

1. Fases de un compilador

1.1. Modelo de análisis y síntesis de la compilación

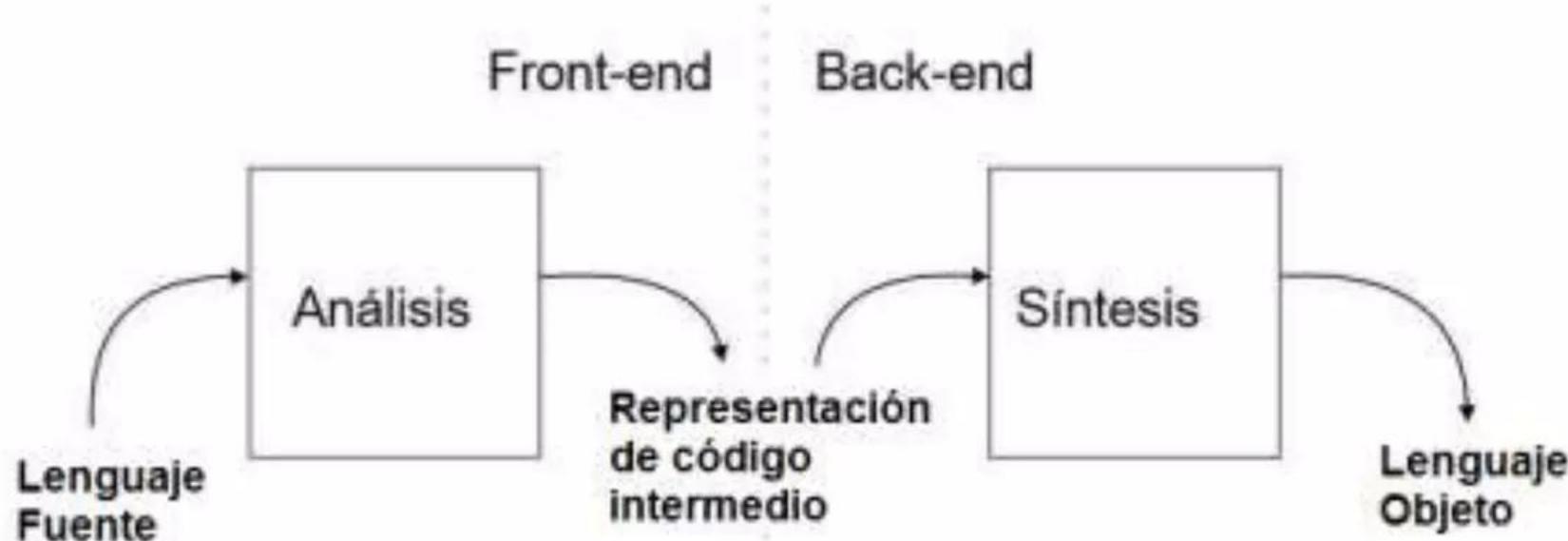
En la compilación hay dos partes:

- Análisis
- Síntesis

La parte del análisis, conocido como la parte **front-end** del compilador, lee el programa fuente y lo divide en partes fundamentales; verificando la parte léxica, sintáctica, tabla de símbolos hasta llegar a la generación de una representación intermedia del programa fuente.

La parte de la síntesis, conocido como la parte **back-end** del compilador, genera al programa de destino con la ayuda de código fuente, representación intermedia y tabla de símbolos.

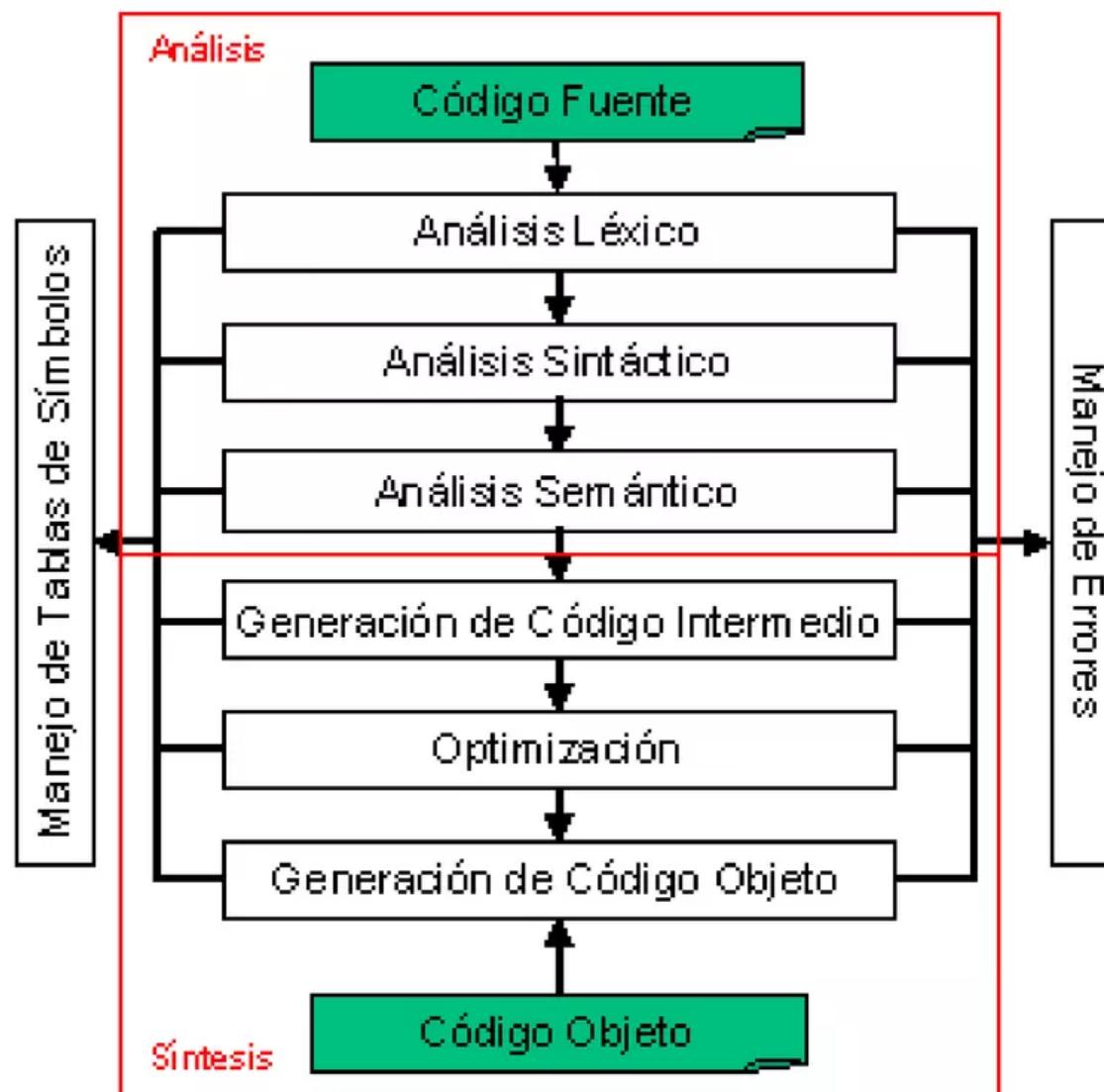
No obstante hay algunos que consideran una parte **middle-end**, que es el lugar donde se producen las optimizaciones, tanto intra-procedurales (propias de cada función) como inter-procedurales (que tienen en cuenta el estado de todas las funciones). En esta parte el compilador nos avisará si hemos cometido errores semánticos o si tenemos variables sin usar, etc.



1.2. Fases de un compilador

La construcción de un compilador involucra la división del proceso en una serie de fases que variará con su complejidad.

Las tres primeras fases de un compilador suelen agruparse en un sola fase llamada Análisis del programa fuente y las tres últimas en una sola fase llamada Síntesis del programa objeto.



Análisis Léxico: Esta fase se encarga de verificar si una cadena de entrada del código fuente pertenece o no al lenguaje, es decir se realiza un análisis símbolo a símbolo indicando el token para cada una de las cadenas reconocidas o un error en caso de no reconocer la cadena.

Cadena: valor = valor + inc;

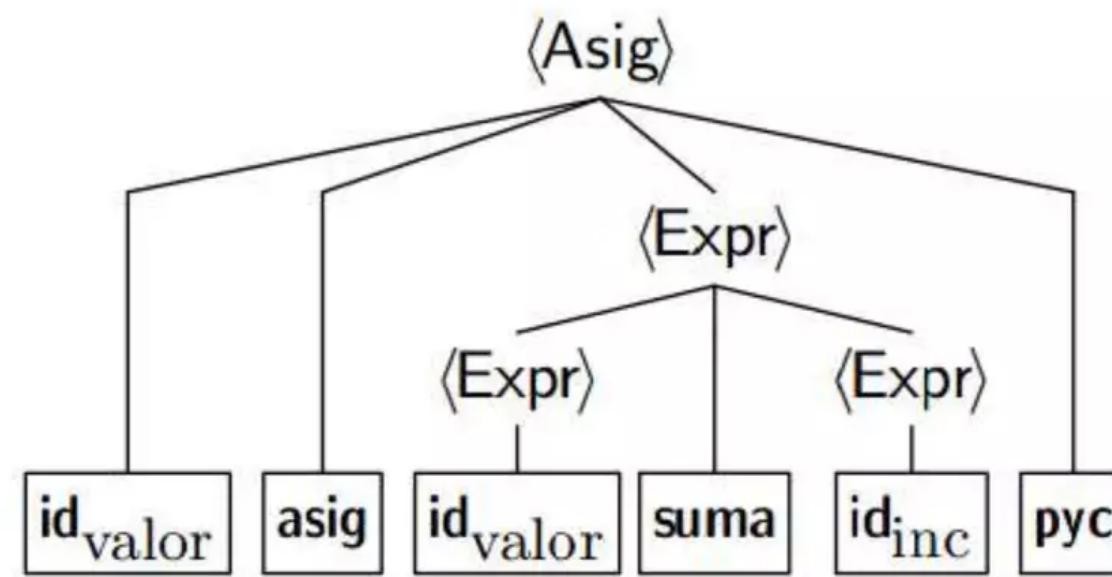
Token	Lexema
Id	"valor"
Asig	'='
Id	"valor"
Suma	'+'
Id	"inc"
pyc	';'

Análisis Sintáctico: En esta fase se analiza la estructura de las expresiones en base a gramáticas en base a reglas que determinan si una cadena de entrada del código fuente es válida. El análisis que se realiza es jerárquico ya que se obtiene árboles de derivación de las mismas gramáticas especificadas en el lenguaje.

$\langle \text{Asig} \rangle \rightarrow \text{id } \text{asig } \langle \text{Expr} \rangle \text{ pyc}$

$\langle \text{Expr} \rangle \rightarrow \text{id}$

$\langle \text{Expr} \rangle \rightarrow \langle \text{Expr} \rangle \text{ suma } \langle \text{Expr} \rangle$



Análisis Semántico: Este análisis es mucho más difícil de formalizar que el sintáctico ya que tiene que verificar que el árbol sintáctico tenga un significado válido dentro de las reglas especificadas en el lenguaje. El análisis semántico verifica que:

En una asignación, el tipo de la variable concuerde con el tipo de la expresión asignada.

Que las variables estén declaradas antes de ser usadas.

Generación de código intermedio: Esta fase se ocupa de generar instrucciones para la máquina virtual genérica a partir del análisis de las primeras tres fases.

Para la cadena $a = b + c$, se puede general el siguiente código intermedio:

1: + b c T1

2: = a T1

Optimización: Se encarga de transformar el código intermedio en uno equivalente que tenga menos líneas de código de menor tamaño y menor tiempo de ejecución.

Optimizando y suprimiendo la variable temporal T1, obtenemos:

1: + b c a

Generación de código objeto: Es la fase final en la que se genera el código objeto el cual utiliza el conjunto de instrucciones específico del CPU que por lo general es código máquina o código en lenguaje ensamblador.

Escribiendo en lenguaje ensamblador la instrucción anterior: + b c a

LOAD b

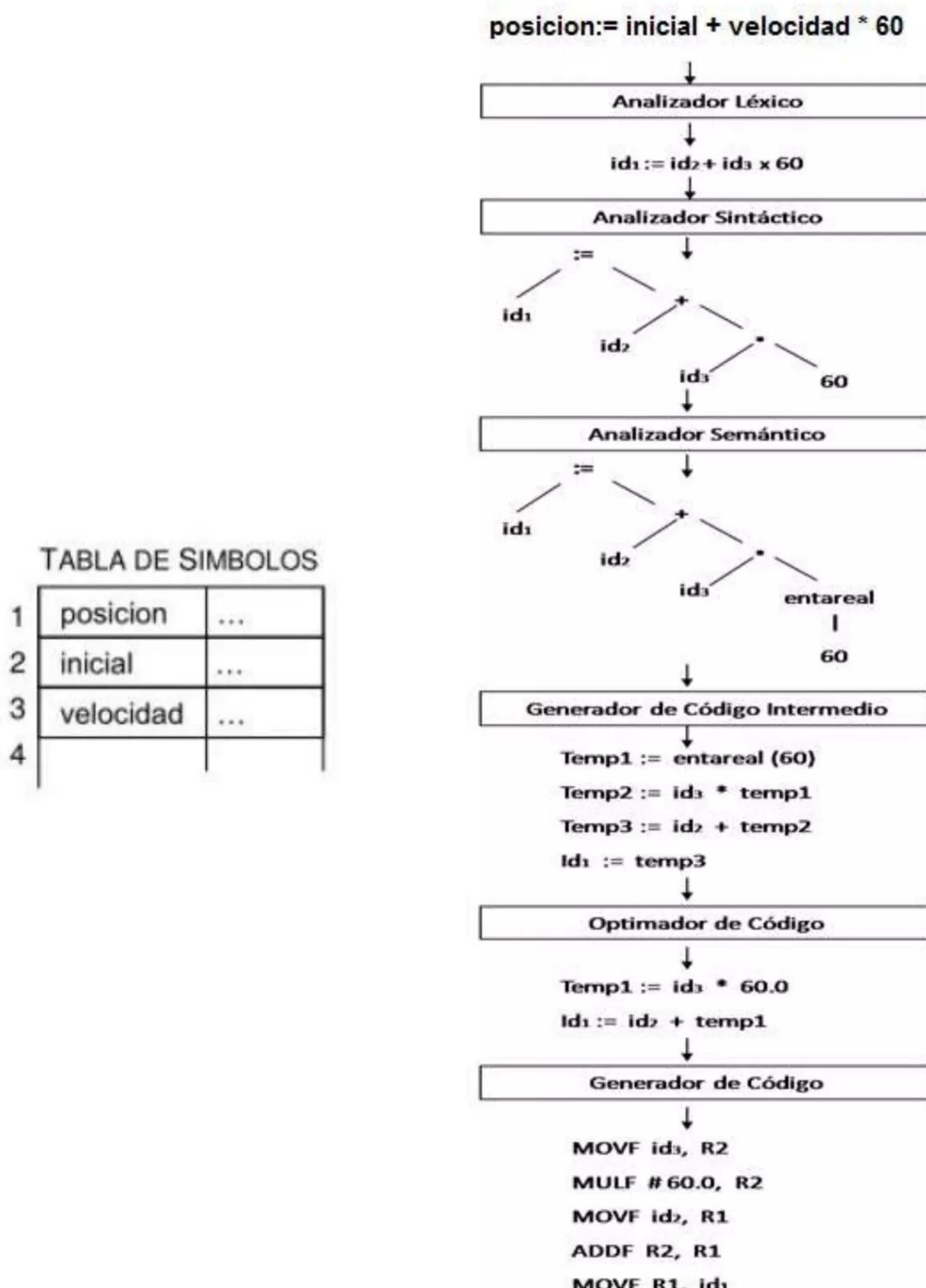
ADD b

STORE a

Tabla de símbolos: La tabla de símbolos es básicamente una estructura de datos donde cada registro representa un identificador y sus campos los atributos del identificador. El analizador léxico coloca los identificadores en la tabla y en las fases siguientes del compilador se agrega información de los identificadores. La información en la tabla de símbolos puede ser usada de forma variada. Por ejemplo, se puede agregar de qué tipo de identificador se trata y asegurarse que el programa fuente los usa de forma válida en ese lenguaje.

Manejo de errores: En cada fase de la compilación pueden encontrarse errores y deben manejarse de forma tal que se pueda seguir con las demás fases, en muchos casos, sin interrumpir el proceso.

Ejemplo de traducción de una *proposición*



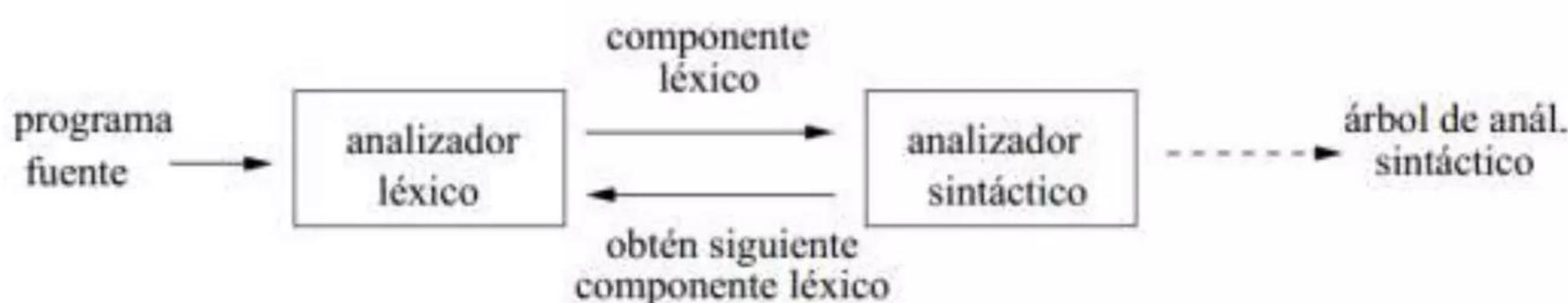
2. Analizador Léxico

2.1. Funciones del analizador léxico

Analizador léxico (scanner) lee la secuencia de caracteres del programa fuente, carácter a carácter, y los agrupa para formar unidades con significado propio llamados componentes léxicos (tokens). Estos componentes léxicos representan:

- Palabras reservadas: if, for, while,...
- Identificadores: variables, nombres de funciones, tipos definidos por el usuario, etiquetas,...
- Operadores: +, -, *, /, ==, =, >, <, ...
- Símbolos especiales: ;, (,), [,], {, }, ...
- Constantes numéricas: literales que representan valores enteros, en coma flotante, etc. 43, 0xF546, -83.E+3, ...
- Constantes de caracteres: literales que representan cadenas concretas de caracteres, "hola mundo", ...

El analizador léxico opera bajo petición del analizador sintáctico devolviendo un componente léxico conforme el analizador sintáctico lo va necesitando para avanzar en la gramática. Los componentes léxicos son los símbolos terminales de la gramática. Suele implementarse como una subrutina del analizador sintáctico. Cuando recibe la orden obtén el siguiente componente léxico, el analizador léxico lee los caracteres de entrada hasta identificar el siguiente componente léxico.



Otras funciones secundarias:

- Manejo del fichero de entrada del programa fuente: abrirlo, leer sus caracteres, cerrarlo y gestionar posibles errores de lectura.
- Eliminar comentarios, espacios en blanco, tabuladores y saltos de línea (caracteres no válidos para formar un token).
- Inclusión de ficheros: # include ...
- La expansión de macros y funciones inline: # define ...
- Contabilizar el número de líneas y columnas para emitir mensajes de error.
- Reconocimiento y ejecución de las directivas de compilación (por ejemplo, para depurar u optimizar el código fuente).

2.1.1. Componentes Léxicos, Patrones, Lexemas

Patrón: es una regla que genera la secuencia de caracteres que puede representar a un determinado componente léxico (una expresión regular).

Lexema: cadena de caracteres que concuerda con un patrón que describe un componente léxico. Un componente léxico puede tener uno o infinitos lexemas. Por ejemplo: las palabras reservadas tienen un único lexema, mientras que los números y los identificadores tienen infinitos lexemas.

Compon. Léxico	Patrón	Lexema
Identificador	Letra seguida de letras o dígitos	Temp, a1
Num_entero	Dígito seguido de mas dígitos	1, 27
If	Letra i seguida de la letra f	If
Do	Letra d seguida de la letra o	Do
Op_div	Carácter /	/
Op_asig	Carácter =	=

Los componentes léxicos se suelen definir como un tipo enumerado. Se codifican como enteros. También se suele almacenar la cadena de caracteres que se acaba de reconocer (lexema), que se usará posteriormente para el análisis semántico.

Es importante conocer el lexema para construir la tabla de símbolos. Los componentes léxicos se pueden representar mediante una estructura registro con tipo de token y lexema:

```
typedef enum {TKN_IF, TKN_THEN, TKN_NUM, TKN_ID,...} TokenType;
```

```
typedef struct {
```

```
    TokenType token;
```

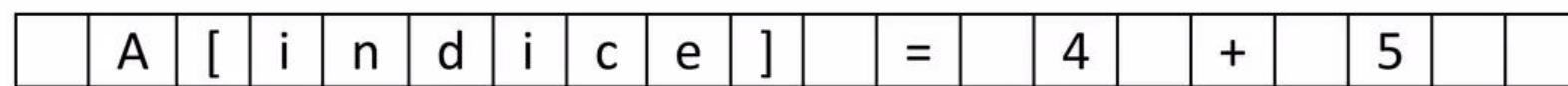
```
    char *lexema;
```

```
} TokenRecord;
```

Sea la entrada:

A[indice] = 4 + 5

Buffer de entrada:



Cada componente léxico va acompañado de su lexema

< TKN_ID , a >

< TKN_CORIZQ , [>

```
< TKN_ID , indice >
< TKN_CORDER , ] >
< TKN_ASIG , = >
< TKN_NUM , 4 >
< TKN_OPMAS , + >
< TKN_NUM , 5 >
```

2.1.2. Expresiones regulares.

Los componentes léxicos se especifican haciendo uso de expresiones regulares. Además de las tres operaciones básicas: concatenación, repetición (*) y alternativa (|) es posible usar los siguientes metasímbolos:

- Una o más repeticiones: +

r+ indica una o más repeticiones de r

$$(0 \mid 1)^+ = (0 \mid 1)(0 \mid 1)^*$$

- Cualquier carácter: .

.* b .* indica cualquier cadena que contiene una letra b

- Un rango de caracteres: [] (clase)

[a-z] indica cualquier carácter en minúscula entre la a y la z.

[a-zA-Z] indica cualquier letra del abecedario minúscula o mayúscula

[0-9] indica cualquier dígito de 0 a 9

[abc] indica a | b | c

- Cualquier carácter excepto un conjunto dado: ~

~ (a|b) indica cualquier carácter que no sea una a ó b

- Opcionalidad: ?

s? indica que la expresión s puede aparecer o no.

En el caso de que aparezca sólo lo hará una vez.

Cuando queremos usar alguno de estos símbolos especiales (+, ?, ...) tenemos que usar la barra de escape, así [a\z] significaría cualquier letra que sea a, guión o z.

Ejemplos:

- Numeros.

$\text{nat} = [0-9]^+$

$\text{signedNat} = (+ \mid -)? \text{nat}$

$\text{number} = \text{signedNat} (\cdot \text{nat})? (\text{E signedNat})?$

- Identificadores

$\text{letter} = [\text{a-zA-Z}]$

$\text{digit} = [0-9]$

$\text{identifier} = \text{letter} (\text{letter} \mid \text{digit})^*$

- Palabras reservadas

$\text{Tkn_if} = \text{"if"}$

$\text{Tkn_while} = \text{"while"}$

$\text{Tkn_do} = \text{"do"}$

- Comentarios

$\{(\sim\})^*$ comentarios en el lenguaje Pascal

- Delimitadores

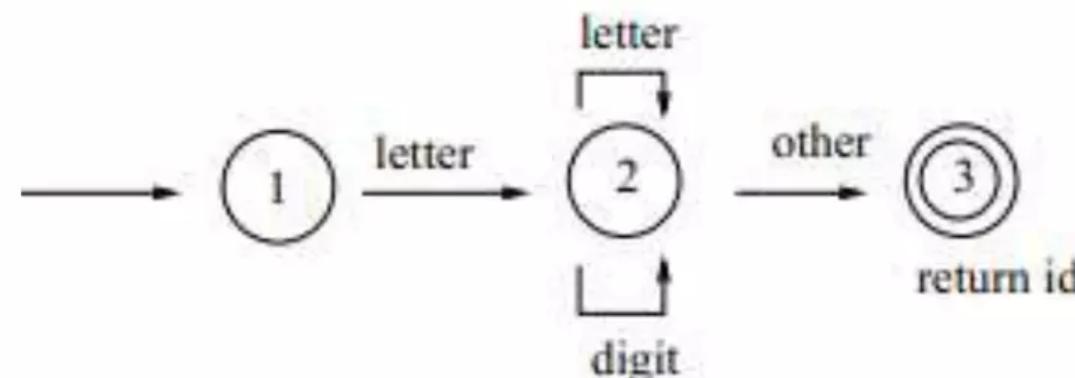
$\text{delimitadores} = (\text{newline} \mid \text{blank} \mid \text{tab} \mid \text{comment})^+$

2.1.3. Autómatas finitos deterministas (AFD).

Los AFD se pueden utilizar para reconocer las expresiones regulares asociadas a los componentes léxicos.

Identificadores

$\text{identifier} = \text{letter} (\text{letter} \mid \text{digit})^*$

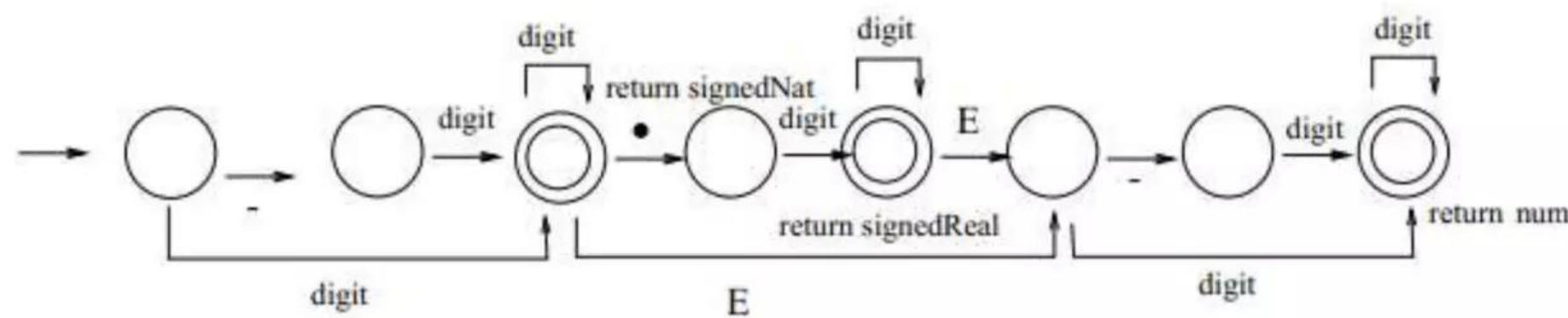


Números naturales y reales

nat = [0-9]+

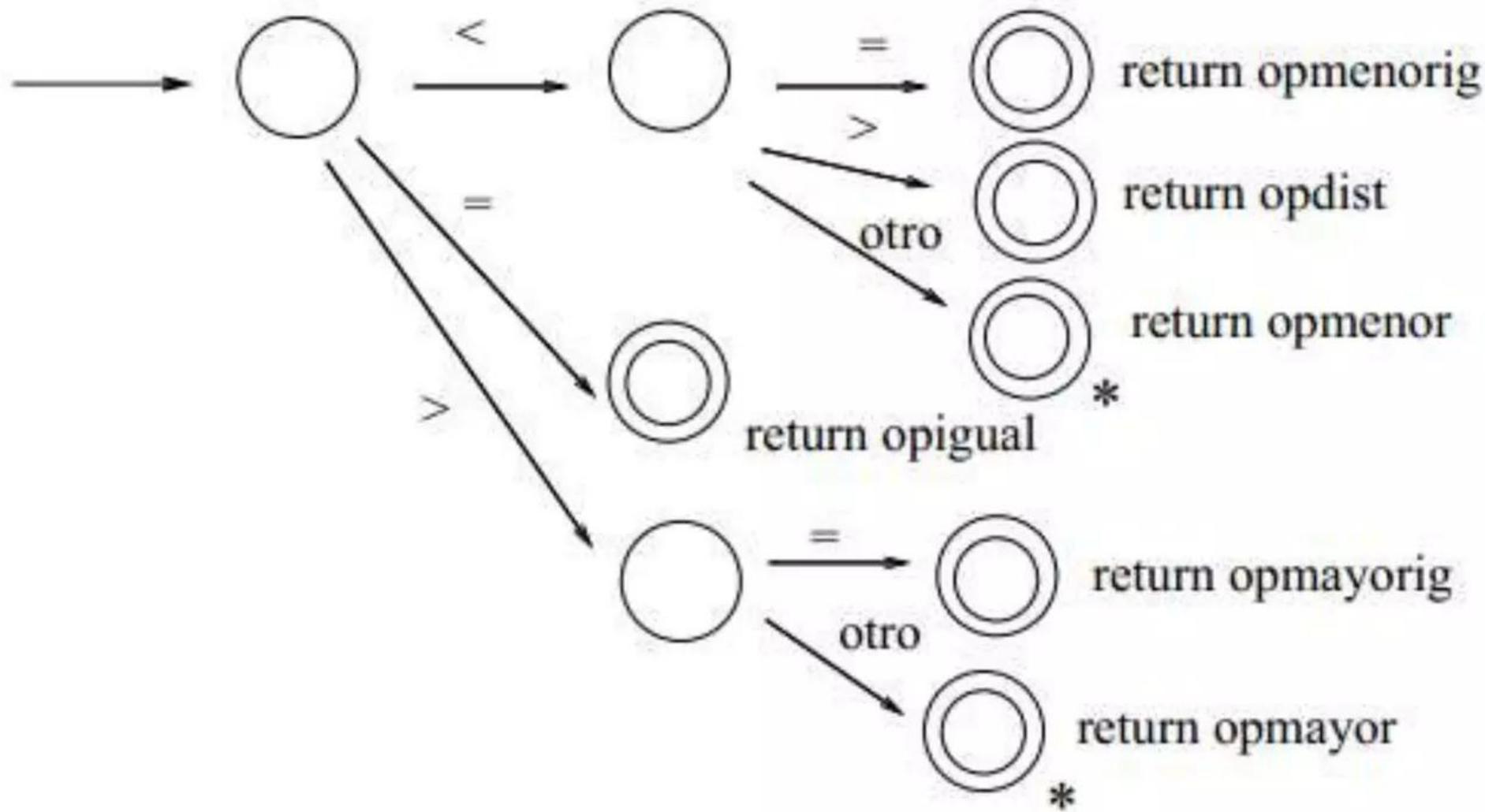
signedNat = (-)? nat

number = signedNat (. nat)? (E signedNat)?



Operadores relacionales

Operadores: < | = | <= | <> | >= | >



El símbolo * indica que se debe hacer un retroceso en la entrada, pues se ha consumido un símbolo demás, que no pertenece al lexema del componente léxico que se devuelve.

Cabe señalar que la implementación de un AFD se puede realizar a partir del diagrama de transición o una tabla de transición.

2.1.4. Aspectos a considerar en la implementación de un analizador léxico

Principio de máxima longitud

Se da prioridad al componente **léxico** de máxima longitud. Por ejemplo: <> se interpreta como el operador “distinto de”, en vez de “menor que” y “mayor que”. La entrada ende se interpreta como el identificador ende y no como la palabra reservada end y la letra e.

Palabras reservadas versus identificadores

Un lenguaje de programación puede tener del orden de 50 palabras reservadas. Para evitar tener un AFD demasiado grande las palabras reservadas se reconocen como identificadores (una palabra reservada también es una letra seguida de más letras) y se comprueba antes de decidir el tipo de token si se trata de una palabra reservada o de un identificador consultando una tabla previamente inicializada con las palabras reservadas.

Entrada de los identificadores en la Tabla de Símbolos

En lenguajes sencillos con solo variables globales y declaraciones, es normal implementar el scanner para que introduzca los identificadores en la Tabla de Símbolos conforme los va reconociendo, si es que no han sido ya introducidos. Después, el analizador sintáctico se encarga de introducir información adicional sobre su tipo, etc., conforme lo va obteniendo. Si se trata de un lenguaje estructurado, el scanner no introduce los identificadores en la Tabla de Símbolos, porque en este tipo de lenguajes, los identificadores pueden tener diferentes significados según su contexto (como una variable, como una función, como un campo de una estructura, . . .). Es el análisis sintáctico, cuando ya ha recogido información sobre su ámbito, cuando introduce los identificadores en la Tabla de Símbolos.

En general, se debe de tener en cuenta las siguientes consideraciones al implementar un Analizar Léxico.

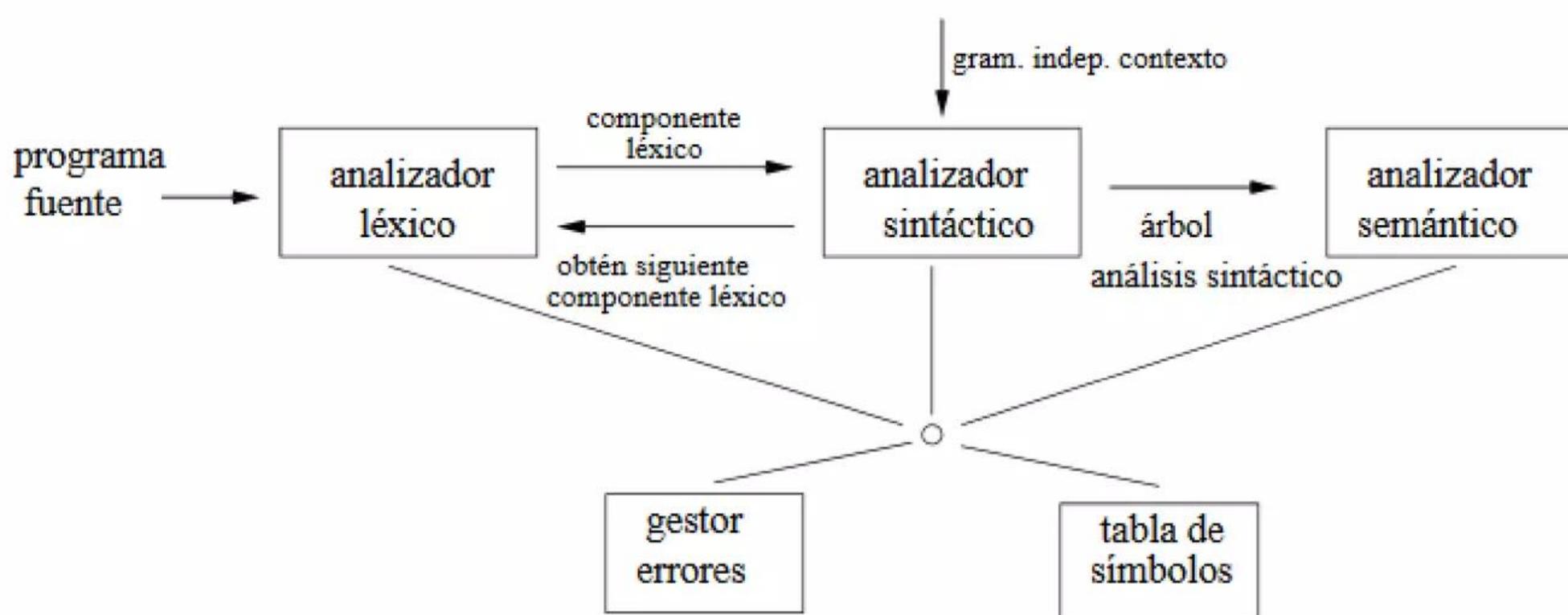
- El análisis léxico es una subrutina del análisis sintáctico que devuelve el tipo de componente léxico y el lexema cada vez que es llamada.
- Usar un tipo enumerado para los tipos de componentes léxicos. Usar un tipo enumerado para los estados del analizador.
- En el analizador léxico debe haber una función que se encarga de gestionar el buffer de entrada.
- Almacenar en variables el número de línea y columna para emitir mensajes de error.
- Las palabras reservadas se reconocen como identificadores y antes de devolver un identificador se comprueba si es una palabra reservada o un identificador consultando en una tabla previamente inicializada con las palabras reservadas.
- Hay casos en los que es necesario reinsertar un carácter en el buffer de entrada.

3. Analizador Sintáctico

3.1. El proceso de análisis sintáctico

Son funciones del analizador sintáctico:

- Comprobar si la cadena de componentes léxicos proporcionada por el analizador léxico puede ser generada por la gramática que define el lenguaje fuente (Gramática Independiente del Contexto, GIC).
- Construir el árbol de análisis sintáctico que define la estructura jerárquica de un programa y obtener la serie de derivaciones para generar la cadena de componentes léxicos.
- Informar de los errores sintácticos de forma precisa y significativa y deberá estar dotado de un mecanismo de recuperación de errores para continuar con el análisis.



El análisis sintáctico se puede considerar como una función que toma como entrada la secuencia de componentes léxicos producida por el análisis léxico y produce como salida el árbol sintáctico. En la realidad, el análisis sintáctico hace una petición al análisis léxico del componente léxico siguiente en la entrada conforme lo va necesitando en el proceso de análisis, conforme se mueve a lo largo de la gramática.

3.2. Especificación sintáctica de los lenguajes de programación

La mayoría de las construcciones de los lenguajes de programación se pueden representar con una gramática independiente del contexto (GIC). La mayoría de las construcciones de los lenguajes de programación implican recursividad y anidamientos.

$$G = \{S, V_T, V_{NT}, P\}$$

S: el axioma o símbolo de inicio

V_T : conjunto de terminales, los componentes léxicos

V_{NT} : conjunto de no-terminales

P: conjunto de reglas de producción de la forma

$$V_{NT} \rightarrow X_1, \dots, X_n, \text{ con } X_i \in (V_T \cup V_{NT})$$

¿Expresiones regulares o gramáticas independientes del contexto? Las expresiones regulares no permiten construcciones anidadas tan comunes en los lenguajes de programación: paréntesis equilibrados, concordancia de pares de palabras clave como begin - end, do - while, etc. Por ejemplo: consideremos el problema de los paréntesis equilibrados en una expresión aritmética. El hecho de que haya un paréntesis abierto obliga a que haya un paréntesis cerrado. Si intentamos escribir una expresión regular, lo más próximo sería: $\{a^*ba^*\}$ pero no se garantiza que el número de a's antes y después sea el mismo. Las expresiones regulares NO SABEN contar. NO es posible especificar la estructura de un lenguaje de programación con sólo expresiones regulares.

¿Son suficientes las gramáticas independientes del contexto? NO, existen ciertos aspectos de los lenguajes de programación que no se pueden representar con una GIC. A estos aspectos se les llama aspectos semánticos, son aspectos en general dependientes del contexto. Por ejemplo, el problema de declaración de los identificadores antes de usarlos.

¿Por qué no usar gramáticas contextuales? Su reconocimiento es demasiado complejo para ser eficiente en un problema práctico.

3.3. Derivaciones

Dada $\alpha, \beta \in (V_T \cup V_{NT})^*$ cadenas arbitrarias de símbolos gramaticales, se dice que

$\alpha A \beta \xrightarrow{*} \alpha \gamma \beta$ es una derivación, si existe una producción $A \rightarrow \gamma$.

Sea $S \xrightarrow{*} \alpha$,

si $\alpha \in (V_T \cup V_{NT})^*$ se dice que α es una *forma de frase* o *forma sentencial*.

Si $\alpha \in (V_T)^*$ se dice que α es una *frase* o *sentencia*.

Existen dos tipos de derivaciones:

- Derivación más a la izquierda: se sustituye en cada paso el no-terminal más a la izquierda de la forma de frase. La denotaremos por: $\alpha \xrightarrow{* mi} \beta$

- Derivación más a la derecha: se sustituye en cada paso el no-terminal más a la derecha de la forma de frase. La denotaremos por: $\alpha \xrightarrow{* md} \beta$

Por ejemplo para la gramática:

$$E \rightarrow \mathbf{id} \mid \mathbf{num} \mid E + E \mid (E) \mid - E$$

Si queremos derivar la frase $-(id+id)$, entonces una derivación más a la izquierda es:

$$E \Rightarrow - E \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id)$$

Y una derivación más a la derecha es:

$$E \Rightarrow - E \Rightarrow -(E + E) \Rightarrow -(E + id) \Rightarrow -(id + id)$$

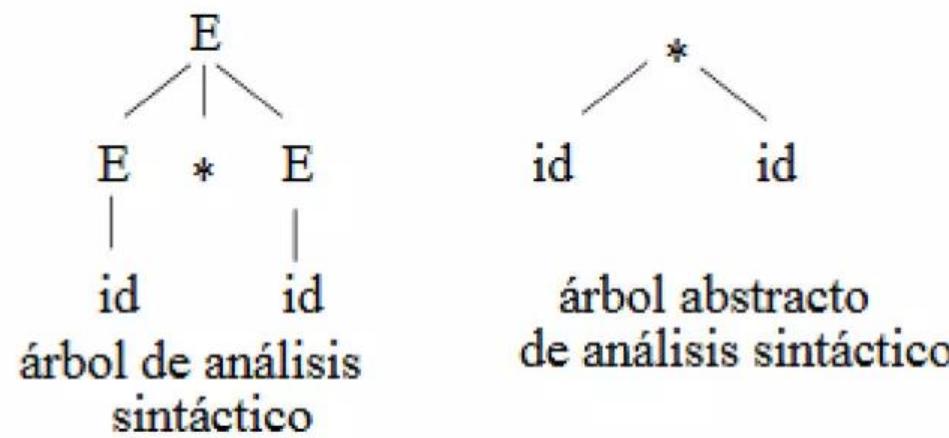
3.3.1. Árboles de análisis sintáctico

Un árbol de análisis sintáctico indica cómo a partir del axioma de la gramática se deriva una frase (cadena) del lenguaje. Dada una gramática independiente del contexto, un árbol de análisis sintáctico es un árbol tal que:

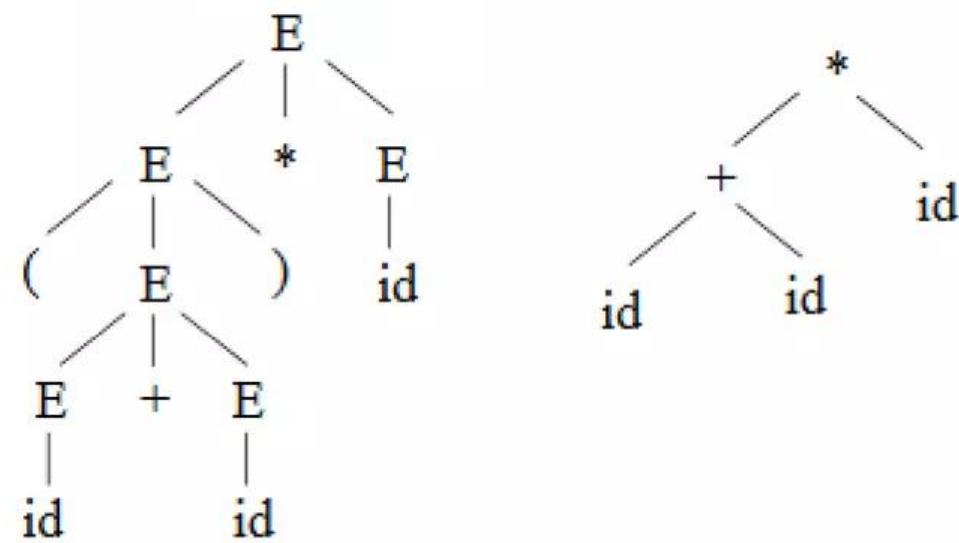
1. La raíz está etiquetada con el símbolo inicial.
2. Cada hoja está etiquetada con un componente léxico. Las hojas de izquierda a derecha forman la frase (el programa fuente).
3. Cada nodo interior está etiquetado con un no-terminal.
4. Si A es un no-terminal y X_1, X_2, \dots, X_n son sus hijos de izquierda a derecha, entonces existe la producción $A \rightarrow X_1, X_2, \dots, X_n$, con $X_i \in (V_T \cup V_{NT})$.

El árbol de análisis sintáctico contiene en general mucha más información que la estrictamente necesaria para generar el código. Se puede construir una estructura más sencilla, *los árboles abstractos de análisis sintáctico* o simplemente *árbol sintáctico*. Ejemplo: expresiones aritméticas (la misma semántica pero con menor complejidad)

Entrada: id * id



Entrada: (id + id)* id



3.4. Gramáticas limpias y bien formadas

Llamamos *símbolo vivo* al símbolo a partir del cual se puede derivar una cadena de terminales. Llamamos *símbolo muerto* a los símbolos no-vivos, no generan una cadena del lenguaje. Llamamos *símbolo inaccessible* si nunca aparece en la parte derecha de una producción. A las gramáticas que contienen estos tipos de símbolos se les llama gramáticas sucias.

Teorema 1: si todos los símbolos de la parte derecha de una producción son símbolos vivos, entonces el símbolo de la parte izquierda también lo es.

Algoritmo para detectar símbolos muertos

1. Hacer una lista de no-terminales que tengan al menos una producción con sólo símbolos terminales en la parte derecha.
2. Dada una producción, si todos los no-terminales de la parte derecha pertenecen a la lista, entonces podemos incluir en la lista al no-terminal de la parte izquierda de la producción.
3. Cuando ya no se puedan incluir más símbolos en la lista mediante la aplicación del paso 2, la lista contendrá los símbolos no-terminales vivos y el resto serán símbolos muertos.

Teorema 2: si el símbolo no-terminal de la parte izquierda de una producción es accesible, entonces todos los símbolos de la parte derecha también lo son.

Algoritmo para detectar símbolos inaccesibles

1. Se inicializa una lista de no-terminales que sabemos que son accesibles con el axioma.
2. Si la parte izquierda de una producción está en la lista, entonces se incluye en la misma al no-terminal que aparece en la parte derecha de la producción.
3. Cuando ya no se pueden incluir más símbolos a la lista mediante la aplicación del paso 2, entonces la lista contendrá todos los símbolos accesibles y el resto de los no-terminales serán inaccesibles.

Para limpiar una gramática primero se eliminan los símbolos muertos y después los símbolos inaccesibles.

Una gramática *está bien formada* si es limpia y además no contiene producciones- ϵ .

Ejercicio: Limpiar la gramática

$$S \rightarrow a A B \mid A$$

$$A \rightarrow c B d$$

$$B \rightarrow e \mid f S$$

$$C \rightarrow g D \mid h D t$$

$$D \rightarrow x \mid y \mid z$$

- $S \rightarrow a A B \mid b C$
- $A \rightarrow c D e \mid f$
- $B \rightarrow g H \mid h$
- $C \rightarrow i J k \mid l$
- $D \rightarrow m \mid n A$
- $E \rightarrow o P \mid p$
- $F \rightarrow q \mid r F$
- $H \rightarrow s \mid t$
- $J \rightarrow u \mid v$
- $P \rightarrow w \mid x P$

3.5. Gramáticas ambiguas

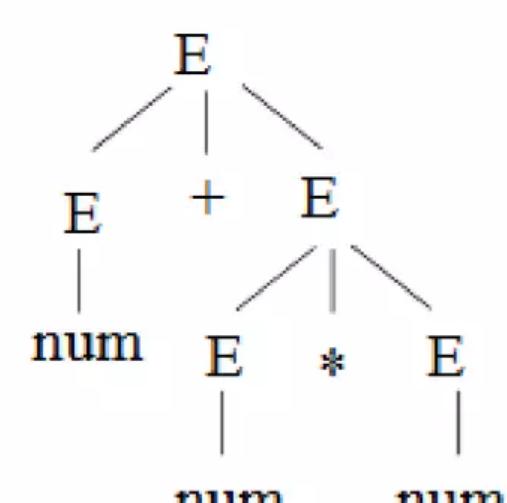
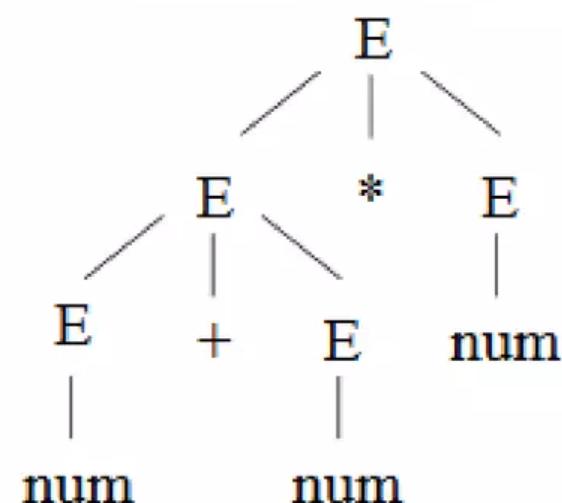
Una gramática es *ambigua* cuando para una determinada sentencia produce más de un árbol de derivación.

La gramática siguiente es ambigua:

$$E \rightarrow id \mid num \mid E + E \mid E * E \mid (E) \mid - E$$

Supongamos la sentencia: $id + id * id$

Entrada: $num + num * num$



El significado semántico es DIFERENTE. No existe una única traducción posible. Se genera código diferente.

No existe un algoritmo que determine con certeza en un plazo finito de tiempo si una gramática es ambigua o no. A lo sumo que se ha llegado en el estudio de la ambigüedad es que hay algunas condiciones que de cumplirse determinan la ambigüedad, pero en el caso de no cumplirse no se puede decir que la gramática es no ambigua.

Necesidad de evitar las gramáticas ambiguas. ¿Cómo?

- Transformando la gramática o
- Estableciendo precedencias entre operadores y de asociatividad.

Se puede eliminar la ambigüedad transformando la gramática, agrupando todos los operadores de igual precedencia en grupos y asociando a cada uno una regla, de forma que los que tengan menor precedencia aparezcan más cercanos al símbolo de inicio, precedencia en cascada. Esto conlleva el aumento de la complejidad de la gramática y con ello en la del árbol sintáctico. La gramática deja de ser intuitiva.

Ejemplo:

Sea la gramática ambigua:

$$E \rightarrow \mathbf{num} \mid E + E \mid E - E \mid E * E \mid E / E$$

Si transformamos la gramática de expresiones aritméticas, ésta dejará de ser ambigua.

$$\text{exp} \rightarrow \text{exp} + \text{term} \mid \text{exp} - \text{term} \mid \text{term}$$

$$\text{term} \rightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor}$$

$$\text{factor} \rightarrow (\text{exp}) \mid \mathbf{num}$$

3.6. Clasificación de métodos de análisis sintáctico

Atendiendo a la forma en que se construye el árbol de análisis sintáctico los métodos de análisis sintáctico se clasifican:

- Métodos descendentes: se parte del símbolo de inicio de la gramática que se coloca en la raíz y se va construyendo el árbol desde arriba hacia abajo hasta las hojas, eligiendo la derivación que da lugar a una concordancia con la cadena de entrada. Se basa en la idea de *predice* una derivación y establece una concordancia con el símbolo de la entrada (*predict/match*). El análisis sintáctico descendente corresponde con un recorrido prefijo del árbol de análisis sintáctico (primero expandimos el nodo que visitamos y luego procesamos los hijos).
- Métodos ascendentes: se construye el árbol de análisis sintáctico desde las hojas hasta la raíz. En las hojas está la cadena a analizar y se intenta reducirla al símbolo de inicio

de la gramática que está en la raíz. Se trata de *desplazar*-se en la cadena de entrada y encontrar una subcadena para aplicar una *reducción*(una regla de producción a la inversa), (*shift-reduce*). El análisis sintáctico ascendente corresponde con un recorrido postorden del árbol (primero reconocemos los hijos y luego mediante una reducción reconocemos el padre).

Atendiendo a la forma en que procesan la cadena de entrada se clasifican en:

- Métodos direccionales: procesan la cadena de entrada símbolo a símbolo de izquierda a derecha.
- Métodos no-direccionales: acceden a cualquier lugar de la cadena de entrada para construir el árbol. Necesitan tener toda la cadena de componentes léxicos en memoria. Más costosos, no se suelen implementar.

Atendiendo al número de alternativas posibles en una derivación se clasifican en:

- Métodos deterministas: dado un símbolo de la cadena de entrada se puede decidir en cada paso cuál es la alternativa/derivación adecuada a aplicar, sólo hay una posible. No se produce retroceso y el costo es lineal.
- Métodos no-deterministas: en cada paso de la construcción del árbol se deben probar diferentes alternativas/derivaciones para ver cual es la adecuada, con el correspondiente aumento del costo.

Descendentes		Ascendentes	
No direccionales	Unger	CYK	
Direccionales	No deterministas Deterministas	<i>Predice/Concuerda</i> <i>1º en anchura</i> <i>1º en profundidad</i> <i>Predice/Concuerda</i> <i>Gram. LL(1)</i>	<i>Desplaza/Reduce</i> <i>1º en anchura</i> <i>1º en profundidad</i> <i>Desplaza/Reduce</i> <i>Gram. LR(K)</i> <i>LR(0),SLR(1),LALR(1)</i>

4. Análisis Sintáctico Descendente

4.1. Análisis Descendente

La especificación de un lenguaje se realiza mediante gramáticas independientes de contexto. Estas gramáticas permiten recursividad y estructuras anidadas.

La recursividad va a implicar:

- Algoritmos de reconocimiento más complejos, que hagan uso de llamadas recursivas o usen explícitamente una pila, la pila de análisis sintáctico.
- La estructura de datos usada para representar la sintaxis del lenguaje ha de ser también recursiva, en vez de lineal.

Fundamento de los métodos descendentes: *autómata predice/concuerda*

En cada paso del proceso de derivación de la cadena de entrada se realiza una *predicción* de la posible producción a aplicar y se comprueba si existe una *concordancia* entre el símbolo actual en la entrada con el primer terminal que se puede generar a partir de esa regla de producción, si existe esta concordancia se avanza en la entrada y en el árbol de derivación, en caso contrario se vuelve hacia atrás y se elige una nueva regla de derivación.

En los métodos deterministas (no hay vuelta atrás) teniendo en cuenta un sólo símbolo de preanálisis (componente léxico que se está analizando en la cadena de entrada en ese momento), sabemos exactamente en todo momento qué producción aplicar. El símbolo de preanálisis se actualiza cada vez que se produce una concordancia.

Los métodos descendentes se caracterizan porque analizan la cadena de componentes léxicos de izquierda a derecha, obtienen la derivación más a la izquierda y el árbol de derivación se construye desde la raíz hasta las hojas.

Problemas de decisión: ¿Qué producción elegir cuando hay varias alternativas?

4.2. Problemas en el análisis descendente

- La recursividad a izquierdas da lugar a un bucle infinito de recursión.
- Problemas de indeterminismo cuando varias alternativas en una misma producción comparten el mismo prefijo. No sabríamos cuál elegir. Probaríamos una y si se produce un error haríamos backtracking. Si queremos que el método sea determinista hay que evitarlas.

Una solución sería transformar las gramáticas para la eliminación de la recursividad y el indeterminismo. Estas transformaciones no modifican el lenguaje que se está reconociendo, pero sí que cambian el aspecto de la gramática, que es en general menos intuitiva.

4.2.1. Recursividad a izquierdas y su eliminación

Una gramática es recursiva por la izquierda si tiene un no-terminal A tal que existe una producción $A \rightarrow A\alpha$.

Algoritmo para eliminar recursividad a izquierdas:

- **primer paso:** Se agrupan todas las producciones de A en la forma:

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

donde ninguna β_i comienza con A .

- **segundo paso:** se sustituyen las producciones de A por:

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$$

Ejercicio: Eliminar la recursividad a izquierdas de

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \text{num} \mid \text{id}$$

Una gramática recursiva a izquierdas nunca puede ser LL(1). En efecto, asume que encontramos un *token* t , tal que predecimos la producción $A \rightarrow A\beta$, después de la predicción, al apilar la parte derecha de la producción, el símbolo A estará en la cima de la pila, y por tanto, se predecirá la misma producción, y tendríamos de nuevo A en la cima, así hasta el infinito, produciéndose un desbordamiento de la pila de análisis sintáctico.

4.2.2. Indeterminismo en las alternativas. Factorización

Cuando dos alternativas para un no-terminal empiezan igual, no se sabe qué alternativa expandir. Nuevamente la idea es reescribir la producción para retrasar la decisión hasta haber visto de la entrada lo suficiente para poder elegir la opción correcta.

Algoritmo para factorizar la gramática:

- **primer paso:** para cada no-terminal A buscar el prefijo α más largo común a dos o más alternativas de A , $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_n | \gamma$
- **segundo paso:** Si $\alpha \neq \epsilon$, sustituir todas las producciones de A , de la forma $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_n | \gamma$, donde γ representa todas la alternativas de A que no comienzan con α por:

$$A \rightarrow \alpha A' | \gamma$$

$$A' \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$$

Ejemplo: Sea la gramática.

$$\text{Stmt} \rightarrow \text{if E then Stmt else Stmt} | \text{if E then Stmt} | \text{otras}$$

Se puede reescribir la gramática como:

$$\text{Stmt} \rightarrow \text{if E then Stmt Stmt'} | \text{otras}$$

$$\text{Stmt}' \rightarrow \text{else Stmt} | \epsilon$$

4.3. Gramáticas LL(1)

Las gramáticas LL(1) permiten construir de forma automática un analizador determinista descendente con tan sólo examinar en cada momento el símbolo actual de la cadena de entrada (símbolo de preanálisis) para saber qué producción aplicar.

- La primera **L**: método direccional, procesamos la entrada de izquierda a derecha.
- La segunda **L**: obtenemos la derivación más a la izquierda
- **(1)** : usamos un símbolo de preanálisis para decidir la producción a aplicar.

Teorema 1: Una gramática LL(1) no puede ser recursiva a izquierdas y debe estar factorizada.

Teorema 2: Una gramática LL(1) es no ambigua.

Ejemplo de gramática LL(1).

$$S \rightarrow a B$$

$$B \rightarrow b | a B b$$

4.3.1. Conjuntos PRIMEROS Y SIGUIENTES

¿Cómo garantizar que una gramática es LL(1)?

Primero, debemos calcular los conjuntos PRIMEROS y SIGUIENTES

Sea $\alpha \in (V_N \cup V_T)^*$, PRIMEROS(α) indica el conjunto de terminales que pueden aparecer al principio de cadenas derivadas de α

Si $\alpha \Rightarrow^* \sigma \sigma_1 \dots \sigma_n$ entonces $\{\sigma\} \in \text{PRIMEROS}(\alpha)$ con $\sigma \in V_T$

Si $\alpha \Rightarrow^* \varepsilon$ entonces $\{\varepsilon\} \in \text{PRIMEROS}(\alpha)$

Sea $A \in V_N$ entonces SIGUIENTES(A) indica el conjunto de terminales que en cualquier momento de la derivación pueden aparecer inmediatamente a la derecha (después) de A .

Sea $\sigma \in V_T$, $\{\sigma\} \in \text{SIGUIENTES}(A)$ si existe $\alpha, \beta \in (V_N \cup V_T)^*$, tales que $S \Rightarrow^* \alpha A \sigma \beta$.

Algoritmo para calcular el conjunto de PRIMEROS:

Para X un único símbolo de la gramática (terminal o no-terminal). Entonces PRIMEROS(X) consiste de símbolos terminales y se calcula:

- Si X es un terminal o ε , entonces $\text{PRIMEROS}(X) = \{X\}$
- Si X es un no-terminal, entonces para cada producción de la forma $X \rightarrow X_1 X_2 \dots X_n$, $\text{PRIMEROS}(X)$ contiene a $\text{PRIMEROS}(X_1) - \{\varepsilon\}$. Si además para algún $i < n$ todos los conjuntos $\text{PRIMEROS}(X_1) \dots \text{PRIMEROS}(X_i)$ contienen a ε , entonces $\text{PRIMEROS}(X)$ contiene a $\text{PRIMEROS}(X_{i+1}) - \{\varepsilon\}$

Para α cualquier cadena, $\alpha = X_1 X_2 \dots X_n$, de terminales y no-terminales, entonces $\text{PRIMEROS}(\alpha)$ contiene a $\text{PRIMEROS}(X_1) - \{\varepsilon\}$. Para algún $i = 2 \dots n$, si $\text{PRIMEROS}(X_k)$ contiene $\{\varepsilon\}$ para todos los $k = 1 \dots i-1$, entonces $\text{PRIMEROS}(\alpha)$ contiene a $\text{PRIMEROS}(X_i) - \{\varepsilon\}$. Finalmente, si para todo $i = 1 \dots n$, $\text{PRIMEROS}(X_i)$ contiene ε , entonces $\text{PRIMEROS}(\alpha)$ contiene a $\{\varepsilon\}$.

Algoritmo para calcular el conjunto de SIGUIENTES:

Dado $A \in V_{NT}$, entonces SIGUIENTES(A) consiste de terminales y del símbolo # (símbolo de fin de cadena) y se calcula como:

- Si A es el símbolo de inicio, entonces $\{\#\} \in \text{SIGUIENTES}(A)$
- Si hay una producción $B \rightarrow \alpha A \gamma$, entonces $\text{PRIMEROS}(\gamma) - \{\varepsilon\} \in \text{SIGUIENTES}(A)$
- Si existe una producción $B \rightarrow \alpha A$ ó $B \rightarrow \alpha A \gamma$ tal que $\{\varepsilon\} \in \text{PRIMEROS}(\gamma)$ entonces $\text{SIGUIENTES}(B) \subset \text{SIGUIENTES}(A)$

Ejemplo: Calcular los conjuntos PRIMEROS Y SIGUIENTES para la gramática:

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid - T E' \mid \varepsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' | / F T' | \epsilon$$

$$F \rightarrow (E) | \text{num} | \text{id}$$

$\text{PRIMEROS}(E)=\{ ,\text{id},\text{num} \}$

$\text{PRIMEROS}(T)=\{ ,\text{id},\text{num} \}$

$\text{PRIMEROS}(F)=\{ ,\text{id},\text{num} \}$

$\text{PRIMEROS}(E')=\{ +,-, \epsilon \}$

$\text{PRIMEROS}(T')=\{ *,/, \epsilon \}$

$\text{SIGUIENTES}(E)=\{ #,) \}$

$\text{SIGUIENTES}(E')=\{ #,) \}$

$\text{SIGUIENTES}(T)=\{ #,), +, - \}$

$\text{SIGUIENTES}(T')=\{ #,), +, - \}$

$\text{SIGUIENTES}(F)=\{ +,-, *, /,), \# \}$

4.3.2. La condición LL(1)

Para que una gramática sea LL(1) se debe cumplir que:

1. Para las producciones de la forma $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$ se debe cumplir que:

$$\text{PRIMEROS}(\alpha_i) \cap \text{PRIMEROS}(\alpha_j) = \emptyset \text{ para todo } i,j \text{ (} i \neq j \text{)}$$

Esta regla permite decidir que alternativa elegir conociendo sólo un símbolo de la entrada.

2. Si $A \in V_N$, tal que $\{ \epsilon \} \in \text{PRIMEROS}(A)$, entonces:

$$\text{PRIMEROS}(A) \cap \text{SIGUIENTES}(A_j) = \emptyset$$

Esta condición garantiza que para aquellos símbolos que pueden derivar la cadena vacía, el primer símbolo que generan y el siguiente que puede aparecer detrás de ellos sean distintos, sino no sabríamos cómo decidir si vamos a empezar a reconocer el no-terminal o que ya lo hemos reconocido.

Ejercicio. Comprobar si la gramática anterior para generar expresiones aritméticas es LL(1).

4.4. Analizador Sintáctico Predictivo Recursivo

En este método se considera a cada regla de la gramática como la definición de un procedimiento que reconocerá al no-terminal de la parte izquierda.

El lado derecho de la producción especifica la estructura del código para ese procedimiento: los terminales corresponden a concordancias con la entrada, los no-terminales son llamadas a procedimientos y las alternativas a casos condicionales en función de lo que se esté observando en la entrada.

Se asume que hay una variable global que almacena el componente léxico actual (el símbolo de preanálisis), y en el caso en que se produce una concordancia se avanza en la entrada, o en caso contrario se declara un error.

Implementación del analizador sintáctico predictivo recursivo

- Un procedimiento, que llamaremos Parea/Match que recibe como entrada un terminal y comprueba si el símbolo de preanálisis coincide con ese terminal. Si coinciden se avanza en la entrada con el siguiente token, en caso contrario se declara un error sintáctico.
- Un procedimiento para cada no-terminal con la siguiente estructura:
 - Para las reglas de la forma $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$, decidir la producción a utilizar en función de los conjuntos PRIMEROS(α_i). Si preanálisis no pertenece a ningún PRIMEROS(α_i) entonces error sintáctico, excepto si existe la alternativa $A \rightarrow \epsilon$ en cuyo caso no se hace nada.
 - Para cada alternativa α_i del no-terminal, proceder analizando secuencialmente cada uno de los símbolos que aparece en la parte derecha terminales y no-terminales. Si es un no-terminal haremos una llamada a su procedimiento; Si es un terminal haremos una llamada al procedimiento Parea/Match con ese terminal como parámetro.
 - Para lanzar el analizador sintáctico se hace una llamada al procedimiento que corresponde con el símbolo de inicio en la gramática, es decir, el axioma.

Ejercicio.

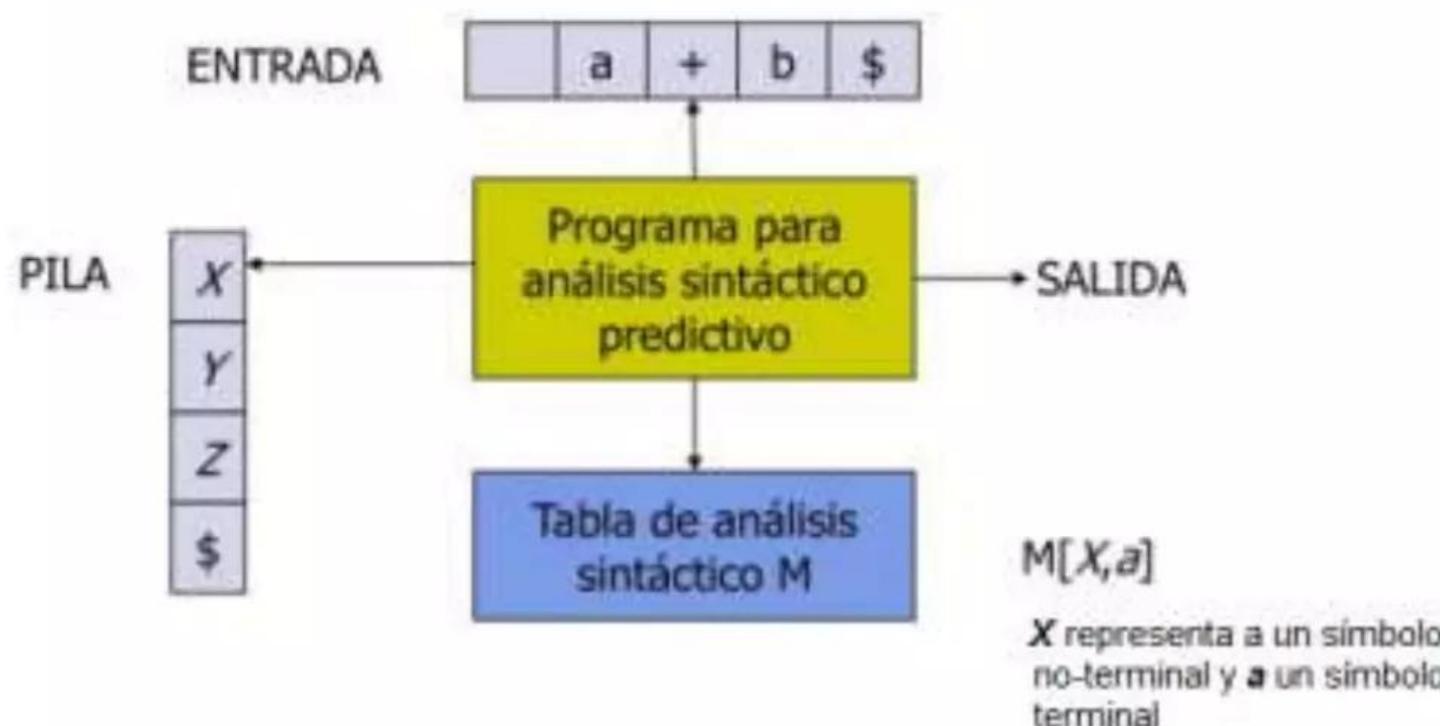
Desarrollar un analizador sintáctico predictivo recursivo para la gramática:

```

S --> x S
S --> A B c
A --> a
B --> b
  
```

4.5. Analizador Sintáctico Predictivo dirigido por Tabla

Se puede construir un analizador sintáctico descendente predictivo no-recursivo sin llamadas recursivas a procedimientos, sino que se utiliza una pila auxiliar de símbolos terminales y no-terminales. Para determinar qué producción debe aplicarse en cada momento se busca en una tabla de análisis sintáctico (parsing table) en función del símbolo de preanálisis que se está observando en ese momento y del símbolo en la cima de la pila se decide la acción a realizar.



- *Buffer de entrada:* que contiene la cadena de componentes léxicos que se va a analizar seguida del símbolo # (ó \$) de fin de cadena.
- *La pila de análisis sintáctico:* que contiene una secuencia de símbolos de la gramática con el símbolo # (ó \$) en la parte de abajo que indica la base de la pila.
- *Una tabla de análisis sintáctico:* que es una matriz bidimensional $M[X, a]$ con $X \in V_{NT}$ y $a \in V_t \cup \{\#\}$. La tabla de análisis sintáctico dirige el análisis y es lo único que cambia de un analizador a otro (de una gramática a otra).
- *Salida:* la serie de producciones utilizadas en el análisis de la secuencia de entrada.

El analizador sintáctico tiene en cuenta en cada momento, el símbolo de la cima de la pila X y el símbolo en curso de la cadena de entrada a (el símbolo de preanálisis). Estos dos símbolos determinan la acción a realizar, que pueden ser:

Si $X == a == \#$ (la pila está vacía y no hay mas símbolos en la entrada), el análisis sintáctico ha llegado al final y la cadena ha sido reconocida con éxito.

Si $X == a \neq \#$, el terminal se desapila (se ha reconocido ese terminal) y se avanza en la entrada obteniendo el siguiente componente léxico.

Si X es no-terminal, entonces se consulta en la tabla $M[X, a]$. La tabla contiene una producción a aplicar o si está vacía significa un error. Si es una producción de la forma $X \rightarrow Y_1 Y_2 \dots Y_n$, entonces se apilan los símbolos $Y_n \dots Y_1$ (notar el orden inverso). Si es un error el analizador llama a una rutina de error.

El análisis sintáctico dirigido por tabla es más eficiente en cuanto a tiempo de ejecución y espacio, puesto que usa una pila para almacenar los símbolos a reconocer en vez de llamadas a procedimientos recursivos.

Tabla de análisis sintáctico

La tabla de análisis sintáctico se forma de la siguiente manera:

Entrada: una gramática G

Salida: la tabla de análisis sintáctico, con una fila para cada no-terminal, una columna para cada terminal y otra para # (ó \$).

Método:

Ampliar la gramática con una producción $S' \rightarrow S$ donde S es el axioma.

Para cada producción de la gramática $A \rightarrow \alpha$ hacer:

- Para cada terminal $a \in PRIMEROS(\alpha)$, añadir la producción $A \rightarrow \alpha$ en la casilla $M[A,a]$
- Si $\varepsilon \in PRIMEROS(\alpha)$, añadir $A \rightarrow \alpha$ en la casilla $M[A,b]$ para todo $b \in SIGUIENTE(A)$

Las celdas de la tabla que hayan quedado vacías se definen como error.

Puede aparecer más de una producción en una misma casilla. Esto supondría que estamos en el caso no-determinista en el que tendríamos que probar una producción y si ésta produce un error, probar la otra... sería un método ineficiente.

Las gramáticas LL(1) garantizan que sólo tenemos una producción por casilla. Así, el costo del análisis es lineal con el número de componentes léxicos de la entrada.

Ejemplo:

Formar la tabla de análisis sintáctico para la siguiente gramática:

$$E \rightarrow T E'$$

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

$$T \rightarrow F T'$$

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

$$F \rightarrow (E) \mid id$$

No es necesario aumentar la gramática, puesto que no existe confusión de producciones partiendo del axioma.

$$PRIMEROS(E)=\{ (, id \}$$

$$PRIMEROS(T)=\{ (, id \}$$

$$PRIMEROS(F)=\{ (, id \}$$

$$PRIMEROS(E')=\{ +, \varepsilon \}$$

$$PRIMEROS(T')=\{ *, \varepsilon \}$$

SIGUIENTES(E)={ \$,) }

SIGUIENTES(E')={ \$,) }

SIGUIENTES(T)={ +,\$,) }

SIGUIENTES(T')={ +, \$,) }

SIGUIENTES(F)={ *, + , \$,) }

Gramática:

Tabla M:

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

No Terminal	Símbolo de entrada					
	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	$F \rightarrow id$			$F \rightarrow (E)$		

Ejemplo:

Haciendo uso de la pila, la entrada y la tabla anterior, procesar la entrada: *id + id * id*

PILA	ENTRADA	SALIDA
\$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\$	$T' \rightarrow \epsilon$
\$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'	\$	$T' \rightarrow \epsilon$
\$	\$	$E' \rightarrow \epsilon$

Análisis sintáctico y acciones semánticas

Las acciones semánticas insertadas en las producciones de la gramática se pueden considerar como si fueran símbolos adicionales y que tienen asociado un procedimiento a ejecutar (Símbolos de acciones). Las llamadas a rutinas semánticas no se les pasa parámetros explícitos, los parámetros necesarios se pueden transmitir a través de una pila semántica. En los analizadores descendentes recursivos, la pila de análisis sintáctico está “escondida” en la pila de llamadas recursivas a los procedimientos. La pila semántica también se puede “esconder”, si hacemos que cada rutina semántica devuelva información.

Un programa para gestionar acciones semánticas sería:

Repetir

```

Sea X el símbolo de la cima de la pila
Si X es un terminal o #
    Si X == preanálisis
        Extraer X de la pila y obtener nuevo componente léxico
    Sino error();
Sino    si M[X,preanálisis] = X → Y1Y2...Yn
        Extraer X de la pila
        Meter Yn...Y2Y1 con Y1 en la cima de la pila
    Sino    si X es símbolo de acción
            Desapilar, llamar a rutina semántica correspondiente
    Sino
        Error();
Hasta _que X == # // Siempre que la pila esté vacía y se haya procesado toda la entrada

```

5. Análisis Sintáctico Ascendente

En el analizador sintáctico ascendente, se construye el árbol de análisis sintáctico de la cadena de entrada desde las hojas hasta la raíz. En las hojas tenemos la cadena a analizar (los símbolos terminales) que se intentan reducir al axioma, que se encontrará en la raíz, si la cadena es correcta sintácticamente.

Se trata de **desplazarse** en la entrada hasta encontrar una subcadena de símbolos que represente la parte derecha de una producción, en ese momento sustituimos esa subcadena por el no-terminal de la parte izquierda correspondiente de la producción (**reducimos**)

Los métodos ascendentes se caracterizan porque analizan la cadena de componentes léxicos de izquierda a derecha y obtienen la derivación más a la derecha.

En el caso de los métodos deterministas (no hay vuelta atrás) se tiene que al analizar un sólo símbolo de preanálisis, sabemos exactamente en todo momento que acción realizar: bien sea desplazarnos en la entrada o bien aplicar una reducción. En el caso de una reducción debemos saber de forma única qué producción aplicar.

Ejemplo:

Considerando la siguiente gramática, realice el árbol de análisis sintáctico y represente el análisis ascendente para la siguiente entrada: id + id +id

$$E \rightarrow E + E$$

$$E \rightarrow E \wedge E$$

$$E \rightarrow id$$

$$E \rightarrow num$$

Conflictos

Cuando aplicamos la técnica ascendente puede presentarse los siguientes conflictos.

- Conflicto desplazar – reducir
- Conflicto reducir – reducir.

5.1. Métodos de análisis sintáctico LR

Los métodos de análisis sintáctico LR permiten reconocer la mayoría de las construcciones de los lenguajes de programación. Son métodos muy potentes, con mayor poder de reconocimiento que los métodos LL (Las gramáticas LL son un subconjunto de las gramáticas LR).

Las gramáticas LR(k) se caracterizan porque:

- L: Procesamos la cadena de entrada de izquierda a derecha.
- R: proporcionan la derivación más a la derecha de la cadena de entrada.
- K: se examinan k -símbolos de la entrada por anticipado para tomar la decisión sobre la acción a realizar.

Ventajas: es un método potente que permite reconocer la mayoría de las construcciones de los lenguajes de programación con $k=0,1$. Es un método sin retroceso y por tanto determinista.

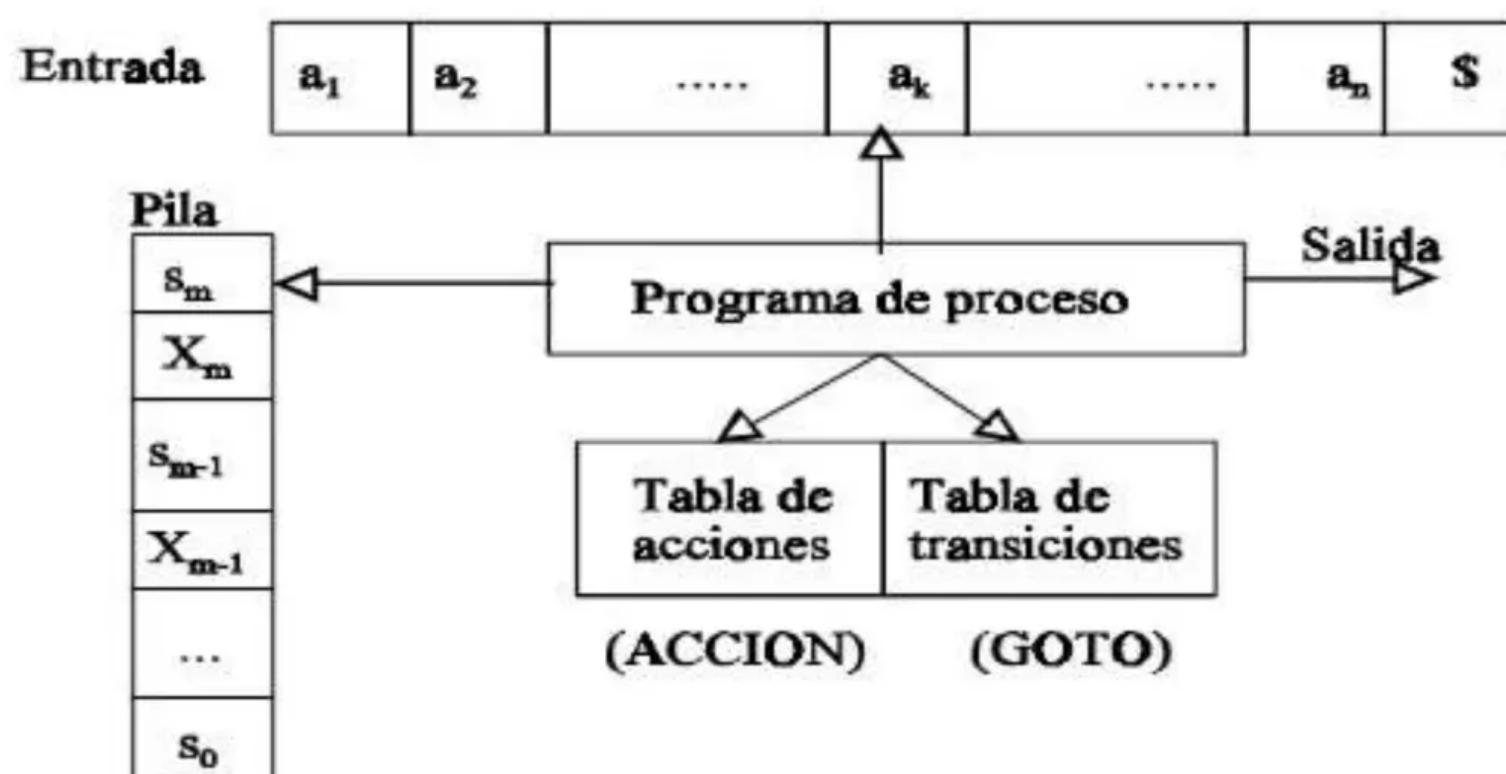
Desventajas: los métodos son difíciles de comprender, de implementar a mano y de realizar una traza. Necesidad de usar generados automáticos de analizadores sintácticos LR. Por ejemplo: Bison (para Linux) y Yacc (yet another compiler compiler) para Unix.

Dentro del análisis sintáctico LR se distinguen cuatro técnicas:

- **El método LR(0)**. Es el más fácil de implementar, pero el que tiene menos poder de reconocimiento. No usa la información del símbolo de preanálisis para decidir la acción a realizar.
- **El método SLR(1)** (Simple LR). Usa ya un símbolo de preanálisis.
- **El método LR(1)**. Es el más poderoso y costoso. El tamaño del autómata a pila para el reconocimiento se incrementa considerablemente.
- **El método LALR(1)** (Look-Ahead LR). Es una versión simplificada del LR(1), que combina el costo en tamaño (eficiencia) de los métodos SLR con la potencia del LR(1).

La diferencia que hace a los métodos LR más poderosos que los LL es que los primeros disponen de la información de la entrada y los estados por lo que ha ido pasando el analizador (la cadena que ha ido reconociendo hasta ahora), mientras que los métodos LL sólo disponen de la información de la entrada.

Modo de operación del análisis sintáctico ascendente



- La **entrada** formada por la serie de componentes léxicos a reconocer.
- Un **programa de proceso** que lee tokens de la cadena de entrada de uno en uno y utiliza una pila para almacenar los símbolos que va reconociendo y los estados por los que pasa el analizador.
- Una **pila** cuyo contenido es de la forma $s_0X_1s_1X_2s_2\dots$ donde cada X_i son símbolos de la gramática que se van reconociendo, $X_i \in (V_T \cup V_{NT})$, y los s_i son los estados por los que pasa el analizador. Los símbolos terminales se introducen en la pila mediante desplazamientos de la cadena de entrada a la pila, los no-terminales se apilan como resultado de hacer una reducción.
- La **tabla** de análisis sintáctico está formada por dos partes. La primera parte es de la forma **acción**, $\text{acción}(s_m, a_i)$ indica que acción se debe de realizar si observamos en la entrada el token a_i y el analizador está en el estado s_m . Las acciones posibles son: desplazar, reducir, error, aceptar. La segunda parte es de la forma **goto**, $ir_a(s_m, V_{NT})$ indica el estado al que tenemos que ir después de hacer una reducción.
- **Salida**: la derivación más a la derecha de la cadena de entrada (de abajo hacia arriba)

En general necesitamos:

- Un mecanismo que nos determine el tipo de acción a realizar: desplazar o reducir.
- En el caso de que tengamos que reducir nos debe proporcionar la subcadena de símbolos a reducir y qué producción utilizar.

Este mecanismo nos lo proporciona el Autómata Finito Determinista de elementos LR(0).

5.2. Elementos LR(0) y construcción del Autómata Finito de elementos LR(0)

Un elemento de análisis sintáctico LR(0) de una gramática G, es una producción de G con un punto en alguna posición del lado derecho.

$$[A \rightarrow \beta_1 \cdot \beta_2]$$

Siendo $A \rightarrow \beta_1 \cdot \beta_2$ una producción. Este punto separa la parte que se ha analizado hasta ese momento, y por tanto está en la cima de la pila, de la parte que se espera aún analizar, que es el resto de la cadena de entrada. Por ejemplo para la producción $A \rightarrow XYZ$ tenemos los posibles siguientes ítems LR(0):

$A \rightarrow \bullet XYZ$ se espera en la entrada una cadena derivable de XYZ

$A \rightarrow X \bullet YZ$ se ha reconocido una cadena derivable de X y se espera en la entrada una cadena derivable de YZ

$A \rightarrow XYZ \bullet$ Item completo. Indica que ya se ha reconocido por completo una cadena derivable de A

Autómata finito de elementos LR(0)

Los **estados** son los propios ítems o elementos LR(0).

Las **transiciones** son de la forma:

Si $X \in V_T$ tenemos una transición

$$[A \rightarrow \alpha \cdot X \beta] \xrightarrow{X} [A \rightarrow \alpha X \cdot \beta]$$

Si $X \in V_{NT}$ tenemos una transición

$$[A \rightarrow \alpha \cdot X \beta] \xrightarrow{X} [A \rightarrow \alpha X \cdot \beta]$$

Y además

$$[A \rightarrow \alpha \cdot X \beta] \xrightarrow{\varepsilon} [X \rightarrow \cdot \gamma] \quad \forall(X \rightarrow \gamma)$$

5.3. Análisis LR(0)

El análisis LR(0) es el más sencillo de los métodos ascendentes. No usa información de la entrada, para decidir la acción a realizar.

Algoritmo de análisis LR(0)

Se S el estado de la cima de la pila. Entonces:

- Si el estado S contiene cualquier ítem del la forma $A \rightarrow \alpha \cdot X\beta$ con X terminal $X \in V_T$, entonces la acción es **desplazar** X a la pila. El nuevo estado al que pasa el analizador, y que se coloca en la cima de la pila, es aquel que contenga un ítem de la forma $A \rightarrow \alpha X \cdot \beta$
- Si el estado S contiene un ítem de la forma $A \rightarrow \alpha \cdot$, entonces la acción es **reducir** aplicando la producción $A \rightarrow \alpha$. Sacamos $2 ||\alpha||$ elementos de la pila y el estado que queda al descubierto debe contener un ítem de la forma $B \rightarrow \gamma \cdot A\sigma$, insertamos el no-terminal en la pila y como nuevo estado, aquel que contenga el ítem de la forma $B \rightarrow \gamma A \cdot \sigma$
- Si no se puede realizar ninguna de las acciones anteriores entonces error.

Se dice que una gramática es LR(0) si la aplicación de las reglas anteriores no son ambiguas, es decir, que si un estado contiene un ítem de la forma $A \rightarrow \alpha \cdot$, entonces no puede contener cualquier otro ítem. Si se produjera eso aparecerían conflictos. En efecto:

- Si tuviera además un ítem de la forma $A \rightarrow \alpha X \cdot \beta$ con X terminal tendríamos un conflicto *desplaza-reduce*.
- Si tuviera además un ítem de la forma $B \rightarrow \beta \cdot$, se produciría un conflicto *reduce-reduce*.

Por tanto, una gramática es LR(0) si cada estado es bien un estado de desplazamiento (contiene sólo ítems que indican desplazar) o bien un estado de reducción (contiene un único ítem completo)

Ejemplo:

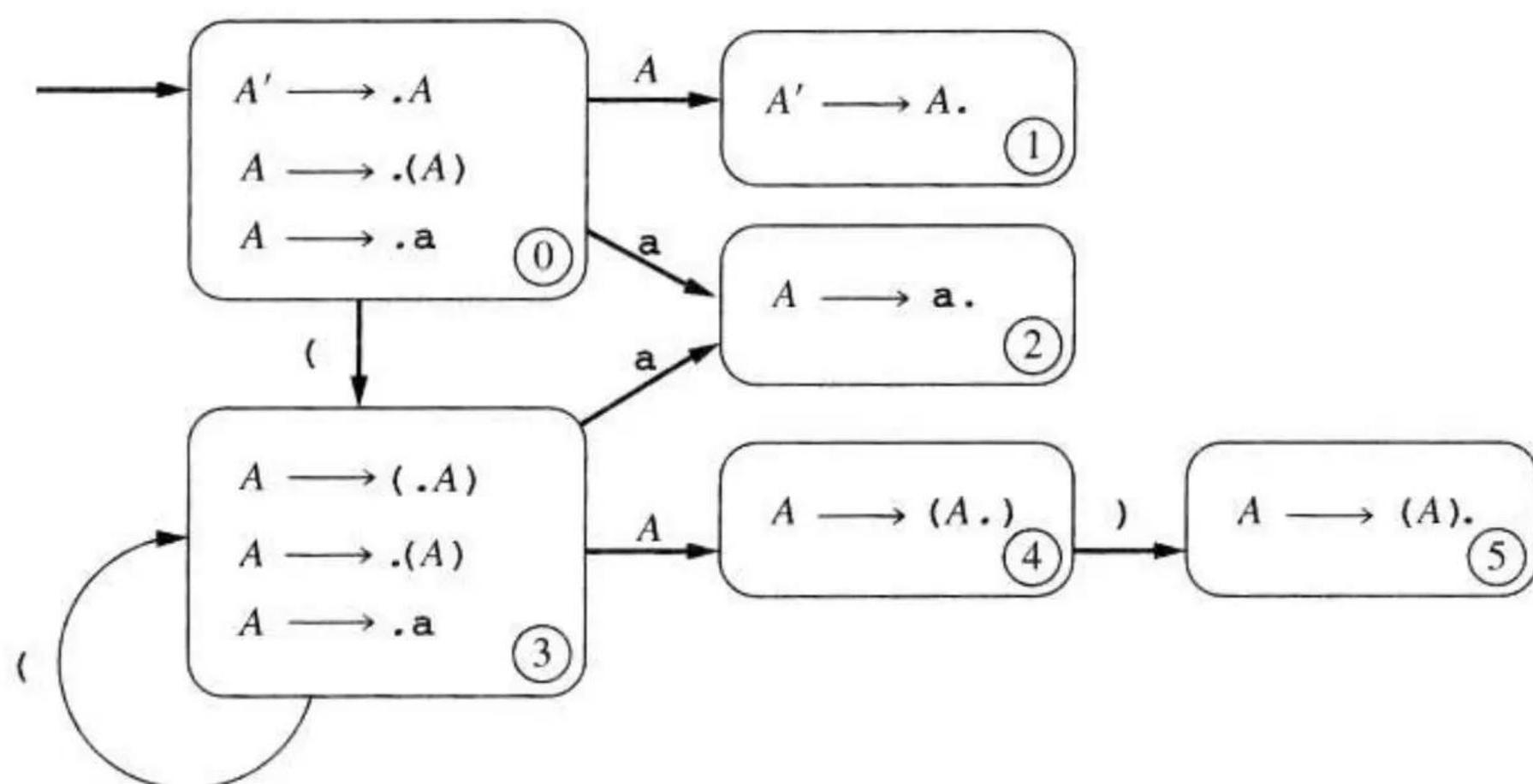
Considere la siguiente gramática para generar paréntesis anidados:

$$A \rightarrow (A) \mid a$$

Realizar los siguientes pasos:

- Aumentar la gramática
- Crear el autómata finito no determinista
- Crear el autómata finito determinista

Como producto del ejercicio anterior, el AFD de elementos LR(0) es el siguiente



A partir del autómata se construye la tabla de análisis sintáctico:

Estado	Acción	Regla	Entrada			Goto
			(a)	
0	desplazamiento					A
1	reducción	$A' \rightarrow A$	3			1
2	reducción	$A \rightarrow a$		2		
3	desplazamiento		3			
4	desplazamiento			2		4
5	reducción	$A \rightarrow (A)$			5	

Debe observarse que no se ha necesitado conocer el tipo de token que teníamos en la entrada (preanálisis) para decidir la acción a realizar.

Ejercicio:

Dada la tabla anterior, determine la validez de la siguiente entrada: ((a))

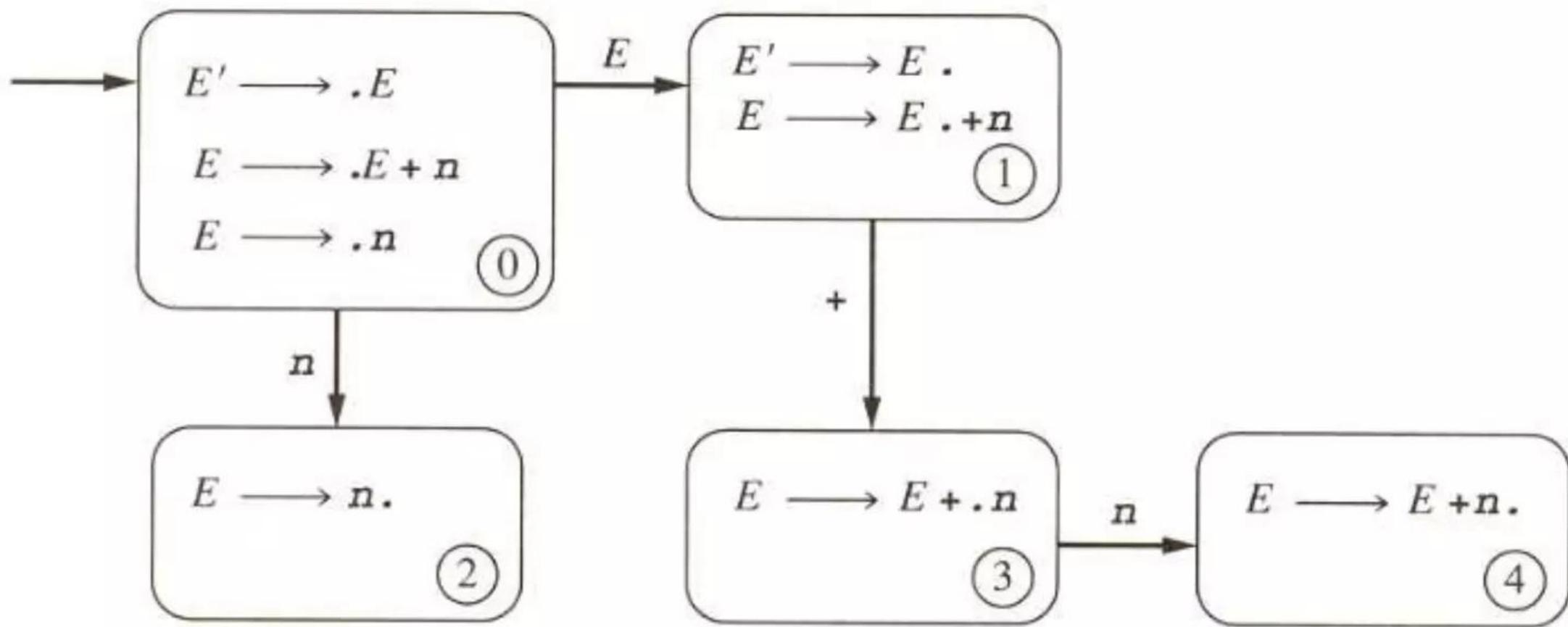
	Pila de análisis sintáctico	Entrada	Acción
1	\$ 0	((a)) \$	desplazamiento
2	\$ 0 (3	(a)) \$	desplazamiento
3	\$ 0 (3 (3)) \$	desplazamiento
4	\$ 0 (3 (3 a 2) \$	reducción $A \rightarrow a$
5	\$ 0 (3 (3 A 4) \$	desplazamiento
6	\$ 0 (3 (3 A 4) 5) \$	reducción $A \rightarrow (A)$
7	\$ 0 (3 A 4) \$	desplazamiento
8	\$ 0 (3 A 4) 5	\$	reducción $A \rightarrow (A)$
9	\$ 0 A 1	\$	aceptar

Limitaciones del análisis LR(0):

Supongamos la gramática siguiente:

$$E \rightarrow E + n \mid n$$

El AFD de elementos LR(0) correspondiente es:



En el estado 1 tenemos un ítem que indica desplazar y otro que indica reducir. Tenemos un conflicto *desplaza/reduce* según el algoritmo LR(0).

Existen determinadas construcciones de los lenguajes de programación que no pueden ser reconocidas por este tipo de método. Así, para solucionar este problema tenemos que recurrir a otro tipo de análisis, como por ejemplo el análisis SLR(1).

5.4. Análisis Sintáctico Ascendente SLR(1)

Este tipo de análisis usa el AFD construido a partir de elementos LR(0) y usa el *token* de la cadena de entrada para determinar el tipo de acción a realizar. Este método consulta el *token* de la entrada antes de realizar un desplazamiento para tener seguro que existe una transición correspondiente en el estado en el que se encuentra el analizador y en el caso de que se tenga que hacer una reducción se comprueba que el *token* actual pertenece al conjunto de SIGUIENTE del no-terminal al que se quiere reducir.

Algoritmo de análisis SLR(1)

Se S el estado de la cima de la pila. Entonces:

- Si el estado S contiene cualquier ítem del la forma $A \rightarrow \alpha \cdot X\beta$ con X terminal $X \in V_T$, y X es el siguiente terminal en la entrada entonces la acción es **desplazar** X a la pila. El nuevo estado al que pasa el analizador, y que se coloca en la cima de la pila, es aquel que contenga un item de la forma $A \rightarrow \alpha X \cdot \beta$
- Si el estado S contiene un ítem de la forma $A \rightarrow \alpha \cdot$, y el siguiente *token* en la entrada está en $\text{SIGUIENTE}(A)$, entonces la acción es **reducir** aplicando la producción $A \rightarrow \alpha$. Sacamos $2 ||\alpha||$ elementos de la pila y el estado que queda al descubierto debe contener un ítem de la forma $B \rightarrow \gamma \cdot A\sigma$, insertamos el no-terminal A en la pila y como nuevo estado, aquel que contenga el item de la forma $B \rightarrow \gamma A \cdot \sigma$
- Si el símbolo en la entrada es tal que no se puede realizar ninguna de las acciones anteriores entonces error.

Se dice que una gramática es SLR(1) si la aplicación de las reglas anteriores no son ambiguas, es decir, una gramática es SLR(1) si cumple para cada estado una de las siguientes condiciones:

- Para cada ítem de la forma $A \rightarrow \alpha \cdot X\beta$ en S con X terminal, no existe un ítem completo $B \rightarrow \gamma \cdot$ en S , con $X \in \text{SIGUIENTE}(B)$. Sino tendríamos un conflicto *desplaza-reduce*.
- Para cada par de ítems completos $A \rightarrow \alpha \cdot$ y $B \rightarrow \beta \cdot$ en S , entonces $\text{SIGUIENTE}(A) \cap \text{SIGUIENTE}(B) = \emptyset$. De lo contrario se produciría un conflicto *reduce-reduce*.

El aspecto de la tabla de análisis sintáctico cambia respecto al análisis LR(0), puesto que un estado puede admitir desplazamientos o reducciones dependiendo del *token* en la entrada. Por tanto, cada entrada debe tener una etiqueta de desplazamiento o reducción

Ejemplo:

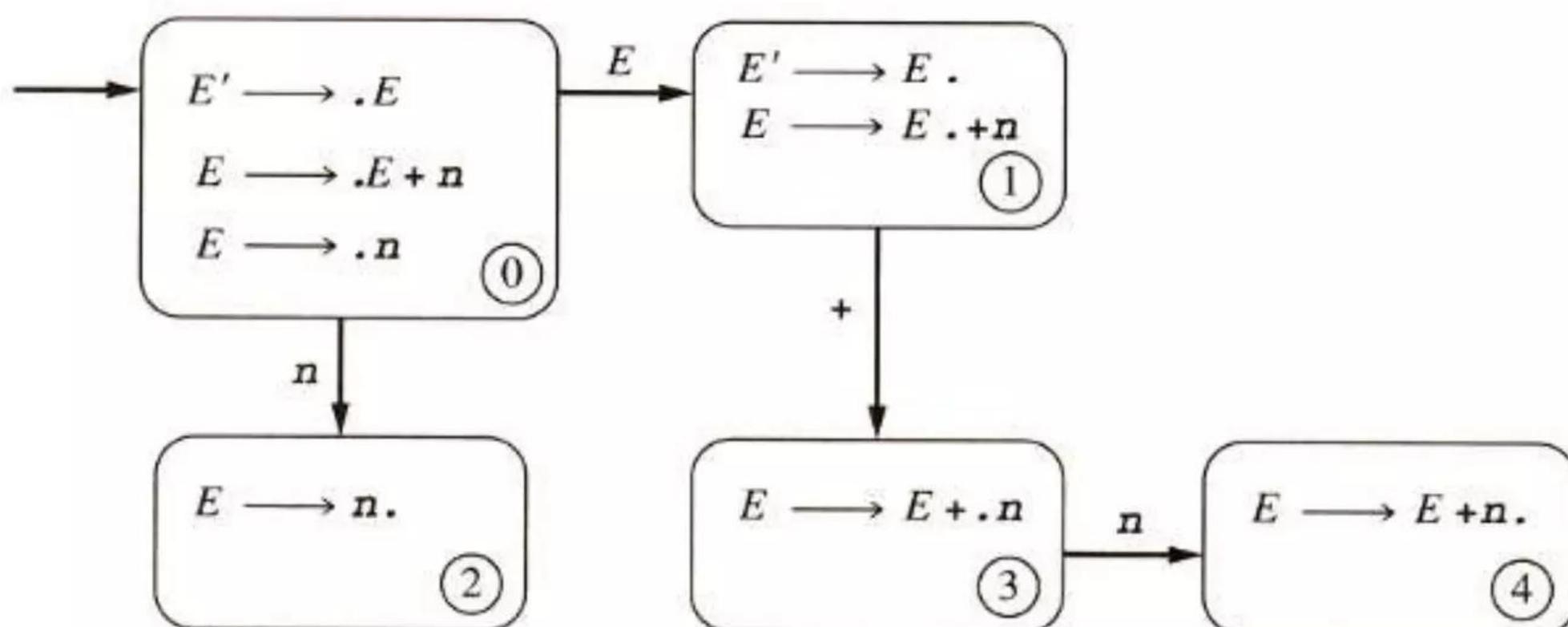
Considere la siguiente gramática:

$$E \rightarrow E + n \mid n$$

Realizar los siguientes pasos:

- Aumentar la gramática
- Crear el autómata finito no determinista
- Crear el autómata finito determinista

Como producto del ejercicio anterior, el AFD de elementos LR(0) es el siguiente:



Comprobemos que según el análisis SLR(1), ahora no existe conflicto desplaza/reduce en el estado 1. En efecto: $\{+\}$ no pertenece a SIGUIENTE(E')

La tabla de análisis sintáctico es:

	Ir a			
	+	n	\$	E
0		D-2		1
1	D-3		Aceptar	
2	R: E → n		R: E → n	
3		D-4		
4	R: E → E + n		R: E → E + n	

Ejercicio:

Dada la tabla anterior, determine la validez de la siguiente entrada: $n+n+n$

Pila de análisis sintáctico	Entrada	Acción
\$ 0	$n + n + n \$$	desplazamiento 2
\$ 0 n 2	$+ n + n \$$	reducción $E \rightarrow n$
\$ 0 E 1	$+ n + n \$$	desplazamiento 3
\$ 0 E 1 + 3	$n + n \$$	desplazamiento 4
\$ 0 E 1 + 3 n 4	$+ n \$$	reducción $E \rightarrow E + n$
\$ 0 E 1	$+ n \$$	desplazamiento 3
\$ 0 E 1 + 3	$n \$$	desplazamiento 4
\$ 0 E 1 + 3 n 4	$\$$	reducción $E \rightarrow E + n$
\$ 0 E 1	$\$$	aceptar

Limitaciones del análisis SLR(1):

El análisis SLR(1) es simple y eficiente. Es capaz de describir prácticamente todas las estructuras de los lenguajes de programación. Pero existen algunas situaciones donde este tipo de análisis no es suficientemente poderoso y falla, especialmente en aquellos casos donde se produce un conflicto reduce/reduce. Los conflictos reduce/reduce no son habituales y suelen ser un síntoma de un mal diseño de la gramática.

5.5. Análisis Sintáctico LR

Este es el método más poderoso que existe, a costa de que se incrementa la complejidad (un factor de 10 respecto a los métodos que usan el AFD de items LR(0)).

Las limitaciones del método LR(0) y SLR(1) radican en que tienen en cuenta el símbolo de preanálisis después de la construcción del AFD de items LR(0), que por sí misma ignora los símbolos siguientes en la entrada.

Elementos LR(1) y construcción del autómata finito de elementos LR(1)

Un elemento de análisis sintáctico LR(1) de una gramática G, es un par consistente de un ítem LR(0) y un símbolo de preanálisis.

$$[A \rightarrow \alpha \cdot \beta, a]$$

Siendo $A \rightarrow \alpha \cdot \beta$ in ítem LR(0) y a el token de preanálisis.

Un ítem completo $[A \rightarrow \alpha \beta \cdot, a]$, indica que ya se ha reconocido por completo una cadena derivable de A y reduciré siempre que en la entrada tenga el token a .

Autómata finito de elementos LR(1)

Los *estados* son los propios ítems LR(1).

Las *transiciones* son de la forma:

Si $X \in V_T$ tenemos una transición

$$[A \rightarrow \alpha \cdot X\gamma, a] \xrightarrow{X} [A \rightarrow \alpha X \cdot \gamma, a]$$

Si $X \in V_{NT}$ tenemos una transición

$$[A \rightarrow \alpha \cdot X\gamma, a] \xrightarrow{X} [A \rightarrow \alpha X \cdot \gamma, a]$$

Y además transiciones vacías

$$[A \rightarrow \alpha \cdot X\gamma, a] \xrightarrow{\varepsilon} [X \rightarrow \cdot \beta, b] \quad \forall X \rightarrow \beta \quad y \quad \forall b \in PRIMERO(\gamma a)$$

El estado inicial del AFD se obtiene ampliando la gramática, como para el caso de items LR(0), y poniendo como símbolo de preanálisis #.

$$[S' \rightarrow \cdot S, \#]$$

Que significa que reconoceremos una cadena derivable de S seguida del símbolo #.

Algoritmo de análisis sintáctico LR(1)

Es básicamente igual al algoritmo de análisis SLR(1), excepto que usa el símbolo de preanálisis en vez del conjunto de SIGUIENTE.

Sea S el estado de la cima de la pila. Entonces:

- Si el estado S contiene cualquier ítem del la forma $[A \rightarrow \alpha \cdot X\beta, a]$ con X terminal $X \in V_T$, y X es el siguiente terminal en la entrada, entonces la acción es **desplazar** X a la pila. El nuevo estado al que pasa el analizador, y que se coloca en la cima de la pila, es aquel que contenga un item de la forma $[A \rightarrow \alpha X \cdot \beta, a]$
- Si el estado S contiene un ítem de la forma $[A \rightarrow \alpha \cdot, a]$, y el siguiente *token* en la entrada es a , entonces la acción es **reducir** aplicando la producción $A \rightarrow \alpha$. Sacamos 2 $|\alpha|$ elementos de la pila y el estado que queda al descubierto debe contener un ítem de la forma $[B \rightarrow \gamma \cdot A\sigma, b]$, insertamos el no-terminal A en la pila y como nuevo estado, aquel que contenga el item de la forma $[B \rightarrow \gamma A \cdot \sigma, b]$
- Si el símbolo en la entrada es tal que no se puede realizar ninguna de las acciones anteriores entonces error.

Se dice que una gramática es LR(1) si la aplicación de las reglas anteriores no son ambiguas, es decir, una gramática es LR(1) si cumple para cada estado una de las siguientes condiciones:

- Para cada ítem de la forma $[A \rightarrow \alpha \cdot X\beta, a]$ en S con X terminal, entonces no existe un ítem completo $[B \rightarrow \gamma \cdot, X]$ en S . Si no tendríamos un conflicto *desplaza-reduce*.
- No existen dos ítems completos de la forma $[A \rightarrow \alpha \cdot, a]$ y $[B \rightarrow \beta \cdot, a]$ en S . De lo contrario se produciría un conflicto *reduce-reduce*.

La tabla de análisis sintáctico tiene el mismo aspecto que para el caso SLR(1).

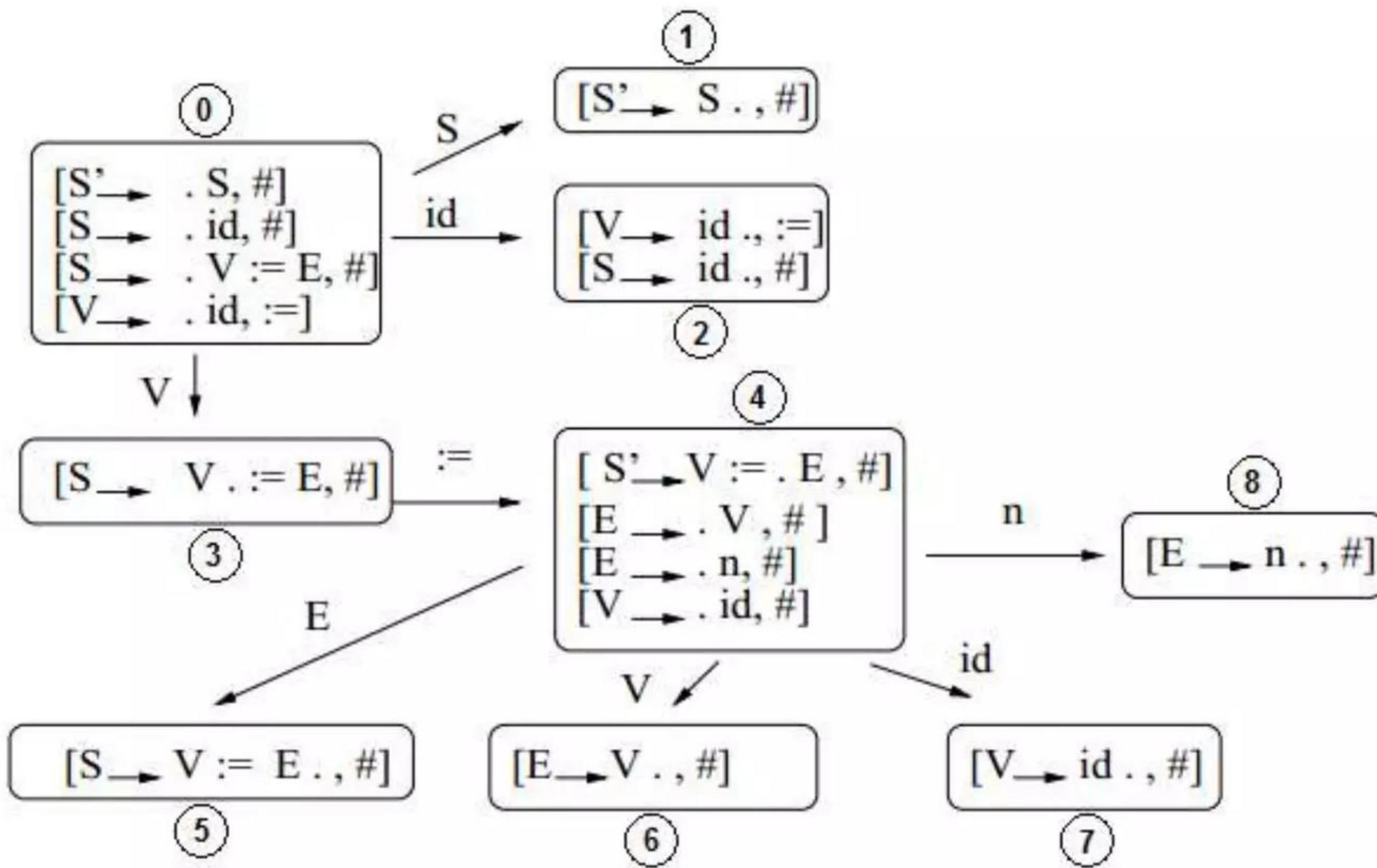
Ejemplo:

Formar el AFD de conjuntos de elementos LR(1) para la siguiente gramática

$$S \rightarrow \text{id} \mid V := E$$

$$V \rightarrow \text{id}$$

$$E \rightarrow V \mid n$$



Obsérvese que el estado 2 presenta dos elementos completos pero se evita el conflicto.

5.6. Análisis Sintáctico LALR(1)

Una modificación del método LR(1), llamada método LALR(1) mantiene el poder del LR(k) y preserva la eficiencia del SLR(1). El análisis LALR(1) se basa en la observación de que en muchos casos el tamaño grande del AFD de ítems LR(1) se debe a la existencia de muchos estados diferentes que tienen igual la primera componente, los ítems LR(0), y difieren sólo de la segunda componente, los símbolos de preanálisis.

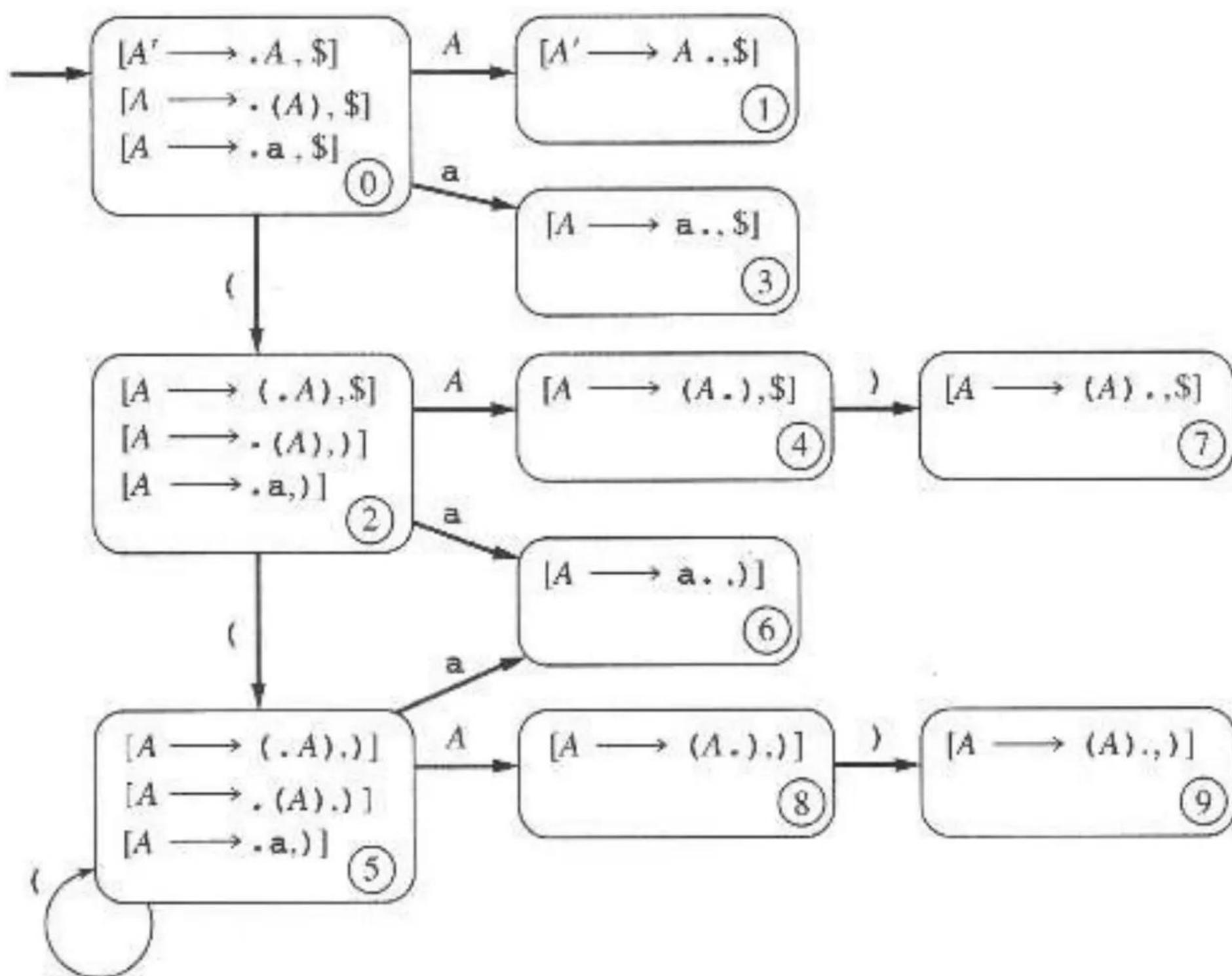
Lo que se hará es identificarlos como un único estado combinando los símbolos de preanálisis. Si el proceso está bien hecho, llegaremos a un autómata como el de ítems LR(0), excepto que cada estado tienen símbolos de preanálisis.

El algoritmo de análisis LALR(1) es idéntico al LR(1).

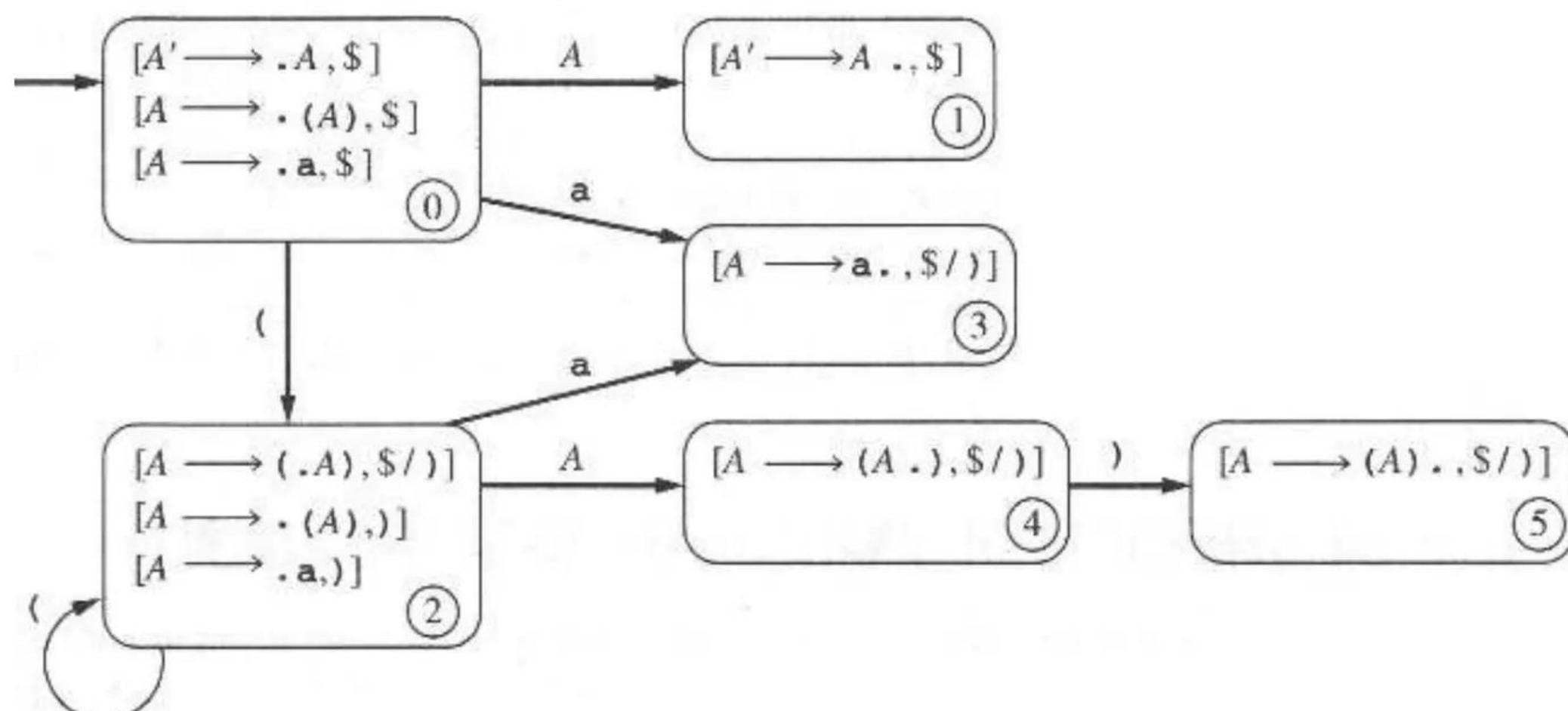
Considere la gramática de paréntesis anidados:

$$A \rightarrow (A) \mid a$$

El AFD de conjuntos de elementos LR(1) es:



El AFD de conjunto de elementos LALR(1) es:



Obsérvese que:

- Los estados 2 y 5, 4 y 8, 7 y 9, 3 y 6 del AFD LR(1), proporciona el AFD LALR(1).
- En el ejemplo se denota las búsquedas hacia delante múltiples escribiendo una / entre ellas. Así, el elemento LALR(1) [$A \rightarrow \alpha . \beta, a / b / c$] tiene un conjunto de búsqueda hacia delante compuesto de los símbolos a, b y c.

Bibliografía

Aho, Lam, Sethy, Ullman (2008) “compiladores: principios, técnicas y herramientas”

Kenneth C. Louden (2004) “Construcción de compiladores: Principios y práctica”

Alfonseca; Ortega, Pulido (2006) “Compiladores e intérpretes: Teoría y práctica”