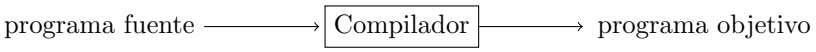
Apuntes de compiladores

Tomás Barros

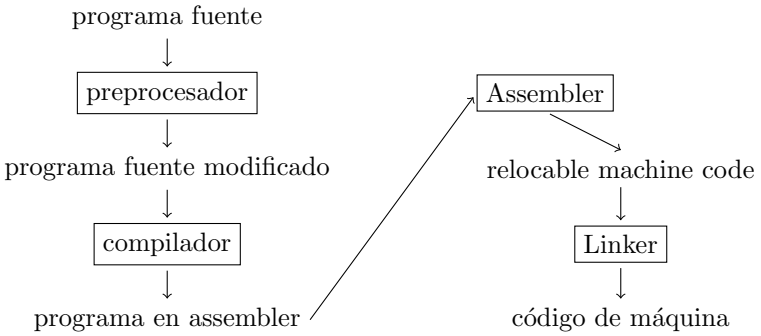
4 de septiembre de 2008

1. Introducción

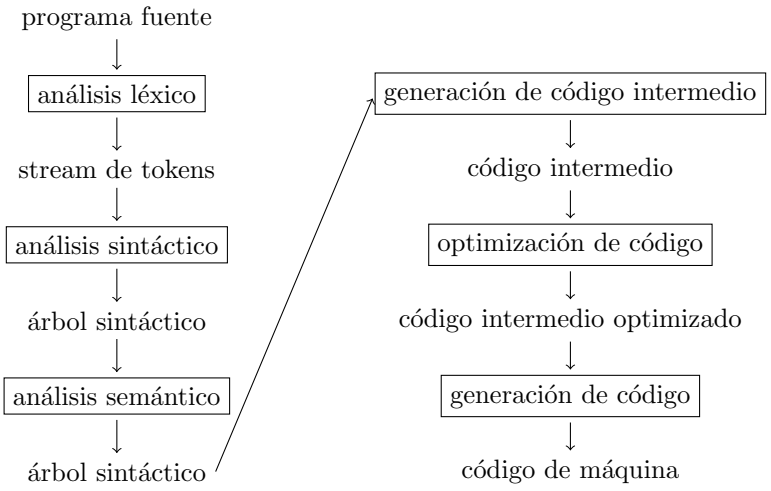
Un compilador es simplemente un programa que puede leer un programa en un lenguaje y traducirlo a otro lenguaje



1.1. procesos de un programa



1.2. arquitectura de un compilador



1.3. Análisis léxico

Es el primer paso, consiste en reconocer "palabras"

Esto es una oración.

Notar que:

- "E" inicia la oración
- " " separa las palabras
- "." termina la oración

El analizador léxico divide el programa de entrada en "tokes"

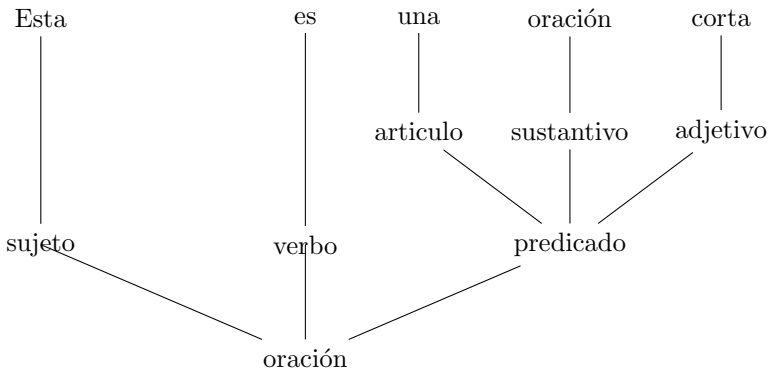
```
If x == y then z = 1; else z = 2;
```

Tokens:

```
If      x       ==      y
then    z       =       1
;       else    z       =
2       ;
```

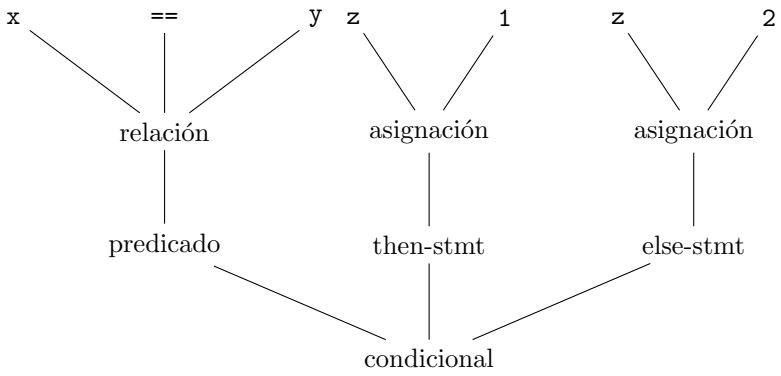
Una vez reconocidas las palabras es necesario reconocer la oración:

```
Esta es una oración corta
```



Es igual para programas

```
If x == y then z = 1; else z = 2;
```



1.4. análisis semántico

Una vez que se ha entendido la estructura de una oración, es necesario entender el significado de esta oración

- Sin embargo, esto suele ser muy difícil para los compiladores

Los compiladores suelen hacer un análisis muy semántico muy simple y limitado.

Por ejemplo:

Pedro le dijo a Juan que dejara su tarea en su casa

- ¿de quien es la tarea?
- ¿en la casa de quien?

2. Análisis léxico

Consiste en reconocer "tokens" en strings de entrada
Ejemplo:

```
if (i == j)
  z = 0;
else
  z = 1;
```

La entrada es simplemente una entrada de caracteres:

\tif (i == j);\n\t\tz=0;\n\telse\n\t\tz=1;

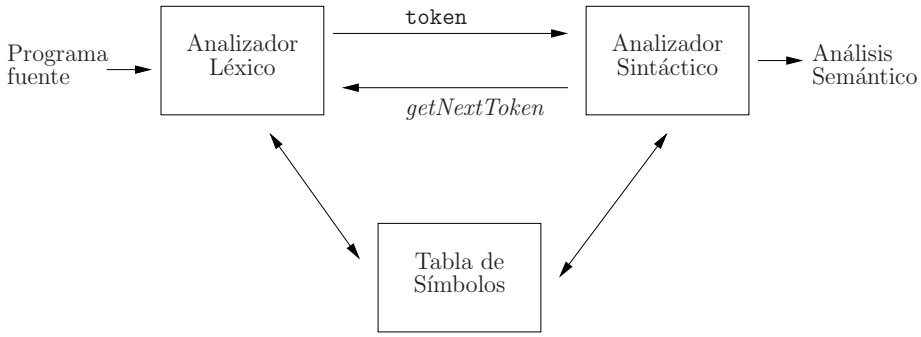
Objetivo: Particionar este string en substrings, donde los substrings son los "tokes". Un Token es una categoría sintáctica:

- En español: sustantivo, verbo, adjetivo, ...
- En un lenguaje de programación: Identificador, Entero, Palabra Reservada, Espacio en Blanco, ...

Los tokens corresponden a una tupla <conjunto,valor>, algunos conjuntos útiles son:

- Indentificador: strings de letras o dígitos, partiendo con una letra
- Entero: un string no vacío de dígitos
- Palabras Reservadas: `else`, `if`, `begin`, ...
- Espacios en blanco: Una secuencia no vacía de blancos, tabular, nueva línea, ...

2.1. Rol del analizador léxico



2.2. Realización

Para el análisis léxico, se usan expresiones regulares para especificar los tokens.

Definición 1. *Alfabeto. Un alfabeto* Σ *es un conjunto (finito) de símbolos*

Definición 2. *Palabra. Una palabra sobre un alfabeto* Σ*, es una secuencia (finita) de símbolos pertenecientes a* Σ

El conjunto de todas las palabras sobre un alfabeto Σ se denota Σ^*.

Definición 3. *Lenguaje. Un Lenguaje sobre el alfabeto* Σ *es un subconjunto de* Σ^* *(*$L \subset \Sigma^*$*)*

Asumiremos que $\forall\ L, \epsilon \in L$, donde ϵ es la palabra de largo 0.
Ejemplos:

- Alfabeto:
  - Caracteres del español
  - ASCII
  - Binario $\{0, 1\}$
- Lenguajes:
  - Frases del español
  - Programas en C

Notar que no toda combinación de caracteres del alfabeto español pertenecen al lenguaje español

Lo mismo sucede en lenguajes de programación. Necesitamos una notación para especificar cuales subconjuntos son los que nos interesan

2.2.1. Algebra de Kleene

Sean L y M dos lenguajes.

Definición 4. *Unión.* $L \cup M = \{s | s \in L \vee s \in M\}$

Definición 5. *Concatenación.* $LM = \{st | s \in L \wedge t \in M\}$

Definimos $L^0 = \{\epsilon\}, L^1 = L, L^i = L^{i-1}L$

Definición 6. *Clausura.* $L^* = \bigcup_{i=0}^{\infty} L^i$

2.2.2. Expresiones Regulares

Definición 7. *Las expresiones regulares sobre un alfabeto Σ es el conjunto más pequeño de expresiones incluyendo:*

ϵ	
$'c'$	*donde* $c \in \Sigma$
$A\|B$	*donde A, B, son expresiones regulares sobre* Σ
AB	*donde A, B, son expresiones regulares sobre* Σ
A^*	*donde A es un aexpresión regular sobre* Σ

Las expresiones regulares definen un lenguaje según la siguiente semántica:

Definición 8. *Semántica de expresiones regulares.*

$L(\epsilon)$	$=$	$\{$ ”$\}$
$L('c')$	$=$	$\{$ “c”$\}$
$L(A\|B)$	$=$	$L(A) \cup L(B)$
$L(AB)$	$=$	$\{st\|s \in L(A) \wedge t \in L(B)\}$
$L(A^*)$	$=$	$\bigcup_{i=0}^{\infty} L(A^i)$

Algunas leyes para expresiones regulares:

$r\|s$	$\|$ es commutativo
$r\|(s\|t) = (r\|s)\|t$	$\|$ es asociativo
$r(st) = (rs)t$	Concatenación es asociativa
$r(s\|t) = rs\|rt; (s\|t)r = sr\|tr$	Concatenación distribuye sobre $\|$
$\epsilon r = r\epsilon = r$	ϵes la identidad para la concatenación
$r^* = (r\|e)^*$	ϵ está garantizado en una clausura
$r^{**} = r^* *$	es idempotente

Ejemplos:

- $(a|b) = \{a, b\}$
- $(a|b)(a|b) = \{aa, ab, ba, bb\}$
- $a|a * b$, el string a o que empieza con cero o más a terminando en una b
- Sea $digits = (0|1|\ldots|9)(0|1|\ldots|9)*$, entoces los números enteros y reales (2321, 21213.32, 334.54E2, etc.) serían:

$$digits * (.digits * |\epsilon)((E(+|-|\epsilon)digits)|\epsilon)$$

2.3. Implementación

A partir de una expresión regular, hacer un programa que sea capaz de decidir si una palabra pertenece o no a el lenguaje generado por tal expresión regular.

Extendemos las expresiones regulares con:

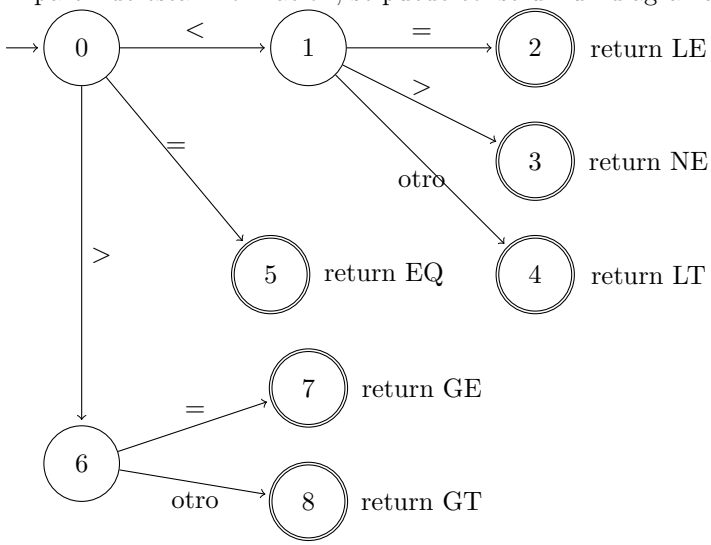
- $A? = A|\epsilon$

- $A+ = AA*$
- $[a-z] =' a'|'b'|\ldots|'z'$
- Exclusión, $[\wedge a-z]$ = complemento de $[a-z]$

Veamos el caso para relaciones lógicas, la tabla de tokens es:

Lexeme	Token	Valor
<	Rel.	LT
<=	Rel.	LE
=	Rel.	EQ
<>	Rel.	NE
>	Rel.	GT
>=	Rel.	GE

A partir de esta información, se puede construir un diagrama de transición.

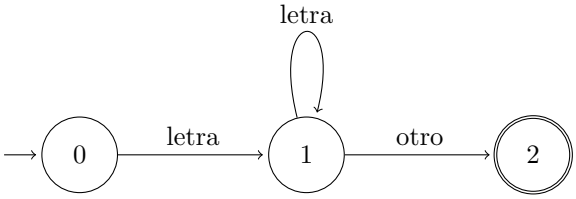


La implementación en C sería:

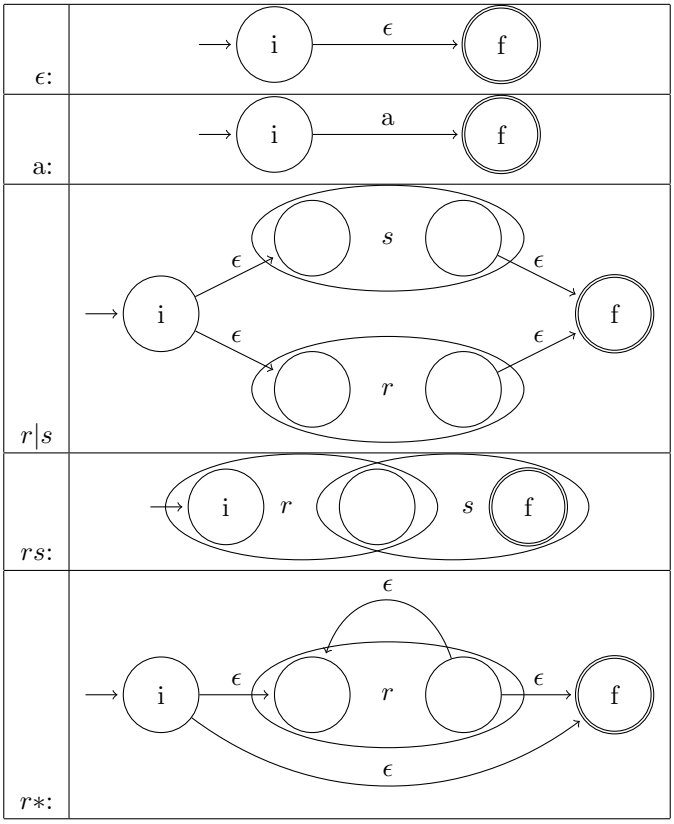
```
while (1) {
switch (state) {
case 0:
  c = getc(fd);
  if (c == EOF) exit(0);
  if ( c == '<') state = 1;
  else if ( c == '=') state = 5;
  else if (c == '>') state = 6;
  else if (c == '\n') ;
  else printf("Error, no reconoce lexeme '%c'\n",c);
  break;
  ...
case 1:
```

```
    c = getc(fd);
    if ( c == '=') state=2;
    else if ( c == '>') state=3;
    else state = 4;
    break;
  case 2:
    state=0;
    return(LE);
    ...
  case 4:
    ungetc(c,fd); /* backtraking */
    state=0;
    return(LT);
  ...
```

Para identificadores podemos usar el autómata:

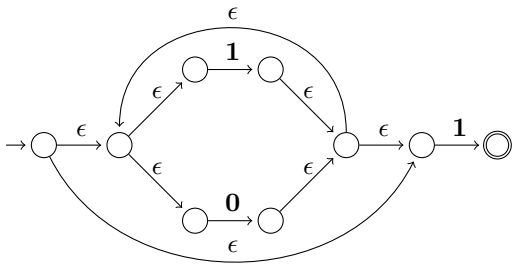


En general, todas las expresiones regulares pueden ser traducidas a autómatas con las siguientes fórmulas del algoritmo McNaughton-Yamada-Thompson:

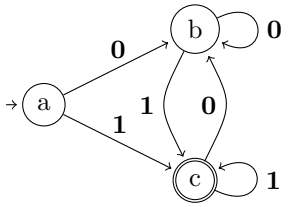


Con estas fórmulas se construye un autómata finito no determinístico (NFA) de aceptación de los tokens. Notar que cuando se reconocen varios tokens, por ejemplo t_1 y t_2, es equivalente a la expresión regular $(t_1|t_2)$.

Ejemplo **(1|0)*1**:



Este NFA se transforma a un DFA:



Luego para reconocer tokens, simplemente "corro" la entrada en el autómata y voy viendo cuando acepta.

3. Análisis sintáctico

Recordemos la definición de una gramática

Definición 9. *Una gramática libre de contexto $G = (T, N, S, R)$ consiste en:*

- *un conjunto de terminales T*
- *un conjunto de no terminales N*
- *un símbolo de inicio S*
- *un conjunto de producciones R o reglas de derivación, cada una de la forma forma:*

  $$X \to Y_1 Y_2 \dots Y_n$$

  donde $X \in N$ e $Y_i \in T \cup N \cup \{\epsilon\}$

Por ejemplo, una gramática para operaciones matemáticas simples puede ser (1) (que también se puede escribir con el operador |[1] como en (2)):

$$\begin{aligned} S &\to E \\ E &\to E + E \\ E &\to E * E \\ E &\to (E) \\ E &\to id \end{aligned} \qquad (1) \qquad\qquad \begin{aligned} S &\to E \\ E &\to E + E \\ &\mid E * E \\ &\mid (E) \\ &\mid id \end{aligned} \qquad (2)$$

Las expresiones gramaticales derivan en árboles sintácticos, por ejemplo para la expresión:

$$id * id + id$$

La derivación se hace reemplazando los no terminales por la regla de producción hasta "calzar" la expresión, segun el ejemplo:

$$S \to E \to E + E \to E * E + E \to id * E + E \to id * id + E \to id * id + id$$

[1] o-exclusivo

Esta es una derivación por la izquierda porque a cada paso reemplazamos el no terminal de más a la izquierda. De esta derivación se forma el árbol sintáctico mostrado en Figura 2.

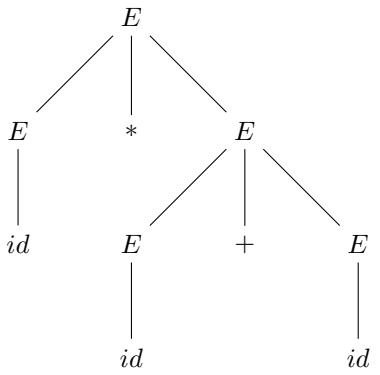


Figura 1: Derivación por la derecha

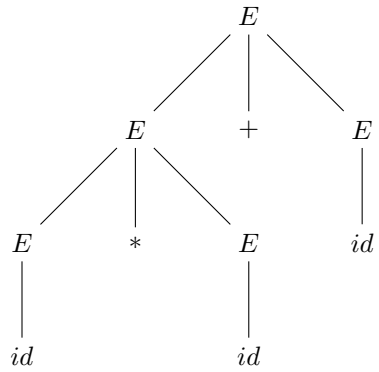


Figura 2: Derivación por la izquierda

De la misma forma se puede derivar por la derecha obteniendo el árbol mostrado en Figura 1.

Cuando una gramática permite formar dos árboles distintos para la misma expresión, se dice que es ambigüa, lo cual no es deseable (la compilación se vuelve no determinística).

Una de las formas más usuales de corregir la ambigüedad es reescribir la gramática en forma no ambigúa, en nuestro ejemplo podemos reescribirla dando prioridad a $*$ sobre $+$:

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow E + F \mid F \\ F &\rightarrow id * F \mid id\ (E) * F \mid (E) \\ &\quad \mid (E) \\ &\quad \mid id \end{aligned} \tag{3}$$

Con esta gramática la expresion $id * id + id$ sólo puede ser derivada al álbol en Figura 2

To contínue.. lef assosiation, if-then-else

4. Top-Down Parsing

En top-down parsing el árbol sintáctico se va construyendo de la raíz hacia las hojas (como en la sección anterior).

En cada nodo no terminal se intenta aplicar alguna de las reglas de derivación (si una falla, pruebo la siguiente).

Si encuentro terminal en la regla y calza con el input, avanzo el input.
El algoritmo para un no terminal A sería:

Algorithm 1 A()

```
1: for all S ∈ R / S → X_1 X_2 ... X_k do
2:   for i = 1 to k do
3:     if X_i es un no terminal then
4:       X_i();
5:     else
6:       if X_i = a y a es el token input actual then
7:         avanzar el input al siguiente símbolo
8:       else
9:         retroceder puntero y probar siguiente derivación S
10:      end if
11:    end if
12:  end for
13: end for
14: Si ninguna S pasó ⇒ error sintáctico
```

El algoritmo anterior se llama un **analizador sintáctico recursivo**, sin embargo, no puede resolver gramáticas recursivas por la izquierda ya que entraría en un ciclo infinito.

4.1. Eliminando la recursión por la izquierda

Afortunadamente una gramática recursiva por la izquierda se puede reescribir para eliminar la recursión "inmediata" por la izquierda:
Una gramática recursiva inmediatamente por la izquierda tiene la forma:

$$A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_m | \beta_1 | \beta_2 | \dots | \beta_n$$

La recursión se elimina reemplazando la producción A por:

$$A \rightarrow \beta_1 A' | \beta_2 A' | \dots | \beta_n A'$$
$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \dots | \alpha_m A' | \epsilon$$

Esta técnica elimina la recursión por la izquierda inmediata, sin embargo una gramática como en la ecuación 4 es además indirectamente recursiva (por ejemplo $A \Rightarrow Ba \Rightarrow Aba$).

$$\begin{aligned} A &\rightarrow Ba \mid Aa \mid c \\ B &\rightarrow Bb \mid Ab \mid d \end{aligned} \tag{4}$$

En general, una gramática indirectamente recursiva por la izquierda tendrá la forma:

$$\begin{aligned} A_0 &\to A_1\alpha_1 \mid \ldots \\ A_1 &\to A_2\alpha_2 \mid \ldots \\ &\ldots \\ A_n &\to A_0\alpha_{n+1} \mid \ldots \end{aligned} \tag{5}$$

Si la gramática no tiene ciclos ($A \Rightarrow^+ A$) y no tiene producciones ϵ, entonces el siguiente algoritmo elimina la recursión:

Algorithm 2 ELIMINA_RECURSION_IZQ()

1: poner en algún orden los no terminales $A_1, A_2, \ldots, A_n$
2: **for** $i = 1$ to n **do**
3: **for** $j = 1$ to $i - 1$ **do**
4: reemplace cada producción de la forma $A_i \to A_j\alpha$, donde $A_j \to \beta_1 \mid \ldots \mid \beta_k$ por:
$$A_i \to \beta_1\alpha \mid \ldots \mid \beta_k\alpha$$
5: elimine la recursión inmediata por la izquierda entre las producciones A_i
6: **end for**
7: **end for**

Usando el algoritmo 2 en (4) obtenemos la gramática (6):

$$\begin{aligned} A &\to BaA' \mid cA' \\ A' &\to aA' \mid \epsilon \\ B &\to cA'bB' \mid dB' \\ B' &\to bB' \mid aA'bB' \mid \epsilon \end{aligned} \tag{6}$$

4.2. Descenso predictivo

Para evitar el backtraking del descenso recursivo, se puede usar otro tipo de analizador sintáctico que se llama **analizador sintáctico predictivo**

Este analizador ve el próximo input antes de decidir que regla de derivación aplicar. Antes de introducir la técnica, introduciremos dos funciones: $FIRST$ y $FOLLOW$

Definición 10. $FIRST(X) = \{t \in \textit{Terminales} \mid X \Rightarrow^* t\alpha\} \cup \{\epsilon \mid X \Rightarrow^* \epsilon\}$

Definición 11. $FOLLOW(X) = \{t \ \in \textit{Terminales}; \mid S \Rightarrow^* \beta Xt\delta\}$

Computo de $FIRST$ y $FOLLOW$

Algorithm 3 FIRST(X)

1: **if** X es un terminal **then**
2: $FIRST(X) = \{X\}$
3: **end if**
4: **if** X es un no terminal y $X \to Y_1 Y_2 \dots Y_n, n \geq 1$ **then**
5: **for all** $Y_i \mid a \in FIRST(Y_i) \wedge Y_1 \dots Y_{i-1} \Rightarrow^* \epsilon$ **do**
6: agrega a a FIRST(X)
7: **end for**
8: **if** $Y_1 \dots Y_n \Rightarrow^* \epsilon$ **then**
9: agrega ϵ a FIRST(X)
10: **end if**
11: **end if**

Notar que si $\epsilon \in FIRST(X)$, entonces existe una derivación $X \Rightarrow^* \epsilon$. En este caso se dice que X es **anulable**

Algorithm 4 FOLLOW()

1: $\$ \in FOLLOW(S)$
2: **while** Hayan cambios en los conjuntos $FOLLOW$ **do**
3: **for all** $A \to X_1 X_2, \dots, X_n$ **do**
4: **for all** $X_i \mid X_i$ es no terminal **do**
5: $FIRST(X_{i+1} X_{i+2} \dots X_n) - \{\epsilon\} \in FOLLOW(X_i)$
6: **if** $\epsilon \in FIRST(X_{i+1} X_{i+2} \dots X_n) \vee i = n$ **then**
7: $FOLLOW(A) \in FOLLOW(X_i)$
8: **end if**
9: **end for**
10: **end for**
11: **end while**

Con los conjuntos de $FIRST$ y $FOLLOW$ construimos una tabla de parsing predictiva con el algoritmo:

Algorithm 5 M()

1: **for all** $A \to \alpha$ **do**
2: $\forall a \in FIRST(A), M[A, a] = A \to \alpha$
3: **if** $\epsilon \in FIRST(\alpha)$ **then**
4: $\forall b \in FOLLOW(A), M[A, b] = A \to \alpha$
5: **end if**
6: **end for**
7: si hay celdas vacias en M, marcarlas como error.

Por ejemplo:

$$
\begin{aligned}
E &\rightarrow \quad TE' \\
E' &\rightarrow \quad +TE' \mid \epsilon \\
T &\rightarrow \quad FT' \\
T' &\rightarrow \quad *FT' \mid \epsilon \\
E &\rightarrow \quad (E) | id
\end{aligned}
\tag{7}
$$

- $FIRST(F) = FIRST(T) = FIRST(E) = \{(, id\}$
- $FIRST(E') = \{+, \epsilon\}$
- $FIRST(T') = \{*, \epsilon\}$
- $FOLLOW(E) = FOLLOW(E') = \{), \$\}$
- $FOLLOW(T) = FOLLOW(T') = \{+,), \$\}$
- $FOLLOW(F) = \{+, *,), \$\}$

$M \Rightarrow$

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

Con esta tabla M se puede hacer un análisis sintáctico llamado **análisis predictivo guíado por la tabla sintáctica**. Para hacer el análisis sintáctico usamos la tabla M, un stack P y la entrada I, usamos el algoritmo 6.

Esta técnica no funciona cuando hay dos derivaciones que empiezan por el mismo terminal, por ejemplo:

$$
\begin{aligned}
S \rightarrow &\text{ if } E \text{ then } S \text{ else } S \\
&\mid \text{if } E \text{ then } S
\end{aligned}
\tag{8}
$$

Para solucionar eso se usa **left factoring** que es reescribir la gramática para posponer la desición hasta que se tenga suficiente input. La técnica es simple

$$
A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \qquad \Rightarrow \qquad \begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 \mid \beta_2 \end{aligned}
$$

5. Notas

Es indecidible si una gramática es ambigua (Hopcroft) una gramática LR(0) es no ambigua (Hopcroft)

Algorithm 6 LL(M,P,I)

1: $ip \leftarrow$ inicio de I
2: $put(\$, P); put(S, P)$
3: **while** $head(P) != \$$ **do**
4: **if** $head(P) = ip$ **then**
5: $pop(P)$
6: $ip++$
7: **else**
8: **if** $head(P)$ es terminal **then**
9: $error$
10: **else**
11: **if** $M(head(P), ip) = error$ **then**
12: $error$
13: **else**
14: **if** $M(head(P), ip) = X \rightarrow Y_1 Y_2 \dots Y_n$ **then**
15: $pop(P)$
16: $put(Y_1 Y_2 \dots Y_n)$
17: **end if**
18: **end if**
19: **end if**
20: **end if**
21: **end while**