rama del árbol o subárbol que quede por analizar con el factor de peso más grande; en caso de igualdad se toma la rama de la derecha y continuaría hasta llegar al final de la expresión.

Algoritmo de etiquetado:

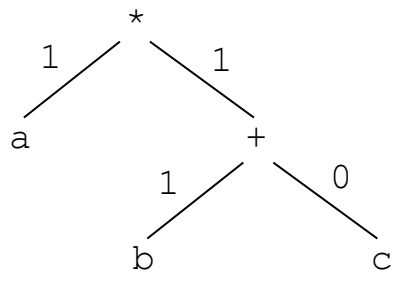
IF n es una hoja THEN
    IF n es el hijo más a la derecha de su padre THEN
        etiqueta (n) = 0
    ELSE
        etiqueta (n) = 1
ELSE
    Sean n_1, n_2, ..., n_k los hijos de n ordenados por etiqueta, de modo que etiqueta (n_1) ≥ etiqueta(n_2) ≥ ... ≥ etiqueta (n_k), hacemos:

$$\text{etiqueta}(n) = \max_{1 \le i \le k} (\text{etiqueta}(n_i) + i - 1)$$

NOTA: En caso de tratarse de un árbol binario, si los hijos tienen igual peso, el padre tendrá ese mismo peso + 1, si los hijos tienen distinto peso, el padre tendrá el peso del mayor.

Ejemplo.- a * (b + c)

El árbol de Nakata es:



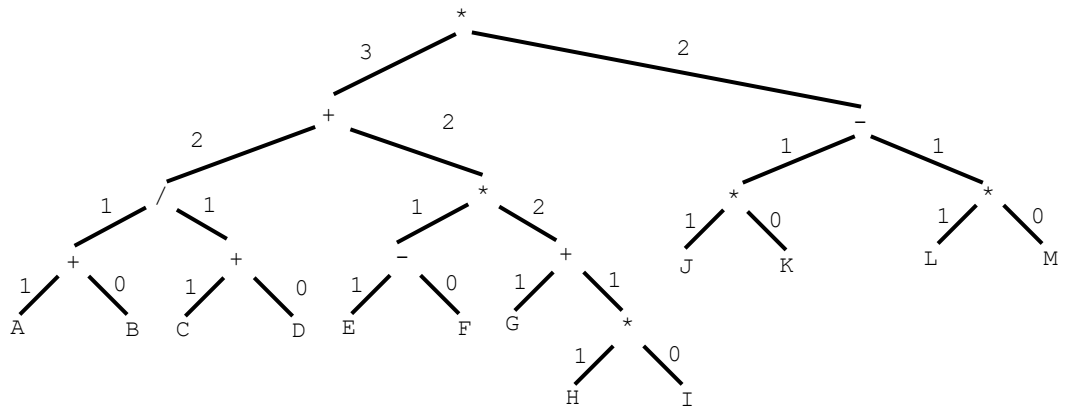
Haciendo el recorrido por Nakata, la expresión queda: (b + c) * a

Veamos ahora el código que generaría la expresión original y la expresión reordenada por Nakata:

a * (b + c)	(b + c) * a
MOV a, R0	MOV b, R0
MOV R0, T1	ADD c, R0
MOV b, R0	MUL a, R0
ADD c, R0	MOV R0, T1
MUL T1, R0	
MOV R0, T2	

Se puede observar como la operación generada por Nakata precisa menos código y menos variables temporales para su ejecución.

Ejemplo.- ((A + B) / (C + D) + (E - F) * (G + H * I)) * (J * K – L * M)



Recorremos el árbol de arriba hacia abajo, siguiendo la rama con mayor peso y obtenemos la siguiente expresión:

((((H * I) + G) * (E – F)) + ((C + D) / (A + B))) * ((L * M) – (J * K))

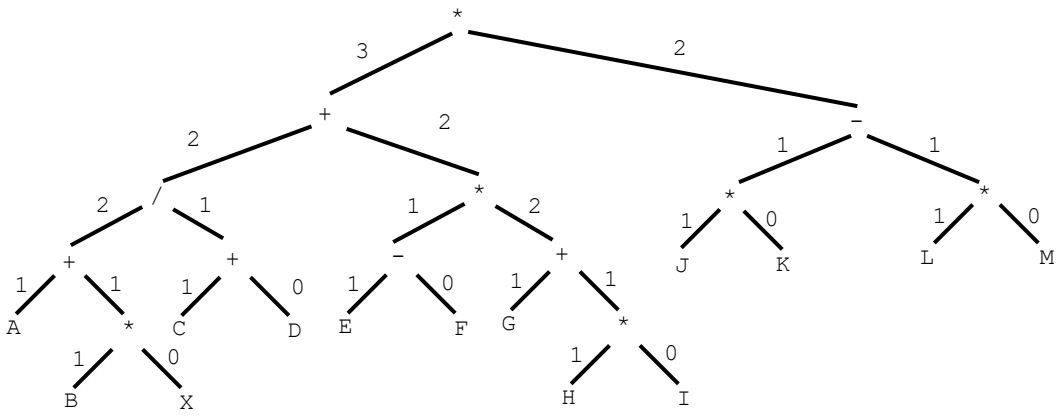
Nakata no tiene en cuenta la conmutatividad o no conmutatividad de las operaciones y nos ha cambiado una operación de división y una resta. Sin embargo, si generamos código suponiendo un acumulador, en este caso funciona por la definición que hemos hecho de nuestra máquina objeto:

MOV H, R0	MOV C, R0	MOV L, R0
MUL I, R0	ADD D, R0	MUL M, R0
ADD G, R0	MOV R0, T3	MOV R0, T5
MOV R0, T1	MOV A, R0	MOV J, R0
MOV E, R0	ADD B, R0	MUL K, R0
SUB F, R0	DIV T3, R0	SUB T5, R0
MUL T1, R0	ADD T2, R0	MUL T4, R0
MOV R0, T2	MOV R0, T4	MOV R0, T6

Ejemplo.- Vamos a hacer una modificación en la operación, de la forma siguiente:

((A + (B * X)) / (C + D) + (E - F) * (G + H * I)) * (J * K – L * M)

El árbol quedaría de la siguiente forma:



Recorremos el árbol de arriba hacia abajo, siguiendo la rama con mayor peso y obtenemos la siguiente expresión:

((((H * I) + G) * (E – F)) + (((B * X) + A) /(C + D))) * ((L * M) – (J * K))

Vemos que ahora ha realizado la división de forma “correcta”, sin embargo, al generar código con la máquina objeto que hemos definido, tenemos que intercambiar los operandos en la división. Veamos el código:

MOV H, R0	MOV B, R0	MOV L, R0
MUL I, R0	MUL X, R0	MUL M, R0
ADD G, R0	ADD A, R0	MOV R0, T6
MOV R0, T1	MOV R0, T3	MOV J, R0
MOV E, R0	MOV C, R0	MUL K, R0
SUB F, R0	ADD D , R0	SUB T6, R0
MUL T1, R0	**MOV R0, T4**	MUL T5, R0
MOV R0, T2	**MOV T3, R0**	MOV R0, T7
	DIV T4, R0	
	ADD T2, R0	
	MOV R0, T5	

NOTA: Las operaciones que están en negrita son las que nos permiten realizar la operación de división de forma correcta. Al hacer el recorrido del árbol sintáctico, llegados a “/” tiene que darse cuenta de que no es conmutativa e invertir los operandos para realizarlo correctamente.

Ejercicio.- Vamos a optimizar el siguiente código:

```
FOR I=L0 TO L1 DO BEGIN
        FOR J=L2 TO L3 DO BEGIN
                M[I] =X1 * X2 * (X3+X4)
                N[I,J] = M[I] ** 2 * D
        END;
END;
```

Vemos que una gran parte del código se puede ejecutar fuera de los bucles, o por lo menos una gran parte. Además, podemos optimizar por Nakata los cálculos y transformar el "elevado al cuadrado" por una simple multiplicación. Nos quedaría los siguiente:

```
K1 = (X3 + X4) * X1 * X2
K2 = K1 * K1 * D
FOR I=L0 TO L1 DO BEGIN
        M[I] = K1
        FOR J=L2 TO L3 DO BEGIN
                M[I,J] = K2
        END;
END;
```

Vamos a generar código en cuartetos:

1. (+, X3, X4, T1)
2. (*, T1, X1, T2)
3. (*, T2, X2, T3)
4. (=, T3, , K1)
5. (*, K1, K1, T4)
6. (*, T4, D, T5)
7. (=, T5, , K2)
8. (=, L0, , I)
9. (JL, 20, L1, I)
10. (=, K1, , M[I])
11. (=, L2, , J)
12. (JL, 17, L3, J)
13. (=, K2, , M[I,J])
14. (+, J, 1, T7)
15. (=, T7, , J)
16. (JP, 12, ,)
17. (+, i, 1, T8)
18. (=, T8, , I)
19. (JP, 9, ,)
20. (END, , ,)

NOTA: Es preciso hacer notar que no hemos generado el código de forma estricta pues no hemos calculado las direcciones de la matrices. Además, vemos que, en los bucles con cuartetos, hemos realizado la comprobación al principio del bucle "FOR", con lo que, un FOR I=2 TO 1 nunca se ejecutaría.

7.2 Ejemplo de optimización manual.

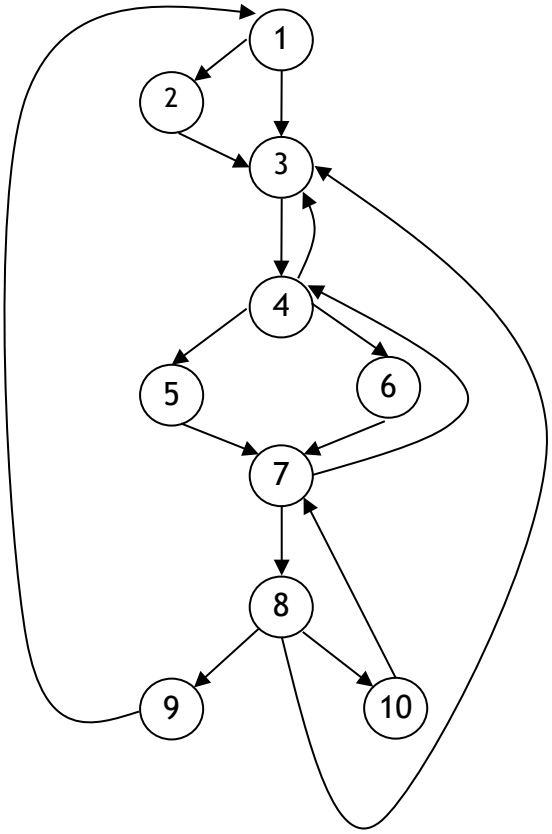
--- Libro: Compiladores (**primera edición**), Aho-Ullman Pag 606-616 ---

7.3 Lazos en los grafos de flujo.

Vamos a comenzar con algunas definiciones necesarias para este apartado.

Definición: decimos que un nodo d de un grafo de flujo **domina** al nodo n (escribimos "d domina n") si todo camino desde el nodo inicial del grafo de flujo a n pasa por d.

NOTA: hay que tener en cuenta que todo nodo se domina a sí mismo y que el nodo inicial domina a todos.

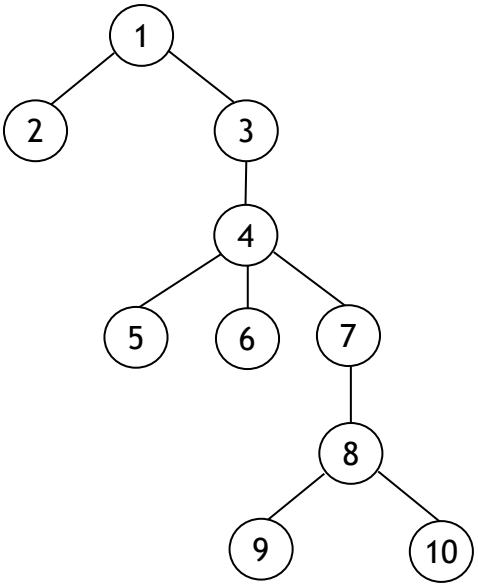


Ejemplo.-

Aquí podemos ver como:

El nodo 1 domina a todos los demás nodos.
El nodo 2 sólo se domina a sí mismo.
El nodo 3 domina a todos excepto al 1 y el 2.
El nodo 4 domina a todos menos al 1, 2 y el 3.
El nodo 5 y el nodo 6 sólo se dominan a ellos mismos.
El nodo 7 domina al 7, 8, 9 y 10
El nodo 8 domina al 8, 9 y 10.
El nodo 9 y el nodo 10 sólo se dominan a sí mismo.

Vamos a ver ahora el denominado “árbol de dominación”, que es la representación gráfica de la información anterior.



Una de las aplicaciones principales de la información sobre dominadores es determinar los lazos de un grafo de flujo. Los lazos, para poder asumir ciertas condiciones iniciales que posibiliten la optimización, tienen que cumplir dos propiedades importantes (denominándose “ciclos naturales”):

1. Un lazo tendrá un solo punto de entrada (encabezamiento) y además este punto de entrada domina a todos los nodos dentro del lazo, será su única entrada.
2. Debe existir al menos una forma de iterar el lazo, es decir, por lo menos existirá un camino hacia el encabezamiento.

Para localizar los lazos en un grafo de flujo, un método es localizar aristas cuyas cabezas dominen a sus colas. Esto es, si tenemos una arista de la forma: a $\rightarrow$ b (b es la cabeza y a es la cola), si sabemos que b domina a entonces significa que ahí hay un lazo.

En el grafo anterior podemos ver que: Tenemos una arista de 7 a 4 y 4 domina 7, de la misma forma, hay una arista de 10 a 7 y 7 domina 10. Los otros lazos que nos encontramos aparece con las aristas de 4 a 3, de 8 a 3 y de 9 a 1.

7.4 Análisis global del flujo de datos.

Para realizar el análisis del flujo de datos y, por lo tanto, realizar una buena optimización, precisamos tener información disponible sobre las variables, expresiones, etc, que será el punto de partida de la optimización.

Nos centraremos especialmente en los bucles, un punto en donde las optimizaciones suelen ser importantes.

Con el análisis de flujo de datos conseguiremos eliminar código inactivo, obtener subexpresiones comunes, etc.

La información del flujo de datos se obtendrá resolviendo ecuaciones que relacionan la información en varios puntos de un programa. Veamos la siguiente ecuación:

$$sal[S] = gen[S] \cup (ent[S] - desact[S])$$

Dependiendo de lo que pretendamos optimizar, variarán las nociones de "generar" y "desactivar".

Es perciso hacer notar que el análisis se realiza habitualmente a nivel de bloque y no de proposición, cuando hablamos de sal[S] estamos hablando de un punto final único, algo que sí existe en los bloques.

Hay casos especiales como son los punteros y las matrices, los cuales hacen más complicado el análisis. También nos pueden complicar el análisis las llamadas a procedimientos. Por ejemplo:

```
x = 5
...
call función(x)
...
x = 6
```

Cuando llamamos a la función puede haber ocurrido que el valor de x cambie.

En cualquier programa consideraremos que existen puntos entre 2 proposiciones (cada proposición tiene, al menos, un punto antes y después).

Un **camino** de un punto p_1 a un punto p_n es una secuencia de puntos p_1, p_2,..., p_{n-1}, de forma que p_i es que punto que precede a una proposición y p_{i+1} el que la sigue en el mismo bloque, o bien p_i es el final de un bloque y p_{i+1} es el comienzo del bloque siguiente.

Si, para comenzar, realizamos un "análisis de alcance máximo de definiciones", llamamos **definición de una variable x** a una proposición que asigna un valor a **x**, o puede asignarlo.

Serán *definiciones no ambiguas*: x=7, scanf("%d", fpepe), lectura de fichero, etc. Serán *ambiguas* las llamadas a procedimientos con x como parámetro (por dirección, no por valor) o sin la variable x como parámetro, pero está en su alcance.

Llamamos **alcance de una definición d a un punto p**, si existe un camino desde el siguiente punto a d hasta p tal que la definición no se desactive. Estamos

hablando de “alcance de definiciones” y, por aclarar, se trata de analizar el “alcance **máximo** de las definiciones”, hasta dónde pueden llegar.

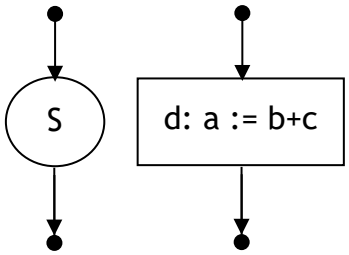
Decimos que una **definición se desactiva** si durante el camino a otro punto se realiza otra definición del mismo elemento.

7.4.1 Alcance *máximo* de definiciones en estructuras de control.

Para los ejemplos de este apartado utilizaremos la siguiente sintáxis:

S → **id** := E | S ; S | **if** E **then** S **else** S | **do** S **while** E
E → **id** + **id** | **id**

Vamos a ver las definiciones inductivas, o dirigidas por la sintaxis, de los conjuntos ent[S], sal[S] y genera[S] y desact[S] para las diferentes estructuras:

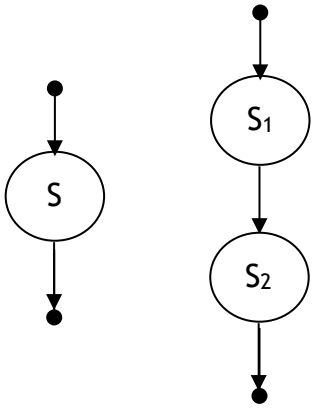


gen[S] = {d}
desact [S] = D_a – {d}

sal[S] = gen[S] ∪ (ent[S] – desact[S])

Aquí estamos diciendo que lo que se genera en esa proposición es la definición que hemos llamado “d”. Lo que se desactiva son todas la definiciones que haya en el programa de la variable “a” menos la de la propia proposición.

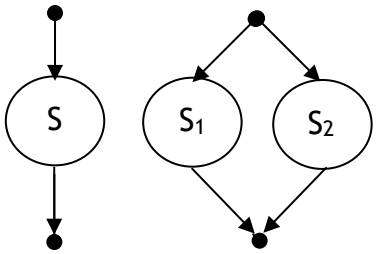
Lo que sale de S es lo que generamos dentro (la definición de “a”) unido a todo lo que entra menos lo que desactivamos.



gen [S] = gen[S_2] ∪ (gen[S_1] – desact[S_2])
desact[S] = desact[S_2] ∪ (desact[S_1] – gen[S_2])

ent[S_1] = ent[S]
ent[S_2] = sal[S_1]
sal[S] = sal[S_2]

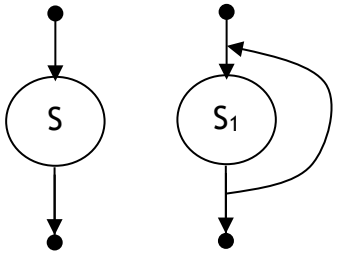
Lo que se genera en S_2 ya es lo que genera S, además de lo que genere S_1, siempre y cuando no sea desactivado por S_2. Idem con la desactivación.



gen [S] = gen[S_1] ∪ gen[S_2]
desact[S] = desact[S_1] ∩ desact[S_2]

ent[S_1] = ent[S]
ent[S_2] = ent[S]
sal[S] = sal[S_1] ∪ sal[S_2]

Aquí utilizamos una política conservadora, asumimos todo lo de S_1 y S_2 (cuando realmente se ejecuta uno de los dos) y en la desactivación, sólo consideramos que se ha desactivado lo que realmente se desactiva en los dos (pase por el que pase). Esto es correcto si pretendemos ver, por ejemplo, el rango de valores que puede tomar una variable. Si vemos que todas las definiciones en un punto determinado después del "if" son siempre x=9 (por las dos ramas), podemos utilizar 9 en lugar de x. Para este tipo de optimizaciones no hacerlo así nos produciría optimizaciones incorrectas. Si lo que pretendemos es utilizar el valor de la asignación hacerlo así no sería correcto pues "es posible esa rama no se haya ejecutado".



gen[S] = gen[S_1]
desact [S] = desact[S_1]

ent[S_1] = ent[S] ∪ sal[S_1]
sal[S] = sal[S_1]

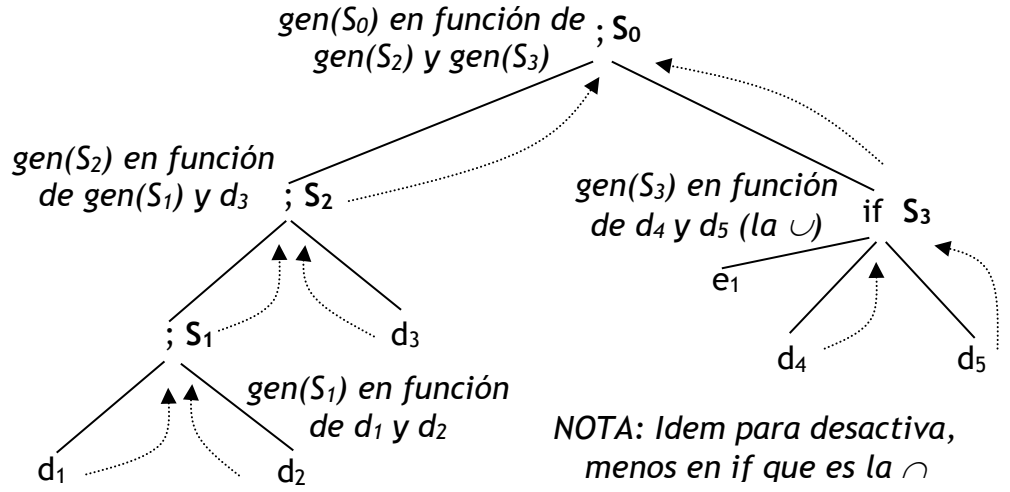
Aquí, la existencia del lazo no afecta a lo que se genera o desactiva en S, así como su salida.

Aquí puede observarse como la ent[S] no es la misma que la ent[S1], y mediante la ecuación anterior, o sea ent[S_1] = ent[S] ∪ sal[S_1], no se puede calcular la entrada sin tomar en cuenta primero la salida.

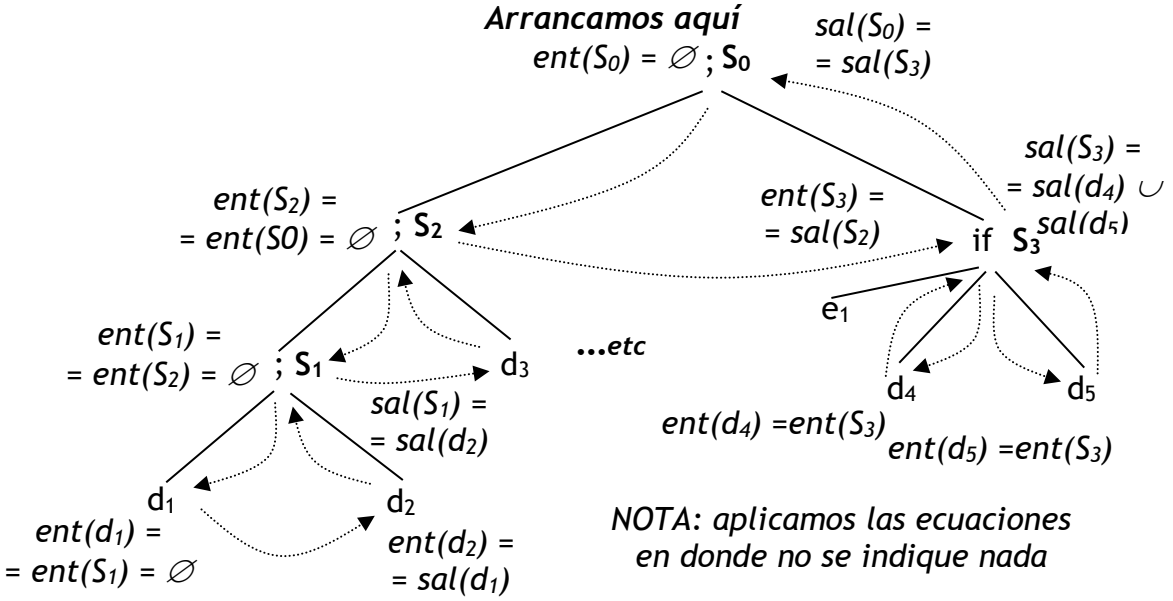
1.1.1.1 Recorridos en el árbol para el alcance máximo de definiciones.

Veamos como obtener los atributos en el árbol sintáctico y su cálculo mediante recorridos en el árbol mediante un ejemplo:

1º. Calculamos genera y desactiva de forma sintetizada con un primer recorrido ascendente de árbol sintáctico.



2º. Partimos de que ent[S_0] = ∅, siendo S_0 el programa completo (a la entrada del programa completo no hay ninguna definición. A partir de ahí recorremos el árbol en profundidad de izquierda a derecha y vamos obteniendo la entrada de forma heredada, y calculando la salida de forma sintetizada escalando en el árbol.



Como ya se ha indicado, es muy importante darse cuenta de que, en el caso de los lazos (do), esta secuencia no es aplicable porque ent[S_1] = ent[S] ∪ sal[S_1], es decir, necesitamos la salida para conseguir la entrada. Luego veremos en este caso como se soluciona con una definición equivalente.

Para resolver el problema con los lazos, existe una definición equivalente para ent[S_1] que nos va a servir para realizar los cálculos normalmente (primero calculamos de forma sintetizada genera y desactiva y luego, heredando la entrada, calculamos la salida), que es:

$ent[S_1] = ent[S] \cup gen[S_1]$

Vamos a demostrarlo. Partimos de que:

$ent[S_1] = ent[S] \cup sal[S_1]$
$sal[S_1] = gen[S_1] \cup (ent[S_1] - desact[S_1])$

vamos a escribirlo como:

$E1 = E \cup X1$
$X1 = G1 \cup (E1 - D1)$

E1 y X1 son variables, las otras tres son constantes.

Al comenzar la primera iteración, X1 = 0 y como entrada de la primera iteración tenemos:

$E1^1 = E$

La salida de la primera iteración es:
$X1^1 = G1 \cup (E1^1 - D1) = G1 \cup (E - D1)$

La entrada de la segunda iteración es:
$E1^2 = E \cup X1^1 = E \cup G1 \cup (E - D1) = E \cup G1$

La salida de la segunda iteración es:
$X1^2 = G1 \cup (E1^2 - D1) = G1 \cup (E \cup G1 - D1) = G1 \cup (E - D1)$

Si continuamos con las iteraciones veremos que no hay variación alguna. Por lo tanto, hemos demostrado que:

$ent[S_1] = ent[S] \cup gen[S_1]$

7.4.2 Notación vectorial para representar genera y desactiva.

La notación que utilizaremos para los conjuntos **genera** y **desactiva** será en formato vectorial:

$V = (a_1, a_2, ..., a_n)$

n: número de definiciones
Si $a_i = 0$ significa que se ha generado o desactivado (según sea uno u otro vector) una definición, si su valor es 1 es todo lo contrario.

Con esta representación, los cálculos se realizarán de la siguiente forma:

A - B se calcula como	VA /\ ¬ VB	(AND lógico con VB negado)
A ∪ B se calcula como	VA \/ VB	(OR lógico entre VA y VB)

A $\cap$ B se calcula como VA /\ VB (AND lógico entre VA y VB)

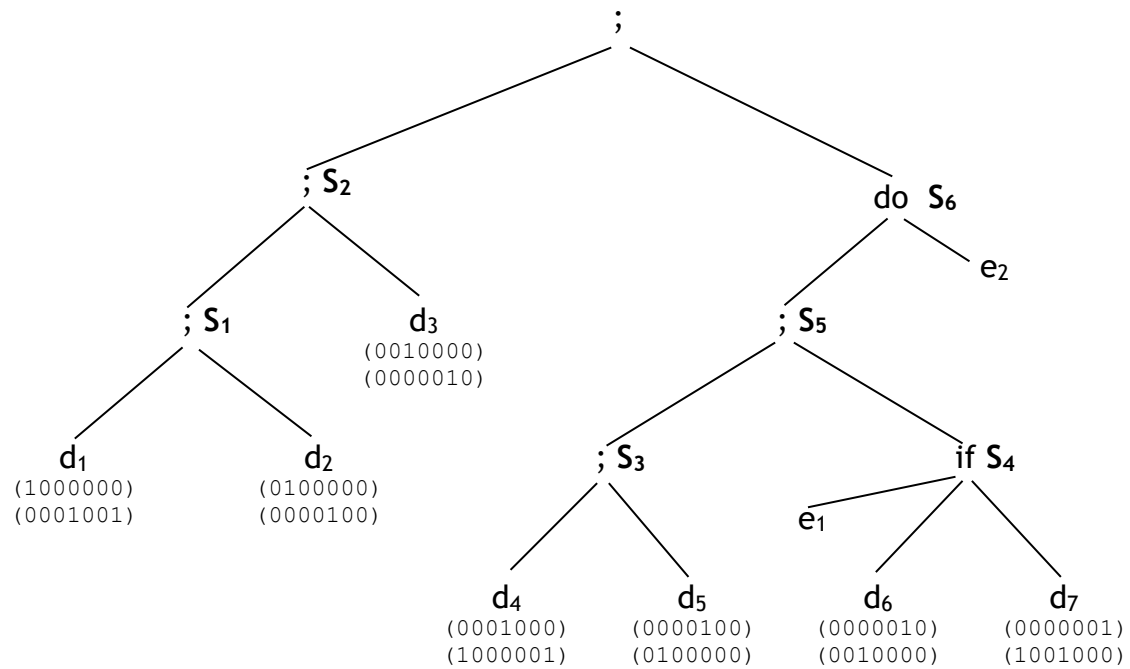
Ejemplo.-

```
/* d1 */ i := m-1;
/* d2 */ j := n;
/* d3 */ a := u1;
do
/* d4 */       i := i+1;
/* d5 */       j := j-1;
               if e1 then
/* d6 */               a := u2;
               else
/* d7 */               i := u3;
while e2
```

NOTA: con d_1, d_2, ... , d_n indicamos las definiciones que se realiza en esa proposición (ese será el orden utilizado en los vectores **genera** y **desactiva**).

También es necesario hacer notar que ambos conjuntos son atributos sintetizados, que se obtienen de forma ascendente.

Veamos el árbol sintáctico, con las definiciones en las hojas, y los vectores genera y desactiva:



gen[S_1] = gen[d_2] $\cup$ (gen[d_1] – desact[d_2]) =
= (0100000) $\cup$ ((1000000) – (0000100)) = (1100000)

desact[S_1] = desact [d_2] $\cup$ (desact[d_1] – gen[d_2]) =
= (0000100) $\cup$ ((0001001) – (0100000)) = (0001101)

$gen[S_2] = gen[d_3] \cup (gen[S_1] - desact[d_3]) =$
$= (0010000) \cup ((1100000) - (0000010)) = (1110000)$

$desact[S_2] = desact\ [d_3] \cup (desact[S_1] - gen[d_3]) =$
$= (0000010) \cup ((0001101) - (0010000)) = (0001111)$

$gen[S_3] = gen[d_5] \cup (gen[d_4] - desact[d_5]) =$
$= (0000100) \cup ((0001000) - (0100000)) = (0001100)$

$desact[S_3] = desact[d_5] \cup (desact[d_4] - gen[d_5]) =$
$= (0100000) \cup ((1000001) - (0000100)) = (1100001)$

$gen\ [S_4] = gen[d_6] \cup gen[d_7] = (0000010) \cup (0000001) = (0000011)$
$desact[S_4] = desact[d_6] \cap desact[d_7] = (0000000)$

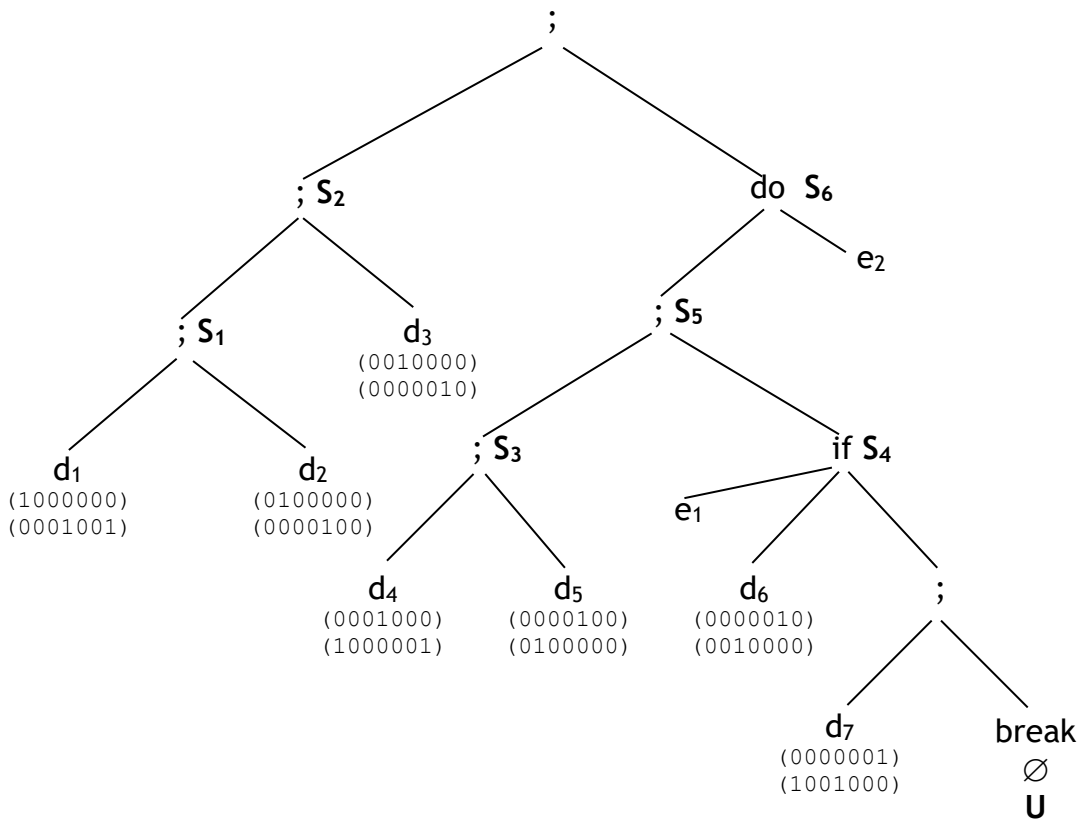
$gen[S_5] = gen[S_4] \cup (gen[S_3] - desact[S_4]) =$
$= (0000011) \cup ((0001100) - (0000000)) = (0001111)$

$desact(S_5) = desact\ [S_4] \cup (desact[S_3] - gen[S_4]) =$
$= (0000000) \cup ((1100001) - (0000011)) = (1100000)$

$gen[S_6] = gen[S_5] = (0001111)$
$desact[S_6] = desact[S_5] = (1100000)$

Vamos a realizar una modificación en nuestro código, introduciendo una instrucción **break** dentro del lazo do-while (salta al final del lazo), de la siguiente forma:

```
/* d1 */ i := m-1;
/* d2 */ j := n;
/* d3 */ a := u1;
do
/* d4 */       i := i+1;
/* d5 */       j := j-1;
               if e1 then
/* d6 */               a := u2;
               else begin
/* d7 */               i := u3;
                       break
               end
while e2
```



Para realizar los cálculos, donde esta la instrucción **break** decimos que no se genera nada (∅), en nuestro ejemplo (0000000) y se desactiva todo (**U**), en nuestro ejemplo (1111111) y realizamos los cálculos normalmente. Esta es una postura también conservadora porque no se puede llegar al final de una secuencia de proposiciones que finalice con una proposición break.

7.5 Solución iterativa de las ecuaciones de flujo de datos.

En los siguientes apartados, presentamos métodos iterativos rápidos para realizar tres tipos de análisis, partiendo de la información a nivel de bloques (secuencias de instrucciones).

7.5.1 Análisis de alcance *máximo* de definiciones.

Vamos a definir bloques (B) básicos, considerando cada bloque como una proposición que es cascada de una o varias proposiciones de asignación. También se definen sal[B], gen[B], desact[B] y ent[B] de la misma forma que en los apartados anteriores.

Algoritmo iterativo para alcance de definiciones:

Partimos de que se ha calculado gen[B] y desact[B] para cada bloque B. Y además partimos de que:

ent[B] = $\cup_P$ sal[P] La unión de los P, bloques predecesores de B.

sal[B] = gen[B] $\cup$ (ent[B] – desact[B])

Algoritmo:

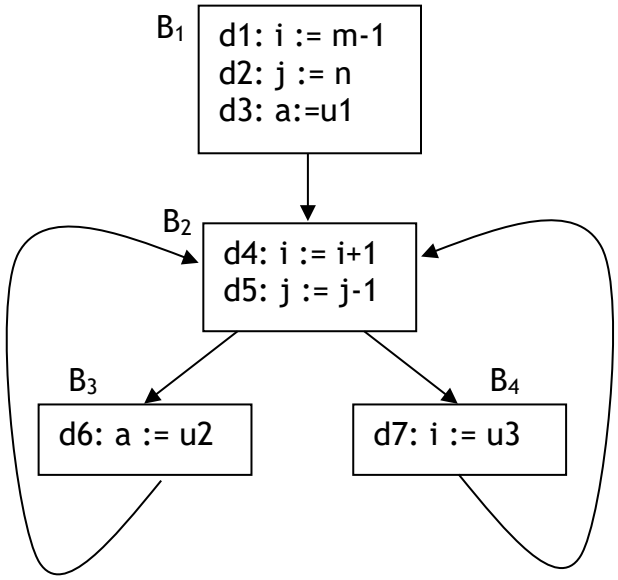
Entrada: genera y desactiva para cada B_i del grafo.
Salida: ent[B_i], sal[B_i]

```
ent[Bi] = ∅
for cada Bi do sal[Bi] = gen[Bi];
cambio := true;
while cambio do begin
        cambio := false;
        for cada bloque B do begin
                ent[Bi] := ∪P sal[P];
                salant := sal[Bi];
                sal[Bi] := gen[Bi] ∪ (ent[Bi] – desact[Bi]);
                if sal[Bi] ≠ salant then cambio:= true
        end
end
```

Ejemplo.-

```
/* d1 */ i := m-1;
/* d2 */ j := n;
/* d3 */ a := u1;
do
/* d4 */       i := i+1;
/* d5 */       j := j-1;
               if e1 then
/* d6 */               a := u2;
               else
/* d7 */               i := u3;
while e2
```

Vamos a ver el grafo de flujo:



$gen[B_1] = \{d_1, d_2, d_3\}$
$desact[B_1] = \{d_4, d_5, d_6, d_7\}$

$gen[B_2] = \{d_4, d_5\}$
$desact[B_2] = \{d_1, d_2, d_7\}$

$gen[B_3] = \{d_6\}$
$desact[B_3] = \{d_3\}$

$gen[B_4] = \{d_7\}$
$desact[B_4] = \{d_1, d_4\}$

En formato vectorial, los conjuntos genera y desactiva son:

BLOQUE	**genera**	**desactiva**
B1	1110000	0001111
B2	0001100	1100001
B3	0000010	0010000
B4	0000001	1001000

Vamos a ver como calcular las entradas en los distintos bloques según sus predecesores:

$ent[B_1] = \varnothing$
$ent[B_2] = sal[B_1] \cup sal[B_3] \cup sal[B_4]$
$ent[B_3] = sal[B_2]$
$ent[B_4] = sal[B_2]$

Si aplicamos el algoritmo:

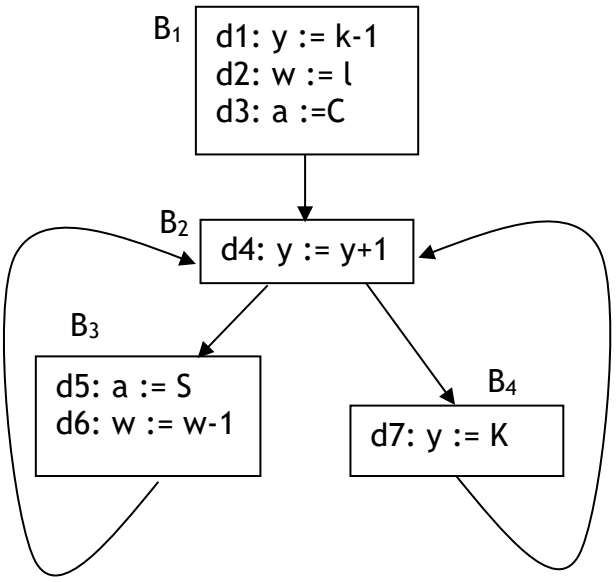
	Inicialmente		*Iteración 1*		*Iteración 2*	
BLOQUE	**ent**	**sal**	**ent**	**sal**	**ent**	**sal**
B1	∅	1110000	∅	***1110000***	∅	***1110000***
B2	∅	0001100	1110011	***0011110***	1111111	***0011110***
B3	∅	0000010	*0011110*	***0001110***	0011110	***0001110***
B4	∅	0000001	*0011110*	***0010111***	0011110	***0010111***

Las casillas en cursiva y negrita, conjuntos salida, tienen el mismo valor, con lo cual paramos el algoritmo y ya tenemos la entrada y la salida para cada uno de los bloques. NOTA: La entrada de la Iteración 1 en B3 y B4 (cursiva) se puede tomar en relación a la salida de B2 de esa misma iteración o de la anterior (estrictamente debería tomarse de la anterior pero no afecta al resultado).

Ejemplo.-

```
/* d1 */      y := k-1;
/* d2 */      w := l;
/* d3 */      a := C;
              do
/* d4 */              y := y+1;
                      if condicion1 then
/* d5 */                                      a := S;
/* d6 */                                      w := w-1;
                      else
/* d7 */                      y := K;
              while condicion2
```

El grafo de flujo sería el siguiente:



$gen[B_1] = \{d_1, d_2, d_3\}$
$desact[B_1] = \{d_4, d_5, d_6, d_7\}$

$gen[B_2] = \{d_4\}$
$desact[B_2] = \{d_1, d_7\}$

$gen[B_3] = \{d_5, d_6\}$
$desact[B_3] = \{d_3, d_2\}$

$gen[B_4] = \{d_7\}$
$desact[B_4] = \{d_1, d_4\}$

En formato vectorial, los conjuntos genera y desactiva son:

BLOQUE	genera	desactiva
B1	1110000	0001111
B2	0001000	1000001
B3	0000110	0110000
B4	0000001	1001000

Vamos a ver como calcular las entradas en los distintos bloques según sus predecesores:

$ent[B_1] = \varnothing$
$ent[B_2] = sal[B_1] \cup sal[B_3] \cup sal[B_4]$
$ent[B_3] = sal[B_2]$
$ent[B_4] = sal[B_2]$

Si aplicamos el algoritmo:

	Inicialmente		*Iteración 1*		*Iteración 2*	
BLOQUE	**ent**	**Sal**	**Ent**	**sal**	**ent**	**sal**
B1	∅	1110000	∅	***1110000***	∅	***1110000***
B2	∅	0001000	1110111	***0111110***	1111111	***0111110***
B3	∅	0000110	0111110	***0001110***	0111110	***0001110***
B4	∅	0000001	0111110	***0110111***	0111110	***0110111***

Las casillas en cursiva y negrita, conjuntos salida, tienen el mismo valor, con lo cual paramos el algoritmo y ya tenemos la entrada y la salida para cada uno de los bloques.

7.5.2 Análisis de expresiones disponibles.

Una expresión x+y está disponible en un punto p si todo camino, no necesariamente sin lazos, desde el nodo inicial hasta p evalua x+y y después de la última evaluación antes de p no hay asignaciones posteriores de x ni de y.

Un bloque B_i desactiva una expresión x+y si asigna un valor a x o a y.

Un bloque B_i genera una expresión x+y si evalua x+y y no genera una redefinición posterior de x o de y.

e_gen[B_i] es el conjunto de las expresiones generadas en el bloque B_i.
e_desact[B_i] es el conjunto de las expresiones desactivadas en el bloque B_i.

U es el conjunto universal , todas las expresiones que hay en el programa.
∅ es el complementario de **U**.

Las ecuaciones a aplicar son:

$sal[B_i] = e\_gen[B_i] \cup (ent[B_i] - e\_desact[B_i])$
$ent[B_i] = \cap_p sal[P_i]$ si Bi no es el bloque inicial
$ent[B_1] = \varnothing$ siendo B_1 el bloque inicial

Aquí utilizamos la $\cap$ de la salida de los predecesores porque para que una expresión esté disponible en un bloque tiene que salir de todos sus predecesores (una vez más, utilizamos una estrategia conservadora).

Ejemplo.-

(1) a = b + c
-1- Disponible la expresión 1
(2) b = a + d
-2- Disponible la expresión 2 (se ha cambiado b y la 1 ya pasa a no estar disponible)

(3) c = b + c

-3- Disponible la expresión 2 (la 3 ya no es disponible inmediatamente pues hemos cambiado el valor de c)

(4) d = a – d

-4- Ninguna es disponible (se ha cambiado el valor de d y la dos ya no está disponible y la propia expresión 4 ya cambia inmediatamente el valor de d).

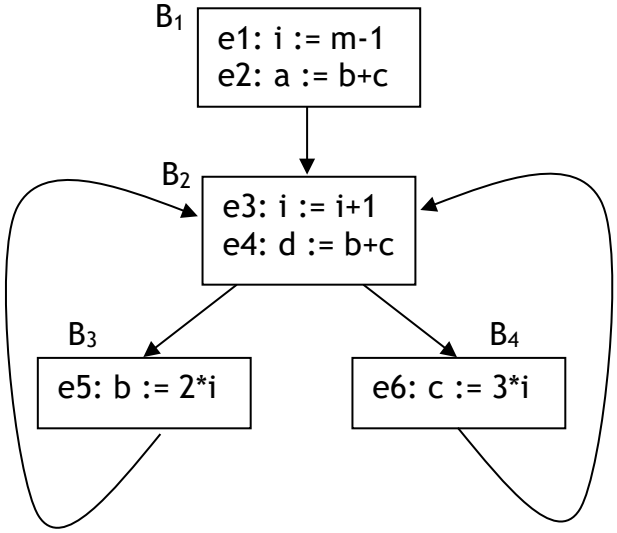
Algoritmo iterativo para expresiones disponibles:

ENTRADA: Grafo de flujo G, e_genera[B_i], e_desactiva[B_i]
ent[B_i] = ∅; /* B_1 es el bloque inicial */
sal[B_1] = e_gen[B_1];
for B ≠ B_1 do sal[B] = **U** – e_desact[B];
cambio = true;

```
while cambio do begin
        cambio = false
        for B ≠ B1 do begin
                ent[B] = ∩p sal[P];
                salant = sal[B];
                sal[B] = e_gen[B] ∪ (ent[B] – e_desact[B]);
                if salant ≠ sal[B] then cambio = true;
        end;
end;
```

Ejemplo.-

```
/* e1 */            i := m-1;
/* e2 */            a := b+c;
            do
/* e3 */            i := i+1;
/* e4 */            d := b+c;
            i       f e1 then
/* e5 */                    b := 2*i;
                    else
/* e6 */                    c := 3*i;
            while e2
```



$e\_gen[B_1] = \{e_1, e_2\}$
$e\_desact[B_1] = \{e_3, e_5, e_6\}$

$e\_gen[B_2] = \{e_4\}$
$e\_desact[B_2] = \{e_3, e_5, e_6\}$

$e\_gen[B_3] = \{e_5\}$
$e\_desact[B_3] = \{e_2, e_4\}$

$e\_gen[B_4] = \{e_6\}$
$e\_desact[B_4] = \{e_2, e_4\}$

En formato vectorial, e_genera y e_desactiva son:

BLOQUE	**e_genera**	**e_desactiva**
B1	110000	001011
B2	000100	001011
B3	000010	010100
B4	000001	010100

Vamos a ver como calcular las entradas en los distintos bloques según sus predecesores:

$ent[B_1] = \varnothing$
$ent[B_2] = sal[B_1] \cap sal[B_3] \cap sal[B_4]$
$ent[B_3] = sal[B_2]$
$ent[B_4] = sal[B_2]$

Para las salidas aplicamos:

$sal[B_i] = e\_gen[B_i] \cup (ent[B_i] - e\_desact[B_i])$

	Inicialmente		*Iteración 1*		*Iteración 2*	
BLOQUE	**ent**	**Sal**	**Ent**	**sal**	**ent**	**sal**
B1	∅	110000	∅	***110000***	∅	***110000***
B2	∅	110100	100000	***100100***	100000	***100100***
B3	∅	101011	110100	***100010***	100100	***100010***
B4	∅	101011	110100	***100001***	100100	***100001***

En las dos últimas iteraciones vemos que **sal** se mantiene, entonces paramos el algoritmo y ya tenemos las expresiones que entran y salen en cada bloque.

7.5.3 Análisis de variables activas.

Una variable a es activa en un punto p si se utiliza el valor a en algún camino que comience en p, en caso contrario se dirá que a está desactiva.

Como se puede deducir, el análisis en este caso se realiza en dirección opuesta al flujo de ejecución del programa. Como sabemos que una variable está activa en un punto del programa si se usa en un punto posterior, esa información la recogeremos analizando el programa en este sentido.

Entrada es el conjunto de variables activas al comienzo del bloque.

Salida es el conjunto de variables activas a la salida del bloque.

Definidas son el conjunto de variables a las que se les ha asignado definitivamente un valor en en bloque.

Uso son el conjunto de variables cuyos valores se utilizan antes de cualquier definición de la variable.

Las ecuaciones a aplicar son las siguientes:

$ent[B_i] = uso[B_i] \cup (sal[B_i] - def[B_i])$
$sal[B_i] = \cup_S ent[S_i]$

La primera ecuación nos indica que una variable está activa al entrar en un bloque si se utiliza antes de una redefinición en el bloque o si está activa al salir del bloque y dentro no se redefine.

La segunda ecuación nos indica que una variable está activa al salir de un bloque si, y sólo si, está activa al entrar en uno de sus sucesores.

Algoritmo iterativo de análisis de variables activas:

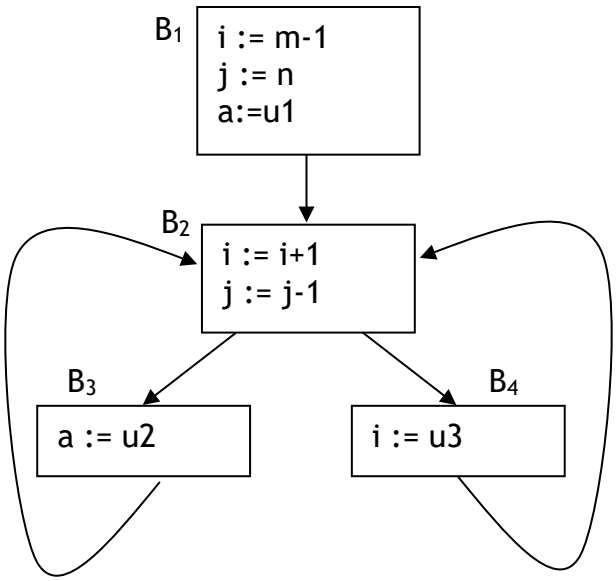
Entrada: Un grafo de flujo G, $def[B_i]$ y $uso[B_i]$
Salida: sal[Bi], es decir, el conjunto de variables activas a la salida de cada bloque B del grafo de flujo.

```
for cada bloque B do sal[B] := ∅;
for cada bloque B do ent[B] := uso[B];
while ocurren cambios en los conjuntos ent do
        for cada bloque B do begin
                sal[B] := ∪S ent[S];
                ent[B] := uso[B] ∪ (sal[B] – def[B])
        end
end
```

Ejemplo.-

```
i := m-1;
j := n;
a := u1;
do
        i := i+1;
        j := j-1;
                if e1 then
                        a := u2;
                else
                        i := u3;
while e2
```

Vamos a ver el grafo de flujo:



$def[B_1] = \{i, j, a\}$
$uso[B_1] = \{m, n, u1\}$

$def[B_2] = \{i, j\}$
$uso[B_2] = \{i, j\}$

$def[B_3] = \{a\}$
$uso[B_3] = \{u2\}$

$def[B_4] = \{i\}$
$uso[B_4] = \{u3\}$

Para la notación vectorial, vamos a utilizar el orden de aparición de las variables en el programa, esto es:

(i, m, j, n, a, u1, u2, u3)

En formato vectorial, los conjuntos definición y uso son:

BLOQUE	definición	uso
B1	10101000	01010100
B2	10100000	10100000
B3	00001000	00000010
B4	10000000	00000001

Vamos a ver como calcular las salidas en los distintos bloques según sus sucesores:

$sal[B_1] = ent[B_2]$
$sal[B_2] = ent[B_3] \cup ent[B_4]$
$sal[B_3] = ent[B_2]$
$sal[B_4] = ent[B_2]$

Si aplicamos el algoritmo:

	Inicialmente		*Iteración 1*		*Iteración 2*	
BLOQUE	**sal**	**Ent**	**sal**	**ent**	**sal**	**ent**
B1	∅	01010100	10100000	01010100	10100011	***01010111***
B2	∅	10100000	00000011	10100011	10100011	***10100011***
B3	∅	00000010	10100000	10100010	10100011	***10100011***
B4	∅	00000001	10100000	00100001	10100011	***00100011***

	Iteración 3	
BLOQUE	**sal**	**ent**
B1	10100011	***01010111***
B2	10100011	***10100011***
B3	10100011	***10100011***
B4	10100011	***00100011***

Las casillas en cursiva y negrita, conjuntos entrada, tienen el mismo valor, con lo cual paramos el algoritmo y ya tenemos la entrada y la salida para cada uno de los bloques.

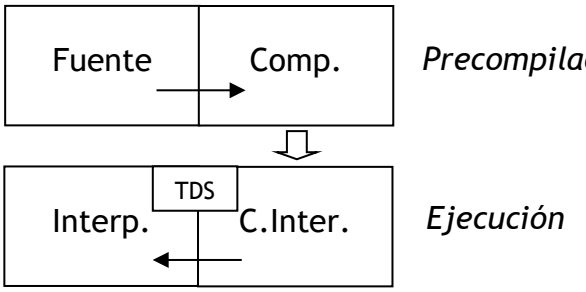
8 INTÉRPRETES.

Los intérpretes, en la primera generación de ordenadores, han tenido mucho éxito por los problemas de memoria existentes. Muchas veces el programa fuente y el intérprete ocupaban memos memoria que el programa fuente y el compilador e incluso que el objeto.

Hoy en día han caído en desuso debido precisamente a la misma causa, hoy no tenemos problemas con la memoria de los ordenadores.

Además tienen el problema de la detección de errores. Puede llegarse a un punto del programa en que exista un error y todo el trabajo anterior no sirve para nada.

Por ello, se optó por una opción intermedia, se realiza una compilación previa a un lenguaje intermedio, que además servirá para detectar errores. Luego el intérprete actúa sobre este lenguaje intermedio. El esquema es el siguiente:



Es decir, tenemos dos tipos de intérprete:

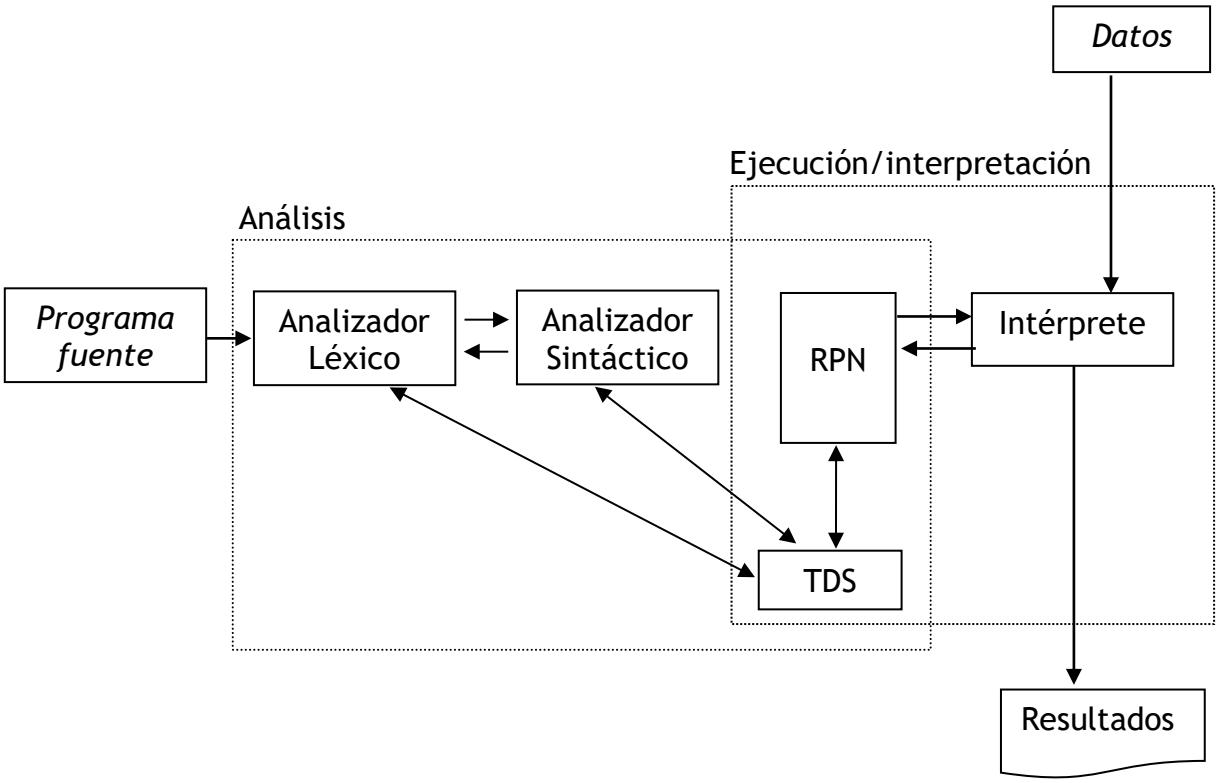
a) Intérprete puro.- En el que no hay precompilación.
b) Inérprete actual.- Tenemos un código intermedio que hará el reconocimiento léxico, sintáctico, etc. Posteriormente el módulo de interpretación coge línea a línea el código intermedio y lo va ejecutando.

8.1 Estructura de un intérprete actual.

En un intérprete actual tendremos una primera fase de análisis que será similar a la que nos encontramos en los compiladores.

Posteriormente se realiza la fase de interpretación y ejecución, que utilizará la TDS generada anteriormente y en la que hemos supuesto la utilización de un lenguaje intermedio en RPN.

Veamos el siguiente esquema:



Ejemplo.-

Un programa en BASIC:

```
A=9 ;
B=8 ;
B=B+A ;
WRITE B ;
GOTO 1000
```

Se traduce a código intermedio en RPN, en forma compacta, de la siguiente forma:

A 9 := ; ; B 8 := ; B B A + := ; B WRITE ; 9 GOTO ;

El intérprete lo ejecutaría utilizando la siguiente programación (PCP es el puntero a la cabeza de la pila):

```
case V[pos] of
        ident, entero: apilar(V[pos]); p := p+1;
        "+": sumar los elementos PCP y PCP-1 y sustituir en la pila;
        "write": representar en pantalla el PCP;
        ":=": evaluar elemento PCP y ponerlo en la dirección del de PCP-1;
end
```