UNIVERSIDAD DE MAGALLANES
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE COMPUTACIÓN

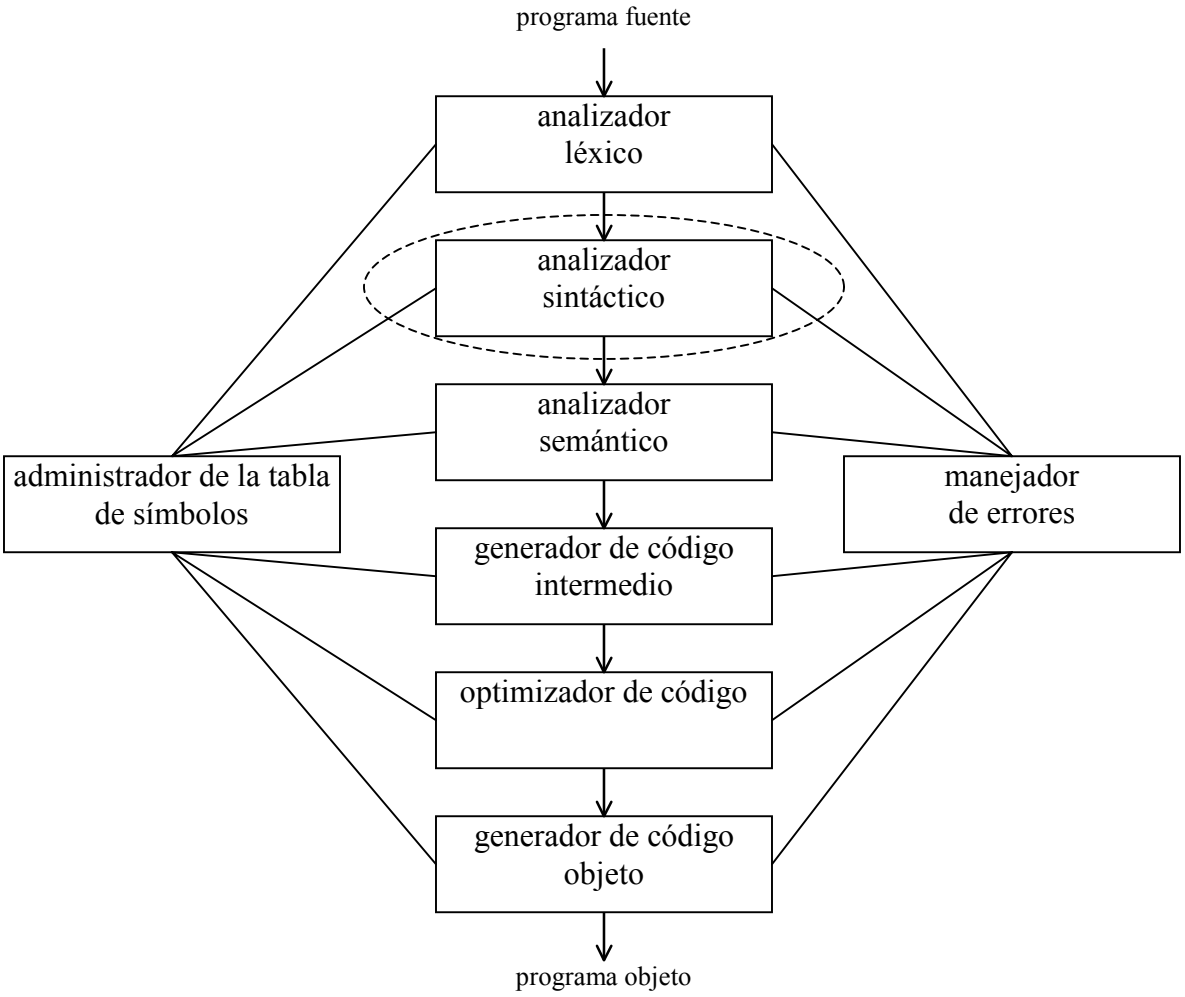
FUNCIONES DEL ANALIZADOR SINTÁCTICO

Elaborado el Domingo 19 de Septiembre de 2004

I.- FUNCIÓN DEL ANALIZADOR SINTÁCTICO

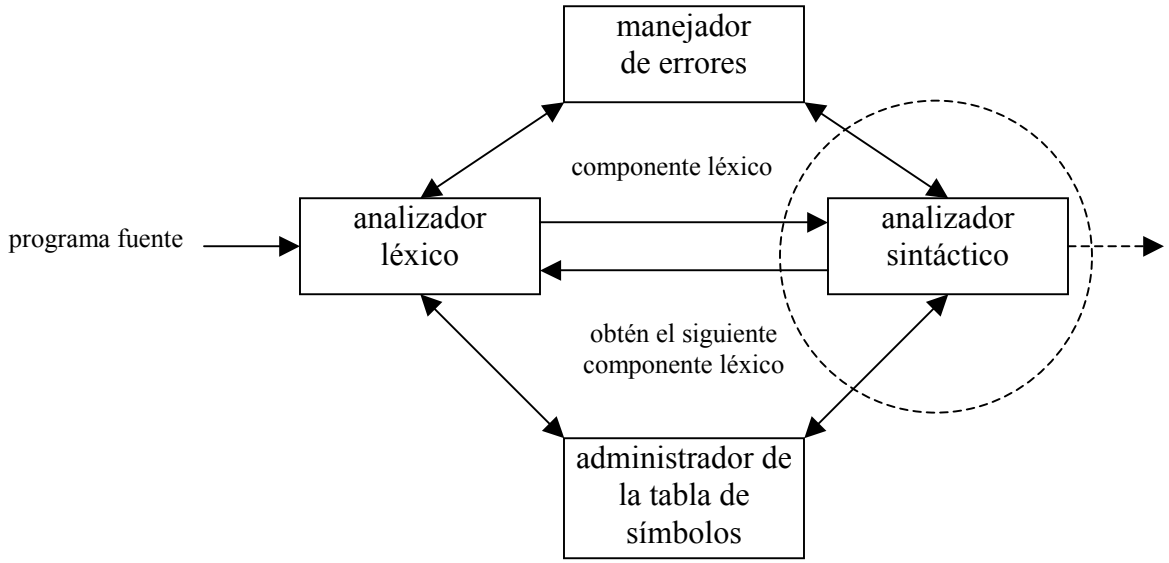
(Las figuras fueron extraídas de "Compiladores: Principios, técnicas y Herramientas", Aho, Sethi y Ullman)

Dado un diagrama de fases de compilación como el siguiente:



Fases de un compilador

Se debe notar que la segunda fase de un compilador es el análisis sintáctico. Su principal función es analizar la secuencia de componentes léxicos de la entrada para verificar que cumplen con las reglas gramaticales especificadas. Esta interacción se aplica bajo un esquema donde el analizador léxico es una subrutina o corutina del analizador sintáctico. Recibida la orden "obtén el siguiente componente léxico" del analizador sintáctico, el analizador léxico lee los caracteres de entrada hasta que pueda identificar el siguiente componente léxico.



Interacción de un analizador léxico con un analizador sintáctico

En general, las principales funciones que realiza un analizador sintáctico son las siguientes.

- Recibir los componentes léxicos y producir como salida una representación del árbol sintáctico que reconoce la entrada de acuerdo a la gramática especificada.
- Interactuar con la Tabla de Símbolos. Una estructura que mantiene todos los símbolos presentes en la entrada.
- Chequear que los tipos de datos están asignados correctamente para evitar la pérdida de información o los errores semánticos.
- Generar un Código Intermedio, ya sea para una máquina virtual o real, que permita la ejecución o interpretación de la entrada.
- Informar de los errores encontrados en la entrada.

II.- GRAMÁTICAS LIBRES DE CONTEXTO

Según la *clasificación de Noam Chomsky*, los lenguajes de programación como C, no pueden ser definidos mediante lenguajes regulares, debido a sus estructuras *inherentemente recursivas*. Pero si pueden ser descritos por *gramáticas libres de contexto*.

G) **Tipo 0** (Chomsky) Sin restricciones.

M) Máquinas de Turing

L) Problemas Recursivamente Enumerables.

Describe cualquier suceso computable.

G) **Tipo 1** Sensibles al contexto

M) Autómata acotado linealmente

L) Dependiente de contexto

G) **Tipo 2** De contexto libre

M) Autómatas a pila

L) Independientes de contexto.

Sintaxis de lenguajes como Pascal, C...

G) **Tipo 3** Gramáticas regulares

M) Autómatas finitos, probabilísticos y Células de McCulloch-Pitts

L) Expresiones regulares.

Analizador léxico (indica si se reconoce la palabra)

Editores de Texto (Reconocen /Reemplazan la palabra)

Clasificación de los Lenguajes

(extraído de Apunte "Autómatas", trabajo realizado por Alicia de Alvaro Martín, Lara Barrio Márquez, María Elena Rincón Arribas)

Las gramáticas libres de contexto nos proporcionarán un modo para diseñar lenguajes de programación ofreciendo las siguientes ventajas:

- Una gramática es una especificación sintáctica precisa y fácil de entender de un lenguaje de programación.

- A partir de algunas clases de gramáticas se puede construir automáticamente un analizador sintáctico que determine si un programa fuente está bien construido.
- Los cambios a la gramática son más fáciles de realizar cuando se tiene una descripción gramatical del lenguaje.

Definición de gramática libre de contexto

Una gramática libre de contexto es un modelo matemático formado por:

1. Un conjunto de *componentes léxicos* Σ, denominados *símbolos terminales*.
2. Un conjunto de *símbolos no terminales* denominado N.
3. Un conjunto de producciones, en el que cada producción consta de un no terminal, llamado *lado izquierdo* de la producción, una flecha y una secuencia de componentes léxicos y no terminales, o ambos, llamado *lado derecho* de la producción. (P : N → (N x Σ) *)
4. Un símbolo no terminal *S* que se considera el *símbolo de inicio* (o *símbolo inicial*).

Sea entonces G una gramática, tal que G = { Σ, N, P, S } donde:

Σ = { id, núm, (,), -, +, *, / }
N = { expresión, operador }
P = {
expresión → expresión operador expresión,
expresión → (expresión),
expresión → - expresión,
expresión → id,
expresión → núm,
operador → -,
operador → +,
operador → *,
operador → /,
}
S = { expresión }

Comentarios respecto a una gramática

Para evitar tener que establecer siempre cuáles son los elementos de cada conjunto se empleará la siguiente convención:

- Las producciones del símbolo de inicio se listarán primero.
- Los símbolos no terminales son el conjunto de todos los símbolos diferentes que aparecen al lado izquierdo de las producciones.
- El resto de los símbolos forman el conjunto de los terminales.

Por comodidad en la notación se pueden agrupar las producciones de un mismo símbolo no terminal utilizando el operador | que se leerá como “o”. Por ejemplo:

```
expresión → expresión operador expresión
          | ( expresión )
          | - expresión
          | id
          | núm
operador → - | + | * | /
```

Para efectos prácticos es más recomendable dejar cada producción, por más simple que sea, en una línea. Por ejemplo:

```
operador →  -
          | +
          | *
          | /
```

La producción que contiene una cadena vacía hace uso de la representación ε para dicha cadena. Por ejemplo:

```
puntero →  ε
         | *
         | * puntero
```

Nótese la recursividad de las definiciones en las producciones mostradas.

Gramáticas BNF

Una definición como la mostrada en la sección anterior se denomina *gramática BNF* (Backus-Naur Form), por estar escrita con la notación introducida por estos autores. Después que se descubrió la equivalencia entre estas gramáticas y las gramáticas independientes de contexto se ha utilizado hasta nuestros días como estándar de diseño. El símbolo → puede ser reemplazado por la notación ::=. Los símbolos no terminales pueden ser encerrados entre < y > para que puedan ser reconocidos más fácilmente por las personas que leen la definición.

```
<expresión> ::= <expresión> <operador> <expresión>
              | ( <expresión> )
              | - <expresión>
              | id
              | núm
<operador> ::= - | + | * | /
```

Además existe la posibilidad de indicar mediante la notación { } $^{i - j}$ la cantidad mínima (i) y máxima (j) de veces que se puede repetir un símbolo. Por ejemplo:

$$<S> ::= \{a\}^{3-4}$$

Lo cual sería una forma abreviada de escribir lo siguiente:

<S> ::= a a a
| a a a a

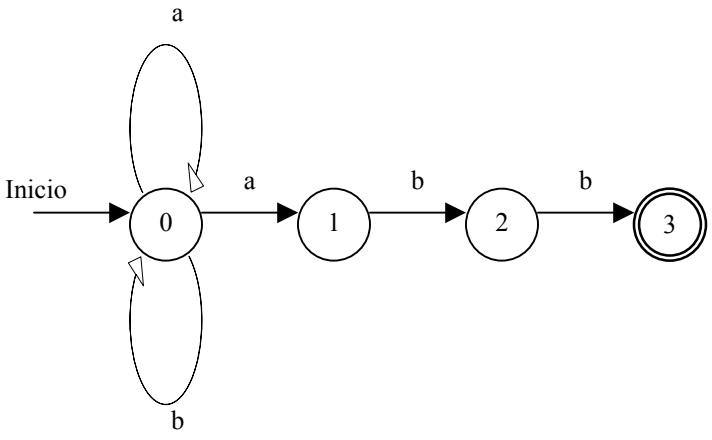
III.- LENGUAJES LIBRES DE CONTEXTO

Según la *clasificación de Noam Chomsky*, los lenguajes regulares serían un subconjunto restringido de los lenguajes libres de contexto. Por tanto toda construcción que se pueda describir mediante una expresión regular también se puede describir por medio de una gramática. Por ejemplo, la expresión regular (a|b)*abb y la gramática

$A_0 \rightarrow a\ A_0 \mid b\ A_0 \mid a\ A_1$
$A_1 \rightarrow b\ A_2$
$A_2 \rightarrow b\ A_3$
$A_3 \rightarrow \varepsilon$

Describen el mismo lenguaje, el conjunto de cadenas de caracteres a y b que terminan en abb.

Se puede convertir de manera mecánica un autómata finito no determinista (AFN) en una gramática que genere el mismo lenguaje reconocido por el AFN. La gramática anterior se construyó a partir de la siguiente figura:



Para cada estado i del AFN se creó un símbolo no terminal (A_0, A_1, A_2 y A_3). Si el estado i tiene una transición hacia el estado j con el símbolo de entrada a, se crea la producción $A_i \rightarrow a\ A_j$. Si el estado i va al estado j con la entrada ε se introduce la producción $A_i \rightarrow A_j$. Si i es un estado de aceptación se introduce la producción $A_i \rightarrow \varepsilon$. Si i es el estado de inicio, entonces A_i es el símbolo inicial de la gramática.

Los lenguajes de programación, en cambio pueden ser descritos por gramáticas libres de contexto, pero no por lenguajes regulares debido a la recursividad de sus estructuras. Pero sin perder de vista el punto central de la asignatura, y sólo a modo de ejemplo, se puede indicar que existen lenguajes que sin ser tan complejos no pueden ser descritos por la notación de las expresiones regulares.

Sea el lenguaje $L = \{a^n b^n \mid n \geq 1\}$ es independiente del contexto con la gramática

$S \rightarrow a\ S\ b \mid a\ b$

Pero no puede ser representado por una expresión regular. Nótese que L acepta entradas del tipo {ab, aabb, aaabbb, aaaabbbb, aaaaabbbbb, etc.}, lo cual en esencia implica que una expresión regular no podría servir para saber si las llaves de los bloques de C están bien equilibradas, puesto que tiene que haber *igual cantidad* de llaves que abren y llaves que cierran.

Derivaciones

Para comprender que es lo que aceptan las gramáticas libres de contexto existen dos posibilidades de análisis: las derivaciones por la izquierda, reemplazando no terminales en cada paso sucesivo, ó los árboles de análisis sintáctico.

Por ejemplo: para saber si la cadena aaabba es aceptada por la gramática

$$S \rightarrow a\ S\ b \mid a\ b$$

Se realizan las siguientes derivaciones a partir del símbolo de partida S:

$S \Rightarrow a\ S\ b \Rightarrow a\ a\ S\ b\ b \Rightarrow a\ a\ a\ b\ b\ b$

El resultado puede expresarse de manera más concisa como:

$$S \overset{+}{\Rightarrow} a\ a\ a\ b\ b\ b$$

Es decir, aaabbb se deriva en uno o más pasos a partir de S. Si la gramática estuviera definida de esta otra forma:

$$S \rightarrow a\ S\ b \mid \varepsilon$$

Se realizan las siguientes derivaciones a partir del símbolo de partida S:

$S \Rightarrow a\ S\ b \Rightarrow a\ a\ S\ b\ b \Rightarrow a\ a\ a\ S\ b\ b\ b \Rightarrow a\ a\ a\ \varepsilon\ b\ b\ b \Rightarrow a\ a\ a\ b\ b\ b$

El resultado puede expresarse de manera más concisa como:

$$S \overset{*}{\Rightarrow} a\ a\ a\ b\ b\ b$$

Es decir, aaabbb se deriva en cero o más pasos a partir de S. Para el ejemplo, la decisión de usar la notación de + o * es irrelevante. Se han usado de un modo didáctico para introducir su uso y lectura.

Por ejemplo: para saber si la cadena (a + 3) * b es aceptada por la gramática

expresión → expresión operador expresión
| (expresión)
| - expresión
| id
| núm
operador → - | + | * | /

Se realizan las siguientes derivaciones a partir del símbolo de partida expresión:

expresión ⇒ expresión operador expresión
⇒ (expresión) operador expresión
⇒ (expresión operador expresión) operador expresión
⇒ (id operador expresión) operador expresión
⇒ (id + expresión) operador expresión
⇒ (id + núm) operador expresión
⇒ (id + núm) * expresión
⇒ (id + núm) * id

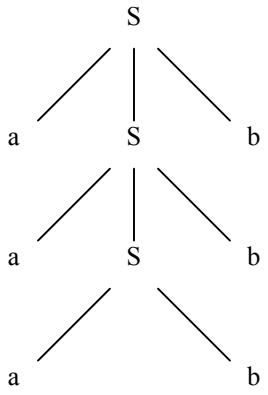
Árboles de análisis sintáctico

La segunda forma de analizar si una cadena es aceptada por una gramática libre de contexto son los *árboles* de análisis sintáctico. Estos árboles se contruyen con el símbolo de partida como *raíz*. Cada no terminal produce un *subárbol*, de tal forma que cuando se hayan *expandido* todos los no terminales cada *hoja* será un símbolo terminal de la gramática y cada *nodo interno* será un símbolo no terminal.

Por ejemplo: para saber si la cadena aaabbb es aceptada por la gramática

$$S \rightarrow a\ S\ b \mid a\ b$$

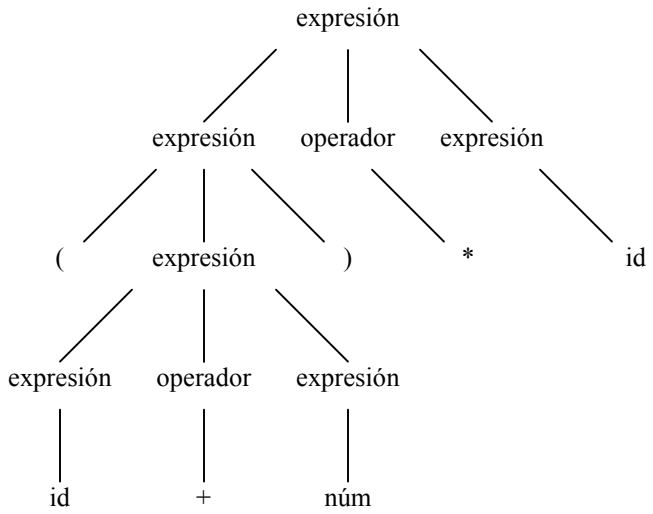
Se realizan las siguientes derivaciones a partir del símbolo de partida S:



En el caso de la cadena (a + 3) * b con la gramática

expresión → expresión operador expresión
| (expresión)
| - expresión
| id
| núm
operador → - | + | * | /

Se realizan las siguientes derivaciones a partir del símbolo de partida expresión:



IV.- EJERCICIOS PROPUESTOS

1.- Dada la siguiente gramática G:

<S> ::= <A> <B>
<A> ::= X | Y
<B> ::= Z | W

Obtenga TODAS las cadenas válidas para este lenguaje.

2.- Dada la siguiente gramática G:

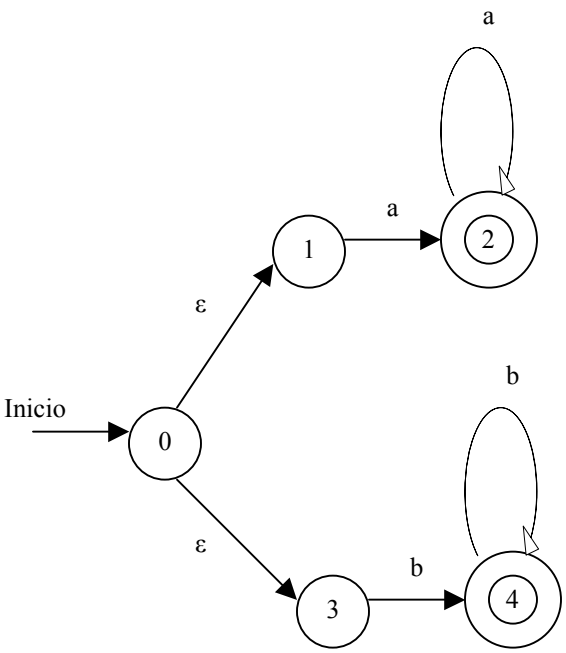
<sentencia> ::= <sujeto> <predicado>
<sujeto> ::= perro | pájaro | elefante
<predicado> ::= vuela | ladra | come

Obtenga TODAS las cadenas válidas para este lenguaje.

3.- Describa el conjunto formado por la siguiente gramática:

<identificador> ::= <letra> {<literal>}$^{0-5}$
<letra> ::= a | b | c
<literal ::= <letra> | <dígito>
<dígito> ::= 0 | 1 | 2 | 3 | 4

4.- Convierta a la notación BNF el siguiente AFN.



5.- Dada la siguiente gramática G:

D → A DD
DD → id
| (D)
| DD ( )
| DD [T]
A → ε
| *
| * A
T→ ε
| núm
| id

Compruebe que las siguientes expresiones se pueden derivar con G.

a) **argv
b) (*daytab)[13]
c) *daytab[13]
d) *comp()
e) (*comp)()
f) (*(*x())[])()
g) (*(*x[3])())[5]

6.- Construya un AFD que reconozca claúsulas y reglas de un programa PROLOG:

cláusula → id = (x {, x } $^{0-n}$).
regla → id = (x {, x } $^{0-n}$) :- id = (x {, x } $^{0-n}$) { , id = (x {, x } $^{0-n}$) } $^{0-m}$.
x → id
| num