UNIVERSIDAD NACIONAL AUTONOMA
DE MEXICO

FACULTAD DE CIENCIAS

Traductor de BNF Modificado a Diagramas de Tren

T E S I S

QUE PARA OBTENER EL TÍTULO DE

A C T U A R I O

P R E S E N T A :

BENJAMÍN MORENO ARANDA

FACULTAD DE CIENCIAS
UNAM

DIRECTORA DE TESIS: DRA. ELISA VISO GUROVICH

2005

VNIVERSIDAD NACIONAL
AVTNOMA DE
MEXICO

ACT. MAURICIO AGUILAR GONZÁLEZ
Jefe de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:

Traductor de BNF Modificado a Diagramas de Tren

realizado por **Benjamín Moreno Aranda**

con número de cuenta **07217689-2** , quien cubrió los créditos de la carrera de: **Actuaría**

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director Propietario	**Dra. Elisa Viso Gurovich**
Propietario	**M. en C. José de Jesús Galaviz Casas**
Propietario	**M. en C. Javier García García**
Suplente	**Lic. en C.C. Karla Ramírez Pulido**
Suplente	**M. en I. María de Luz Gasca Soto**

Consejo Departamental de Matemáticas

Act. Jaime Vázquez Alamilla

FACULTAD DE CIENCIAS
CONSEJO DEPARTAMENTAL
DE
MATEMATICAS

Este libro está dedicado a:

Mis padres a quienes les agradezco el amor y apoyo que me brindaron siempre para que mis sueños se volvieran realidad.

Mi esposa Claudia y mis hijos: Ingrid Ximena, Carlo Van Eyck y Paris Adán.

AGRADECIMIENTOS

A todos mis maestros

Por la luz con la que iluminaron mi camino.

A mis asesores

Por aceptar participar en la revisión de mi trabajo y dedicar parte de su valioso tiempo en atenderme para indicarme sus valiosas observaciones y correcciones.

A mi directora de tesis

Por aceptar dirigir mi tesis. Por compartir tan generosamente conmigo su sabiduría y conocimientos. Por su infinita paciencia al soportar mis abandonos a este trabajo, que aunque involuntarios y justificados desgraciadamente no iban acompañados de una explicación. Le pido una vez más que perdone, siempre le estaré infinitamente agradecido.

A la UNAM

Por brindarme la oportunidad de estudiar esta maravillosa carrera.

Contenido

Introducción

Sistema que se pretende obtener y su utilidad

El objetivo principal de este trabajo es proporcionar una herramienta que permita ***traducir*** producciones de una sintaxis de ***BNF extendido*** a ***diagramas de tren***. Cabe aquí preguntarse ¿para qué queremos una herramienta semejante? Bueno, la mayoría de los manuales de computación definen las reglas sintácticas de su lenguaje en términos similares a las siguientes definiciones:

```
identifier: nondigit
        identifier nondigit
        identifier digit
nondigit : one of
        a b c d e f g h i j k l m n o p q r s t u v w x y z _
        A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
digit   : one of
        0 1 2 3 4 5 6 7 8 9
```

El ejemplo anterior se tomó de la guía de programación del manual de Borland C++ versión 4.5; dicho ejemplo en una notación más precisa como el de BNF extendido, debería estar escrito de la siguiente forma:

```
<identifier>::=<nondigit>|<identifier><nondigit>|<identifier><digit>
<nondigit> ::=a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
             |A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<digit>    ::=0|1|2|3|4|5|6|7|8|9
```

Esta notación si bien cumple con la definición formal, es poco intuitiva. Con el lenguaje de programación ***Pascal*** surgió una manera gráfica de representar BNF, llamada ***diagramas de***

tren. Si traducimos por ejemplo la definición de <identifier> a diagramas de tren, obtenemos lo siguiente:

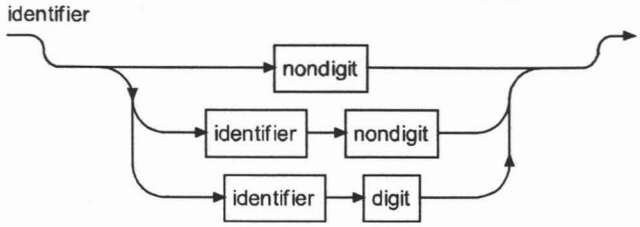


FIGURA *i*.1

Obviamente los diagramas de tren, desde el punto de vista didáctico ofrecen ventajas sobre la notación BNF, ya que expresan más claramente la forma en que se define la sintaxis de las producciones de un lenguaje, como sucede aquí con la definición de un identificador.

Por otra parte se puede intentar una abstracción mayor, ya que si traducimos directamente de BNF en forma natural la definición de lo que es un identificador; necesitaríamos en el ejemplo anterior 3 gráficas; podemos comprimir los tres diagramas en uno solo, si realizamos algunas modificaciones a la notación de BNF, que pretendemos utilizar para graficar los diagramas de tren, por lo tanto, se propone aquí una notación de BNF modificada, que permita por ejemplo, en el caso anterior, diagramar a un identificador como:

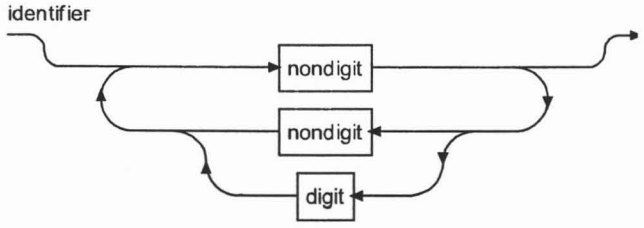


FIGURA *i*.2

Principalmente dicha herramienta podría utilizarse para definir la sintaxis de un lenguaje en forma gráfica sin necesidad de tener que involucrarse en lo tedioso de una elaboración gráfica; es decir, tendríamos una herramienta que en forma sencilla y elegante, nos permita obtener gráficas de producciones escritas en ***BNF extendido***. El trabajo es de hecho un intérprete cuya entrada son expresiones de BNF modificado y el código que genera son las instrucciones que dibujan los gráficos.

Obviamente dicha herramienta debe estar programada para utilizarla en una computadora que nos dé la posibilidad de ver estas gráficas tanto en la pantalla como en forma impresa. Se optó por programar en ***Borland Delphi Copyright*** ® para ejecutar en ***Windows*** ®, dado lo comercial y popular que ha resultado dicho sistema operativo.

Fundamentación teórica general

Capítulo 1

En este capítulo presentaremos los fundamentos teóricos necesarios para la traducción automática de lenguajes en general.

1.1 Qué son los lenguajes de programación

Aquí vamos a hablar de lo que son los lenguajes de programación y la clasificación de los mismos de acuerdo a ciertas características en su diseño. La primera pregunta que surge es, ¿para qué necesitamos un lenguaje de programación? Bueno, por lo general utilizamos un lenguaje de programación porque deseamos comunicarnos con la computadora, para que nos auxilie en la solución de un problema determinado.

Dicho lenguaje nos impone ciertas reglas que debemos seguir si queremos que la computadora haga exactamente lo que deseamos. Es algo razonable no sólo para comunicarnos con la computadora, sino para comunicarnos en general con las personas; en una plática informal, tal vez no sea relevante la manera en que nos comunicamos, ni tampoco si utilizamos las palabras adecuadas.

Sin embargo, aun cuando nos comunicamos con una persona debemos encontrar la manera de transmitirle ya sea con palabras, gestos, muecas o mímica nuestros deseos si es que queremos que la persona haga lo que necesitamos o queremos. Por ejemplo, supongamos que queremos saber cuánto vamos a pagar por la tenencia de nuestro automóvil este año, pero tenemos flojera de buscar el dato nosotros mismos, así que queremos que alguien lo haga por nosotros. ¿Qué tendríamos que hacer? Dependiendo de la persona a la que le vayamos a pedir el favor, podrían darse algunas de las siguientes alternativas contrastantes (no son todas, obviamente) :

- Alguien quien tenga muchas ganas de ayudarme y que sepa de estas cosas, tal vez sólo tenga que decirle: "¿Me ayudas a ver cuánto tengo que pagar por la tenencia de mi automóvil?" La persona se encargaría del resto.
- Alguien me va a ayudar porque no tiene otra opción, a lo mejor me dice:
  - ¿Tienes la tabla, o en dónde me la dan?
  - Dame los datos de tu auto
  - ¿Vas a pagar derechos y tenencia? (nada más por molestar).
  - ¿Tu coche ya no paga o sí? (también nada más por molestar).

Nótese que si me toca alguien que no tenga voluntad de ayudarme o que no tenga capacidad para resolver este tipo de problemas, la lista de preguntas puede crecer considerablemente.

¿Qué pasaría si quisiera que el problema lo resolviera la computadora?

Tendría que decirle paso por paso lo que tendría que hacer; claro únicamente lo necesario, ni más ni menos, y tendría la ventaja de que lo haría para mi auto o para cualquier auto (si es que lo planteo así) y posiblemente (no necesariamente) lo haría más rápido que una persona. Lo bueno es que como carece de emociones no me haría preguntas por molestar (a no ser que yo quiera que lo haga a la hora que le doy las instrucciones, para que moleste a los usuarios del sistema con preguntas innecesarias).

Pero ¿qué entiende la computadora? En esencia sólo sabe interpretar secuencias de ceros y unos, a lo que se le conoce como ***lenguaje de máquina***. Cada instrucción en un programa se representa por un código numérico y direcciones numéricas (también son números) se utilizan para hacer referencia a localidades de memoria.

Obviamente programar de esta manera limitaría el uso de las computadoras, no cualquiera podría hacerlo y además hacerlo así sería más lento de lo que es. Por lo tanto, se hizo necesario facilitar un poco las cosas introduciendo símbolos mnemotécnicos que representaran en forma más natural el puro lenguaje numérico y así nació el ***lenguaje ensamblador***, en donde a cada operación se le asigna un código simbólico, como por ejemplo *ADD* para la suma.

Pero aún cuando el lenguaje ensamblador permite una comunicación menos críptica que la del lenguaje de máquina, no es de ninguna manera fácil de manejar ni representa la mejor opción para comunicarse con la computadora, al menos a nivel masivo.

Es por eso que nacieron los lenguajes de alto nivel como ALGOL, FORTRAN y LISP entre otros. Algunas de las ventajas que presentan son:

- Su aprendizaje es más sencillo, ya que no hay necesidad de escribir código numérico ni simbólico como en el lenguaje ensamblador.
- El conjunto de instrucciones es mucho más amplio ya que se cuenta con instrucciones condicionales, de repetición basada en una condición, instrucciones anidadas y estructuras en bloques.
- Por lo general no son dependientes del hardware de la máquina.
- El programador no necesita saber convertir los datos de su representación externa a su representación interna.
- Es más fácil corregir los errores de programación.
- Permite una descripción modular de las tareas de programación, lo que permite dividir el trabajo de programación.

Inclusive hay lenguajes llamados de cuarta generación, lenguajes orientados a usuarios, lenguajes orientados a problemas en donde el lenguaje utilizado se apega básicamente a las reglas de juego del problema específico en cuestión que se intenta resolver. Es decir, son lenguajes diseñados a propósito del problema que intentan resolver.

Hasta aquí todo va bien, pero si la computadora sólo entiende secuencias de ceros y unos ¿cómo es que hemos estado hablando de lenguaje ensamblador y de lenguajes de alto nivel? ¿Cómo es posible que entienda otros lenguajes?

Todo esto es posible gracias a los traductores, que toman como entrada un programa fuente y lo convierten en un programa objeto; a esto se le conoce como ***tarea de compilación***. Tipos de traductores son los ***compiladores*** (de los lenguajes de alto nivel) y los ***intérpretes***. El ***programa objeto*** se ejecuta en lo que se conoce como tiempo de corrida o ejecución. La siguiente figura ilustra el proceso de compilación. Nótese que el ***programa fuente*** y los datos (la información de entrada que deseamos procesar para obtener un resultado) se procesan en diferentes tiempos.

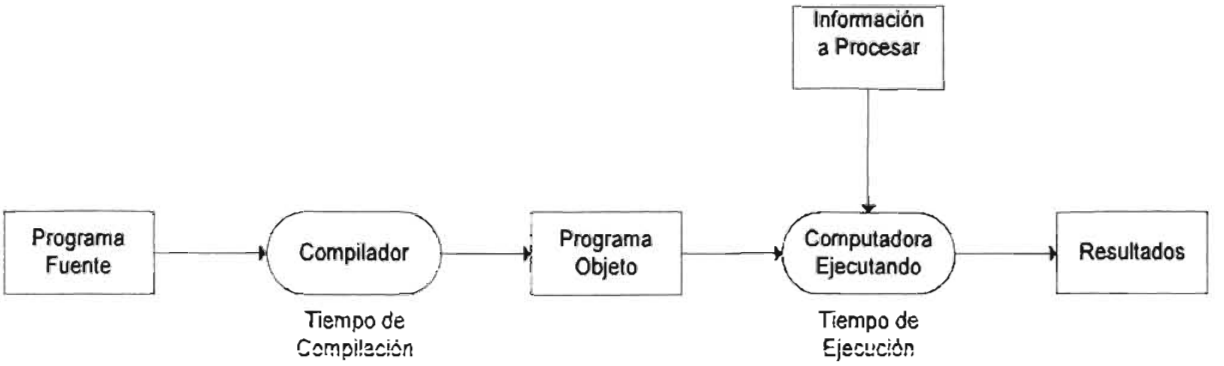


FIGURA 1.1.1

Un ***intérprete*** procesa una forma interna del programa fuente conforme lo va traduciendo, no se genera ningún programa objeto. Podemos visualizar este proceso interpretativo así:

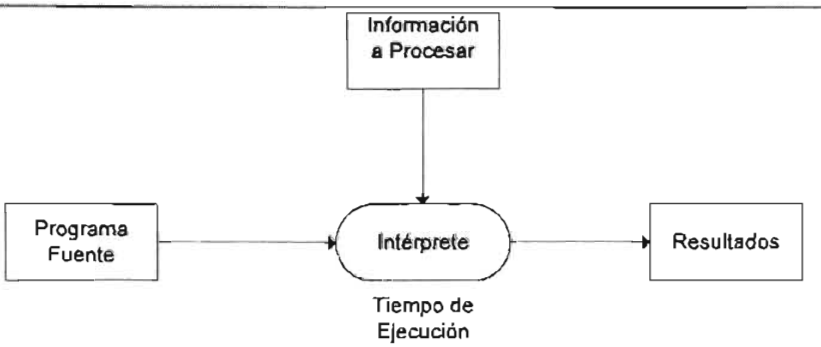


FIGURA 1.1.2

Cabe mencionar que hoy en día, para acelerar la ejecución de programas interpretados, muchas veces se recurre a códigos intermedios. Volviendo a nuestro planteamiento original, para que la computadora nos diga cuánto tenemos que pagar de tenencia, tendríamos que indicarle paso por paso (mediante un lenguaje de programación, preferentemente un lenguaje de alto nivel como C, BASIC y PASCAL) las acciones que tendría que ejecutar para llegar al resultado que deseamos.

1.2 El proceso de compilación.

Se puede definir el proceso de compilación esquemáticamente de la siguiente manera:

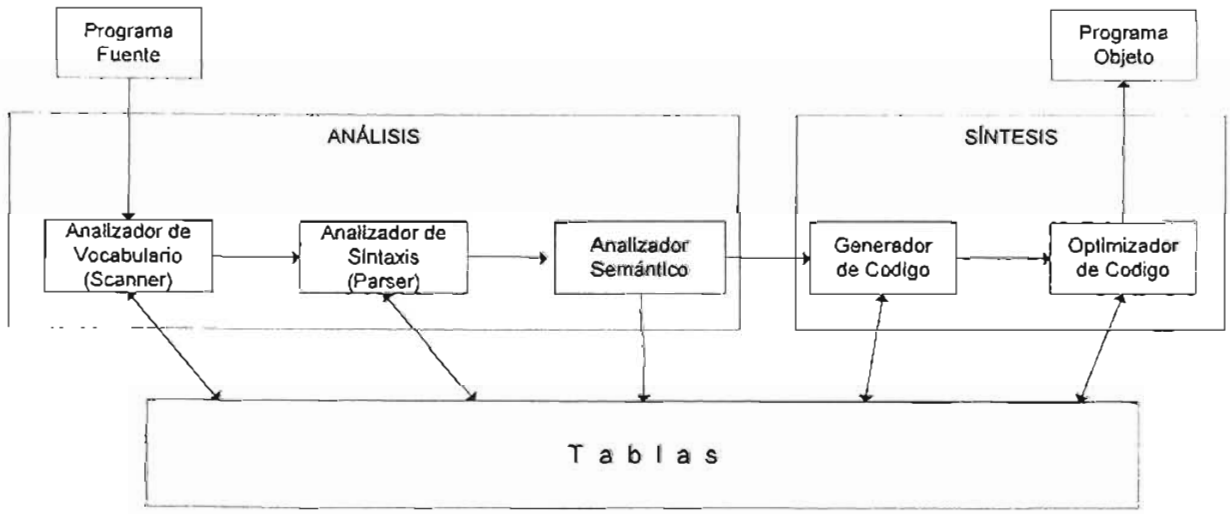


FIGURA 1.2.1

Como podemos observar, hay dos procesos básicos, uno es el análisis del programa fuente y el otro corresponde a la síntesis de el programa objeto correspondiente.

El ***análisis*** se encarga de descomponer el programa fuente en sus partes básicas, las cuales toma la ***síntesis*** para construir los módulos que integran el programa objeto; este proceso de síntesis se facilita con la construcción y mantenimiento de varias tablas que contienen información que se va recolectando a lo largo del proceso.

Recordemos que el ***programa fuente***, que es el medio que utilizamos para comunicarnos con la computadora, es en esencia una cadena de símbolos de un lenguaje; cada uno de estos símbolos está representado generalmente por una letra, un dígito o ciertos carácteres especiales tales como +, -, * y /, los cuales combinados representan ya sea nombres de variables, etiquetas, constantes, palabras reservadas, operadores, estructuras de bloque, etc... Estas construcciones y elementos válidos del lenguaje se describen en la definición del lenguaje al igual que en el español o lenguaje natural; no cualquier combinación de palabras corresponde a una oración; en los lenguajes de programación se especifica, en su

definición, cuáles son las combinaciones válidas, en términos de clases de símbolos (lo que en español corresponde a las clases gramaticales como el verbo, sustantivo, etc.)..

Así pues el programa fuente es el punto de partida del proceso, ya que sirve de entrada a un ***analizador de vocabulario*** o ***scanner***, cuya función principal consiste en separar esta cadena de texto o carácteres en símbolos válidos del lenguaje tal como los que mencionamos con anterioridad; además inicia la tarea de la construcción de las tablas que mencionamos con anterioridad, ya que asigna a cada símbolo un número único de representación interna; esto significa que por ejemplo a un nombre de variable puede asignarle el número 1, a una constante el número 2, etc., además de indicar su posición dentro de la tabla y la línea dentro del programa en la que aparece. Un ejemplo está dado por la siguiente instrucción escrita en C++ (que por cierto, nunca terminaría de ejecutarse, a pesar de estar correctamente construida).

```
while       ( c    !=    2 )
 j   =   5;
```

Tal vez se traduciría en los siguientes símbolos:

Símbolo	Tipo	Posición en la tabla	Línea en el programa
while	18	102	23
c	23	131	23
!=	44	121	23
2	18	567	23
j	23	100	24
=	45	667	24
5	18	454	24
;	12	232	24

En la mayoría de los lenguajes de alto nivel de hoy en día, el compilador reconoce dónde empiezan y terminan los símbolos porque éstos se encuentran separados por caracteres que llamamos ***separadores***, el más típico siendo uno o más blancos. Siendo el blanco un separador, el scanner los ignora. Una cadena de carácteres corresponde a un símbolo que empieza y termina con comillas o apóstrofes. En el caso de los blancos contenidos en estas cadenas, éstos no son ignorados.

En general el scanner ***ignora también los comentarios***, ya que éstos no representan partes ejecutables de un programa. Además también procesa las macros que tenga implementadas el lenguaje, realizando las sustituciones correspondientes y generando los símbolos y su información correspondiente conforme a lo mencionado anteriormente.

Esta información que ha generado el scanner, la toma el ***analizador sintáctico (parser)***, para determinar la manera en que esta secuencia de símbolos genera la estructura general del programa fuente. Por ejemplo, para determinar la estructura de una oración escrita en español, estaríamos interesados en identificar ciertas clases tales como el "***sujeto***", el "***predicado***", el "***verbo***", el "***adjetivo***".

En el ***análisis de la sintaxis***, estamos interesados en agrupar a los símbolos en clases sintácticas tales como expresión, instrucción y procedimiento. El ***analizador sintáctico*** da como resultado final un ***árbol sintáctico*** (o su equivalente) en el cual sus hojas son los símbolos y en el que cada rama representa un tipo de clase sintáctica. Hagamos el ejemplo siguiente de lo que haría un parser a una oración común y corriente del lenguaje español.

Supongamos que damos las siguientes reglas sintácticas para analizar una pequeña parte del idioma español.

```
Oración =  Sujeto + Predicado
Sujeto = Articulo + Sustantivo
Predicado = Verbo + Preposición + PronombrePosesivo> +Sustantivo
Articulo = "El" o "La"
Sustantivo = "niña" o "gato"  o "amo"
Verbo = "ama" o "aborrece"
Preposición = "a"
PronombrePosesivo = "su".
```

El árbol sintáctico de dicha oración sería:

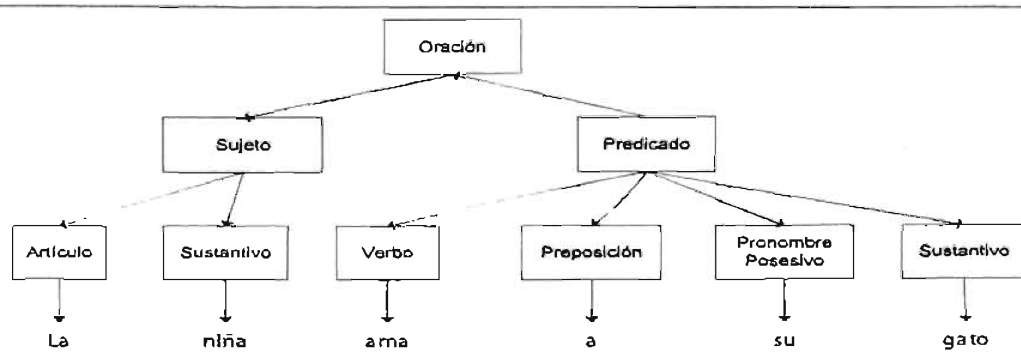


FIGURA 1.2.2

Y otras oraciones válidas serían:

- La niña aborrece a su amo.
- El gato ama a la niña.
- La niña aborrece a su gato.

Volviendo a los lenguajes de computación, el análisis de la instrucción

```
( X * Y ) + ( Z * W)
```

produciría tentativamente las clases sintácticas ***<factor>***, ***<término>*** y ***<expresión>*** de una manera similar al ejemplo que mostramos anteriormente; es decir como un árbol sintáctico. Esta generación del árbol sintáctico la describiremos después cuando definamos un conjunto de reglas conocidas como gramática. En realidad el parser utiliza este ***conjunto de reglas o gramática*** para determinar la estructura del programa fuente. A este proceso se le conoce en computación como ***parsing***, por lo tanto nos referiremos al analizador sintáctico como parser.

Volviendo al proceso que estamos describiendo, una vez que se ha generado este ***árbol sintáctico***, la función del ***analizador semántico*** (ya aclaramos que sólo realiza una parte limitada de este tipo de análisis) es determinar el significado del programa fuente. Por ejemplo al analizar una expresión como

```
( X * Y ) + ( Z * W)
```

debe determinar las acciones que le indican los operadores aritméticos involucrados, para lo cual invoca a una rutina que verifica:

- Primero si los operandos han sido declarados y de que tipo son.
- Si no son del mismo tipo tal vez se encargue de homogenizarlos.
- También que tengan valores.

Su labor puede ir un poco más allá y tal vez inclusive genere una forma intermedia o un adelanto de la generación de código; al realizar toda esta labor el analizador semántico continúa utilizando las tablas y creando o actualizando otras.

Los resultados creados por el analizador semántico sirven para que el ***generador de código*** traduzca a ***lenguaje ensamblador o máquina*** y para que por último el ***optimizador de código*** dé los últimos toques al programa objeto que se genere finalmente.

Éste es a grandes rasgos el proceso de compilación. Pero, ¿qué tiene que ver con el trabajo que estamos desarrollando?

Recuérdese que el objetivo es graficar producciones escritas en BNF modificado. Luego entonces, tenemos un lenguaje ***BNF modificado***, una gramática (la que vamos a definir más adelante) y deseamos obtener un resultado de la computadora. ¿Cómo podríamos empatar esto contra el proceso de compilación? Bueno, nuestro proceso podría quedar de la siguiente manera.

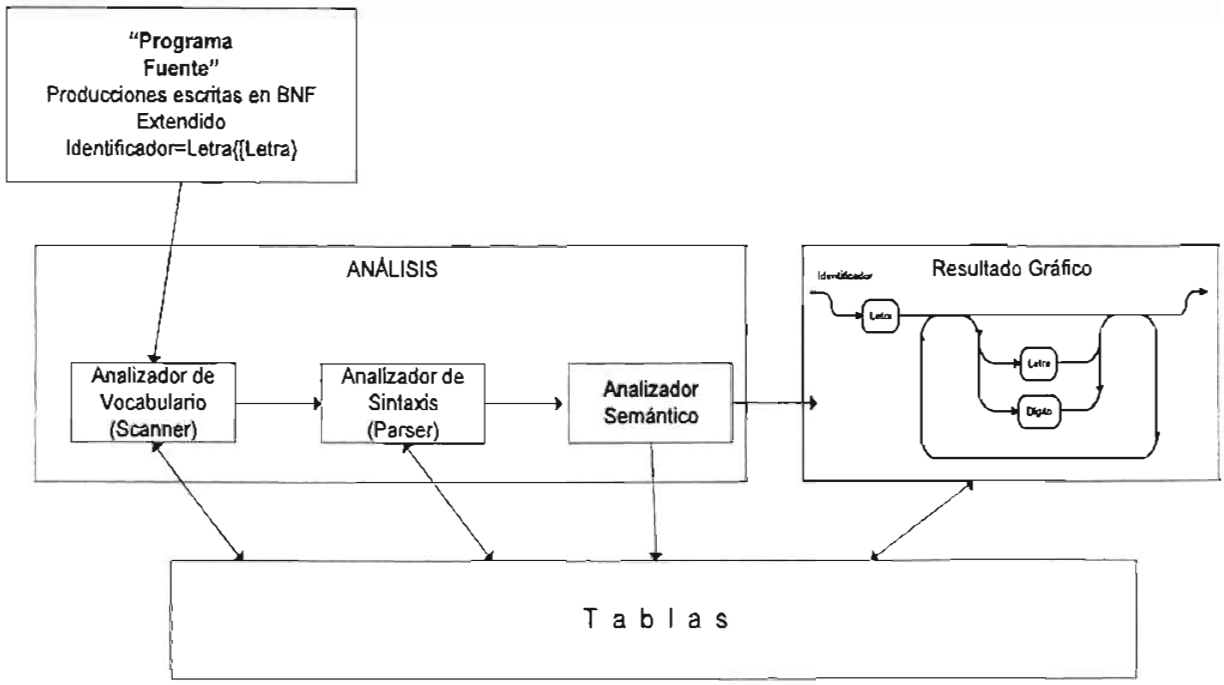


FIGURA 1.2.3

Podemos pensar que en nuestro caso, un programa consiste de un enunciado de BNF y el ***código objeto*** consiste de la ***gráfica de trenes*** correspondiente. La ejecución de los diagramas consistiría en usarlos para generar elementos del lenguaje descrito por ellos. En este sentido (restringido) es que hablamos de que tenemos un ***compilador (o traductor) de BNF a diagramas de trenes***.

1.3 Conceptos básicos de cadenas y relaciones

Empezaremos por definir lo que es un alfabeto y operaciones con símbolos. Un ***alfabeto*** V es un conjunto de símbolos, finito y no vacio. El conjunto $V=\{a,e,i,o,u\}$ es un ejemplo familiar de lo que es un alfabeto (es a su vez un subconjunto del alfabeto español). La ***concatenación*** de dos carácteres alfabéticos por ejemplo 'x' y 'm' forman la sucesión de caracteres 'xm'. La operación de concatenación se aplica también a sucesiones de carácteres. Por ejemplo, 'cd' concatenado con 'yz' nos da '$cdyz$'. Denotamos al operador de concatenación por medio del símbolo especial •. Esto nos permite escribir expresiones tales como 'df'•'hk' lo cual es idéntico a '$dfhk$'.

Una ***cadena*** (u oración) de un alfabeto V es o una letra del alfabeto V o una ***sucesión de letras*** derivada de la concatenación de cero o más carácteres de dicho alfabeto. Dado el alfabeto $V=\{aeiou\}$, ejemplos de cadenas son 'a', 'ae', 'iou'.

Sea $V•V=V^2$ el conjunto de todas las cadenas de longitud 2 sobre V, $V•V•V=V^3$ el conjunto de todas las cadenas de longitud 3 sobre V, generalizando sea $V•V•...•V=V^n$ el conjunto de todas las cadenas de longitud n sobre V. Denotamos a la ***cerradura*** de V como V^+ y la definimos como $V^+ = V \cup V^2 \cup V^3 \cup \ldots$

Decimos que el conjunto V^* es el ***conjunto cerradura*** de V, si incluimos a ε, ***la cadena vacía***, (que es una cadena especial). Esto es, $V^* = \varepsilon \cup V \cup V^2 \cup V^3 \cup \ldots = \varepsilon \cup V^+$.

La cadena vacía es el elemento ***identidad*** en la operación de concatenación, es decir, $x•\varepsilon = \varepsilon•x = x$, para cualquier cadena x que sea un elemento de V^*, y se le conoce como el ***elemento identidad*** en el sistema formado por el conjunto V^* y la operación de concatenación. La ***asociatividad*** es otra de las propiedades que tiene este sistema; esto es, $(x•y)•z = x•(y•z) = x•y•z$ para $x,y,z \in V^*$.

Sean x,y, y z cadenas sobre un alfabeto, donde $z = xy$. A la cadena x se le llama ***prefijo*** o cabeza de z. Si $y \neq \varepsilon$, entonces a x se le llama ***prefijo propio*** o cabeza propia. Similarmente, a y se le conoce como ***sufijo*** o cola de z y si $x \neq \varepsilon$, entonces a y se le llama sufijo propio o cola propia de z.

Antes de continuar vale la pena recordar la definición del ***producto cartesiano*** de dos conjuntos A y B, el cual se define como el conjunto de todos los posibles pares ordenados

(a,b) de tal manera que a es un miembro de A y b es un miembro de B. Esto es,

$$A \times B = \{ (a, b) \mid a \in A \text{ y } b \in B \}$$

Podemos ahora pasar a definir el concepto de relación, que nos proporciona un mecanismo para describir conexiones entre objetos y es un concepto elemental en matemáticas.

Definición 1.3.1 Una ***relación binaria*** de A a B es un subconjunto R de $A \times B$. A es el ***dominio*** de R y B es el codominio o ***rango*** de R. Cuando $A = B$ decimos que R es una relación en A.

Para todo $a \in$ A y b $\in$ B escribiremos $a\ R\ b$ para significar que $(a,b) \in R$. Intuitivamente podemos decir que $a\ R\ b$ significa que a está relacionado con b y que R especifica la manera en que se relacionan.

Dado que una relación es un conjunto de pares ordenados, podemos aplicarle las operaciones usuales de los conjuntos. Por lo tanto si R_1 y R_2 denotan dos relaciones, entonces:

$R_1 \cap R_2$ define una relación tal que $x\ (R_1 \cap R_2)\ y$ si y sólo si $x\ R_1\ y \ \wedge \ x\ R_2\ y$

$R_1 \cup R_2$ define una relación tal que $x\ (R_1 \cup R_2)\ y$ si y sólo si $x\ R_1\ y \ \vee \ x\ R_2\ y$

$R_1 - R_2$ define una relación tal que $x\ (R_1 - R_2)\ y$ si y sólo si $x\ R_1\ y \ \wedge \ x\ \not{R}_2\ y$

donde $x\ \not{R}_2\ y$ denota que x no se relaciona con y en la relación R_2.

En particular estamos interesados en las relaciones donde $A=B$, es decir relaciones sobre un solo conjunto. Definimos las siguientes propiedades: sea $R \subseteq A \times A$ una relación sobre el conjunto A, decimos que:

- R es reflexiva, si para toda $x \in A$, $x\ R\ x$, *es decir*; $(x,x) \in R$.
- R es simétrica, si para toda $x, y \in A$, cuando $x\ R\ y$ entonces $y\ R\ x$.
- R es antisimétrica, si para toda $x, y \in A$, cuando $x\ R\ y$ y $y\ R\ x$, entonces $x = y$.
- R es transitiva, si para toda $x, y, z \in A$, cuando $x\ R\ y$ y $y\ R\ z$, entonces $x\ R\ z$.

También podemos representar a esta relación como una matriz, a la que llamamos ***matriz de la relación***. Si consideramos que r_{ij} son los elementos de dicha matriz, tenemos que

$$r_{ij}=\begin{cases} 1, si \;\; x_i \; R \; y_j \\ 0, si \;\; x_i \;\; R \; y_j \end{cases}$$

es el elemento en el renglón i-ésimo y la columna j-ésima, esta es una matriz de $n \times n$, con n igual al número de elementos de A.

Por último consideramos otra operación sobre las relaciones, la ***composición***, que se define como sigue: Sea R una relación de A a B y S una relación de B a C. A la relación $R \circ S$, se le llama relación compuesta de R y S, donde

$$R \circ S = \{ (x,z) \mid x \in A \;\; y \;\; z \in C \;\; y \text{ existe } y \in B \text{ tal que } (x,y) \in R \;\; y \;\; (y,z) \in S \}$$

Ésta es una operación binaria sobre relaciones y produce una nueva relación a partir de dos relaciones. Esta operación de composición de relaciones es ***asociativa***, es decir

$$(R \circ S) \circ T = R \circ (S \circ T) = R \circ S \circ T$$

En efecto, si $(R \circ S) \circ T$ es no vacía, existen $(x,y) \in R$ y $(y,z) \in S$ y $(z,w) \in T$. Esta suposición significa que $(x,z) \in R \circ S$ y que $(x,w) \in (R \circ S) \circ T$. Por otra parte, también podemos afirmar que $(y,w) \in S \circ T$ y por lo tanto $(x,w) \in R \circ (S \circ T)$, con lo cual queda demostrada la asociatividad.

Supongamos ahora que tenemos una relación R de A a B y que S es una relación de B a C. Sean $A = \{ a_1, a_2, \ldots, a_m \}$, $B = \{ b_1, b_2, \ldots, b_n \}$ y $C = \{ c_1, c_2, \ldots, c_p \}$. De acuerdo a nuestra notación matricial para relaciones tenemos que $r_{ij} = 1$ si $a_i \, R \, b_j$ y $s_{jk} = 1$ si $b_j \, R \, c_k$. Si denotamos por rs_{ik} a los elementos de la matriz de la relación $(R \circ S)$, entonces podemos decir que $rs_{ik} = 1$, si

$a_i \, R \, b_1$ y $b_1 \, R \, c_k$, es decir si $a_i \, R \, c_k$

o $a_i \, R \, b_2$ y $b_2 \, R \, c_k$, es decir si $a_i \, R \, c_k$

. . .

o $a_i \, R \, b_n$ y $b_n \, R \, c_k$, es decir si $a_i \, R \, c_k$

Ateniéndonos a nuestra notación matricial, tenemos

$r_{i1}=1$ y $s_{1k}=1$

o $r_{i2}=1$ y $s_{2k}=1$

. . .

o $r_{in}=1$ y $s_{nk}=1$

Por lo tanto podemos decir que

$$rs_{ik} = \bigvee_{j=1} r_{ij} \wedge s_{jk}\ ,\quad i=1,2,...,m;\ k=1,2,...,p$$

Llamamos ***multiplicación booleana de matrices*** a esta manera de calcular los elementos de la composición de relaciones, ya que si sustituimos el símbolo $\bigvee$ por el de $\sum$ y el de $\wedge$ por el de x, obtenemos nuestra conocida multiplicación de matrices.

Definimos ahora

$$R \circ R = R^2,\ R \circ R \circ R = R \circ R^2 = R^3,\ \ldots,\ R \circ R^{n-1} = R^n$$

Definición 1.3.2 Sea A un conjunto finito y R una relación sobre A. Se le llama ***cerradura transitiva*** de R en A, a la relación $R^+ = R \cup R^2 \cup R^3 \cup \ldots$.

Teorema 1.3.3 La cerradura transitiva R^+,de una relación R en un conjunto finito A, es transitiva. Para cualquier otra relación transitiva P en A, tal que $R \subseteq P$, tenemos que $R^+ \subseteq P$. Por lo tanto R^+, es la relación transitiva más pequeña que contiene a R.

En la fase de reconocimiento sintáctico se pueden utilizar los conceptos de cerradura transitiva de relaciones aplicados a la gramática; por ejemplo para construir los conjuntos Primeros Símbolos(FIRST) y Símbolos Siguientes(FOLLOW).

1.4 Definición de lenguaje y gramática

La especificación de un lenguaje de programación involucra la definición precisa de los siguientes elementos:

- El conjunto de símbolos (alfabeto) que se utilizan para construir programas correctos, desde el punto de vista sintáctico.
- Las reglas para combinar símbolos y obtener enunciados correctos.
- Las reglas que dan el significado de los enunciados definidos en el inciso anterior.

Se puede considerar a un ***lenguaje*** como un subconjunto de la cerradura de un alfabeto (incluyendo la cadena vacía). Decimos que un lenguaje L es un conjunto de cadenas u oraciones basadas en un alfabeto finito V_T, de tal manera que $L \subseteq V_T^*$.

Existen dos tipos de lenguajes, los lenguajes que constan de un número finito de oraciones y los que constan de un número infinito de oraciones; los lenguajes de programación y en particular el lenguaje ***BNF*** modificado que se propone en este estudio, caen en esta segunda categoría (y prácticamente todos los que son interesantes). El problema fundamental de este tipo de lenguajes, es que ***dada una cadena u oración se pueda decidir en un número finito de pasos si dicha cadena pertenece o no al lenguaje en cuestión***.

Los lenguajes finitos se describen por medio de la numeración de cada una de sus oraciones; sin embargo para los lenguajes infinitos, tal numeración no es posible y es necesario buscar un método de especificación que sea finito. Una ***gramática*** consta de un conjunto finito de reglas de producción de oraciones válidas, que definen la sintaxis de un lenguaje; constituye un método de especificación, conocido como generador, pues siguiendo las reglas de producción generamos oraciones válidas del lenguaje.

Gramáticas diferentes pueden generar el mismo lenguaje, pero generar diferentes estructuras para las oraciones del lenguaje. El estudio de gramáticas constituye un área importante de las ciencias de la computación llamada ***teoría de lenguajes formales***. Estamos interesados en la gramática como un sistema matemático para definir un lenguaje y como un dispositivo para dotar de una estructura útil a las oraciones del lenguaje.

La estructura de una oración en un lenguaje como el español, se define en términos

de sujeto, predicado, frase, etc.; en un lenguaje de programación, tal estructura se define en términos de procedimientos, instrucciones, expresiones, etc.. En cualquiera de los dos casos es deseable describir tales estructuras y poder decidir si una oración pertenece o no al lenguaje. La estructura gramatical de una oración se estudia generalmente analizando las diferentes partes de una oración y sus interrelaciones.

Por ejemplo, supongamos que tenemos la oración "El niño regaló su libro".

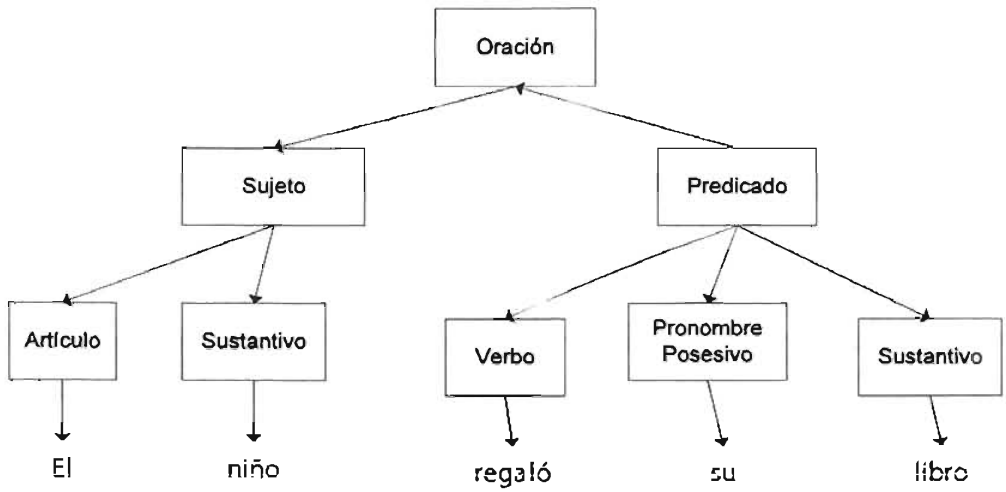


FIGURA 1.4.1.

El diagrama muestra la estructura sintáctica de la oración en forma arborescente, y por lo tanto se le conoce como ***árbol sintáctico***. Cada nodo del árbol representa un símbolo de la gramática. Las palabras "El", "niño", etc., son los símbolos básicos del lenguaje.

Si intentamos establecer las reglas de sintaxis de este pequeño subconjunto del lenguaje español, tendríamos:

- Una **Oración** está compuesta de un **Sujeto** seguido de un **Predicado**
- Un **Sujeto** está compuesto de un **Artículo** seguido de un **Sustantivo**
- Un **Predicado** está compuesto de un **Verbo** seguido de un **Pronombre Posesivo** seguido de un **Sustantivo**.
- Un **Artículo** puede ser **El** o **La**.
- Un **Sustantivo** puede ser **niño** o **monja** o **libro**.

- **Verbo** puede ser **regaló** o **compró**.
- **Pronombre posesivo** puede ser **su** o **mi**.

La estructura de este pequeño lenguaje se define utilizando símbolos tales como **Oración**, **Sujeto**, **Predicado**, etc., que representan ***clases sintácticas*** o ***reglas de reemplazo***. Cada clase sintáctica está compuesta de un número de estructuras alternativas y cada estructura consiste de un conjunto ordenado de símbolos básicos del lenguaje o de otras clases sintácticas. A estas estructuras alternativas se les conoce como ***reglas de sintaxis o reglas de producción o reemplazo***. La clase sintáctica, las palabras `está compuesto de un`, `seguido de`, `puede ser`, etc., nos permiten describir a este pequeño subconjunto del lenguaje español. Al lenguaje que describe a otro se le conoce como ***Metalenguaje***. Por ejemplo, una partitura musical sería el metalenguaje utilizado para describir el lenguaje musical. El diagrama o árbol sintáctico anterior, describe la estructura sintáctica de una oración, pero no su significado.

Por medio de estas reglas gramaticales podemos producir o derivar una oración del lenguaje. Un programador de computación tiene la obligación de producir programas que se adhieran a las producciones (reglas gramaticales) de el lenguaje. El compilador de un lenguaje tiene la obligación de determinar si una oración es sintácticamente correcta basada en las reglas gramaticales establecidas. Si la sintaxis es correcta, el compilador produce código.

Considérese el problema de tratar de generar o producir la oración que analizamos anteriormente, "El niño regaló su libro", a partir del conjunto de producciones dado. Esto se logra empezando con la clase sintáctica **Oración** que nos dice:

Una **Oración** está compuesta de un **Sujeto** seguido de un **Predicado**.

Si empezamos a reemplazar los símbolo de la izquierda por los símbolos que nos indican la reglas de esta sintaxis tendremos, que una Oración está compuesta de:

1) Un **Sujeto** seguido de un **Predicado**.

2) Un **Sujeto**, seguido de un **Verbo**, seguido de un **Pronombre Posesivo**, seguido de un **Sustantivo**.

3) Un **Sujeto** seguido de un **Verbo**, seguido de un **Pronombre Posesivo**, seguido de *libro*.

4) Un **Sujeto** seguido de un **Verbo**, seguido de *su*, seguido de *libro*.

5) Un **Sujeto** seguido de *regaló*, seguido de *su*, seguido de *libro*.

6) Un **Artículo**, seguido de un **Sustantivo**, seguido de *regaló*, seguido de *su*, seguido de *libro*.

7) Un **Artículo**, seguido de *niño*, seguido de *regaló*, seguido de *su*, seguido de *libro*.

8) *El*, seguido de *niño*, seguido de *regaló*, seguido de *su*, seguido de *libro*.

En este ejercicio, hemos reemplazado siempre la clase sintáctica que se encontraba en la extrema derecha, pudimos haber tomado la de extrema izquierda y hubiéramos llegado al mismo resultado; este orden no escrito y no definido en las reglas, no es por el momento importante para la discusión, más adelante cuando avancemos en el proceso de definición de reconocimiento veremos que tiene una importancia especial. También hemos simplificado el proceso generando el símbolo básico adecuado en los casos en los que podíamos escoger entre uno o varios símbolos básicos, como en el caso de la clase sintáctica **verbo**.

Ejemplos de oraciones válidas para este pequeño lenguaje del idioma español son:

- La monja regaló su niño.
- El niño compró su monja.
- Mi libro regaló su monja.

Todas esta oraciones aunque válidas no tienen mucho sentido. Sin embargo, es más fácil especificar las reglas de reemplazo que definen a un lenguaje si pasamos por alto este detalle; y dejamos la tarea de evaluar el significado de las oraciones de un lenguaje al analizador semántico.

El conjunto de oraciones que pueden generarse por las reglas de el ejemplo anterior, es finito. Cualquier lenguaje interesante, consiste generalmente de un conjunto infinito de oraciones. La importancia de un dispositivo finito como la gramática, es que permite estudiar la estructura de un lenguaje que consiste de un conjunto infinito de oraciones.

Denotemos ahora a los símbolos **Letra**, **Dígito** e **Identificador**. Las producciones que siguen son ***recursivas*** y producen un conjunto infinito de nombres dado que la clase sintáctica **Identificador** se encuentra presente tanto en la izquierda como en la derecha de

ciertas producciones.

- Un **Identificador** puede ser una **Letra**
  o un **Identificador** seguido de un **Dígito**
  o un **Identificador** seguido de una **Letra**.
- Una **Letra** puede ser a o b o c o . . . o z.
- Un **Dígito** puede ser 0 o 1 o 2 o . . . o 9.

Es fácil ver que la clase sintáctica **Identificador** define un conjunto infinito de cadenas o nombres en las que cada nombre consiste de una letra seguida de cualquier número de letras o dígitos. Este conjunto es una consecuencia de utilizar la ***recursividad*** en la definición de las producciones En efecto, la recursividad es fundamental para definir un lenguaje infinito por medio del uso de una gramática.

Pasando a la formalización de esta ideas, digamos que V_{ST} es un conjunto no vacío de símbolos *(símbolos terminales)* llamado el ***alfabeto de símbolos terminales***. El ***metalenguaje*** que se utiliza para generar cadenas en el lenguaje contiene un conjunto de ***clases sintácticas*** o variables llamadas símbolos no terminales. Se identifica al conjunto de clases sintácticas por V_{CS}, y se le conoce como ***alfabeto de clases sintácticas*** , sus elementos definen la ***sintaxis*** (estructura) del lenguaje.

Los conjuntos V_C y V_T son disjuntos, es decir $V_C \cap V_T = \emptyset$. Al conjunto $V_G=V_C \cup V_T$ que contiene a las clases sintácticas y los simbolos terminales se le llama ***alfabeto de la gramática***. En adelante utilizaremos la siguiente notación: letras mayúsculas **(A,B,C,...)** para las clases sintácticas; las primeras letras minúsculas $(a,b,c,...)$ para los símbolos terminales , las últimas letras minúsculas **(x,y,z,...)** para sucesiones de símbolos terminales; y letras griegas **(α, β, γ,...)** para cualquier sucesión de símbolos que se encuentra en V_G^*. La longitud de una cadena la denotaremos por $|\alpha|$.

Definición 1.4.1 Una ***gramática*** está definida por la cuarteta $G=(V_C , V_T , S, P)$ donde:

S es el ***símbolo inicial*** y $S \in V_C$, también se le conoce como el ***símbolo distinguido***.

P es un conjunto no vacío y finito de ***reglas de producción***.

Definición 1.4.2 Una ***regla de producción*** es un par ordenado (α, β) tal que:

$$(\alpha, \beta) \in (V_G^* - V_C - V_G^*) \times V_G^*$$

Es decir, $\alpha = \varphi_1 A \varphi_2$ donde $\varphi_1, \varphi_2 \in (V_C \cup V_T)^*$, $A \in V_C$ y $\beta \in (V_C \cup V_T)^*$, por lo regular se escribe como $\alpha \longrightarrow \beta$.

Para nuestro ejemplo del identificador podemos escribir su gramática como:

$G=(V_C, V_T, S, P)$

$V_C = \{\mathbf{I}, \mathbf{L}, \mathbf{D}\}$

$V_T = \{0, 1, \ldots, 9, a, b, \ldots, z\}$

$S=\mathbf{I}$

$P=\{$ $\mathbf{I} \longrightarrow \mathbf{L}$, $\mathbf{I} \longrightarrow \mathbf{IL}$, $\mathbf{I} \longrightarrow \mathbf{ID}$, $\mathbf{L} \longrightarrow a$, $\mathbf{L} \longrightarrow b$, ..., $\mathbf{L} \longrightarrow z$, $\mathbf{D} \longrightarrow 0$, $\mathbf{D} \longrightarrow 1$, ..., $\mathbf{D} \longrightarrow 9\}$

En este punto reemplazamos las definiciones textuales por definiciones en donde utilizamos los símbolos "$\longrightarrow$", y ",". Donde el símbolo "$\longrightarrow$" reemplaza a "***Un ..... puede ser***" o "***Un ..... está compuesto por***". Y "**,**" reemplaza a la palabra **o** que posibilita las alternativas que puede tener una producción. Veamos algunas definiciones más que vamos a necesitar.

Definición 1.4.3 Sea $G=(V_C, V_T, S, P)$ una gramática con α y $\beta \in V_G^*$. Decimos que β es una ***derivación directa*** de α, o que α ***deriva directamente en*** β, lo cual escribimos como $\alpha \Longrightarrow \beta$, si y sólo si existe una producción $\delta \longrightarrow \rho \in P$ tal que $\alpha = \varphi_1 \delta \varphi_2$, $\beta = \varphi_1 \rho \varphi_2$, con $\varphi_1, \varphi_2 \in V_G^*$.

Para nuestra gramática del identificador, tenemos como ejemplos de derivadas directas, las siguientes:

α	β	$\delta \longrightarrow \rho$	φ_1	φ_2
I	L	I → L	ε	ε
L L	Lx	L → x	L	ε
L D L	L1L	D→ 1	L	L
L D D L	L2DL	D→ 2	L	DL

La definición anterior nos indica que $\Longrightarrow$ es una relación binaria en el conjunto de cadenas de la gramática, es decir $\Longrightarrow \subseteq V_G^* \times V_G^*$. Al decir que $\alpha \Longrightarrow \beta$ en lugar de $(\alpha, \beta) \in \Longrightarrow$, estamos utilizando una notación infija.

Los conceptos pueden extenderse para producir una cadena β no directamente a partir de α , sino considerando más pasos.

Definición 1.4.4 Decimos que β es una ***derivación*** de α, o que α ***deriva en*** β, lo cual escribimos como $\alpha \overset{*}{\Longrightarrow} \beta$, si y sólo se verifica una de las dos condiciones siguientes:

- $\alpha = \beta$, (se trata de la misma cadena), o
- Existen cadenas $\varphi_0, \varphi_1, \ldots, \varphi_n \in V_G^*$, tal que $\varphi_0=\alpha$, $\varphi_n=\beta$ y para toda $0 \leq i \leq n$, se cumple que $\varphi_i \Longrightarrow \varphi_{i+1}$. Llamamos a la sucesión $\varphi_0 \Longrightarrow \varphi_1 \Longrightarrow \ldots \Longrightarrow \varphi_n$, ***derivación de longitud n***.

Está claro que $\overset{*}{\Longrightarrow}$ es una relación binaria en V_G^* y que es además la cerradura reflexiva y transitiva de la relación de derivación directa, lo cual significa que $\overset{*}{\Longrightarrow}$ es la menor relación que cumple con:

- Si $\alpha \Longrightarrow \beta$, entonces $\alpha \overset{*}{\Longrightarrow} \beta$.
- $\overset{*}{\Longrightarrow}$ es reflexiva, ya que para toda $\alpha \in V_G^*$, se cumple que $\alpha \overset{*}{\Longrightarrow} \alpha$.
- $\overset{*}{\Longrightarrow}$ es transitiva, ya que si $\alpha \overset{*}{\Longrightarrow} \beta$ y $\beta \overset{*}{\Longrightarrow} \delta$, entonces $\alpha \overset{*}{\Longrightarrow} \delta$.

Regresando a nuestra gramática del identificador, se puede demostrar que la cadena **a13** se deriva de I, siguiendo la sucesión de derivación:

I $\Longrightarrow$ ID $\Longrightarrow$ IDD $\Longrightarrow$ LDD $\Longrightarrow$ aDD $\Longrightarrow$ a1D $\Longrightarrow$ a 1 3

Nótese que en tanto se tenga un símbolo no terminal en la cadena, podemos producir una nueva cadena a partir de éste (suponiendo que no tenemos reglas de la forma A$\longrightarrow$A). Por otro lado, si una cadena contiene solo símbolos terminales, entonces podemos considerar que la derivación está completa y que ya no podemos producir ninguna otra cadena.

Definición 1.4.5 Sea $G=(V_C , V_T , S, P)$ una gramática. Una palabra $\alpha \in (V_C \cup V_T)^*$ se denomina ***forma sentencial*** de la gramática, si y sólo si se cumple que $S \overset{*}{\Longrightarrow} \alpha$. Una forma sentencial φ tal que $\varphi \in V_T^*$ se dice que es una ***sentencia o frase***.

Al lenguaje $L(G)$ formado por todas las cadenas de símbolos terminales que son derivables del símbolo inicial de la gramática (sentencias), se le llama ***lenguaje generado por la gramática G***.

$$L(G)=\left\{ \varphi \in V_T^* \mid S \overset{*}{\Rightarrow} \varphi \right\}$$

Por lo tanto, el lenguaje es un subconjunto de el conjunto de todas las cadenas terminales (oraciones) sobre V_T. No siempre es fácil probar que una gramática genera un determinado lenguaje.

Definición 1.4.6 Decimos que dos ***gramáticas*** G y G' son ***equivalentes*** si y sólo si generan el mismo lenguaje, es decir, si y sólo si $L(G)=L(G')$.

1.5 Clasificación de lenguajes y gramáticas

Sea $G=(V_C , V_T , S, P)$ una gramática, clasificamos las gramáticas y los lenguajes generados por ella, de la siguiente manera:

- Gramáticas Tipo 0 (***Gramáticas generales o sin restricciones***)

  Las más generales, son gramáticas sin restricciones. Lo cual significa que sus producciones son de la forma $\alpha \longrightarrow \beta$, donde $\alpha \in (V_G^* \cdot V_C \cdot V_G^*)$ y $\beta \in V_G^*$.

  Los lenguajes generados por estas gramáticas son los ***lenguajes con estructura de frase***.

- Gramáticas Tipo 1 (***Gramáticas dependientes del contexto***)

  Con producciones de la forma

  $$\alpha \longrightarrow \beta$$

  donde

  $|\alpha| \leq |\beta|$ y $|\alpha|$ es la longitud de α.

  Esta restricción sobre una producción previene que β sea vacía. La forma de producción de una gramática dependiente del contexto puede establecerse de otra manera. Esta segunda forma implica que α y β en la producción $\alpha \longrightarrow \beta$ puedan expresarse como $\alpha = \varphi_1 A \varphi_2$ y $\beta = \varphi_1 \delta \varphi_2$ (donde φ_1 y φ_2 son posiblemente

vacías y en donde δ no debe ser vacía; con lo cual el significado dependiente del contexto adquiere mayor sentido, ya que la aplicación de la producción $\alpha = \varphi_1 A \varphi_2$ y $\beta = \varphi_1 \delta \varphi_2$ a una forma sentencial significa que A se rescribe como δ cuando se encuentra entre (contexto) φ_1 y φ_2.

Los lenguajes generados por estas gramáticas se llaman ***lenguajes dependientes del contexto***.

Un ejemplo de este lenguaje es:

$$L(G)=\left\{ a^n b^n c^n \mid n \geq 1 \right\}$$

que está generado por

$$G=\left(\{\mathbf{S}, \mathbf{B}, \mathbf{C}\}, \{a, b, c\}, S, P\right)$$

donde $P=($ $\mathbf{S} \longrightarrow a\mathbf{SBC}$, $b\mathbf{C} \longrightarrow bc$, $\mathbf{S} \longrightarrow ab\mathbf{C}$, $\mathbf{CB} \longrightarrow \mathbf{BC}$, $b\mathbf{B} \longrightarrow bb$, $c\mathbf{C} \longrightarrow cc$ $)$

La siguiente es la derivación para la cadena $a^2b^2c^2$

S	$\Rightarrow$	$a\mathbf{SBC}$	por la regla de producción 1.
	$\Rightarrow$	$aab\mathbf{CBC}$	por la regla de producción 3.
	$\Rightarrow$	$aab\mathbf{BCC}$	por la regla de producción 4.
	$\Rightarrow$	$aabb\mathbf{CC}$	por la regla de producción 5.
	$\Rightarrow$	$aabbc\mathbf{C}$	por la regla de producción 2.
	$\Rightarrow$	$aabbcc$	por la regla de producción 6.

- Gramáticas Tipo 2 (***Gramáticas libres del contexto***)

Sus producciones son de la forma

$$A \longrightarrow \beta$$

donde

$A \in V_C$ y $\beta \in (V_C \cup V_T)^*$, y $S \longrightarrow \varepsilon$, pero S no aparece del lado derecho.

Genera ***lenguajes libres del contexto***. Cada clase sintáctica se escribe directamente según la regla que lo define sin importar el contexto que lo rodea. Todas las producciones tienen en el lado izquierdo un solo símbolo o clase sintáctica.

Este tipo de gramáticas son capaces de especificar la mayor parte de la sintaxis de los lenguajes de programación, dado que esto lenguajes son simples en su estructura.

Un ejemplo de gramática libre de contexto es la que genera el lenguaje:

$$L(G)=\left\{ a^n b a^n \mid n \geq 1 \right\}$$

donde

$$G=\left(\{\mathbf{S}, \mathbf{C}\}, \{a, b\}, S, P\right)$$

y $P=($ $\mathbf{S} \longrightarrow a\mathbf{C}a$, $\mathbf{C} \longrightarrow a\mathbf{C}a$, $\mathbf{C} \longrightarrow \mathbf{b}$ $)$

a^3ba^3 se deriva de la siguiente forma:

S	$\Rightarrow$	$a\mathbf{C}a$	por la regla de producción 1.
	$\Rightarrow$	$aa\mathbf{C}aa$	por la regla de producción 2.
	$\Rightarrow$	$aaa\mathbf{C}aaa$	por la regla de producción 2.
	$\Rightarrow$	$aaabaaa$	por la regla de producción 3.

- Gramáticas Tipo 3 (***Gramáticas regulares***)
  - ***Lineales por la derecha***

    Donde todas las producciones son del tipo:

    $A \longrightarrow bC$

    $A \longrightarrow b$

    $A \longrightarrow \varepsilon$

  - ***Lineales por la izquierda***

    Donde todas las producciones son del tipo:

    $A \longrightarrow Cb$

    $A \longrightarrow b$

    $A \longrightarrow \varepsilon$

donde $A, C \in V_C$ y $b \in V_T$.

Los lenguajes generados por estas gramáticas se llaman ***lenguajes regulares***.

Un ejemplo de estos lenguajes es:

$$L(G)=\left\{ a^n b a^m \mid n, m \geq 1 \right\}$$

que está generado por

$$G=\left(\{\mathbf{S}, \mathbf{A}, \mathbf{B}, \mathbf{C}\}, \{a, b\}, S, P\right)$$

donde $P=($ $\mathbf{S}\longrightarrow a\mathbf{S}, \mathbf{S}\longrightarrow a\mathbf{B}, \mathbf{B}\longrightarrow b\mathbf{C}, \mathbf{C}\longrightarrow a\mathbf{C}, \mathbf{C}\longrightarrow a$ $)$

La siguiente es la derivación para la cadena a^3ba^2

S	$\Rightarrow$	$a\mathbf{S}$	por la regla de producción 1.
	$\Rightarrow$	$aa\mathbf{S}$	por la regla de producción 1.
	$\Rightarrow$	$aaa\mathbf{B}$	por la regla de producción 2.
	$\Rightarrow$	$aaab\mathbf{C}$	por la regla de producción 3.
	$\Rightarrow$	$aaabaC$	por la regla de producción 4.
	$\Rightarrow$	$aaabaa$	por la regla de producción 5.

Es importante notar que una gramáticas tipo *i* es también una gramática tipo *i-1*, dado que las gramáticas tipo *i* se obtienen a partir de las gramáticas tipos *i-1*, agregando restricciones. Sin embargo, *a las gramáticas se les nombra de acuerdo al tipo mayor en el cual encajan.* Lo mismo sucede con los lenguajes que generan las gramáticas; supongamos que tenemos un lenguaje que puede ser generado por una gramática tipo *0*, una tipo *1* y una tipo *2* pero ya no por una tipo *3*; diremos que tal lenguaje es del tipo *2*. Por lo tanto, decimos que los lenguajes se nombran y clasifican de acuerdo a la gramática más restringida que los genera. Algunas consecuencias de esto son:

- Un lenguaje tipo *n*, puede ser generado por una gramática tipo *n*, pero no por una más débil del tipo *n+1*.
- El tener una gramática tipo *n* que genera a un determinado lenguaje no quiere decir que no exista una gramática tipo *n+1* que genere al mismo lenguaje.

1.6 Lenguajes y gramáticas libres del contexto

De todas las gramáticas que hemos visto, las gramáticas libres del contexto, son las más importantes, debido a su aplicabilidad la construcción de compiladores. Estas gramáticas se pueden usar para expresar la mayoría de las estructuras sintácticas de un lenguaje de programación. Sin embargo, estas gramáticas no pueden expresar condiciones dependientes del contexto como son por ejemplo:

- Un nombre de variable debe declararse antes de poder ser utilizado
- Los operandos de una expresión deben tener tipos compatibles
- Que la llamada a un procedimiento contenga exactamente tantos argumentos como parámetros en la definición que se hizo del procedimiento y que además sus tipos coincidan.

Una posible solución a este problema es utilizar gramáticas sensibles al contexto; desgraciadamente no existen algoritmos de reconocimiento eficaces y sencillos para estas gramáticas. Por lo tanto, la definición de la sintaxis de los lenguajes de programación por lo general se divide en dos partes. La mayor parte se especifica en forma de una gramática libre del contexto, a la cual puede aplicársele un algoritmo de reconocimiento eficiente. Así, la gramática describe la estructura general de reconocimiento que permite aceptar oraciones que las condiciones del contexto pueden rechazar. El resto de la especificación consiste en especificar estas restricciones que se imponen a la gramática. Por ejemplo:

```
int x;
x = "three";
```

sintácticamente es correcto pero semánticamente es incorrecto.

Vamos a resumir algunas definiciones fundamentales, relacionadas con las gramáticas libres de contexto, que utilizamos en el estudio de los métodos de análisis sintáctico, hasta llegar a la definición de árbol de derivación:

Definición 1.6.1 Sea una gramática $G=(V_C , V_T , S, P)$, y $\alpha\beta\varphi$ una forma sentencial, donde $\alpha \in V_T^*$, $\beta \in V_C$ y $\varphi \in (V_C \cup V_T)^*$. Una ***derivación izquierda*** se obtiene sustituyendo β por alguna de las partes derechas que la definen.

Definición 1.6.2 Sea una forma sentencial $\alpha\beta\varphi$ en donde $\alpha \in (V_C \cup V_T)^*$, $\beta \in V_C$ y $\varphi \in V_T^*$. Una ***derivación derecha*** se obtiene sustituyendo β por alguna de las partes derechas que la definen.

Una derivación por la izquierda para una forma sentencial, es una derivación que, comenzando con el símbolo inicial, acaba en esa forma sentencial, y en cada derivación directa, siempre se aplica una regla de producción correspondiente a la variable más a la izquierda de la cadena que se está derivando.

Un ejemplo de este tipo de derivaciones, para la gramática

$A \rightarrow BF$

$B \rightarrow EC$

$E \rightarrow a$

$C \rightarrow b$

$F \rightarrow c$

puede verse en

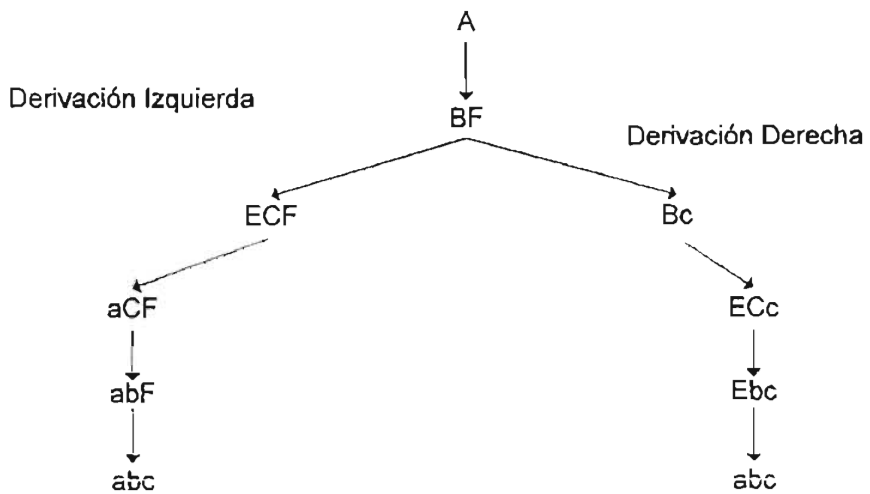


FIGURA 1.6.1

Definición 1.6.3 Sea $\alpha = \varphi_1\ \beta\ \varphi_2$ una forma sentencial de la gramática. Se dice que β es una ***frase*** de la forma sentencial α respecto de la variable A si y solamente si:

$$S \overset{*}{\Longrightarrow} \varphi_1 A \varphi_2$$

$$A \overset{+}{\Longrightarrow} \beta$$

Definición 1.6.4 Se dice que β es una ***frase simple*** si y solamente si:

$$S \overset{*}{\Longrightarrow} \varphi_1 A \varphi_2$$

$$A \Longrightarrow \beta$$

Por ejemplo si tenemos el conjunto de reglas definido por:

$P=\{ S \rightarrow zABz,\ B \rightarrow CD,\ C \rightarrow c,\ D \rightarrow d,\ A \rightarrow a\}$

y la forma sentencial $\alpha = zAcdz$:

$$S \Longrightarrow zABz \Longrightarrow zACDz \Longrightarrow zACdz \Longrightarrow zAcdz$$

- c es frase de α respecto de B
  - En efecto $S \Longrightarrow zABz$, *es decir* $\varphi_1 = zA$ *y* $\varphi_2 = z$
  - y $B \Longrightarrow CD \Longrightarrow cD$
- c es frase simple de α respecto de C
  - En efecto $S \Longrightarrow zABz \Longrightarrow zACDz$, *es decir* $\varphi_1 = zA$ *y* $\varphi_2 = Dz$
  - y $C \Longrightarrow c$
- d es frase simple de α respecto de D
  - En efecto $S \Longrightarrow zABz \Longrightarrow zACDz$, *es decir* $\varphi_1 = zAC$ *y* $\varphi_2 = z$
  - y $D \Longrightarrow d$

Definición 1.6.5 Se llama ***asa*** de una forma sentencial α a la ***frase simple situada más a la izquierda*** de α.

En el ejemplo anterior tenemos que c es le pivote de la forma sentencial $\alpha = zAcdz$ respecto de C, pues es la frase simple situada más a la izquierda de α.

Las gramáticas libres del contexto que se utilizan para describir la sintaxis de los lenguajes de programación generalmente están formadas por un amplio conjunto de reglas, por lo cual pueden existir ocultos distintos problemas, tales como tener reglas que produzcan símbolos que no se usen después, o que nunca se llegue a cadenas terminales. Esto se puede evitar realizando la transformación de la gramática inicial a una ***gramática limpia*** equivalente. Por equivalente queremos decir que ambas gramáticas generan el mismo lenguaje. Vamos a formalizar estos conceptos por medio de las siguientes definiciones:

- ***Símbolo inactivo***: es una clase sintáctica que no genera ninguna cadena de símbolos terminales.
- ***Símbolo activo***: es una clase sintáctica de la cual se puede derivar una cadena de símbolos terminales. Todos los símbolos terminales son símbolos activos. Es decir, son símbolos activos lo que no son inactivos.
- ***Símbolo inaccesible***: es una clase sintáctica a la que no se puede llegar por medio de producciones desde el símbolo inicial.
- ***Símbolo accesible***: es un símbolo que aparece en una cadena derivada del símbolo inicial. Es decir, aquel símbolo que no es inaccesible.
- ***Símbolo inútil***: se denomina así a todo símbolo inactivo o inaccesible.
- ***Símbolo útil***: se denomina así a todo símbolo que es activo y accesible.
- ***Gramática no limpia*** : es toda gramática que contiene símbolos inútiles o reglas de producción de la forma $A \rightarrow B$, llamadas producciones unitarias.
- ***Gramática limpia***: es toda gramática que contiene únicamente símbolos útiles y no contiene producciones unitarias.

A continuación vamos proporcionar algoritmos para depurar y limpiar las gramáticas. Primero necesitamos detectar todos los símbolos inactivos, y después detectar todos los símbolos inaccesibles. Debemos seguir este orden, ya que la eliminación de símbolos inactivos puede generar nuevos símbolos inaccesibles.

Algoritmo de los símbolos activos

Dada una gramática libre del contexto, vamos formular un algoritmo para determinar el conjunto de símbolos activos. Para ello nos basaremos en las siguientes observaciones:

- Cada símbolo en el conjunto $\left\{ A \mid A \rightarrow \alpha \text{ y } \alpha \in V_T \right\}$ es activo.
- Si todas las clases de la parte derecha de una regla de producción son símbolos activos, entonces la clase sintáctica correspondiente de la parte izquierda también es un símbolo activo.

Por lo que nuestro ***algoritmo para determinar los símbolos activos*** queda así:

1. Incluir al conjunto inicial de símbolos activos, las clases sintácticas que tienen al menos una producción con exclusivamente símbolos terminales.
2. Dada una regla de producción, si todas las clases sintácticas de la parte derecha pertenecen al conjunto de símbolos activos, entonces incluimos a la clase sintáctica de la parte izquierda en el conjunto.
3. Cuando no se puedan incluir más símbolos mediante la aplicación del paso 2, el conjunto consistirá de todos los símbolos activos, el resto serán inactivos.

A manera de ejemplo, sea la gramática definida por:

$$P=\{ S\rightarrow aABC,\ S\rightarrow Dd,\ A\rightarrow bBC,\ B\rightarrow e,\ B\rightarrow de,\ C\rightarrow gB,\ C\rightarrow h,\ D\rightarrow AfE,\ E\rightarrow tD,\ E\rightarrow vE\}.$$

Para determinar los símbolos inactivos, aplicamos los pasos:

1. Tenemos dos clases sintácticas con las cuales comenzar el conjunto
   - B por: $B\rightarrow e$ y $B\rightarrow de$.
   - C por: $C\rightarrow h$.
2. Recorremos las reglas de producción y vemos que podemos agregar a:
   - A porque: $A\rightarrow bBC$, y B y C ya están en el conjunto.
   - S porque: $S\rightarrow aABC$, y A, B y C ya están en el conjunto.
3. Ya no podemos incluir más símbolos en la lista, por lo tanto la lista de los simbolos activos es {S, A, B, C} y la de inactivos es {D, E}.

Algoritmo de los símbolos accesibles

El conjunto de símbolos accesibles puede obtenerse fácilmente, observando lo siguiente:

- El símbolo inicial de la gramática es accesible.
- Si la clase sintáctica en la parte izquierda de una regla de producción es accesible, entonces todos los símbolos de la parte derecha también lo son.

Por tanto, el ***algoritmo para determinar los símbolos accesibles*** es:

1. Iniciar el conjunto con un único no terminal, el símbolo inicial.
2. Si la clase sintáctica de una regla de producción está en el conjunto, incluir en el mismo a todas las clases sintácticas que aparezcan en la parte derecha.
3. Cuando ya no se puedan incluir más símbolos mediante la aplicación del paso 2, el conjunto contiene todos los símbolos accesibles, y el resto será inaccesible.

Veamos el siguiente ejemplo, de la gramática definida por:

$$P=\{ S\rightarrow aAB,\ \ S\rightarrow A,\ A\rightarrow cBd,\ B\rightarrow e,\ B\rightarrow fS,\ C\rightarrow gD,\ C\rightarrow hD,\ D\rightarrow x,\ D\rightarrow y,\ D\rightarrow z\}.$$

Para determinar los símbolos inactivos, aplicamos los pasos:

1. Lista inicial = $\{S\}$
2. Recorremos las reglas de producción y vemos que podemos agregar a:
   - A porque: $S\rightarrow A$, y S ya está en el conjunto.
   - B porque: $A\rightarrow cBd$, y A ya está en el conjunto.
3. Ya no podemos incluir más símbolos en el conjunto, por lo tanto el conjunto de los símbolos accesibles es $\{S, A, B\}$ y la de inaccesibles es $\{C, D\}$.

Así pues, obtenemos una gramática limpia al aplicar primero el algoritmo de los símbolos activos, luego el de los símbolos accesibles y por último eliminando todas la reglas de producción unitarias. Este es un orden de aplicación estricto, no podemos cambiarlo.

Pasamos ahora al problema de reconocimiento de una oración que da como resultado la construcción de un árbol sintáctico para esa oración; a este proceso también lo identificamos como análisis sintáctico.

Definición 1.6.6 Un árbol de derivación D para una gramática libre de contexto $G=(V_C, V_T, S, P)$, es un árbol ordenado y etiquetado que satisface:

- La raíz de D está etiquetada con S.
- Cada nodo interior está etiquetado con un no-terminal X_i de V_C.
- Cada nodo hoja está etiquetado con un terminal X_i de V_T o con ε.
- Si A es el no-terminal que etiqueta a un nodo interior y $X_1 \cdots X_n$ son las etiquetas de los hijos de ese nodo, de izquierda a derecha, entonces debe existir la regla $A \rightarrow X_1 \cdots X_n \in P$, donde los X_i pueden ser terminales o no-terminales.
- Como caso particular, si existe la regla $S \rightarrow \varepsilon \in P$, entonces el nodo A tiene un solo hijo etiquetado con ε.

Considérese la siguiente gramática con reglas de producción:

$P=\{ S \rightarrow S + S \ , \ S \rightarrow a \}$, donde $a \in V_T$.

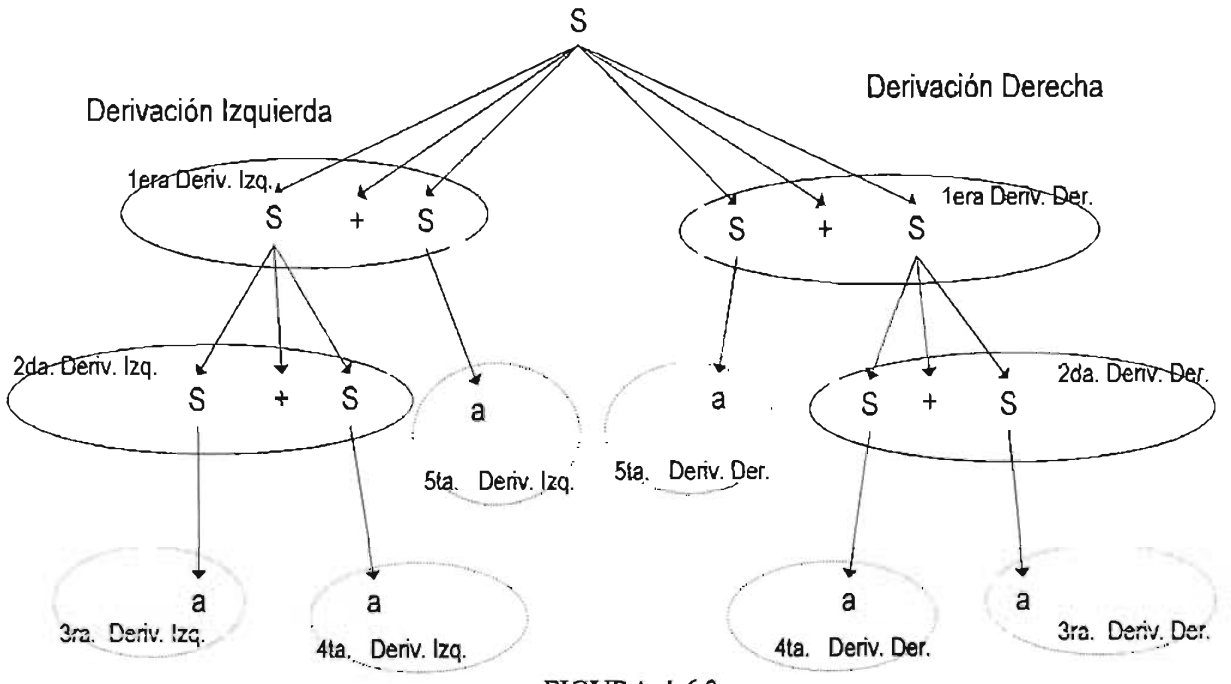


FIGURA 1.6.2

Es notorio que en este caso tenemos dos árboles sintácticos diferentes para la misma oración. La existencia de más de un árbol sintáctico para alguna oración del lenguaje puede

provocar que el compilador genere un conjunto de instrucciones (código objeto) diferente cada vez para esa oración; si el compilador debe realizar traducciones válidas de las oraciones de un lenguaje, el lenguaje debe estar definido sin ambigüedades.

Definición 1.6.7 Una oración generada por una gramática es ***ambigua*** si existe más de un árbol sintáctico para ella. Una ***gramática es ambigua***, si genera al menos una oración ambigua; en caso contrario es no ambigua.

Llamamos ambigua a la gramática, no al lenguaje que genera. Un lenguaje puede estar generado por gramáticas ambiguas y por no ambiguas. Sin embargo hay lenguajes que son inherentemente ambiguos, es decir no existen gramáticas no ambiguas que puedan generarlos.

¿Existe algún algoritmo que nos permita conocer si una gramática libre del contexto es ambigua? No, no existe tal algoritmo. Pero sí existen un conjunto de condiciones que si la gramática cumple nos permiten afirmar que la gramática es no ambigua. Estas condiciones son suficientes más no necesarias; es decir, una gramática puede no cumplirlas y ser de todos modos no ambigua. Veremos cuál es este conjunto de condiciones que deben cumplir las gramáticas libres del contexto cuando veamos el problema del reconocimiento o análisis sintáctico.

Veamos por último otro ejemplo de una gramática ambigua y las complicaciones que puede provocar. Consideremos la gramática para expresiones aritméticas sin paréntesis que consiste de los operadores +, × con identificadores de una sola letra, es decir

P={ exp→exp + exp , exp→exp × exp , exp→i }, donde $i \in V_T$.

Supongamos que la oración que deseamos reconocer es $i \times i + i$, y vamos a tomar siempre la derivación más a la izquierda en cada ocasión. Tenemos dos posibles derivaciones a saber:

exp ⇒ exp × exp
⇒ i × exp
⇒ i × exp + exp
⇒ i × i + exp
⇒ i × i + i

y

$$\begin{aligned} \text{exp} &\Rightarrow \text{exp} + \text{exp} \\ &\Rightarrow \text{exp} \times \text{exp} + \text{exp} \\ &\Rightarrow i \times \text{exp} + \text{exp} \\ &\Rightarrow i \times i + \text{exp} \\ &\Rightarrow i \times i + i \end{aligned}$$

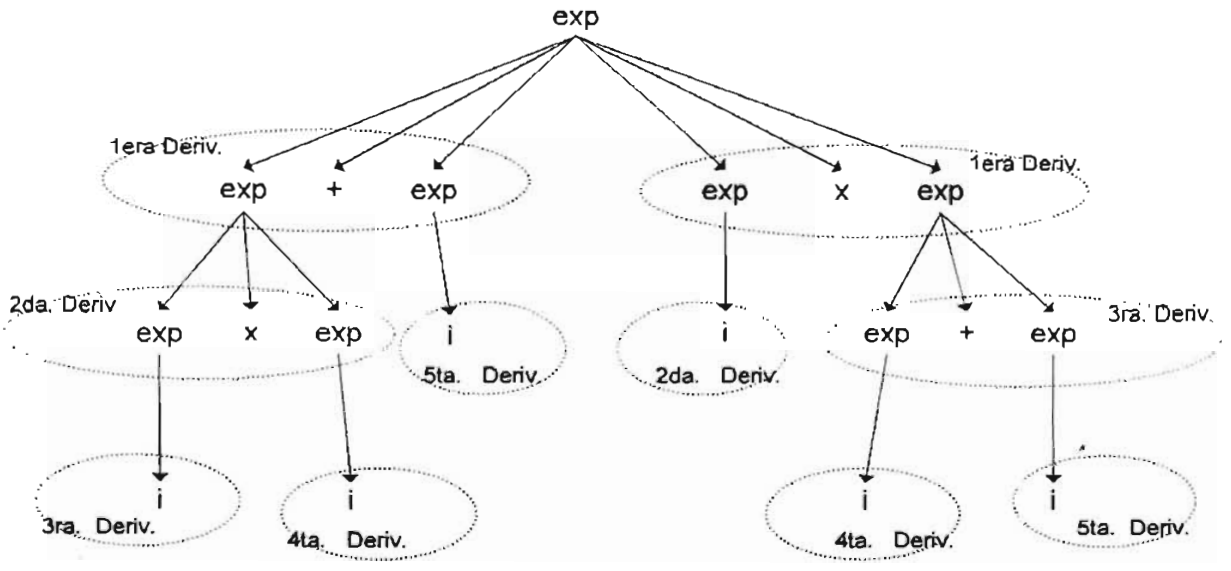


FIGURA 1.6.3

Tenemos dos árboles sintácticos para la oración $i \times i + i$, por lo tanto la gramática es ambigua. Y es ambigua porque no sabemos si evaluar primero la multiplicación o la suma. Podemos modificarla de tal manera que la multiplicación tenga precedencia sobre la suma:

P={ **exp**→**term**, **exp**→**exp** + **term** , **term**→**term** × **fact**, **term**→**fact**, **fact**→i }.

De esta manera un **term** puede involucrar la multiplicación de dos **fact**, pero hasta que resolvemos esta derivación podemos pensar en sumar un **term** con otro, con lo cual le hemos dado precedencia a la multiplicación sobre la suma. El árbol sintáctico sería ahora:

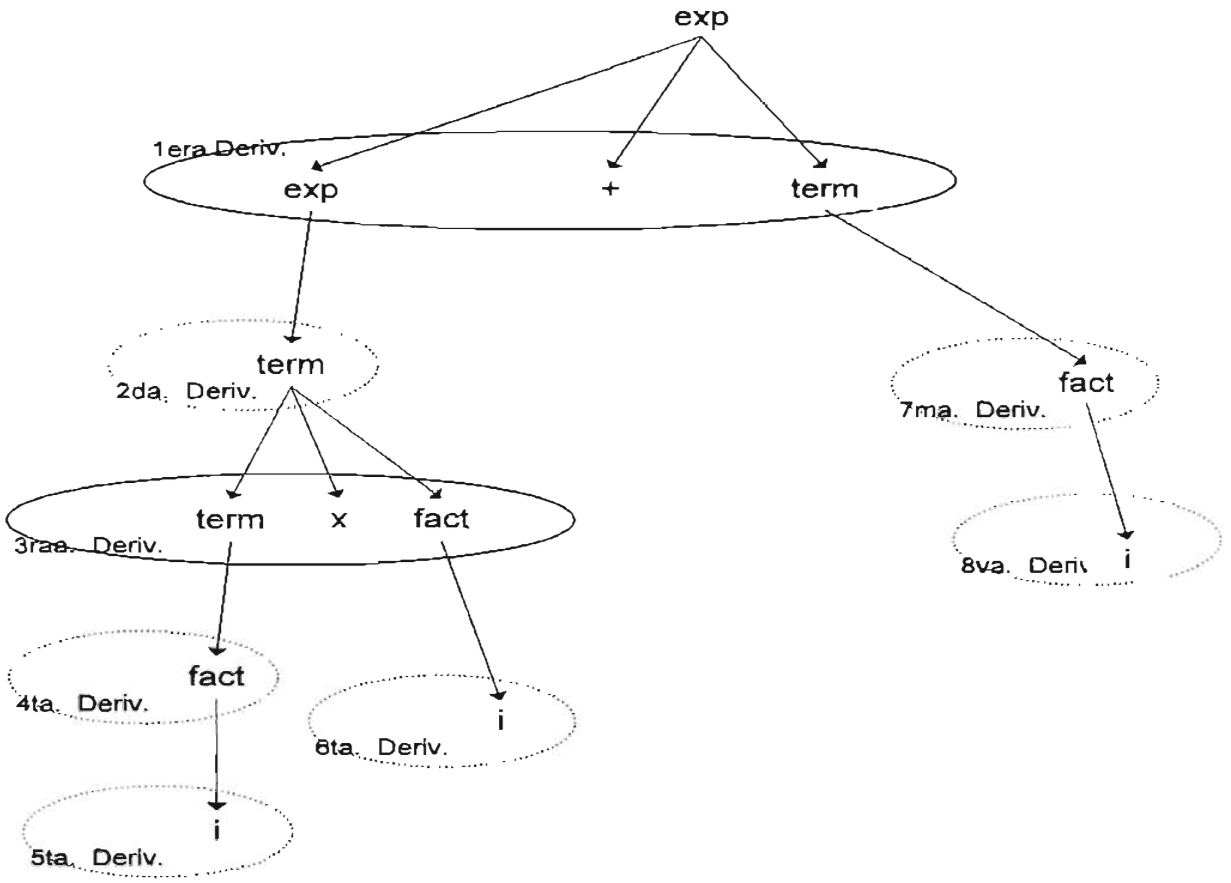


FIGURA 1.6.4

Como podemos observar, al derivar una oración, el rol que juega el árbol sintáctico es vital para representar adecuadamente la sintaxis del lenguaje. El siguiente paso es saber cómo construimos este árbol a partir de una gramática y una oración del lenguaje dadas, lo cual trataremos en la sección del ***análisis sintáctico*** o ***parsing***.

Compilador de descripciones sintácticas

Capítulo 2

2.1 Notación BNF y EBNF

La notación BNF, sirve para describir un lenguaje; las siglas significan "Backus Naur Form". John Backus desarrolló primordialmente la mayor parte de la descripción de la notación(basado en un trabajo del matemático Emile Post), Peter Naur la adoptó y le hizo ligeras mejoras para su reporte de ALGOL 60, que son casi universalmente aceptadas ahora. Desde entonces se ha utilizado para definir formalmente las gramáticas de los protocolos de comunicaciones, de algunas partes de las gramáticas de los lenguajes naturales y de los lenguajes de programación, que por lo general son gramáticas libres del contexto.

En realidad BNF es un metalenguaje; un ***metalenguaje*** sirve para describir otro lenguaje o algunos aspectos de otro lenguaje. En BNF el símbolo ::=' se utiliza en lugar de →' para indicar "está definido como". Adicionalmente, las clases sintácticas se escriben entre paréntesis angulares y los símbolos terminales entre comillas . El metasímbolo | significa "o", es decir una alternativa. Por ejemplo una definición típica escrita en BNF para un identificador quedaría así:

<identificador>::=<letra> | <identificador><letra> | <identificador><dígito>

<letra>::= "a" | "b" | "c" | ... | "y" | "z"

<dígito>::= "0" | "1" | "2" | ... | "8" | "9"

Aun cuando es posible describir la sintaxis de los lenguajes de programación con esta notación, se utiliza una variante conocida como ***BNF extendido (EBNF)***, que permite una

especificación más compacta y amigable. Ambas notaciones son equivalentes aunque se utiliza EBNF por conveniencia no porque incremente el poder expresivo de la gramática definida.

Veamos cuales son algunas de las ventajas de utilizar EBNF. Supongamos que tenemos la siguiente descripción escrita en BNF para la clase sintáctica <**expresión**>:

<**expresión**>::= <**término**> |<**expresión**> "+" <**término**>

La clase sintáctica <*expresión*> puede contener un número arbitrario de clases sintácticas <*término*>, por lo que hacemos uso de la recursividad.

Si agregamos los metasímbolos { y } a nuestra notación y definimos a {**x**} como cero o más ocurrencias de **x**, podemos reescribir a la clase sintáctica <**expresión**> como:

<**expresión**>::= <**término**> { "+" <**término**>}

Entre los metasímbolos { y } puede ir cualquier cadena de símbolos incluso el metasímbolo |, así por ejemplo { "*letra*" | "*dígito*"}, significa una sucesión de cero o más símbolos, cada uno de los cuales puede ser letra o dígito.

También agregamos los metasímbolos *[* y *]* a nuestra notación y definimos a *[***x***]* como cero o una ocurrencias de **x**. Dicho de otra manera, *[***x***]* significa que **x** es un símbolo opcional. Por ejemplo para describir un bloque de instrucciones en Pascal podemos escribir en notación BNF:

<**block**> ::= "**BEGIN**" <**opt-stats**> "**END**".

<**opt-stats**> ::= <**stats-list**>| ε.

<**stats-list**> ::= <**statement**>| <**statement**> ";" **stats-list** .

Aprovechándonos de EBNF, podríamos escribir:

<**block**> ::= "**BEGIN**" *[*<**stats-list**>*]* "**END**".

<**stats-list**> ::= {<**statement**> ";"} <**statement**> .

Resumen de la notación EBNF propuesta.

SÍMBOLO	SIGNIFICADO	EJEMPLOS
Cadena	Símbolo Terminal	"=", "while"
Nombre	Clase Sintáctica	*ident, statement*
=	Separa los lados de una producción	$A=bcd$
.	Termina una producción	
|	Separa alternativas	$a \mid b \mid c$, a o b o c
(...)	Grupo de Alternativas	$a(b \mid c)$, $ab \mid ac$
[...]	Parte Opcional	$[a]b$, $ab \mid b$
{...}	Se repite cero o más veces	$\{a\}b$ ° $b \mid ab \mid aab \mid aaab \mid \ldots$

2.2 Análisis sintáctico con descenso recursivo

El análisis sintáctico generalmente se divide en dos partes:

- ***Análisis léxico***, que se relaciona con un autómata finito determinista basado en una gramática regular.
- ***Análisis sintáctico*** , que lleva a cabo con autómata de pila , basado en una gramática libre de contexto que por lo general se describe por medio de BNF o EBNF.

Los analizadores sintácticos se clasifican de acuerdo a:

- La manera en que procesan las cadenas de carácteres del lenguaje, en:
  - ***Métodos direccionales***: símbolo a símbolo de izquierda a derecha.
  - ***Métodos no-direccionales***: tienen acceso a cualquier lugar de la cadena de entrada.
- El número de alternativas posibles en una derivación:
  - ***Métodos deterministas***: dado un símbolo de la cadena de entrada se puede decidir en cada paso cual es la alternativa/derivación adecuada a aplicar, no hay retroceso y el tiempo de proceso es lineal y depende de la longitud de la cadena.

- ***Métodos no-deterministas***: en cada paso de la construcción del árbol sintáctico se prueban diferentes alternativas/derivaciones para ver cual es la adecuada, con el correspondiente aumento de tiempo del proceso.

Hay tres tipos generales de analizadores sintácticos:

- ***Analizadores Generales de Gramáticas Libres de Contexto***

  Los algoritmos más referenciados para reconocer las gramáticas libres de contexto son:

  - El algoritmo CYK introducido en 1963 por Cocke, Younger y Kasami, únicamente opera con gramáticas libres de contexto en forma normal de Chomsky.
  - El algoritmo de Early, trata con gramáticas libres de contexto en general.

  No se consideran eficientes para ser utilizados de forma generalizada ya que su tiempo de proceso crece en forma exponencial dependiendo de la longitud de la cadena de entrada.

- ***Analizadores Descendentes***

  Construyen el árbol sintáctico de la sentencia a reconocer desde el símbolo inicial (raíz), hasta llegar a los símbolos terminales (hojas) que forman la sentencia, usando derivaciones más a la izquierda.

  Entre estos se encuentran el descenso recursivo y los analizadores LL(k), estos últimos pueden realizar un análisis sin retroceso en forma descendente.

- ***Analizadores Ascendentes***

  Parten desde las hojas hasta llegar a la raíz, para construir el árbol sintáctico de la sentencia a reconocer; el principal problema que enfrentan es el retroceso. Para solucionarlo hay distintos tipos de gramáticas entre las cuales las más utilizadas son las LR(k) ya que realizan eficientemente el análisis ascendente sin retroceso.

Vamos a describir únicamente los analizadores sintácticos con descenso recursivo sin retroceso. El problema del retroceso es que a partir del nodo raíz, el analizador sintáctico no elige las producciones adecuadas para alcanzar la sentencia a reconocer. Cuando se da cuenta de que se ha equivocado de producción, tiene que deshacer las producciones aplicadas hasta encontrar otras producciones alternativas, volviendo a tener que reconstruir

parte del árbol sintáctico. El retroceso puede afectar a otros módulos del compilador tales como la lectura en sí misma, la tabla de símbolos o el código generado, teniendo que deshacerse también los procesos generados en estos módulos.

Para ejemplificar el retroceso, tomemos la siguiente gramática: $G=(V_C, V_T, S, P)$, donde:

V_C={<Programa>, <Declaraciones>, <Procedimientos>}

V_T={**Módulo, d, p, ;, Fin**}

S=<Programa>

con reglas de producción P :

<Programa> ::= **Módulo** <Declaraciones>;<Procedimientos>**Fin**

<DECLARACIONES> ::= **d** | **d**; <Declaraciones>

<Procedimientos> ::= **p** | **p**; <Procedimientos>

Deseamos analizar la cadena de entrada : **Módulo d ; d ; p ; p Fin .**

Tomamos la siguiente convención para saber en qué punto nos encontramos de la cadena de entrada y de la cadena producida; anteponemos un # antes del símbolo:

- que intentamos reconocer de la cadena de entrada; así:

  Módulo d ; d ; # p ; p Fin

  significa que intentamos reconocer el primer símbolo terminal p.

- de los símbolos terminales producidos y que no hemos comparado con la cadena de entrada o de las clases sintácticas que no hemos reemplazado por su regla de producción; así

  Módulo #d ; <Procedimientos> Fin

  significa que d es el símbolo terminal que hemos generado y que no ha sido comparado con la cadena de entrada.

Procedemos de la siguiente manera:

1. Cadena producida = #<Programa>

   y cadena de entrada = # **Module d ; d ; p ; p Fin**

2. La única regla que podemos aplicar es la primera regla de producción de la gramática:

   #<Programa> ⟹ # **Módulo**<Declaraciones>;<Procedimientos>**Fin**

   Módulo es un símbolo terminal, y coincide con el primero de la cadena de entrada. Recorremos # en ambas cadenas por lo que tenemos

   Cadena de entrada=**Módulo # d ; d ; p ; p Fin**

   Cadena Producida =**Módulo#**<Declaraciones>;<Procedimientos>**Fin**

3. La derivación más a la izquierda que podemos aplicar es **<Declaraciones>**, ¿pero cuál aplicamos?

   a) <Declaraciones> ⟹ **d**

   b) |<Declaraciones> ⟹ **d** ; <Declaraciones>

   tomamos la primera derivación. Con lo que tenemos

   Cadena Producida =**Módulo#d;**<Procedimientos>**Fin**

   como coincide con nuestro símbolo por reconocer, recorremos nuevamente # al siguiente símbolo de la cadena de entrada y de la cadena producida tenemos:

   Cadena de entrada=**Módulo d #; d ; p ; p Fin**

   Cadena Producida =**Módulo d#;**<Procedimientos>**Fin**

   Nuevamente coinciden con nuestro símbolo por reconocer, recorremos nuevamente # al siguiente símbolo de la cadena de entrada y de la cadena producida, y tenemos:

   Cadena de entrada=**Módulo d ; #d ; p ; p Fin**

   Cadena Producida =**Módulo d; #**<Procedimientos>** Fin**

4. Ahora la derivación más a la izquierda que podemos aplicar es <Procedimientos>, tenemos dos opciones, tomamos la primera y vemos qué pasa; así tenemos

   Cadena Producida =**Módulo d;#p Fin**

   no coincide con la cadena de entrada. Tomamos entonces la segunda que da:

   Cadena Producida =**Módulo d;#p;** <Procedimientos>** Fin**

tampoco coincide con la cadena de entrada. Debemos volver atrás, hasta encontrar la última derivación de una clase sintáctica y comprobar si tiene alguna alternativa más. Si es así debemos elegir la siguiente y probar. En caso contrario, volver más atrás para probar con la clase sintáctica anterior. A este fenómeno de regresar atrás es lo que se ha definido anteriormente como retroceso. Si en este retroceso llegamos a que no hemos logrado aparear la cadena y ya no hay más opciones por probar estaríamos ante una cadena errónea sintácticamente.

5. Dado que no hemos encontrado el terminal de la cadena de entrada, necesitamos regresar atrás hasta la clase sintáctica analizada anteriormente, que en este caso es **<Declaraciones>** y tomamos la segunda alternativa . Análogamente debemos retroceder también en la cadena de entrada, ya que lo único válido que nos ha quedado del reconocimiento (árbol sintáctico) ha sido el primer símbolo terminal **Módulo**. Derivando con la segunda opción tenemos:

   Cadena de entrada=**Módulo # d ; d ; p ; p Fin**

   Cadena Producida=**Módulo #d;<Declaraciones>;<Procedimien tos>Fin** Con esto vemos que los siguientes dos símbolos de la cadena de entrada y de la cadena producida coinciden, por lo tanto recorremos # y tenemos:

   Cadena de entrada=**Módulo d ; #d ; p ; p Fin**

   Cadena Producida=**Módulo d;#<Declaraciones>;<Procedimien tos>Fin**

6. Ahora la derivación más a la izquierda es nuevamente **<Declaraciones>**, aplicamos la primera opción lo que nos da:

   Cadena de entrada=**Módulo d ; #d ; p ; p Fin**

   Cadena Producida=**Módulo d;#d;<Procedimientos>Fin**

   Nuevamente los siguientes dos símbolos de la cadena de entrada y de la cadena producida coinciden, por lo tanto recorremos # y tenemos:

   Cadena de entrada=**Módulo d ; d ; #p ; p Fin**

   Cadena Producida=**Módulo d;d;#<Procedimientos>Fin**

7. La derivación más a la izquierda es **<Procedimientos>**, aplicamos la primera opción y obtenemos:

   Cadena de entrada=**Módulo d ; d ; #p ; p Fin**

Cadena Producida=**Módulo d;d;#p Fin**

Como coincide con nuestro símbolo por reconocer, recorremos nuevamente # al siguiente símbolo de la cadena de entrada y de la cadena producida tenemos:

Cadena de entrada=**Módulo d ; d ; p #; p Fin**

Cadena Producida=**Módulo d;d;p#Fin**

8. Nuevamente tenemos un problema porque el símbolo terminal **Fin** no coincide con el símbolo por reconocer de la cadena de entrada. Otra vez debemos regresar hasta la clase sintáctica anterior que es <**Procedimientos**> y probar con otra opción, que es la segunda, lo que nos da:

   Cadena de entrada=**Módulo d ; d ; #p ; p Fin**

   Cadena Producida=**Módulo d;d;#p ;<Procedimientos>Fin**

   También nuevamente vemos que los siguientes 2 símbolos de la cadena producida coinciden con la cadena de entrada, por lo tanto, nos queda:

   Cadena de entrada=**Módulo d ; d ; p ; #p Fin**

   Cadena Producida=**Módulo d;d;p ;#<Procedimientos>Fin**

9. Probamos con la primera opción de <**Procedimientos**> y llegamos a:

   Cadena de entrada=**Módulo d ; d ; p ; #p Fin**

   Cadena Producida=**Módulo d;d;p ;#p Fin**

   Con lo cual finaliza nuestro reconocimiento, puesto que los dos últimos símbolos de la cadena de entrada coinciden con los de cadena producida.

De este proceso observamos que el reconocimiento de sentencias de un lenguaje puede dispararse a causa del retroceso, por lo tanto los analizadores sintácticos deben tratar de eliminar las causas que producen el retroceso. Para eliminarlo se ha de elegir correctamente la producción correspondiente a cada clase sintáctica que se expande. Es decir el análisis descendente tiene que ser determinista y sólo debemos tomar una opción en la expansión de cada no terminal.

Esto nos lleva a las ***gramáticas LL(k)*** que son un subconjunto de las gramáticas libres de contexto. Por medio de ellas podemos realizar un análisis descendente determinista (o sin retroceso), el reconocimiento de la cadena de entrada es de izquierda a derecha ("**L**eft

to right") y vamos tomando las derivaciones más hacia la izquierda ("**L**eftmost"), con sólo revisar (sin leer) los siguientes ***k*** símbolos pendientes en la cadena de entrada; de ahí su nombre. Si k=1 estamos hablando de gramáticas LL(1).

Las gramáticas LL(1) nos permiten construir un analizador determinista descendente con tan sólo examinar en cada momento el símbolo actual de la cadena de entrada (que nos entrega el analizador léxico) para saber cual producción aplicar.

A continuación describimos las condiciones que debe cumplir para ser considerada como una gramática LL(1); estas condiciones fueron definidas originalmente por D.E. Knuth.

Primera Condición de Knuth

No permite producciones de la forma $A \rightarrow A\alpha$, donde $A \in V_C$ y $\alpha \in V_G^*$. Esta condición equivale a no admitir la recursividad por la izquierda.

Segunda Condición de Knuth

Los símbolos terminales iniciales de las distintas alternativas de una regla de producción deben formar conjuntos disjuntos. Es decir, si

$$A \rightarrow B\beta \mid C\delta \qquad A,B,C \in V_C \quad \beta, \delta \in V_G^*$$

No debe ocurrir que

$$B \rightarrow d\varphi \qquad d \in V_T$$

$$C \rightarrow d\omega \qquad \varphi, \omega \in V_G^*$$

Esto implica que en todo momento, el símbolo terminal que estamos examinando señala sin ambigüedad, que alternativa hemos de escoger sin posibilidad de error y en consecuencia, sin retroceso.

Tercera condición de Knuth

Si una alternativa de una producción de una clase sintáctica origina la cadena vacía, los símbolos terminales que pueden seguir a dicho clase sintáctica en la producción donde aparezca, han de formar un conjunto disjunto con los terminales que pueden encabezar las distintas alternativas de dicho terminal, es decir:

(***CST Iniciales***$(A) \cap$ ***CST Siguientes***$(A) = \emptyset$), donde CST son las siglas para Conjunto de Símbolos Iniciales.

Veamos un ejemplo del problema que se presenta cuando no se cumple esta condición. Sea la cadena $A_1 \ldots A_2 \ldots A_3 A_4 A_5$ y sea A_3 el símbolo que estamos analizando; además tenemos las producciones:

$A_3 \rightarrow a\varphi \mid \varepsilon$

$A_4 \rightarrow A_3 a\psi$

Como A_3 puede derivar en la cadena vacía y

CST Iniciales(A_3)= { a }

CST Siguientes(A_3)= { a }

no podemos determinar si hemos de elegir la producción de A_3 o de A_4.

Esta condición nos 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 son distintos, si no, no podríamos saber si empezamos a reconocer la clase sintáctica o por el contrario ya la hemos reconocido.

Cuarta condición de Knuth

Ningún clase sintáctica puede tener dos o más alternativas que conduzcan a la cadena vacía. Esta condición deriva de la anterior. Por ejemplo no se permite:

$A \rightarrow B \mid C$

$B \rightarrow \varepsilon \mid D$

$C \rightarrow \varepsilon \mid E$

Es obvio que $A \overset{*}{\Rightarrow} \varepsilon$.

De manera intuitiva hemos manejado CST Iniciales y CST Siguientes, a continuación vamos a definirlos de manera precisa y además proporcionamos los algoritmos necesarios para construirlos.

Se define el Conjunto de Símbolos Terminales Iniciales (en inglés FIRST) de un símbolo

$\alpha \in V_G^*$, como el conjunto de símbolos terminales que pueden aparecer al principio de cadenas derivadas de α. La definición anterior se puede expresar como:

$$\text{CST Iniciales(a)} = \{a \mid \alpha \overset{*}{\Longrightarrow} a \ldots \text{ donde } a \in V_T\}$$

Si $\alpha \overset{*}{\Longrightarrow} a\, \varphi_1 \ldots \varphi_n$ entonces $\{a\} \in$ CST Iniciales(a) con $a \in V_T$

Si $\alpha \overset{*}{\Longrightarrow} \varepsilon$ entonces $\{\varepsilon\} \in$ CST Iniciales(a).

Algoritmo para calcular CST Iniciales

Usamos este cálculo de ***CST Iniciales*** para todos los símbolos gramaticales.

1. Iniciamos el CST Iniciales con el conjunto vacío.
2. Procesamos cada regla de la gramática de la siguiente manera:
   a) Si el lado derecho inicia con un símbolo terminal, agregamos este símbolo al CST Iniciales de la clase sintáctica de la parte izquierda, dado que puede ser el primer símbolo de la forma sentencial derivada por la parte izquierda.
   b) Si la parte derecha inicia con una clase sintáctica, agregamos todos los símbolos que hay en CST Iniciales de esta clase sintáctica al CST Iniciales de la clase sintácticas correspondiente a la parte izquierda.

   Estos son todos los símbolos que pueden ser los terminales iniciales de esta forma sentencial derivada a partir de la parte izquierda.
3. Repetimos hasta que no se pueden añadir más símbolos terminales o ε al conjunto.

Ejemplo del cálculo de CST Iniciales, sea la gramática con P:

S → A B e

A → d B

A → a S

A → c

B → A S

B → b

1. CSTi(A) = {} ∪ CSTi(d B), por A→d B	**9. CSTi(A) = {d a c}**
2. CSTi(d) = {d}, porque d es terminal	**10. CSTi(B) = {} ∪ CSTi(A S), por B→ A S**
3. CSTi(A) = {d}	**11. CSTi(B) = {d a c}**
4. CSTi(A) = {d} ∪ CSTi(a S), por A→a S	**12. CSTi(B) = {d a c} ∪ CSTi(b), por B→ b**
5. CSTi(a) = {a}, porque a es terminal	**13. CSTi(b) = {b}, porque b es terminal**
6. CSTi(A) = {d a}	**14. CSTi(B) = {d a c b}**
7. CSTi(A) = {d a} ∪ CSTi(c), por A→c	**15. CSTi(S) = {} ∪ CSTi(A B e), por S→ ABe**
8. CSTi(c) = {c}, porque c es terminal	**16. CSTi(S) = {d a c}**

Para una gramática libre de contexto con un símbolo inicial S, y una clase sintáctica A, se dice que el Conjunto de Símbolos Terminales Siguientes(A) es el conjunto de símbolos terminales que en cualquier momento de la derivación pueden aparecer inmediatamente a la derecha de (o después de) A. La definición formal se puede expresar como:

CST Siguientes(A) = $\{ a \mid S \overset{*}{\Longrightarrow} \beta A a \varphi$ con $\beta, \varphi \in V_G^* \}$

El CST Siguientes(en inglés FOLLOW) de una clase sintáctica también se pueden definir como los símbolos iniciales del símbolo que sigue a la clase sintáctica.

Algoritmo para calcular CST Siguientes

Cada clase sintáctica tiene un conjunto CST Siguientes que puede calcularse de la siguiente manera:

1. Iniciamos el CST Siguientes con el conjunto vacío.
2. En seguida procesamos cada una de las partes derechas; cuando ésta contiene una clase sintáctica como en $A \rightarrow \ldots B\varphi$, agregamos todos los símbolos de CST Iniciales(φ) a CST Siguientes(B); si además $\varphi \overset{*}{\Longrightarrow} \varepsilon$, se agregan todos los símbolos de CST Siguientes(A) a CST Siguientes(B).
3. El paso anterior se repite hasta que no haya más símbolos por agregar en los conjuntos CST Siguientes.

Tomando la gramática del ejemplo anterior tenemos los siguientes conjuntos CST Siguientes:

1. CSTs(A) = {} ∪ CSTi(B), por S→A B e	**8. CSTs(B) = {e d a c b}**
2. CSTs(A) = {d a c b}	**9. CSTs(S) = {$}**
3. CSTs(A) = {d a c b} ∪ CSTi(S), por B→ A S	**10. CSTs(S) = {$} ∪ CSTs(A), por A→ a S**
4. CSTs(A) = {d a c b}	**11. CSTs(S) = {$ d a c b}**
5. CSTs(B) = {} ∪ CSTi(e), por S→A B e	**12. CSTs(S) = {$ d a c b} ∪ CSTs(B), por B→ AS**
6. CSTs(B) = {e}	**13. CSTs(S) = {$ d a c b e}**
7. CSTs(B) = {e} ∪ CSTs(A), A→ d B	

Pasamos ahora a la definición de los Conjuntos de Símbolos Terminales Directores o CST Directores, de una producción o regla sintáctica; como su nombre lo indica, dirigen al analizador sintáctico para elegir la alternativa adecuada; es el conjunto de símbolos terminales que determinan qué expansión de una regla sintáctica se ha de elegir en un momento dado, con sólo mirar un símbolo hacia adelante. La definición es la siguiente:

Dada una regla sintáctica de la forma $A \rightarrow \alpha$ donde A es una clase sintáctica, y α es una cadena de símbolos terminales y clases sintácticas. Definimos al conjunto de símbolos directores ***CST Directores***(A, α) de una regla de producción $A \rightarrow \alpha$ como:

- CST Iniciales(α) si α no genera la cadena vacía.
- CST Iniciales (α) ∪ CST Siguientes(A) si α genera la cadena vacía.

Continuando con nuestro ejemplo tendríamos que los símbolos directores para cada una de nuestras reglas de producción es:

- CST Directores(**S**, **A B e**) = CST Iniciales(**A**)={**d a c**}
- CST Directores(**A**, **d B**) = CST Iniciales(**d**) = {**d**}
- CST Directores(**A** , **a S**) = CST Iniciales(a) = {**a**}
- CST Directores(**A**, **c**) = CST Iniciales(**c**) = {**c**}
- CST Directores(**B**, **A S**) = CST Iniciales(**A**)={**d a c**}
- CST Directores(**B**, **b**) = CST Iniciales(**b**)={**b**}

La condición necesaria y suficiente para que una gramática limpia sea LL(1), es que los símbolos directores correspondientes a las diferentes expansiones de cada clase sintáctica sean conjuntos disjuntos. Continuando con el ejemplo que venimos manejando, vemos que:

CST Directores(**A**, **d B**) ∩ CST Directores(**A** , **a S**) ∩ CST Directores(**A**, **c**) = {**d**}∩ {**a**}∩ {**c**} = $\emptyset$, y que CST Directores(**B**, **A S**) ∩ CST Directores(**B**, **b**) = {**d a c**}∩ {**b**}=$\emptyset$; por lo tanto podemos afirmar que esta gramática es LL(1).

La definición anterior es equivalente a las condiciones de Knuth para gramáticas LL(1). Con esto concluimos el tratamiento de gramáticas LL(1) y pasamos a describir el analizador sintáctico con descenso recursivo sin retroceso.

La condición necesaria para que el análisis recursivo descendente opere correctamente es que la gramática del lenguaje fuente sea LL(1). El análisis se basa en la ejecución, en forma recursiva, de un conjunto de procedimientos que se encargan de procesar la entrada y determinar si la cadena de entrada pertenece o no al lenguaje. Iniciamos la descripción del analizador, conviniendo lo siguiente:

- Existe un procedimiento *leeSig*, definido como:

  procedimiento *leeSig*;

  { *sig*:= *AnalizadorLexico.leeSig*;

  };

  que le permite en todo momento, al analizador sintáctico saber cual es el símbolo que tiene pendiente por reconocer y que le entrega el analizador léxico por medio de la variable *sig*.

- Existe un procedimiento llamado *EsEntradaValida*, definido como:

  procedimiento *esEntradaValida*(*TerminalEsperado*)

  { Si (*sig* = *TerminalEsperado*)

  LeeSig; // Reconocimos símbolo de la entrada, avanzamos al siguiente

  EnCasoContrario

  despliegaError('Esperaba Símbolo Terminal = ' + *TerminalEsperado*;

  Abortar; // Detenemos el análisis sintáctico.

  }

A continuación definimos la forma de cada procedimiento dependiendo de la modalidad de la regla sintáctica:

- ***Cómo reconocer símbolos terminales***: *a* **:**

  esEntradaValida(a);

  hacemos una llamada para ver si es el símbolo de la cadena de entrada, es una entrada válida.

- ***Cómo reconocer clases sintácticas***: B **;**

  B **;**

  invocamos al procedimiento B, que debe estar definido de acuerdo con estas reglas. En estos casos no hay símbolos terminales que comparar contra la cadena de entrada, por lo que no se avanza ésta.

- Cómo reconocer sucesiones de símbolos, por ejemplo : $A \rightarrow a\ B\ c\ D$ **;**

  Procedimiento *A*;

  { *esEntradaValida(a);*

  B **;**

  esEntradaValida(c);

  D **;**

  }

- ***Cómo reconocer alternativas de símbolos***, por ejemplo : $A \rightarrow \alpha \mid \beta \mid \varphi$

  Si (sig ∈ CST Directores(A, α) entonces intentamos reconocer { ... α ... }

  EnCasoContrario

  Si (sig ∈ CST Directores(A, β) entonces intentamos reconocer { ... β ... }

  En Caso Contrario

  Si (sig ∈ CST Directores(A, φ) entonces intentamos reconocer { ... φ ... }

  EnCasoContrario

  despliegaError('Esperaba Simbolo Terminal = '+ *TerminalEsperado*;

  Abortar; // Detenemos el análisis sintáctico.

  Ejemplo: Retomemos la gramática:

  S → A B e

  A → d B

A → a S

A → c

B → A S

B → b

El procedimiento para la clase sintáctica B, sería:

```
Procedimiento B;
{   Si (sig = d || sig = a) || sig = c)  // ¿sig se encuentra en CST Directores(B, AS)?
    {   A;              // Continuamos el reconocimiento con la clase sintáctica A
        S;              // Continuamos el reconocimiento con la clase sintáctica S
    }
    EnCasoContrario
    {   Si      (sig=b)         // ¿Sig se encuentra en CST Directores(B, b)?
                esEntradaValida(b);     // Validamos contra la cadena de entrada
        EnCasoContrario
                despliegaError('Esperaba Símbolo Terminal = '+ TerminalEsperado;
                Abortar; // Detenemos el análisis sintáctico.
    }
}
```

- Cómo reconocer opciones escritas en EBNF : [β]

Si (*sig* ∈ CST Directores(β) entonces intentamos reconocer { ... β... }

Ejemplo: Si $A \rightarrow [b\ e]\ c$, el procedimiento para A, nos quedaría:

```
Procedimiento A
{       Si (Sig=b)
{       esEntradaValida(b);     // Validamos contra la cadena de entrada
        esEntradaValida(e);     // Validamos contra la cadena de entrada
}
esEntradaValida(c);     // Validamos contra la cadena de entrada
}
```

- Cómo reconocer iteraciones escritas en EBNF : { β }

Mientras (*sig* ∈ CST Directores(β)) intentamos reconocer { ... β... }.

Ejemplo, sea

$C \rightarrow b\ \{B\}\ a.$

$B \rightarrow d \mid e.$

En este caso, CST Directores(B)={d e}, por lo tanto:

```
Procedimiento C;
{       esEntradaValida(b);
        Mientras (sig=b || sig =a)
        B;
        esEntradaValida(e);
}
```

2.3 Descripción general del sistema

Nuestra misión primordial es traducir de BNF extendido a diagramas de trenes; para lograrlo debemos diseñar e implementar un sistema que le permita al usuario interactuar con un archivo de gramática (que es un archivo de texto escrito de acuerdo a las reglas gramaticales del EBNF que aquí se propone) y modificar y escoger las clases sintácticas que desea traducir a diagramas de trenes.

A partir de esta descripción básica podemos deducir lo siguiente: el sistema debe ser capaz de permitir al usuario:

1. Abrir, crear y guardar un archivo de gramática.
2. Elegir la clase sintáctica cuya regla de producción desea traducir a diagramas de trenes.
3. Insertar, eliminar, modificar clases sintácticas.
4. Modificar la correspondiente regla de producción, escrita en EBNF, de las clases sintácticas.

Traducir a diagramas de trenes la clase sintáctica elegida es la función primordial del sistema. Para hacerlo debemos contar con el Analizador Léxico que alimente al Analizador

Sintáctico con el siguiente símbolo en la cadena de entrada (que no es otra que la regla de producción escrita en el EBNF propuesto), para que éste haga el reconocimiento y construya el árbol sintáctico y la tabla se símbolos, con el cual el Generador de Diagramas (en el caso de un compilador normal sería el generado de código) dibuja los diagrama de trenes.

Sin embargo, el Analizador Sintáctico anterior reconoce sólo una clase sintáctica y su correspondiente regla de producción; deseamos que así sea porque el diagrama de tren de cada clase sintáctica es independiente del resto; es decir, nuestro reconocimiento no necesita leer todas las clases sintácticas y sus correspondientes reglas para determinar si se puede generar el diagrama de tren de una clase sintáctica en particular.

Pero recuérdese que deseamos que el usuario pueda interactuar con el archivo que está trabajando, no que sólo lea de un archivo externo al sistema. Para eso, no nos sirve el analizador sintáctico mencionado con anterioridad. Necesitamos poder crear, leer y guardar un archivo de gramática, y saber en todo momento cuáles son las clases sintácticas que contiene y sus correspondientes reglas asociadas; es más, agregamos a nuestra lista de deseos que el usuario pueda incrustar comentarios en su archivo. Para eso necesitamos otro analizador sintáctico que nos permita manipular archivos de gramática.

Este segundo Analizador Sintáctico debe poder leer un conjunto de clases sintácticas y sus correspondientes reglas de producción y generar una lista de clases sintácticas.

Continuando con nuestra descripción genérica, debemos especificar qué entendemos por traducir a diagramas de trenes. La respuesta obvia es: que dibuje el diagrama de tren en la pantalla; pero eso no basta, debemos darle al usuario la posibilidad de conservar el diagrama. Es decir, debemos poder guardar el diagrama, para que el usuario lo use como desee posteriormente. Hay varias opciones, pero la que implementamos es la más sencilla y adecuada; requiere de un esfuerzo mínimo de programación y proporciona una calidad de imagen aceptable que es compatible con la mayoría de las aplicaciones gráficas y de diseño de Windows. Nuestra solución es guardar la imagen del diagrama en formato de Windows Metafile.

Ya tenemos los requerimientos básicos del sistema, a los cuales agregados los siguientes, ayudan a que el sistema sea más útil y amigable para el usuario. Los listamos a continuación.

1. Que se pueda elegir de una lista de clases sintácticas - que debemos construir al hacer el reconocimiento- la que se desea utilizar.
2. Que la lista de clases sintácticas se pueda ordenar alfabéticamente (obviamente al ordenar se pierde el orden inicial).
3. Que el tamaño de las ventanas donde se muestran los principales resultados se pueda

modificar.

4. Que el árbol sintáctico generado por el reconocimiento esté visible a disposición del usuario.
5. Que la ventana de edición de las reglas de producción señale con diferentes colores los metasímbolos de la gramática así como los símbolos terminales y los comentarios.
6. Que se puedan configurar las principales parámetros gráficos de la aplicación como son:
   - Estilo, tamaño y color de la fuente utilizada para desplegar el texto en los diagramas de trenes.
   - Estilo, tamaño y color de las líneas con que se dibujan las figura de los diagramas.
   - Estilo, tamaño y color de la fuente utilizada para desplegar el texto en editor de las reglas de producción.
   - Proporcionar pistas al usuario, sobre el uso de los controles, cuando pasa el mouse sobre ellos.
   - Implementar combinaciones de teclas de atajo para las funciones principales del sistema.

Podemos resumir todos estos requerimientos conceptualmente en la Figura 2.1.1 que se muestra en la siguiente página. Como podemos ver necesitamos:

- Una interfaz gráfica para que el usuario interactúe con el sistema. Parte de la interfaz se describe en la Figura 2.1.1.
- Un analizador sintáctico y léxico para manejar los archivos de gramática y que construya la lista de clases sintácticas y sus correspondientes reglas asociadas a partir del archivo leído.
- Un analizador sintáctico y léxico para reconocer una clase sintáctica y su regla de producción asociada, y que construya el árbol sintáctico y la tabla de símbolos correspondiente.
- Un generador de diagramas que dibuje los diagramas de trenes a partir del árbol sintáctico y la tabla de símbolos generados.

Dado que Delphi es un ambiente de programación orientado a objetos, toda la programación tiene este enfoque, por lo que los analizadores, el generador y toda la interfaz giran alrededor de ejemplares de objetos de clases que ya existen y que vamos a utilizar (como es el caso de todos los controles de la interfaz gráfica) o que vamos a crear como son las clases ***tAnalizadorLexico***, ***tAnalizadorSintactico*** y la que determinemos para dibujar los diagramas a la que por el momento llamamos ***tGenerador***.

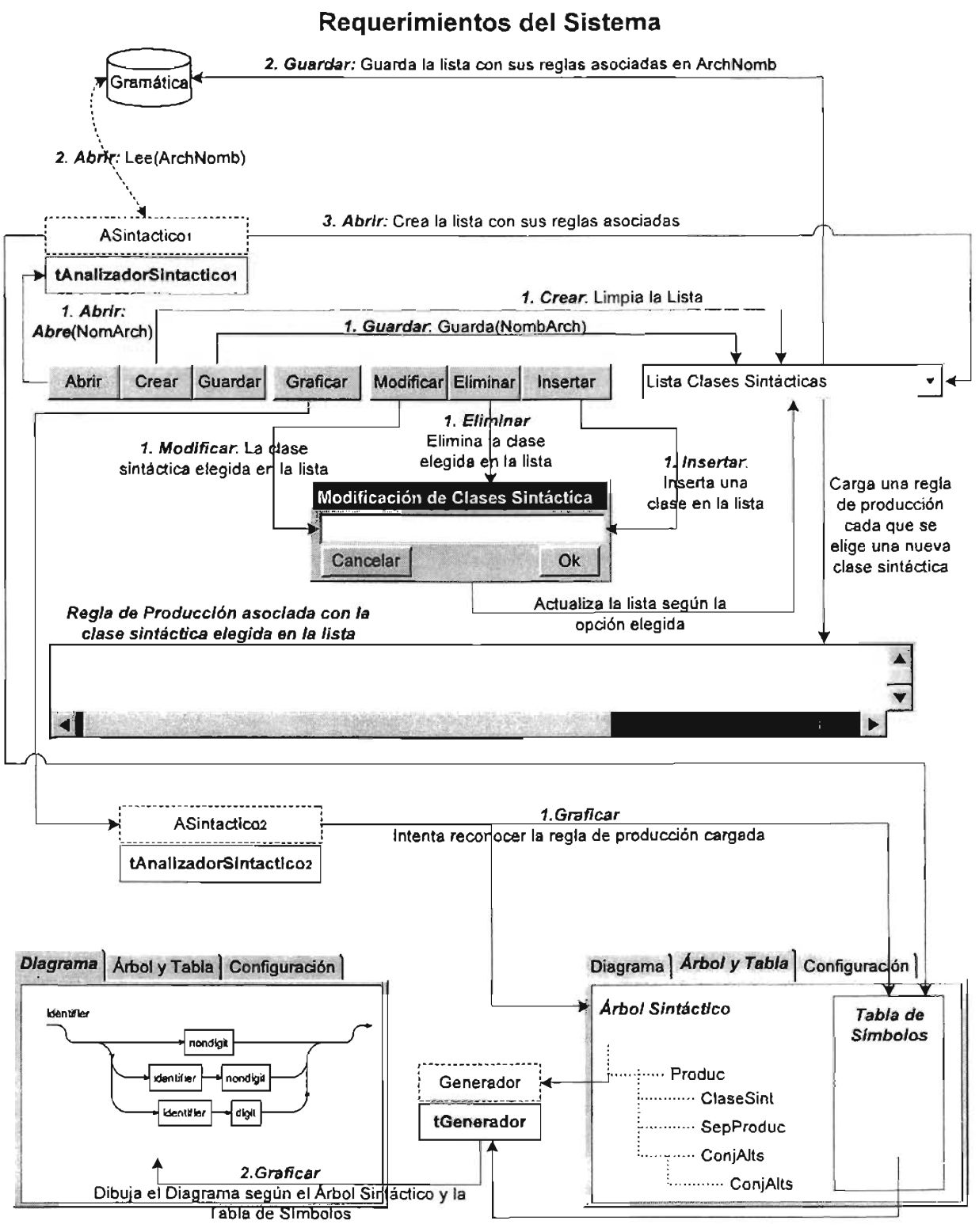

Requerimientos del Sistema
Gramática
2. Guardar: Guarda la lista con sus reglas asociadas en ArchNomb
2. Abrir: Lee(ArchNomb)
ASintactico1
tAnalizadorSintactico1
3. Abrir: Crea la lista con sus reglas asociadas
1. Crear: Limpia la Lista
1. Abrir: Abre(NomArch)
1. Guardar: Guarda(NombArch)
Abrir
Crear
Guardar
Graficar
Modificar
Eliminar
Insertar
Lista Clases Sintácticas
1. Modificar: La clase sintáctica elegida en la lista
1. Eliminar Elimina la clase elegida en la lista
1. Insertar: Inserta una clase en la lista
Carga una regla de producción cada que se elige una nueva clase sintáctica
Modificación de Clases Sintáctica
Cancelar
Ok
Actualiza la lista según la opción elegida
Regla de Producción asociada con la clase sintáctica elegida en la lista
ASintactico2
tAnalizadorSintactico2
1.Graficar
Intenta reconocer la regla de producción cargada
Diagrama
Árbol y Tabla
Configuración
identifier
nondigit
identifier
nondigit
identifier
digit
2.Graficar
Dibuja el Diagrama según el Árbol Sintáctico y la Tabla de Símbolos
Generador
tGenerador
Diagrama
Árbol y Tabla
Configuración
Árbol Sintáctico
Tabla de Símbolos
Produc
ClaseSint
SepProduc
ConjAlts
ConjAlts

FIGURA 2.3.1

2.4 Gramática EBNF propuesta

Vamos a manejar dos definiciones de nuestra gramática; una que nos va a permitir leer un archivo de reglas de producción de una gramática descrita en el EBNF propuesto, a la cual denotamos por G_1, y otra para generar el diagrama de tren correspondiente a una clase sintáctica elegida por el usuario, la cual denotamos por G_2. Ambas gramáticas definen el EBNF propuesto; la primera la definimos así:

$G_1=(V_C, V_T, S, P)$, donde

V_C = { Gramática, ReglaProduccion },

V_T = { "|", ".", **FinArchivo**, "(", ")", "[", "]", "{", "}", **CaracterIgnorado**, **Comentario**, **SimboloTerminal**, **ClaseSintactica**, **MetaSimboloParteDerecha**},

S = Gramatica,

y P contiene las siguientes reglas:

1.1.1 Gramatica = { ReglaProduccion} **FinArchivo** .

1.1.2 ReglaProduccion = { **CaracterIgnorado** }
ClaseSintactica
{ **CaracterIgnorado** }
"="
{ **Comentario** |
CaracterIgnorado |
SimboloTerminal |
ClaseSintactica |
MetaSimboloParteDerecha }
"."

Describimos a mayor detalle los símbolos terminales:

1.2.1 ClaseSintactica = CaracterIdentificador { CaracterIdentificador} .

1.2.2 SimboloTerminal = ("“" | "‘") CadenaSinFinArchivo ("“" | "‘") .

1.2.3 Comentario = ("//" CadenaSinFinArchivoNiSaltoLinea **SaltoLinea**) |

("/*" CadenaSinFinArchivo "*/") .

1.2.4. MetaSimboloParteDer = "(" | ")" | "[" | "]" | "{" | "}" .

1.2.5 CaracterIdentificador = **CaracterAlfabetico** | **CaracterNumerico** | "-" | "_" .

1.2.6 CadenaSinFinArchivo = {**Cualquier car. excepto FinArchivo**} .

1.2.7 CadenaSinFinArchivoNiSaltoLinea = {**Cualquier car. excepto FinArchivo y SaltoLinea**} .

Nuestra segunda gramática es : $G_2=(V_C , V_T , S, P)$, donde

V_C = { Produccion, ConjuntoAlternativasSimbolos, SucesionSimbolos, Simbolo},

V_T = {"|", ".", "(", ")", "[", "]", "{", "}",**CaracterIgnorado, Comentario, SimboloTerminal, ClaseSintactica**},

S = Produccion,

y P contiene las siguientes reglas:

2.1.1 Produccion = **ClaseSintactica** "=" [ConjuntoAlternativasSimbolos] "." .

2.1.2 ConjuntoAlternativasSimbolos = SucesionSimbolos { '|' SucesionSimbolos } .

2.1.3 SucesionSimbolos = Simbolo { Simbolo } .

2.1.4 Simbolo = '#' |
SimboloTerminal |
ClaseSintactica |
'(' ConjuntoAlternativasSimbolos ')' |
'[' ConjuntoAlternativasSimbolos ']' |
'{' ConjuntoAlternativasSimbolos '}' .

Los símbolos terminales son los mismos que para G_1.

Ambas gramáticas son tan pequeñas que no es necesario comprobar si son gramáticas limpias; por lo tanto pasamos directamente a verificar si se trata de gramáticas LL(1), para lo cual deben cumplir con las cuatro condiciones de Knuth. Antes de verificar las condiciones vamos a calcular los conjuntos ***CST Iniciales*** y ***CST Siguientes*** para ambas gramáticas, los cuales además se utilizan en la parte del análisis sintáctico; asimismo por conveniencia y economía vamos a reescribir sus reglas de producción como:

G_1:

Grm = { **RgP** } **FnA** .

RgP = { **Cig** } **CLs** { **Cig** } "=" { **Com** | **Cig** | **Trm** | **CLs** | **MsD**} ":" .

G_2:

Prd = **CLs** "=" [CAS] "." .

CAS = **SuS** { '|' **SuS** } .

SuS = **Smb** { **Smb**} .

Smb = **Trm** | **CLs** | '(' **CAS** ')' | '[' **CAS** ']' | '{' **CAS** '}' .

Cálculo del conjunto de símbolos iniciales para G1:

X	→	α	*CSTi(α)*	*Agregar CSTi(α) a CSTs(X)*	**Observaciones**
RgP		{Cig} CLs {Cig} "=" { Com | Clg | Trm | CLs | MSD} ":"	CLs {Cig} "=" { Com | Clg | Trm | CLs | MSD} ":"	{CLs}	1) Por {Cig}, suponemos que no aparece
			Cig Cls {Clg} "=" { Com | Clg | Trm | Cls | MSD} ":"	{CLs, Cig}	2) Por {Cig}, suponemos que aparece 1 vez o más
Grm		{RgP}FnA	FnA	{FnA}	1) Por {RgP} suponemos que no aparece
			RgP	{FnA, CLs, Cig}	2) Por {RgP}, suponemos que aparece 1 vez o más

Cálculo del conjunto de símbolos siguientes para G1:

Y	→	α	X	β	CSTi(β)	Agregar a CSTs(X)	Observaciones
Gm		...	RgP	FnA	FnA	{FnA}	1) Por {RgP}, suponemos que aparece solo 1 vez; y ε ∉ CSTi(FnA)
			RgP	RgP	RgP	{FnA, CLs, Cig}	2) Por {RgP} suponemos que aparece 2 o más veces; y ε ∉ CSTi(RgP)
			Grm			{FnA}	Por ser el símbolo inicial.

Cálculo del conjunto de símbolos iniciales para G2:

X	→	α	CSTi(α)	Agregar CSTi(α) a CSTs(X)	Observaciones
Smb		Trm | CLs | '(' CAS ')' | '[' CAS ']' | '{' CAS '}' .	{Trm} U {CLs} U {'('} U {'['} U {'{'} .	{Trm, CLs '(', '[', '{'}	Es la unión de todas las opciones.
SuS		Smb { Smb}.	{Trm, CLs '(', '[', '{' }	{Trm, CLs '(', '[', '{' }	
CAS		SuS { '|' SuS } .	{Trm, CLs '(', '[', '{' }	{Trm, CLs '(', '[', '{' }	
Prd		CLs "=" CAS "." .	{CLs}	{CLs}	

Cálculo del conjunto de símbolos siguientes para G2:

Y	→	α	X	β	¿CSTi(β) contiene ε?	Agregar a CSTs(X)	Observaciones
Sus		...	Smb.	ε	Si	{}.	1) Por {Smb}, suponemos que no aparece, entonces agregamos CSTi(ε)-{ε} y CSTs(Sus). Todavía desconocemos CSTs(Sus).
			Smb	Smb ...	Si, porque CSTi(smb) = {Trm, CLs '(', '[', '{' }	Agregamos {Trm, CLs '(', '[', '{'}	2) Por {Smb} suponemos que aparece por lo menos 1 vez. Sigue pendiente por agregar CSTs(Sus).
CAS		ε	SuS	ε	Si	{}.	1) Por {|SuS}, suponemos que no aparece, entonces agregamos CSTi(ε)-{ε} y CSTs(CAS). Todavía desconocemos CSTs(CAS).
		ε	SuS	'|' SuS ...	No, porque es igual a '|'	{ '|' }	2)

Prd		CLs"="	CAS	'.'	No, porque es igual a '.'	{'.'}	
Smb		'('	CAS	')'	No, es igual a ')'	{'.', ")"}	
		'['	CAS	']'	No, es igual a ']'	{'.', ")", "]"}	
		'{'	CAS	'}'	No, es igual a '}'	{'.', ")", "]", "}"}	
			Prd			{FnA}	Por ser el símbolo inicial
			SuS			{'|', '.', ")", "]", "}"}	Lo teníamos pendiente
			Smb			{Trm, CLs '(', '[', '{', '|', '.', ")", "]", "}"}	Lo teníamos pendiente

Resumiendo tenemos :

		Símbolos	
		Iniciales	Siguientes
	Grm	{FnA, CLs, Cig}	{FnA}
	Rgp	{CLs, Cig}	{FnA, CLs, Cig}
	Prd	{CLs}	{FnA}
	CAS	{Trm, CLs '(', '[', '{' }	{ '.' , ")" , "]" , "}" }
	SuS	{Trm, CLs '(', '[', '{' }	{ '|' , '.' , ")" , "]" , "}" }
	Smb	{Trm, CLs '(', '[', '{' }	{Trm, CLs '(', '[', '{', '|' , '.',")", "]", "}"}

Ahora vamos a probar que ambas gramáticas cumplen las cuatro condiciones de Knuth para ser consideradas como gramáticas LL(1).

Primera condición de Knuth

No permite producciones de la forma $A \rightarrow A\alpha$; ambas gramáticas cumplen con esta condición; es decir no tenemos recursividad por la izquierda.

Segunda condición de Knuth

Los símbolos terminales iniciales de las distintas alternativas de una regla de producción deben formar conjuntos disjuntos. Es decir, si

$$A \rightarrow B\beta \mid C\delta \qquad A,B,C \in V_C \quad \beta, \delta \in V_G^*$$

entonces CSTi(B) $\cap$ CSTi(C) = ϕ.

En G_1 tenemos:

1. **Gramatica** = { **ReglaProduccion**} **FinArchivo** .

   es decir,

   Gramatica = **ReglaProduccion** **FinArchivo** | **FinArchivo**.

   y

   CSTi(**ReglaProduccion**) ∩ CSTi(**FinArchivo**)={ **CLs, Cig** } ∩ { **FinArchivo** } =∅ .

2. **RgP** ={ **Cig** } **CLs** . . .

   es decir,

   RgP = **Cig** **CLs** . . . | **CLs** . . . y CSTi(**Cig**) ∩ CSTi(**CLs**)={**Cig**}∩{**CLs**}=∅ .

En G_2 tenemos:

1. **Smb** = **Trm** | **CLs** | '(' **CAS** ')' | '[' **CAS** ']' | '{' **CAS** '}' | ε .

   y ,

   CSTi(**Trm**) ∩ CSTi(**CLs**) ∩CSTi('(')∩ CSTi('[')∩ CSTi('{')=∅ .

Tercera condición de Knuth

Si una alternativa de una producción de una clase sintáctica origina la cadena vacía, los símbolos terminales que pueden seguir a dicha clase sintáctica en la producción donde aparezca, han de formar un conjunto disjunto con los terminales que pueden encabezar las distintas alternativas de dicho terminal, es decir:

(***CST Iniciales***(A) ∩ ***CST Siguientes***(A) = ∅), donde CST son las siglas para Conjunto de Símbolos Iniciales.

No tenemos ningún caso en G1 ni en G2.

Cuarta condición de Knuth

Ninguna clase sintáctica puede tener dos o más alternativas que conduzcan a la cadena vacía. No tenemos ninguna clase sintáctica de ese tipo. ***Por lo tanto concluimos que ambas gramáticas son LL(1).***

Una vez que hemos comprobado que nuestra gramática puede ser considerada como LL(1) podemos empezar su implementación.

Manual del programador

Capítulo 3

3.1 Implementación del analizador léxico

Recordemos que el análisis sintáctico se divide en dos partes, una de las cuales tratamos aqui. El analizador léxico, tiene como función primordial proporcionar al analizador sintáctico el siguiente símbolo terminal reconocido en la cadena de entrada. En nuestro caso necesitamos construir 2 analizadores léxicos. Pero antes recordemos que el ***Análisis léxico***, se relaciona con un autómata finito determinista basado en una gramática regular.

Típicamente se definen a los elementos de V_T en términos de expresiones regulares; se convierten estas expresiones regulares en un autómata finito no determinista; éste se convierte en un autómata finito determinista; se minimiza su número de estados para finalmente implementarlo. Nuestra gramática es muy pequeña por lo que no tiene caso complicar tanto el proceso. Vamos a proceder de la siguiente manera: en nuestra definición de la gramática los elementos correspondientes a V_T ya se describen como expresiones regulares; por lo tanto a continuación describimos el autómata finito determinista que representan para directamente pasar a su implementación.

Si tomamos de la sección anterior la parte que corresponde al análisis léxico para G_1, tenemos:

V_T = { "|", ".", **FinArchivo**, "(", ")", "[", "]", "{", "}", **CaracterIgnorado, Comentario, SimboloTerminal, ClaseSintactica, MetaSimboloParteDerecha**}.

1.2.1 ClaseSintactica = CaracterIdentificador { CaracterIdentificador} .

1.2.2 SimboloTerminal = ("'" | "‘") CadenaSinFinArchivo ("'" | "‘") .

1.2.3 Comentario = ("//" CadenaSinFinArchivoNiSaltoLinea SaltoLinea) |
("/*" CadenaSinFinArchivo "*/") .

1.2.4.MetaSimboloParteDer = "(" | ")" | "|" | "[" | "{" | "}" .

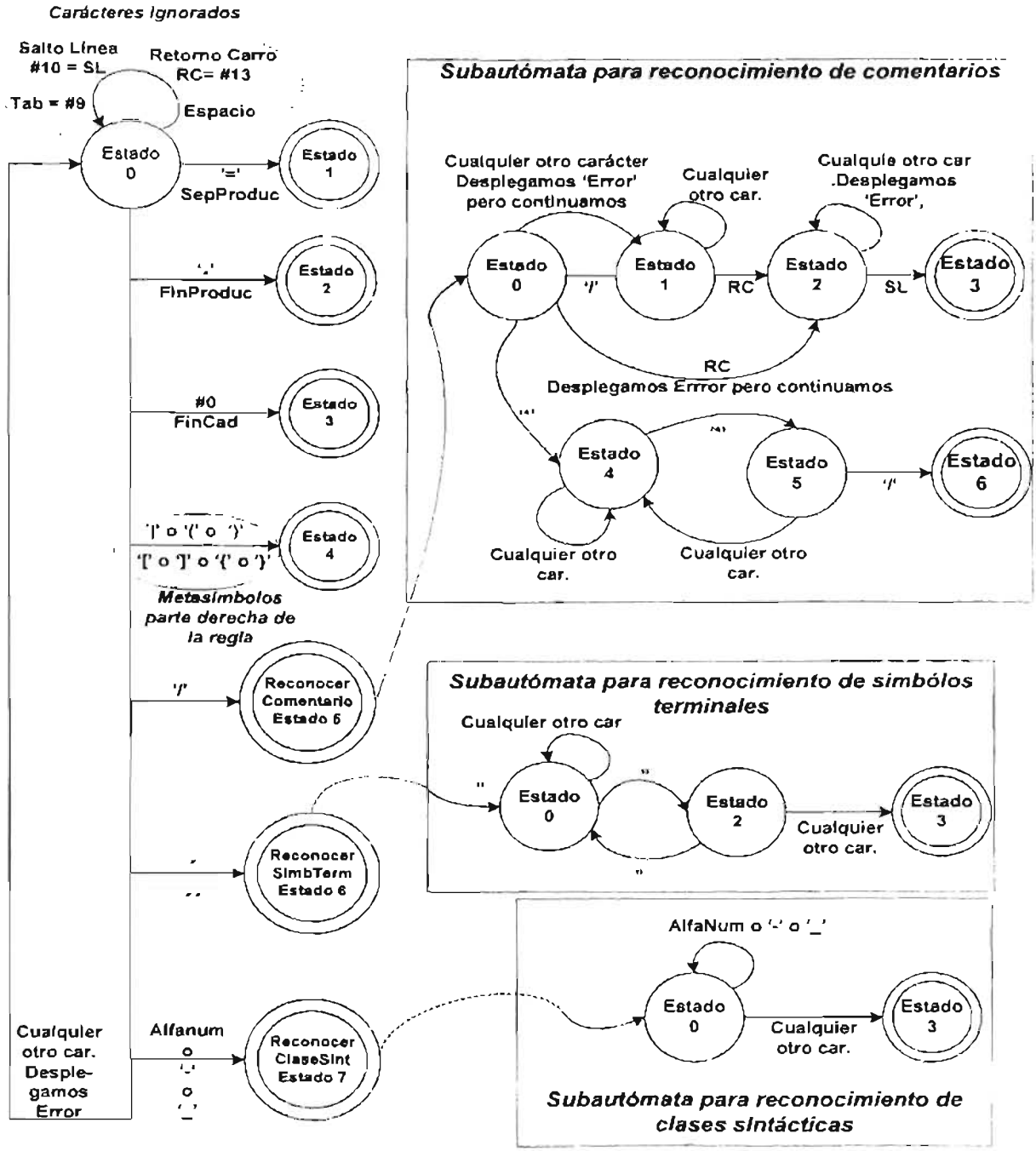
1.2.5 CaracterIdentificador = **CaracterAlfabetico** | **CaracterNumerico** | "-" | "_" .

1.2.6 CadenaSinFinArchivo = {**Cualquier car. excepto FinCadena**}.

1.2.7 CadenaSinFinArchivoNiSaltoLinea = {**Cualquier car. excepto FinCadena y SaltoLinea**}.

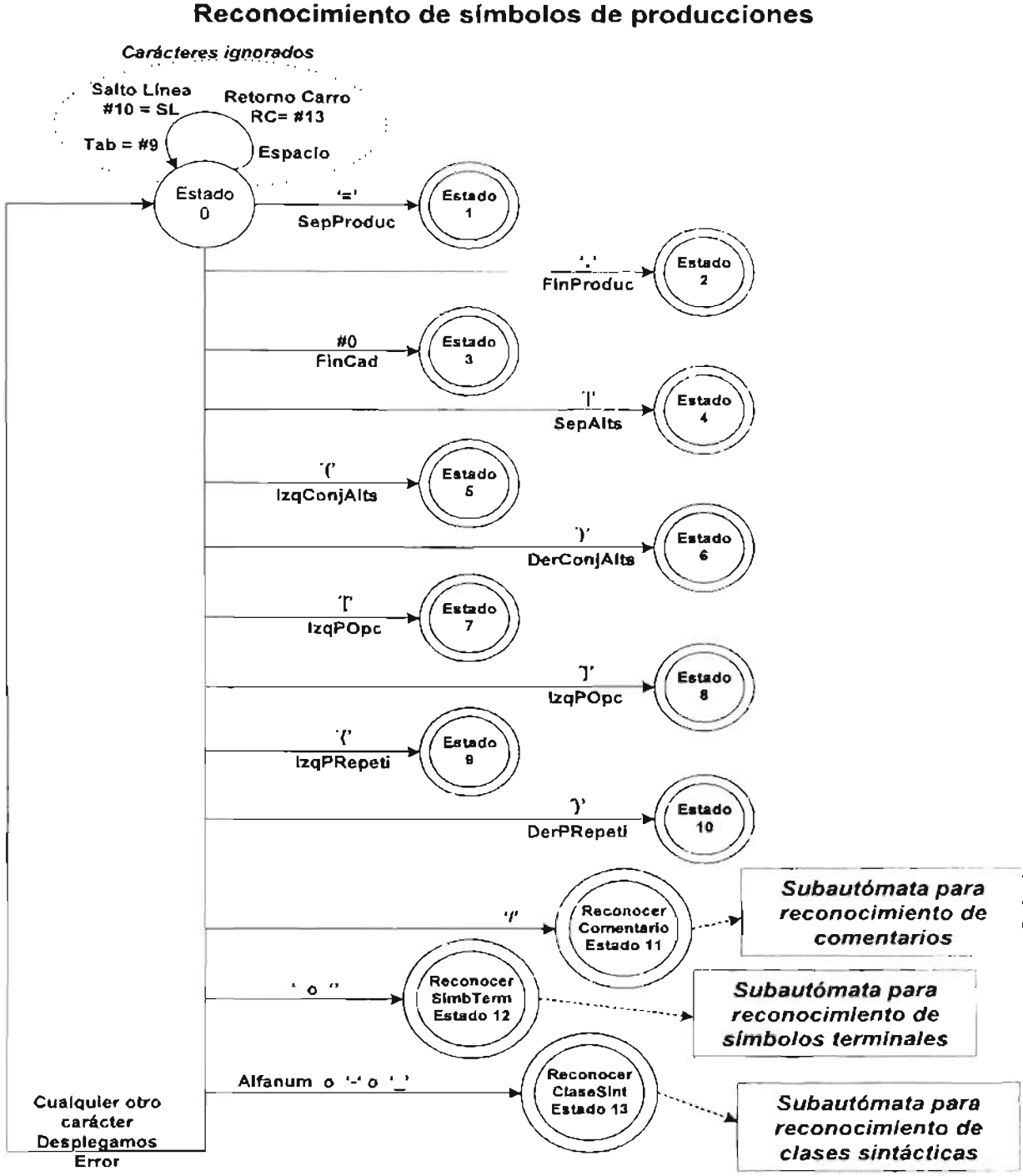
Vemos que podemos representar a nuestro primer analizador léxico de la siguiente forma:

Reconocimiento de símbolos de archivos de gramática



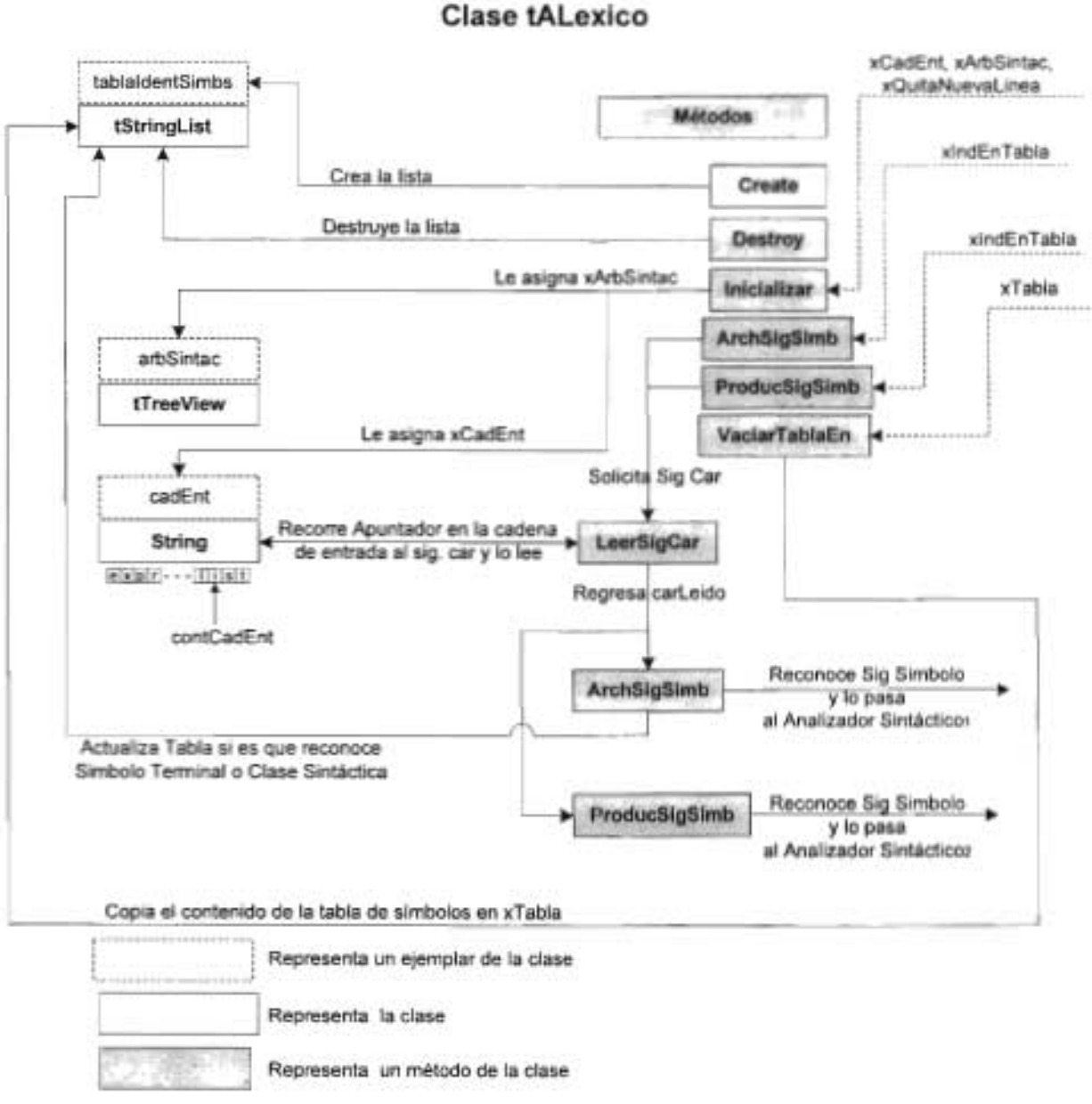
Hacemos lo mismo para G_2 y tenemos:

$V_T = \{$"|", ".", "(", ")", "[", "]", "{", "}", **CaracterIgnorado, Comentario, SimboloTerminal, ClaseSintactica**$\}$



Nótese que los elementos **CaracterIgnorado, Comentario, SimboloTerminal**, de G_1 y G_2 coinciden, por eso sólo los describimos en la parte correspondiente a G_1. Ya tenemos todo para realizar la implementación.

Implementamos a nuestros analizadores a través de una clase con las siguientes características:



La interfaz de nuestra clase en Delphi queda así:

```
tALexico = CLASS(tObject)
  protected
  //Variables relacionadas con el comportamiento del autómata:
    Mensajes          :tMemo;
    Edo               :byte;    //Estado en que se encuentra el reconocimiento.
    yaLeiSig          :boolean; //Indica si leímos el siguiente carácter.
    carLeido          :char;    //Contiene el carácter recién leído.
    cadAsocAlSimb     :string;  //Cadena que vamos construyendo al reconocer.
    cadEnt            :string;  //Cadena que deseamos reconocer.
    contCadEnt        :integer; //Número de caracteres leídos de la cad. ent.
    contRengCadEnt    :integer; //No de renglón en el que estamos leyendo
    contColCadEnt     :integer; //No de columna en la que estamos leyendo
    nombSimbEnt       :string;  //Nombre del símbolo leído en la cadena de ent.

  //Tabla y Contadores de Identificadores de Símbolos
    tablaIdentSimbs   :tStringList;
    contsTipoSimb     :array [1..23] of byte;
    indEnTabla        :integer;          //Índice en la tabla

  // MÉTODOS
    procedure LeerSigCar;
    procedure ErrorLexico(const xEdo: byte; xTipoSimb:tTipoSimb;
                          xEsperado : string);
    procedure InsertarEnTabla(const xCadAsocSimb:string;

var xindEnTabla:integer);
    function FinCadEnt : boolean;
    function ReconocerComentario :tTipoSimb;
    function ReconocerSimbTerm(xSepExtSimb : char):tTipoSimb;
    function ReconocerClaseSint:tTipoSimb;
    function NombTipoSimb(xTipoSimb:tTipoSimb):string;
    function LeerSimbTermTabla(const xindEnTabla:integer) :string;
  public
    constructor Create(const xMensajes:tMemo);
    destructor Destroy; override;
    procedure Inicializar(const xCadEnt:string; xQuitaNuevaLinea:Boolean);
    procedure VaciarTabla;
    function ArchSigSimb(var xindEnTabla:integer)     :tTipoSimb;
    function ProducSigSimb(var xindEnTabla:integer)   :tTipoSimb;
end;
```

El método **ArchSigSimb** corresponde al autómata de la figura 2.3.1, mientras que el

método **ProducSigSimb** corresponde al autómata de la figura 2.3.2; son los métodos que caracterizan el comportamiento de esta clase ya que alimentan al analizador sintáctico con el siguiente símbolo que éste tiene que reconocer. Estos métodos se declaran en la sección pública de la clase porque están disponibles para que los usuarios de la misma puedan solicitar a la clase que realice el trabajo que sabe hacer. Tenemos adicionalmente en esta sección obviamente el método **create**, para poder crear instancias de esta clase y dos métodos adicionales que son **Inicializar** y **VaciarEnTabla**. Antes de pasar a describir a detalle la clase necesitamos algunas definiciones más como son:

```
type
tTipoSimb = (   tsSepProduc=1,      tsSepAlts=2,        tsIzqConjAlts=3,
                tsDerConjAlts=4,    tsIzqPOpc=5,        tsDerPOpc=6,
                tsIzqPRepeti=7,     tsDerPRepeti=8,     tsFinProduc=9,
                tsFinCadena=10,     tsComentario=11,    tsIgnorados=12,
                tsSimbTerm=13,      tsClaseSint=14,     tsGramat=15,
                tsReglaProduc=16,   tsPDerRegla=17,     tsMetaSimbPDer=18,
                tsProduc=19,        tsConjAltsSimbs=20, tsSuceSimbs=21,
                tsSimb=22,          tsSimbInvalido=23,  tsSaltoDiagr=24);
```

tTipoSimb es un tipo enumerado, que define un conjunto ordenado de valores; esto nos permite definir y manejar variables del tipo **tTipoSimb** en donde cada elemento tiene asociado un identificador y un orden dentro del conjunto. Es por eso que los métodos **ArchSigSimb** y **ProducSigSimb** regresan variables de ese tipo.

Ya tenemos manera de saber el tipo de símbolo, además necesitamos poder asociarle un nombre no solo un orden o índice dentro del conjunto, sería incómodo para el usuario del sistema que cuando reportáramos algún problema con el reconocimiento de un tipo de símbolo lo hiciéramos con el índice dentro de este conjunto; por tal motivo declaramos también:

```
const
  nombsTipoSimb : array [1..23] of string =
    ( cSepProduc,                cSepAlts ,
      cIzqConjAlts,              cDerConjAlts,
      cIzqPOpc,                  cDerPOpc,
      cIzqPRepeti,               cDerPRepeti,
      cFinProduc,                '#0',
      cExtComentario+cIntComentario, 'Ignorados',
      'SimbTerm',                'ClaseSint',
      'Gram',                    'Regl',
      'DerReg',                  'MsPd',
      'Prod',                    'CnjA',
```

```
'SucSimbs',          'Simb',
'Error ***',         'SaltoDiagr');
```

Si se pregunta qué son **cIdentificador**, **cSepAlts**, etc., son las últimas constantes que nos restan por definir y son:

```
const
cSepProduc='=';          cSepAlts='|';
cIzqConjAlts='(';        cDerConjAlts=')';
cIzqPOpc='[';            cDerPOpc=']';
cIzqPRepeti='{';         cDerPRepeti='}';
cFinProduc='.';          cFinCadena=#0;
cSaltoDiagr='#';

cExtComentario = '/';    cIntComentario = '*';
cExtSimbTerm1 = '"';     cExtSimbTerm2 = '"';
cNumerico = ['0'..'9'];  cAlfabetico = ['A'..'Z', 'a'..'z'];
cAlfaNumerico = cAlfabetico + cNumerico;
cIdentificador = cAlfabetico + cNumerico + ['-'] + ['_'];
cTab        = #09;       cSaltoLinea  = #10;
cRetornoCarro = #13;     cEspacio     = #32;
cNuevaLinea = #13#10;
cIgnorados = [cTab, cSaltoLinea, cRetornoCarro, cEspacio];
cMetaSimbPDer = [  cSepAlts,
                   cIzqConjAlts,   cDerConjAlts,
                   cIzqPOpc,       cDerPOpc,
                   cIzqPRepeti,    cDerPRepeti,
                   cSaltoDiagr];
```

Como podemos ver estas constantes tienen asociados los carácteres y conjuntos de carácteres que nuestro analizador léxico intenta reconocer y además los nombres de estas constantes coinciden con el tipo de símbolo que representan cuando el tipo de símbolo representa un solo carácter. Todas estas constantes no se encuentran en la misma unidad en donde se encuentra la clase **tLexico**; decidimos ubicarlas en una unidad llamada **ConstipU.pas**, a la cual pueden hacer referencia todas las unidades que utilizan estas constantes o variables que va a contener esta unidad.

Ahora si vamos a ver cómo implementamos nuestro Analizador Léxico. Lo primero es poder construir y destruir ejemplares de esta clase para lo cual tenemos los métodos:

```
constructor tALexico.Create(const xMensajes:tMemo);
```

```
begin
  inherited Create;
  Mensajes:=xMensajes;
  tablaIdentSimbs := tStringList.Create;
  //tablaIdentSimbs.Sorted:=true;
  tablaIdentSimbs.CaseSensitive:=true;
end;

destructor tALexico.Destroy;
begin
      tablaIdentSimbs.free;
      inherited;
end;
```

El método **create** no hace más que generar la tabla de identificadores de símbolos cuando se crea un ejemplar de esta clase y asignar el control visible en la aplicación pasado aquí a través del parámetro *xMensajes* a **Mensjaes** para en ese control desplegar todos los mensajes de error y la tabla de símbolos. El método **destroy** destruye el ejemplar creado para la tablas de identificadores.

El usuario de esta clase después de crear un ejemplar de la misma y antes de utilizar los métodos que le permiten obtener de ella el siguiente símbolo en la cadena de entrada, debe inicializar la clase. Nótese que aunque tenemos dos gramáticas que deseamos reconocer estamos implementando sólo una clase con dos métodos diferentes; como vamos a ver la combinación **Inicializar** con alguno de los dos métodos mencionados nos va a permitir reconocer una u otra gramática.

El método **Incializar** es el siguiente:

```
{ Limpia la cadena de entrada, asigna la nueva cadena de entrada coloca al
* apuntador de caracteres al inicio de la cadena de entrada y actualiza la
* bandera que nos indica si ya hemos leído el siguiente carácter:        }
procedure tALexico.Inicializar(const xCadEnt:string; xarbSintac:tTreeView;
            xQuitaNuevaLinea:Boolean);
begin
  // Asignamos la cadena de entrada por reconocer.
      if xQuitaNuevaLinea then
      cadEnt:=CopyRange(xCadEnt, 1, length(xCadEnt)-2)+cFinCadena
```

```
        else
        cadEnt:=xcadEnt+cFinCadena;

    // Colocamos al apuntador al inicio de la cadena.
        contCadEnt:=1;
        contRengCadEnt:=1;
        contColCadEnt:=1;;

        yaLeiSig:=false;
    // Limpiamos la tabla de símbolos.
        tablaIdentSimbs.clear;
end;
```

Inicializar realiza tareas muy importantes como son asignar la nueva cadena de entrada, inicializar los apuntador/contadores de la cadena de entrada, y borrar la tabla de identificadores y el árbol sintáctico. Tener la opción de especificar la cadena de entrada nos permiten, combinado con el método de reconocimiento que elijamos manejar 2 gramáticas para un mismo analizador.

Una vez inicializada la clase puede iniciar el reconocimiento, implementamos primero el reconocimiento de las gramáticas G_1 y G_2.

```
function tALexico.ArchSigSimb(var xindEnTabla:integer):tTipoSimb;
begin
  cadAsocAlSimb:='';
  result:=tsSimbInvalido;
  while (result=tsSimbInvalido) and (not FinCadEnt) do
  begin
    if (not yaLeiSig) then LeerSigCar;
    yaLeiSig:=false;
    case carLeido of
     // Caracteres que provocan aceptación.
      cSepProduc:result:=tsSepProduc;
      cFinProduc:result:=tsFinProduc;
      cFinCadena:result:=tsFinCadena;
      cExtComentario:result:=ReconocerComentario;
      cExtSimbTerm1:result:=ReconocerSimbTerm(cExtSimbTerm1,xIndEnTabla);
      cExtSimbTerm2:result:=ReconocerSimbTerm(cExtSimbTerm2,xIndEnTabla);
      else
        if carLeido in cIdentificador then
          result:=ReconocerClaseSint(xIndEnTabla)
        else begin
```

```
          if carLeido in cMetaSimbPDer then
            result:=tsMetaSimbPDer
          else begin
            if carLeido in cIgnorados then
              result:=tsIgnorados
            else
              ErrorLexico(0, tsSimbInvalido, 'Carácter válido');
          end;
        end;
    end;
  end;

  if result in [tsSepProduc, tsFinProduc, tsFinCadena, tsMetaSimbPDer,
                                          tsIgnorados] then
    cadASocAlSimb:=carLeido;
  nombSimbEnt:=NombTipoSimb(result);
end;

function tALexico.ProducSigSimb(var xindEnTabla:integer):tTipoSimb;
begin
  cadAsocAlSimb:='';
  result:=tsSimbInvalido;
  while (((result=tsSimbInvalido) or(result=tsComentario)  )and (not FinCadEnt)) do
  begin
    if (not yaLeiSig) then LeerSigCar;
    yaLeiSig:=false;
    case carLeido of
     // Carácteres que provocan aceptación.
      cSepProduc:result:=tsSepProduc;
      cFinProduc:result:=tsFinProduc;
      cFinCadena:result:=tsFinCadena;
      cSepAlts:result:=tsSepAlts;
      cIzqConjAlts:result:=tsIzqConjAlts;
      cDerConjAlts:result:=tsDerConjAlts;
      cIzqPOpc:result:=tsIzqPOpc;
      cDerPOpc:result:=tsDerPOpc;
      cIzqPRepeti:result:=tsIzqPRepeti;
      cDerPRepeti:result:=tsDerPRepeti;
      cExtComentario:result:=ReconocerComentario;
      cExtSimbTerm1:result:=ReconocerSimbTerm(cExtSimbTerm1,xIndEnTabla);
      cExtSimbTerm2:result:=ReconocerSimbTerm(cExtSimbTerm2, xIndEnTabla);
      cSaltoDiagr:result:=tsSaltoDiagr;
      else
        if carLeido in cIdentificador then
```

```
        result:=ReconocerClaseSint(xIndEnTabla)
      else begin
        if (not carLeido in cIgnorados ) then
          ErrorLexico(0, tsSimbInvalido, 'Carácter válido');
      end;
    end;
  end;
  nombSimbEnt:=NombTipoSimb(result);
end;
```

La implementación de ambos autómatas es muy similar, sólo que en la gramática G_1, los carácteres ignorados y los comentarios producen un estado final, mientras que en la gramática G_2, son realmente ignorados. Esto es así porque en la gramática G_1 nos interesa respetar los comentarios, saltos de línea, tabuladores y espacios que el usuario utiliza para comentar y formatear sus reglas de producción; mientras que en la gramática G_2, los comentarios y los carácteres ignorados no nos sirven para generar los diagramas de trenes. El manejo de errores es muy sencillo; cuando no se reconoce un carácter se despliega un mensaje de error que indica el tipo de carácter esperado y el reconocimiento continúa hasta que se puede reconocer "algo". Ambas implementaciones utilizan los mismos métodos para reconocer los comentarios, los símbolos terminales y las clases sintácticas, cuyas implementaciones mostramos a continuación.

```
function tALexico.ReconocerComentario:tTipoSimb;
begin
  result:=tsSimbInvalido;
  cadAsocAlSimb:=cExtComentario;
  Edo:=0;
  while ((result=tsSimbInvalido) and (not FinCadEnt)) do
  begin
    if (not yaLeiSig) then LeerSigCar;
    yaLeiSig:=false;
    case Edo of
      0:case carLeido of
        cExtComentario:Edo:=1;      //Comentario del tipo  "// ....
        cIntComentario:Edo:=4;      // Comentario del tipo "/* ... */"
       // Marcamos error pero suponemos que es del tipo "// ...
        cRetornoCarro:ErrorLexico(2,tsComentario,
                                  cExtComentario + cExtComentario);
       // Marcamos error pero también suponemos que es del tipo "// ...
        else ErrorLexico(1,tsComentario,cExtComentario+cIntComentario);
```

```
        end;
       //Comentario del tipo "// ....  buscamos nueva línea
        1: if carLeido = cRetornoCarro then Edo:=2;
       //Buscamos salto de línea
        2: if carLeido = cSaltoLinea
             then result:=tsComentario
             else ErrorLexico(2, tsComentario, 'Salto de Línea');
       //Es un comentario del tipo "/* ... */"
        4: if carLeido = cIntComentario then edo:= 5;
       //                        Finaliza          No Finaliza
        5: if carLeido = cExtComentario then  result:=tsComentario else edo:=4;
      end;
     // Continuamos construyendo la cadena que constituye el comentario
      cadAsocAlSimb:=cadAsocAlSimb+carLeido;
    end;
    DB_WriteStrLn(0, 'Coment ='+ cadAsocAlSimb);
  end;

  function tALexico.ReconocerSimbTerm(const xSepExtSimb : char;
                                      var xindEnTabla:integer):tTipoSimb;
  begin
    result:=tsSimbInvalido;
    cadAsocAlSimb:='';
    Edo:=0;
    while ((result=tsSimbInvalido) and (not FinCadEnt)) do
    begin
      if (not yaLeiSig) then LeerSigCar;
      yaLeiSig:=false;
      case Edo of
       // ¿Finaliza el símbolo terminal?
        0: if carLeido = xSepExtSimb
             then Edo:=1
             else cadAsocAlSimb:=cadAsocAlSimb + carLeido;
        1: if carLeido = xSepExtSimb
         // Falsa alarma
           then begin
             Edo:=0;
             cadAsocAlSimb:=cadAsocAlSimb + xSepExtSimb+ carLeido;
           end else begin
             result:=tsSimbTerm;
             yaLeiSig:=true; //Indicamos que adelantamos la lectura.
           end;
      end;
    end;
```

```
  if result=tsSimbTerm
    then InsertarEnTabla(cadAsocAlSimb, xindEnTabla);
end;


function tALexico.ReconocerClaseSint(var xindEnTabla:integer):tTipoSimb;
begin
  result:=tsSimbInvalido;
  cadAsocAlSimb:=carLeido; // Tomamos el primer carácter de la clase
  Edo:=0;
  while ((result=tsSimbInvalido) and (not FinCadEnt)) do
  begin
    if (not yaLeiSig) then LeerSigCar;
    yaLeiSig:=false;
    if not (carLeido in cIdentificador) then
    begin
      result:=tsClaseSint;
      yaLeiSig:=true; //Indicamos que adelantamos la lectura.
    end else
      cadAsocAlSimb:=cadAsocAlSimb + carLeido;
  end;
  if result=tsClaseSint then
    InsertarEnTabla(cadAsocAlSimb, xindEnTabla);
end;
```

La implementación de los Comentarios considera dos tipos de comentarios:

- Los que inician con doble diagonal ('//') y terminan con el inicio de nueva linea.
- Los que inician con '/*' y concluyen con '*/' y pueden extenderse a lo largo de varias líneas.

Los carácteres asociados con los comentarios son **cExtComentario**('/') y **cIntComentario**('*'). Una vez que se inicia el reconocimiento de un comentario se espera la finalización del mismo, esto es así para evitar estar desplegando errores a lo largo de lo que se espera es un comentario. Por eso, una vez iniciado el reconocimiento, la única manera de llegar a un estado final de reconocimiento es encontrando los carácteres que finalizan un comentario.

El reconocimiento de símbolos terminales considera dos tipos de símbolos terminales, los que se encuentran entre comilla simple (') y los que se encuentran entre comilla doble ("). Sólo se implementa un método, ya que el autómata asociado es el mismo, únicamente

varia el tipo de comilla; de ahí la necesidad de parametrizar el método indicando el tipo de comilla que inicia el reconocimiento. Su implementación es también sumamente sencilla, el reconocimiento continúa hasta que se encuentra las comillas que finalizan el símbolo terminal; pero como dentro de un símbolo terminal podemos tener doble comilla seguida como parte del mismo símbolo terminal, a diferencia del comentario, no podemos concluir que por encontrar una comilla que cierra la que originó el reconocimiento hemos llegado al final; para cerciorarnos debemos leer el siguiente carácter para asegurarnos de que no se trata de una doble comilla que es parte del símbolo terminal; por lo cual una vez que comprobamos que se trata de un carácter diferente no solo debemos dar por concluido el reconocimiento sino que debemos indicar, para la siguiente lectura, que ya leímos el siguiente carácter (lo indicamos por medio de la bandera **yaLeiSiguiente**:=true).

Por último, tenemos la implementación del autómata de reconocimiento de clases sintácticas, que es el más sencillo de todos; se detiene, es decir reconoce cuando el carácter leído no está dentro del conjunto de **cIdentificadores**. Procedemos igual que con los símbolos terminales, indicando que hemos terminado el reconocimiento y encendiendo la bandera de **yaLeiSiguiente**.

Para concluir con esta sección, incluimos la implementación del resto de los métodos de la clase, los cuales no merecen comentarios adicionales.

```
procedure tALexico.VaciarTabla;
  var i:integer;
begin
  Mensajes.lines.add('TABLA DE SÍMBOLOS');
  Mensajes.lines.add('**********************');
  for i:=0 to tablaIdentSimbs.Count-1 do
    Mensajes.lines.add(PadLeft(IntToStr(i),'0',3)+'.- '+
      tablaIdentSimbs.Strings[i]);
  Mensajes.lines.add('**********************');
end;

procedure tALexico.LeerSigCar;
begin
  carLeido:=cadEnt[contCadEnt]; // Leemos el siguiente carácter.
  inc(contCadEnt);          // Incrementamos al contador(apuntador)
  if carLeido = cSaltoLinea then begin
    inc(contRengCadEnt); // Cambiamos de renglón.
    contColCadEnt:=1;   // Nos ponemos en primera columna del nuevo renglón.
```

```
  end else
    inc(contColCadEnt);
end;


function tALexico.FinCadEnt : boolean;
begin
  if (contCadEnt> length(CadEnt)) then result:=true else result:=false;
end;


procedure tALexico.ErrorLexico(const xEdo: byte; xTipoSimb:tTipoSimb;
                               xEsperado : string);
  var msj: string;
begin
  Edo:=xEdo;
  msj:= 'ERROR * * *'+' en ' + nombsTipoSimb[ord(xTipoSimb)]+ cNuevaLinea +
  ' Renglón ' + intToStr(contRengCadEnt) +
  ' y columna ' + intToStr(contColCadEnt) + cNuevaLinea +
  'Esperaba ' + xEsperado + ' y encontré CadEnt('+ IntToStr(contCadEnt)+')='+
  carLeido + '.'+ cNuevaLinea +
  CopyRangeEx(cadEnt, contCadEnt + (contColCadEnt-1), contCadEnt-1);
  Mensajes.Lines.add(msj);
end;


{ Inserta la cadena indicada en xCadAsocSimb si es que no la encuentra en la
* tabla de símbolos.
}
procedure tALexico.InsertarEnTabla(const xCadAsocSimb:string;
                                   var xindEnTabla:integer);
begin
  // ¿El símbolo terminal no existe en la tabla?
  if not tablaIdentSimbs.Find
    (xCadAsocSimb, xindEnTabla) then
      // Lo agregamos a la tabla de símbolos.
      tablaIdentSimbs.Add(xCadAsocSimb);
end;

// Determinamos el nombre del tipo de símbolo de acuerdo a su tipo
function tALexico.NombTipoSimb(xTipoSimb:tTipoSimb):string;
begin
  result:=nombsTipoSimb[ord(xTipoSimb)];
```

```
  if xTipoSimb in[tsSimbTerm,tsClaseSint, tsMetaSimbPDer] then begin
    case xTipoSimb of
      tsSimbTerm    :result:=result+ '="'+LeerSimbTermTabla(indEnTabla)+'"';
      tsClaseSint   :result:=result+'='+LeerSimbTermTabla(indEnTabla);
      tsMetaSimbPDer :result:=result+'='+carLeido;
    end;
  end else
    //result:=result+ '='+carLeido;
    result:=result;
end;

function tALexico.LeerSimbTermTabla(const xindEnTabla:integer):string;
begin
  result:=tablaIdentSimbs.Strings[xindEnTabla];
end;
```

3.2 Implementación del analizador sintáctico

Una vez que contamos con los analizadores léxicos que nos proporcionan el siguiente símbolo a reconocer, el siguiente paso es construir los correspondientes analizadores sintácticos. Éstos están basados en la ejecución en forma recursiva, de un conjunto de procedimientos cuyas reglas de construcción dimos en la sección 1.8. Hay que recordar que nuestros analizadores son:

- Un analizador de archivos de gramática, que únicamente necesita reconocer y construir una lista de clases sintácticas y sus reglas asociadas.
- Un analizador de clases sintácticas, que nos permita dibujar el diagrama de trenes asociado a cada clase sintáctica contenida en dicha lista, a partir del árbol sintáctico generado.

Al igual que con el analizador léxico, implementamos nuestro analizador sintáctico a través de una clase que hereda de la clase **tLexico** y cuyas características mostramos en la figura 3.2.1.

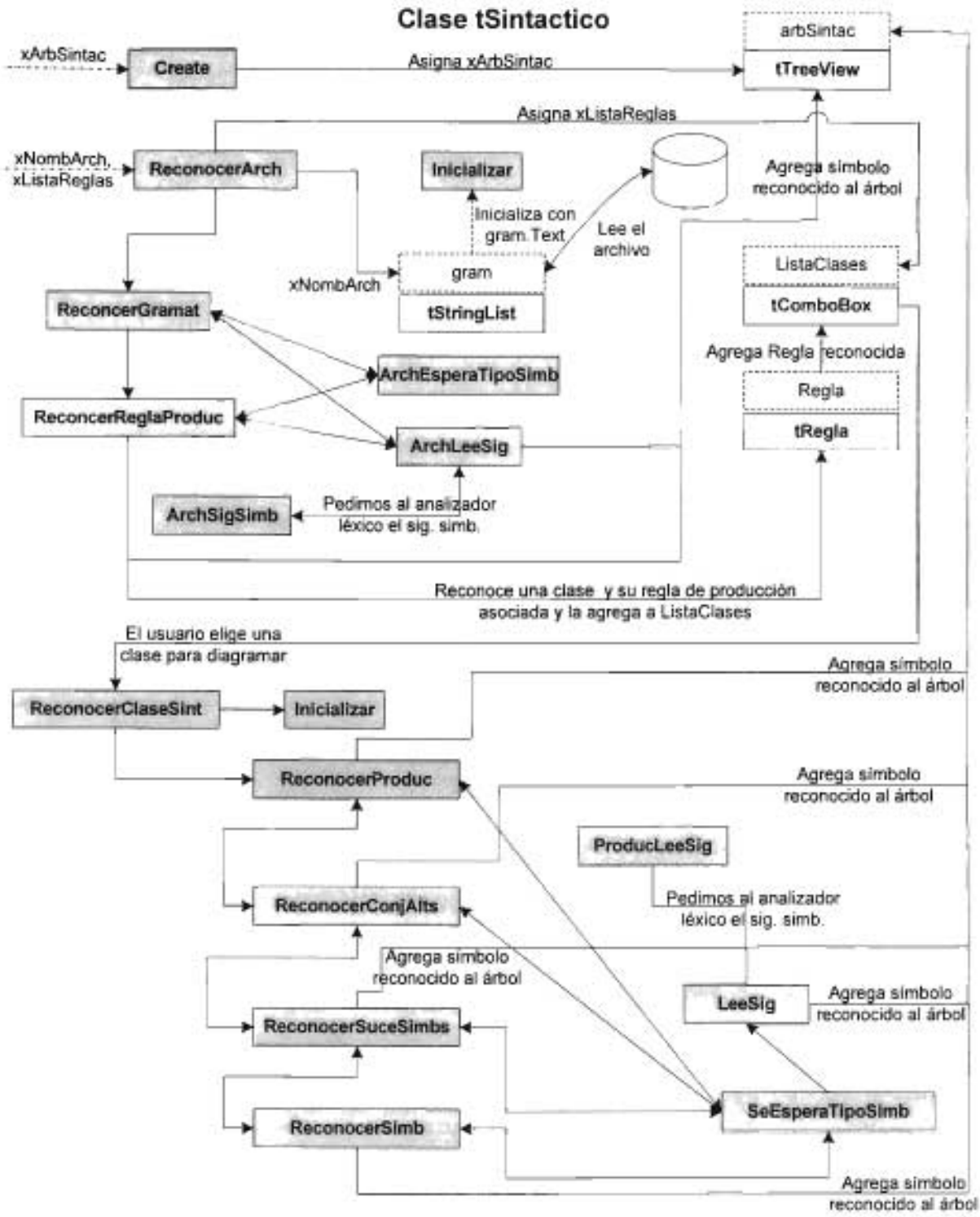

Clase tSintactico
xArbSintac
Create
Asigna xArbSintac
arbSintac
tTreeView
Asigna xListaReglas
xNombArch,
xListaReglas
ReconocerArch
Inicializar
Inicializa con
gram.Text
Lee el
archivo
Agrega símbolo
reconocido al árbol
xNombArch
gram
tStringList
ListaClases
tComboBox
ReconcerGramat
Agrega Regla reconocida
Regla
tRegla
ArchEsperaTipoSimb
ReconcerReglaProduc
ArchLeeSig
ArchSigSimb
Pedimos al analizador
léxico el sig. simb.
Reconoce una clase y su regla de producción
asociada y la agrega a ListaClases
El usuario elige una
clase para diagramar
Agrega símbolo
reconocido al árbol
ReconocerClaseSint
Inicializar
ReconocerProduc
Agrega símbolo
reconocido al árbol
ProducLeeSig
ReconocerConjAlts
Pedimos al analizador
léxico el sig. simb.
Agrega símbolo
reconocido al árbol
LeeSig
Agrega símbolo
reconocido al árbol
ReconocerSuceSimbs
SeEsperaTipoSimb
ReconocerSimb
Agrega símbolo
reconocido al árbol

FIGURA 3.2.1

Mientras que su interfaz en Delphi es:

```
TYPE
  tRegla = class (tObject)
    public
      Produccion : string;
      constructor Create(const xProd:String);
  end;
  tASintactico = CLASS(tALexico)
    private
      contGral          :integer;    // Contador general de símbolos reconocidos
      arbSintac         :tTreeView; // Componente donde creamos el árbol sintáctico
      tipoSimbCadEnt :tTipoSimb; // Tipo de símb. recién reconocido en entrada
      ListaClases    :tComboBox; // Lista de Clases Sintácticas
      pila              :tStack;   { Pila para guardar símbolos asociados con
                                     * inicios de repeticiones.}

     // Métodos del Analizador Sintáctico para G1----------------------------
      procedure DesplegarErrorSintac(const xTipoSimbEsperado:tTipoSimb;
                                     nPadre:tTreeNode);
      procedure ReconocerGramat(nPadre:tTreeNode);
      procedure ReconocerReglaProduc(nPadre:tTreeNode);
      procedure ArchEsperaTipoSimb(const xTipoSimbEsperado:tTipoSimb;
                                     nPadre:tTreeNode);
      procedure ArchLeeSig(nPadre:tTreeNode);

     // Métodos del Analizador Sintáctico para G1----------------------------
      procedure ReconocerProduc(nPadre:tTreeNode);
      procedure ReconocerConjAlts(nPadre:tTreeNode);
      procedure ReconocerSuceSimbs(nPadre:tTreeNode);
      procedure ReconocerSimb(nPadre:tTreeNode);
      procedure SeEsperaTipoSimb(const xTipoSimbEsperado:tTipoSimb;
                                     nPadre:tTreeNode);
      procedure LeeSig(const xTipoSimbEsperado:tTipoSimb;
                                     nPadre:tTreeNode);
      procedure AsocSmb(n:tTreeNode; const xTipoSimb:tTipoSimb);
    public
      constructor Create(const xArbSintac:tTreeView);
      destructor Destroy; override;
      Procedure ReconocerArch(const xNombArch:string;xListaClases:tComboBox);
      procedure ReconocerClaseSint(const  xsCadEnt:string);
  end;
  var ASintactico   : tASintactico;
```

Antes de pasar a la generación de código conviene hacer alguna aclaraciones:

- **ListaClases** es un ejemplar del tipo **tCombBox** que es un componente, con una caja de edición que tiene asociada una lista de cadenas, desplazable verticalmente. Las cadenas en este caso son las clases sintácticas, pero necesitamos también las reglas asociadas. Para tal efecto la clase **tCombBox** permite asociar ejemplares del tipo **tObject** a cada cadena; ésa es la razón de ser de la clase **tRegla**, el poder asociar a cada clase sintáctica; que es un renglón en **ListaClases,** una cadena que representa a su regla asociada. Su implementación es muy sencilla:

```
constructor tRegla.Create(const xProd:String);
begin
  inherited Create;
  Produccion :=xProd;
end;
```

- **arbSintac** es un componente visual que utilizamos para generar nuestro árbol sinttáctico. Este componente contiene nodos del tipo **tTreeNode** y a cada ejemplar del tipo tTreeNode también se le puede asociar un ejemplar del tipo **tObject**; lo cual vamos a aprovechar una vez que utilicemos el árbol sintáctico construido para generar los diagramas.
- La explicación de la función de la variable **pila** la retrasamos hasta la parte de generación de gráficas.

El primer paso es crear ejemplares de esta clase para efectuar el reconocimiento, lo cual hacemos con:

```
constructor tASintactico.Create(const xArbSintac:tTreeView);
begin
  inherited Create;
  arbSintac:=xArbSintac;
  pila:=tStack.create;
end;
```

Esta clase tiene sólo 2 métodos que permiten a los usuarios de esta clase realizar el reconocimiento; uno corresponde a G_1 y es:

```
Procedure tASintactico.ReconocerArch(const xNombArch:string;
                                     xListaClases:tComboBox);
  var  gram:tStringList;
```

```
      nSintact:tTreeNode;
begin
  ListaClases:=xListaClases;// Indicamos cuál es la lista de clases

  gram:=tStringList.create;// Creamos una lista de cadenas
    gram.LoadFromFile(xNombArch); //La cargamos con el archivo deseado
    gram.text:=gram.text + cFinCadena;//Construimos la cadena de entrada
    Inicializar(gram.text, true);//Inicializamos con la cadena de entrada
  gram.free;//Desechamos el ejemplar

  // Si la cadena de entrada no está vacía podemos empezar el reconocimiento.
  if not FinCadEnt then begin
    nSintact:=ArbSintac.Items.AddChild(nil,'ANÁLISIS SINTÁCTICO ARCHIVO');
    tipoSimbCadEnt:=ArchSigSimb; // Pedimos el primer símbolo
    ReconocerGramat(nSintact);//Intentamos reconocer la gramática leída
  end;
end;
```

Este método inicializa al reconocedor léxico con la cadena de entrada **gram.text**, que no es otro que el archivo leído, como una cadena de texto; es por así decirlo nuestro "buffer"; lee el primer símbolo y desencadena el reconocimiento de la gramática al invocar al método **ReconocerGramat** con el parámetro **nSintact**. Este nodo es resultado de la instrucción

```
nSintact:=ArbSintac.Items.AddChild(nil,'ANÁLISIS SINTÁCTICO ARCHIVO');
```

El método

```
AddChild(xNodo: TTreeNode; const S: string): TTreeNode;
```

regresa un apuntador al nodo que se agregó como hijo del nodo especificado en el parámetro **xNodo, mientras que S** es el texto asociado que se muestra al visualizar el nodo en el componente. En este caso, **nSintac** es el nodo raíz de nuestro árbol sintáctico, por lo que su padre es un apuntador a nulo (**nil**) el cual pasa como parámetro al siguiente método de reconocimiento para poder continuar con la construcción del árbol, ya que los nodos subsecuentes serán hijos de este nodo.

Por fin llegamos al método que reconoce a la clase sintáctica **Gramatica**; para generar su código es necesario recordar que se define como:

Gramatica = { **ReglaProduccion**} **FinArchivo** .

y si recordamos que para reconocer iteraciones del tipo {b}, utilizamos :

Mientras (sig Î CST Directores(β)) intentamos reconocer { ... β... }.

y que CST Directores(**ReglaProduccion**) = {**CLs, Cig**}.

Por otro lado, **FinArchivo** es un símbolo terminal y para reconocer símbolos terminales dijimos que teníamos que comparar el símbolo leído en la cadena de entrada y lo comparábamos con el símbolo terminal esperado; en este caso el símbolo terminal esperado es **FinArchivo** y lo comparamos contra el símbolo leído. Esta comparación la efectúa el método ArchEsperaTipoSimb, por lo que nuestro código queda como se muestra a continuación:

```
procedure tASintactico.ReconocerGramat(xPadre:tTreeNode);
  var nGramat:tTreeNode;
begin
  // Creamos nodo de gramática en el árbol sintáctico
  nGramat:=arbSintac.Items.AddChild(xPadre, NombTipoSimb(tsGramat));

  while tipoSimbCadEnt in [tsClaseSint, tsIgnorados] do
    ReconocerReglaProduc(nGramat);

  ArchEsperaTipoSimb(tsFinCadena, nGramat);
end;

procedure tASintactico.ArchEsperaTipoSimb(const xTipoSimbEsperado:tTipoSimb;
                                          xPadre:tTreeNode);
begin
  if tipoSimbCadEnt = xTipoSimbEsperado then
    ArchLeeSig(xPadre)
  else
    DesplegarErrorSintac(xTipoSimbEsperado, xPadre);
end;

procedure tASintactico.ArchLeeSig(xPadre:tTreeNode);
begin
  // Primero insertamos el nodo padre
  if tipoSimbCadEnt<>tsIgnorados then
    arbSintac.Items.AddChild(xPadre, nombSimbEnt);
  tipoSimbCadEnt:=ArchSigSimb;
end;
```

```
procedure tASintactico.DesplegarErrorSintac(const xTipoSimbEsperado:tTipoSimb;
                                                    xPadre:tTreeNode);
  var sMsj:string; nSimb:tTreeNode;
begin

  sMsj:='ERROR *** Esperaba: '+nombsTipoSimb[ord(xTipoSimbEsperado)]+
      ' y encontré: ' + carLeido;
  nSimb:=arbSintac.Items.AddChild(xPadre, sMsj);
  nSimb.Data:=tSmb.Create(tsSimbInvalido);

end;
```

Si el archivo esperado coincide con el obtenido de la cadena de entrada agregamos dicho nodo al árbol y leemos el siguiente; en caso contrario desplegamos un mensaje de error. Nótese que nuestro manejo de errores no es complicado, no hay necesidad, puesto que nuestra gramática tampoco lo es, únicamente reportamos cuando encontramos un error e intentamos continuar el reconocimiento con el siguiente símbolo que esperábamos. El método **DesplegarErrorSintac** realiza esta función y lo utilizamos para ambas gramáticas.

Ahora tenemos el método **ReconocerReglaProduc**, que definimos como:

ReglaProduccion = { **CaracterIgnorado** }
ClaseSintactica
{ **CaracterIgnorado** }
"="
{ **Comentario** |
CaracterIgnorado |
SimboloTerminal |
ClaseSintactica |
MetaSimboloParteDerecha }
"."

Tenemos una nueva modalidad que es el reconocimiento de alternativas:

Cómo reconocer alternativas de símbolos, por ejemplo : $A \rightarrow \alpha \mid \beta \mid \varphi$

Si (sig ∈ CST Directores(A, α) entonces intentamos reconocer { ... α ... }

EnCasoContrario

Si (sig ∈ CST Directores(A, β) entonces intentamos reconocer { ... β ... }

En Caso Contrario

Si (sig ∈ CST Directores(A, φ) entonces intentamos reconocer { ... φ ... }

EnCasoContrario

despliegaError('Esperaba Símbolo Terminal = '+ *TerminalEsperado*;

Abortar; // Detenemos el análisis sintáctico.

Pero además esta modalidad es combinada ya que está dentro de una repetición. Para simplificar un poquito las cosas vamos a escribirla de la siguiente forma; necesitamos reconocer { **a** | **b** | **c** }. Donde **a**, **b**, **c** son símbolos terminales y CST Directores(**a**)=**a.** Como podemos ver se trata de algo muy sencillo ya que siguiendo las reglas que hemos establecido, se reduce a un reconocimiento de la siguiente forma:

Mientras sig ∈ CST Directores(A, αi)

LeeSig;

De donde:

```
procedure tASintactico.ReconocerReglaProduc(xPadre:tTreeNode);
  var  nReglaProd:tTreeNode;
       ClaseSint:string;
       Regla:tRegla;
begin
  // Creamos nodo de regla de producción en el árbol sintáctico
  nReglaProd:=arbSintac.Items.AddChild(xPadre, NombTipoSimb(tsReglaProduc));

  { Inicializamos el objeto que contiene a la regla de producción asociada
  con la clase sintáctica.}
  Regla:=tRegla.create('');

  while tipoSimbCadEnt in [tsIgnorados] do // Desechamos los ignorados
    ArchLeeSig(nReglaProd);
```

```
ClaseSint:=cadAsocAlSimb; // En teoría la cadena asociada es una clase sint.
ArchEsperaTipoSimb(tsClaseSint, nReglaProd);//Verificamos que es clase sint.

while tipoSimbCadEnt in [tsIgnorados] do // Volvemos a desechar los ignorados
  ArchLeeSig(nReglaProd);

ArchEsperaTipoSimb(tsSepProduc, nReglaProd);//Debe venir separador de produc.

while tipoSimbCadEnt in [tsComentario, tsIgnorados, tsClaseSint,
                                                    tsSimbTerm, tsMetaSimbPDer] do
begin
  if tipoSimbCadEnt = tsSimbTerm then
    {En el reconcimiento pierde los delimitadores de símbolo terminal
    volvemos a ponerselos.}
    cadAsocAlSimb:=cExtSimbTerm1+ cadAsocAlSimb +cExtSimbTerm1;

  //Continuamos construyendo la regla asociada con la clase sintáctica
  Regla.Produccion:= Regla.Produccion + cadAsocAlSimb;
  ArchLeeSig(nReglaProd);
end;

ArchEsperaTipoSimb(tsFinProduc, nReglaProd);//Finaliza la regla de producción

{Nos aseguramos que se trata de una clase sintáctica antes de agregarla a la
lista de clases sintácticas; lo cual sucede si el primer caracter es uno de
los permitidos para clases sintáticas.}
if ClaseSint[1] in cIdentificador then
    ListaClases.Items.AddObject(ClaseSint, Regla);
end;
```

Nótese que la primera clase sintáctica que se reconoce es la que se inserta en la lista de clases sintácticas y que todos los símbolos que se ignoran que la anteceden y la suceden se desechan; esto quiere decir que si el usuario crea un archivo sin utilizar la aplicación y aplica formato a la clase sintáctica utilizando estos carácteres; éstos se pierden al entrar al sistema; no así los que conforman la regla sintáctica. El ejemplar **Regla** se construye al inicio del reconocimiento y se va construyendo su contenido conforme se reconocen símbolos después del separador de producción; y finalmente, cuando el reconocimiento termina, se agrega a la lista de Clases Sintácticas el nuevo elemento con el nombre de la clase sintáctica y su regla de producción asociada, por medio de la instrucción:

```
ListaClases.Items.AddObject(ClaseSint, Regla);
```

Con esto concluye la generación de código para la gramática G_1.

Analizamos ahora la generación de código para la gramática G_2. Al igual que la gramática anterior, consta de un solo método que permite a los usuarios de esta clase realizar el reconocimiento y es:

```
procedure tASintactico.ReconocerClaseSint(const xsCadEnt:string);
  var nSintact:tTreeNode;
begin
  SimbAntNodo:=nil;
  contGral:=0;
  Inicializar(xsCadEnt, false);  // Inicializamos al Analizador Léxico

  if not FinCadEnt then begin
    // Iniciamos la construcción del árbol sintáctico
    nSintact:=ArbSintac.Items.AddChild(nil,
                        'ANÁLISIS SINTÁCTICO REGLA DE PRODUCCIÓN');
    tipoSimbCadEnt:=ProducSigSimb;// Leemos el primer símbolo
    ReconocerProduc(nSintact);//Iniciamos el reconocimiento
  end;
end;
```

La única aclaración pertinente es en cuanto al contador, que no sirve para otra cosa más que para saber cuántos símbolos hemos reconocido y mostrarlo en el árbol sintáctico; en lo demás es muy similar al método inicial de reconocimiento de G_1, por lo cual pasamos al siguiente método que tiene que ver con el reconocimiento de producciones; si recordamos nuestra definición tenemos:

Produccion = **ClaseSintactica** "=" [**ConjuntoAlternativasSimbolos**] "."

Si (*sig* ∈ CST Directores(β) entonces intentamos reconocer { ... β... }

Y tenemos que

CST Directores(**ConjuntoAlternativasSimbolos**) = {Trm, CLs '(', '[', '{' }

Lo cual nos lleva al método:

```
procedure tASintactico.ReconocerProduc(xPadre:tTreeNode);
  var nProduc:tTreeNode;
begin
```

```
  inc(contGral);
  // Agregamos un nodo del tipo tsProduc.
  nProduc:=arbSintac.Items.AddChild(xPadre, IntToStr(contGral)+
                                    '.'+NombTipoSimb(tsProduc));
  AsocSmb(nProduc, tsProduc);

  SeEsperaTipoSimb(tsClaseSint, nProduc);

  SeEsperaTipoSimb(tsSepProduc, nProduc);

  if tipoSimbCadEnt in [tsSaltoDiagr, tsClaseSint, tsSimbTerm, tsIzqConjAlts,
                                                    tsIzqPOpc, tsIzqPRepeti] then
    ReconocerConjAlts(nProduc);

  SeEsperaTipoSimb(tsFinProduc, nProduc);

end;


procedure tASintactico.SeEsperaTipoSimb(const xTipoSimbEsperado:tTipoSimb;
                                                          xPadre:tTreeNode);
begin
  if tipoSimbCadEnt = xTipoSimbEsperado then
    LeeSig(xTipoSimbEsperado, xPadre)
  else
    DesplegarErrorSintac(xTipoSimbEsperado, xPadre);
end;


procedure tASintactico.LeeSig(const xTipoSimbEsperado:tTipoSimb;
                                    xPadre:tTreeNode);
  var nSimb:tTreeNode;
begin
  if xTipoSimbEsperado<>tsSaltoDiagr then
  begin
    inc(contGral);
    nSimb:=arbSintac.Items.AddChild(xPadre, IntToStr(contGral)+'.'+nombSimbEnt);
    //Antes de leer el siguiente asociamos el tipo de símbolo
    ASocSmb(nSimb, tipoSimbCadEnt);
  end else begin
    ASocSmb(xPadre, tipoSimbCadEnt);
  end;
  tipoSimbCadEnt:=ProducSigSimb;
end;
```

Hemos aprovechado para listar dos métodos más, porque la explicación va encadenada. Podemos ver que no hay problema con la generación del código asociado al reconocimiento; y que los métodos **SeEsperaTIpoSimb** y **LeeSig** son prácticamente iguales que los de *G1*, únicamente tenemos las siguientes variantes:

- AsocSmb, cuya función explicaremos en la parte que tiene que ver con la generación de diagramas.
- tsSaltoDiagr, que es un símbolo que hemos agregado a nuestra gramática para poder cortar la secuencia de dibujo de un diagrama en lo horizontal y reiniciarlo por la izquierda; esto le sirve al usuario para convertir la longitud cuando un diagrama que se extiende demasiado hacia la derecha en profundidad. Es decir, permite estirar un diagrama hacia abajo para recuperar espacio hacia la derecha.

Para generar el código de **ReconocerConjAlts**, debemos recordar que:

ConjuntoAlternativasSimbolos = SucesionSimbolos { '|' SucesionSimbolos }

y CST Directores('|' **SucesionSimbolos**) = {'|'}. El método queda así:

```
procedure tASintactico.ReconocerConjAlts(xPadre:tTreeNode);
  var i:integer;
     nConjAlts:tTreeNode;
     numSuce:byte;
begin
  inc(contGral);
  // Agregamos un nodo del tipo tsConjAltsSimbs.
  nConjAlts:=arbSintac.Items.AddChild(xPadre, IntToStr(contGral)+'.'
                                + NombTipoSimb(tsConjAltsSimbs));
  ASocSmb(nConjAlts, tsConjAltsSimbs);
  nConjAlts.Data:=tConjAltsSimbs.Create(tsConjAltsSimbs);
  ReconocerSuceSimbs(nConjAlts);
  numSuce:=1;//Contador de Sucesiones de Símbolos
  while (tipoSimbCadEnt = tsSepAlts) do
  begin
    LeeSig(tsSepAlts, nConjAlts);
    ReconocerSuceSimbs(nConjAlts);
    inc(numSuce);
  end;
  tConjAltsSimbs(nConjAlts.Data).numSuce:=numSuce;//Guardamos numSuce
end;
```

Respecto a este método, dejamos pendiente la función de la variable y de la propiedad **numSuce**, para cuando analicemos la generación de los diagramas. Tenemos ahora que generar el código para **ReconocerSuceSimbs** y por nuestra definición tenemos que:

SucesIonSimbolos = Simbolo { Simbolo }

y como ***CST Directores***(Simbolo) = {'#', Trm, CLs '(', '[', '{' }. El método queda así:

```
procedure tASintactico.ReconocerSuceSimbs(xPadre:tTreeNode);
  var nSuceSimbs:tTreeNode;
begin
  inc(contGral);
  // Agregamos un nodo del tipo tsSuceSimbs.
  nSuceSimbs:=arbSintac.Items.AddChild(xPadre,IntToStr(contGral)+'. '
                                                    +NombTipoSib(tsSuceSimbs));
  AsocSmb(nSuceSimbs, tsSuceSimbs);

  ReconocerSimb(nSuceSimbs);
  while tipoSimbCadEnt in [ tsSaltoDiagr, tsSimbTerm, tsClaseSint, tsIzqConjAlts,
                                                    tsIzqPOpc, tsIzqPReperi] do
    ReconocerSimb(nSuceSimbs);
end;
```

Llegamos finalmente al último método que reconoce símbolos, recurrimos nuevamente a nuestra definición de la clase sintáctica **Símbolo** y tenemos:

Símbolo = **'#'** |
SimboloTerminal |
ClaseSintactica |
'(' ConjuntoAlternativasSimbolos ')' |
'[' ConjuntoAlternativasSimbolos ']' |
'{' ConjuntoAlternativasSimbolos '}' | .

Ya que hemos visto cómo generar código para alternativas, tenemos que nuestro método queda así:

```
procedure tASintactico.ReconocerSimb(xPadre:tTreeNode);
  var i:integer;
      nSimb:tTreeNode;
begin
```

```
    inc(contGral);
    // Agregamos un nodo del tipo tsSimb.
    nSimb:=arbSintac.Items.AddChild(xPadre,IntToStr(contGral)+'. ' +
                                                    NombTipoSimb(tsSimb));
    ASocSmb(nSimb, tsSimb);
    case tipoSimbCadEnt of
      tsSaltoDiagr:LeeSig(tsSaltoDiagr, nSimb);
      tsSimbTerm:LeeSig(tsSimbTerm, nSimb);
      tsClaseSint:LeeSig(tsClaseSint, nSimb);
      tsIzqConjAlts:
      begin
        LeeSig(tsIzqConjAlts, nSimb);
        ReconocerConjAlts(nSimb);
        SeEsperaTipoSimb(tsDerConjAlts, nSimb);
      end;
      tsIzqPOpc:
      begin
        LeeSig(tsIzqPOPc, nSimb);
        ReconocerConjAlts(nSimb);
        SeEsperaTipoSimb(tsDerPOpc, nSimb);
      end;
      tsIzqPRepeti:
      begin
        LeeSig(tsIzqPRepeti, nSimb);
        ReconocerConjAlts(nSimb);
        SeEsperaTipoSimb(tsDerPRepeti, nSimb);
      end;
      else DesplegarErrorSintac(tsSimbInvalido, nSimb);
    end;
  end;
```

Con esto concluimos la programación de nuestro analizador sintáctico.

3.3 Cómo dibujar las figuras básicas

Para poder generar los diagramas a partir del árbol sintáctico necesitamos antes que nada saber qué y cómo vamos a dibujar. En Delphi el manejo de gráficos se hace a través de una clase llamada **tCanvas** que además es independiente del dispositivo en el que se desea dibujar; esto quiere decir que a través de esta clase podemos con el mismo método dibujar una línea en la impresora o en una ventana. Nosotros vamos a definir una clase

llamada **tDibujar** que utiliza un ejemplar de **tCanvas** (llamado Lienzo) para implementar todos nuestros dibujos. Podríamos heredar de **tCanvas**, pero nos conviene más manejar un ejemplar de tCanvas como propiedad de nuestra clase tDibujar, porque de esa manera podemos asignar el ejemplar que deseemos de la clase **tCanvas**.

Recuérdese que deseamos poder conservar los diagramas fuera de nuestra aplicación; para tal efecto dijimos que los vamos a guardar en formato de ***Windows Metafile***; esta clase se va encargar también de ese trabajo. Es lo lógico porque se encarga de todo lo relacionado con los dibujos y sus características.

Otra de las tareas que vamos a asociar con nuestra clase tiene que ver con la posibilidad de que el usuario elija la apariencia de los diagramas. De eso hablaremos a profundidad en la sección 3.4, pero aquí explicaremos cuáles son esos parámetros y cómo se calculan estos valores que nos permiten modificar la apariencia del texto, basados exclusivamente en la altura del texto.

Necesitamos primero identificar las figuras básicas que necesitamos para generar las figuras de nuestros diagramas; el siguiente dibujo contiene todas las figuras que necesitamos poder generar.

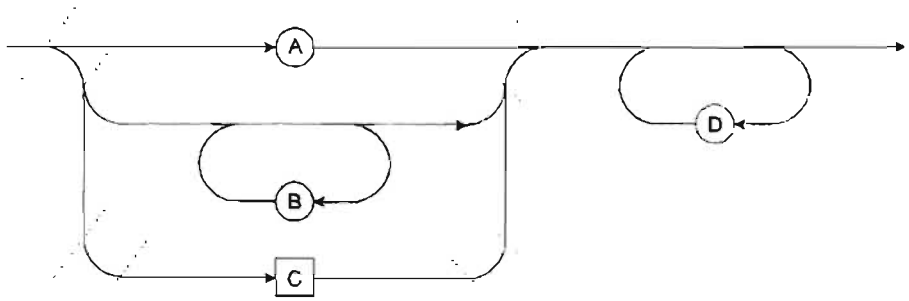


FIGURA 3.3.1

Obviamente las figuras básicas que necesitamos son:

- Línea horizontal ——
- Línea vertical |
- Rectángulo ▭
- Texto *Texto*

- Flecha a la derecha 
- Flecha a la izquierda 
- Arco del cuadrante I: 
- Arco del cuadrante II: 
- Arco del cuadrante III: 
- Arco del cuadrante IV: 

Vamos a analizar ahora cómo debemos dibujar símbolos pertenecientes a nuestra gramática, pero antes debemos definir una serie de conceptos relacionados con los diagramas que vamos a generar. Éstos tienen que ver con la flexibilidad que pretendemos tenga el sistema. Uno de los aspectos que nos afectan en este punto es que tenemos contemplado que el usuario pueda escoger:

- El tipo de fuente, el tamaño y el estilo del texto que desea utilizar al generar los diagramas.
- La distancia horizontal entre el texto y las figuras que rodean al texto que son básicamente rectángulos, círculos y rectángulos con esquinas redondeadas para el caso de símbolos terminales cuya longitud de texto asociada no se pueda enmarcar en un círculo. A este concepto lo vamos a llamar ***Alineación (Ix)*** del texto.
- La distancia vertical entre el texto y las figuras que rodean al texto. A este concepto lo vamos a llamar ***interlineado(Ry)*** del texto.
- La distancia entre las figuras que rodean al texto y las figuras de otro símbolo que se encuentra arriba o abajo. A este concepto lo llamamos ***margen (My)***.
- La longitud de las líneas asociadas a los símbolos terminales y clases sintácticas ***(Cx)***.
- La longitud de las líneas asociadas a las demás figuras ***(Lx)***.
- El tamaño de las flechas ***(Fx)***.

Todas estas consideraciones tienen un impacto directo en la manera como dibujamos porque modifican el tamaño de nuestras figuras; por ejemplo, necesitamos un rectángulo más grande para enmarcar al texto "dígito" cuando el tamaño de la fuente es de 14 puntos

que para cuando es de 8. Reflejamos estos conceptos y su interpretación gráfica en la figura 3.3.2

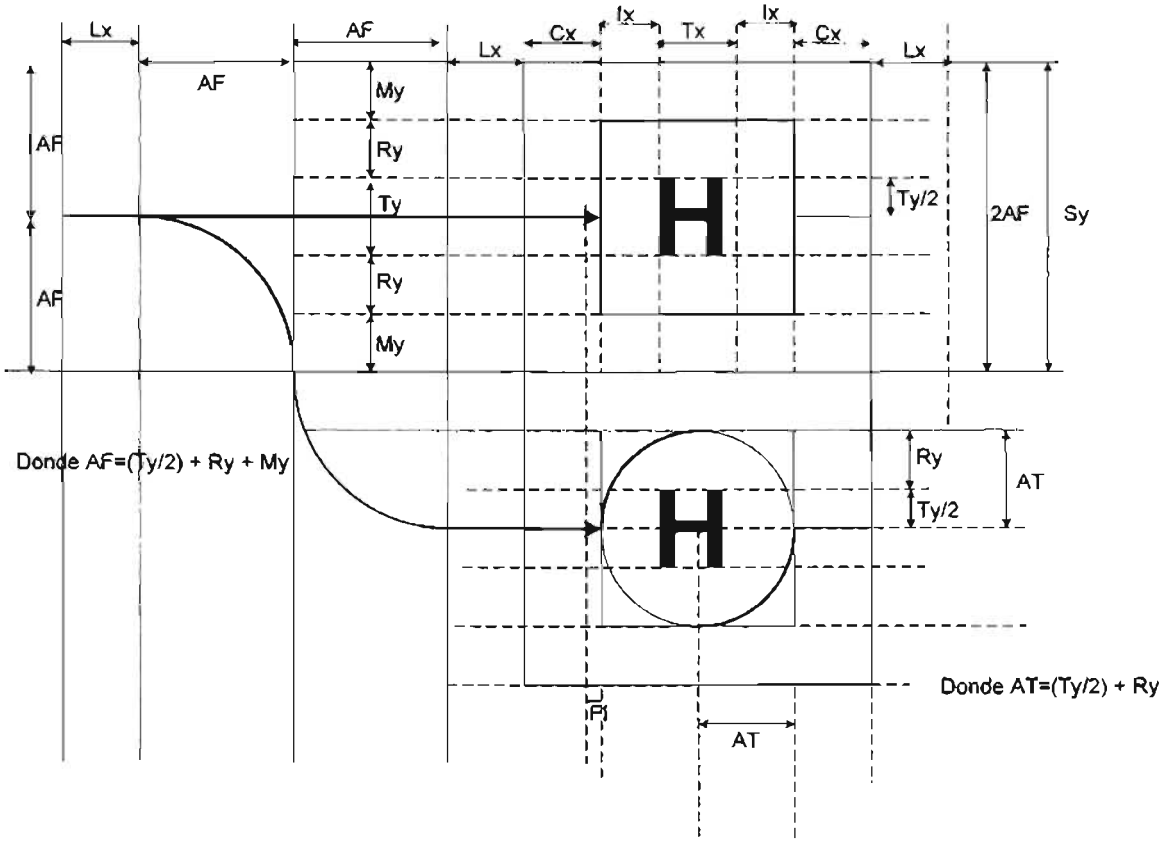


FIGURA 3.3.2

En todos nuestros dibujos vamos a manejar estas medidas; ***AF*** nos da el tamaño de los arcos que utilizamos para dibujar todas las figuras y ***AT*** nos da el tamaño de los arcos para dibujar exclusivamente el círculo o el rectángulo redondeado que rodea el texto de los símbolos terminales. Esta medida depende por completo de la altura del texto, la cual a su vez depende de la fuente elegida, de su estilo y de su tamaño. Los demás parámetros podrían ser independientes del tamaño de la fuente, pero hemos decidido hacerlos dependientes también de la altura de la fuente para que cuando el usuario modifique el tamaño de la fuente todo se ajuste de una manera proporcional, ya que los demás parámetros representan un porcentaje de la altura del texto. El usuario puede elegir el porcentaje de cada uno de ellos. Todas estas medidas las actualizamos cada vez que el usuario elige una nueva fuente, estilo

o tamaño. Por eso Implementamos un procedimiento llamado **Actualizar**, que describimos más adelante. Todas estas medidas las implementamos como variables globales en la unidad **ConstipU.pas** que mencionamos en la sección anterior de la siguiente manera:

```
VAR
  PctIx : byte; { Porcentaje que se aplica a la altura del texto para
                  obtener la alineación del texto}
  PctRy : byte; // Interlineado del texto
  PctMy : byte; // Margen de las figuras
  PctCx : byte; // Longitud de la línea horiz. de las cajas
  PctLx : byte; // Longitud de las otras líneas horizontales
  PctFl : byte; // Tamaño de las flechas


  {Tx :integer;  // Longitud del texto
  Tax:integer;  // Ajuste a la longitud del texto en el símbolo terminal}
  Ty:integer;   // Altura del texto
  Tym:integer;  // Mitad de la altura del texto
  Ix:integer;   // Alineación del texto
  Ry:integer;   // Interlineado del texto
  Cx:integer;   // Longitud de la línea horiz. de las cajas
  Fl:integer;   // Longitud de las flechas
  Lx:integer;   // Longitud de las otra líneas horizontales
  My:integer;   // Margen de las figuras
  AF:integer;   // Longitud de los arcos de las figuras
  AFi:integer;
  AT:integer;   // Longitud de los arcos de los símbolos terminales
  ATi:integer;

  // Sy = My + Ry + Ty + Ry + My : Altura de los símbolos
  Sy:integer;
  SyT:integer;
```

Ya tenemos todos los elementos que necesitamos para definir la interfaz de nuestra clase que es la siguiente:

```
type
 // Implementa los dibujos para cualquier lienzo "canvas"
 tDibujar = class(tObject)
  private
   { Cuadro de diálogo para salvar archivos lo utilizamos cuando copiamos el
```

```
    * diagrama en el portapapeles}
    dlgGuardarComo: TSaveDialog;
    pLienzo : tCanvas; // Apuntador al lienzo donde deseamos dibujar

    {Nos sirve para no perder el lienzo original, ya que cuando copiamos en
    * el portapapeles el el lienzo apunta a un lienzo del tipo Metafile}
    pLienzoPantalla:tCanvas;

    {* * * * * * * Código agregado para posibilitar el envío del gráfico al
                    Portapapeles de Windows<Windows Clipboard>}
    pMetafile          : tMetafile;
    pMetafileCanvas    : tMetafileCanvas;

  public
    constructor Create(const xLienzo:tCanvas);
    destructor  Destroy;override;

    property  Lienzo : tCanvas read pLienzo;

    {Actualiza los parámetros que determinan el tamaño de los dibujos de los
    * diagramas}
    procedure Actualizar;

    procedure Limpiar(const dx, dy:integer);
    function  CalcLongTxt(const xTxt:string):integer;
   // Métodos para dibujar las figuras básicas
    procedure LinHor(const x, y, dx :integer);
    procedure LinVer(const x, y, dy:integer);
    procedure Rectang(const x, y, dx, dy:integer);
    procedure Arco_I(const x, y, c:integer);
    procedure Arco_II(const x, y, c:integer);
    procedure Arco_III(const x, y, c:integer);
    procedure Arco_IV(const x, y,c:integer);
    procedure Texto(const x, y :integer; xTxt:string);
    procedure FlechaIzq(x, y,  dx, dy :integer);
    procedure FlechaDer(x, y,  dx, dy :integer);
    procedure FlechaArr(x, y,  dx, dy :integer);
    procedure FlechaAba(x, y,  dx, dy :integer);
    procedure AbrirMetaArchivo;
    procedure MetaArchivoAsignarMedidas(const dx, dy:integer);
    procedure CerrarMetaArchivo(const xOpcionMetaArchivo:tOpcionMetaArchivo);
    procedure AsignarFuente(const xFuente:tFont);
    procedure AsignarPorcentajes(const xIx, xRy, xMy, xCx, xLx, xFl:integer);
  end;
```

Nuestro método para crear ejemplares de esta clase es:

```
constructor tDibujar.Create(const xLienzo:tCanvas);
begin
  inherited create;
  pLienzo:=xLienzo;
  pLienzoPantalla:=xLienzo;
end;
```

Lo único que hace es inicializar los porcentajes de los parámetros que luego explicaremos para qué sirven; **xLienzo** es un apuntador al ejemplar en donde deseamos dibujar, que en este caso es un control visible dentro de la ventana principal de nuestra aplicación. ¿Por qué duplicamos esta referencia? porque cuando necesitamos generar un archivo a partir de nuestro diagrama, o cuando queremos copiarla al portapapeles, en ese proceso necesitamos que **pLienzo** apunte a un lienzo diferente, y cuando quisiéramos dibujar nuevamente en nuestra ventana tendríamos que volver a asignar el lienzo, pero eso no es necesario porque tenemos su referencia en **pLienzoPantalla**. Esa es la razón de ser de **pLienzoPantalla**.

Antes de especificar como dibujamos estas figuras básicas, establecemos las siguientes convenciones:

- ***Dx*** y ***Dy*** significan respectivamente ***longitud*** y ***altura*** de una figura o símbolo.
- Todas las figuras que manejamos tienen un **punto inicial** denotado por ***(x,y)*** a partir del cual se basa el dibujo de la figura en cuestión.

Para generar líneas, la clase **tCanvas** cuenta con el método **LineTo(x, y:integer)**, que dibuja una línea desde la posición indicada por **PenPos** a la posición indicada por el punto **(x, y)**. **Pen** es el ejemplar por medio del cual **tCanvas** pinta o dibuja en el dispositivo elegido. En nuestro método para dibujar líneas vamos a utilizar este método combinado con otro llamado **MoveTo(x,Y:integer)**. Éste modifica el valor de **PenPos** a la posición indicada por los parámetros **(x, y)**. Combinando estos dos métodos podemos dibujar una línea de la siguiente manera con sólo indicar la posición inicial y su desplazamiento sobre el eje **X** y el **Y**.

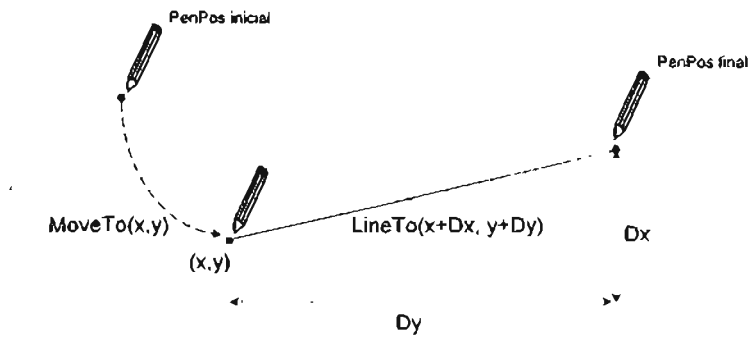


FIGURA 3.3.3

Para diferenciar las líneas verticales de las horizontales, implementamos los siguientes métodos :

```
procedure tDibujar.LineaHor(x, y, dx:integer);
begin
  if pLienzo<>nil then
  begin
    pLienzo.MoveTo(x, y);
    pLienzo.LineTo(x + dx, y);
  end;
end;

procedure tDibujar.LineaVer(x, y, dy:integer);
begin
  if pLienzo<>nil then
  begin
    pLienzo.MoveTo(x, y);
    pLienzo.LineTo(x, y+dy);
  end;
end;
```

Un rectángulo en **tCanvas** se dibuja indicando las coordenadas del vértice superior izquierdo (punto inicial) y las coordenadas del vértice inferior derecho (punto final); sin embargo para nosotros el punto inicial será el punto medio entre el vértice superior izquierdo y el vértice inferior izquierdo, en vez de indicar cuál es el punto final vamos a proporcionar el tamaño del rectángulo, indicados como ***dx***, ***dy*** en la figura 3.3.3, donde vemos cómo se dibuja en Delphi y cómo lo dibujamos nosotros; obviamente utilizando el mismo método pero con parámetros diferentes.

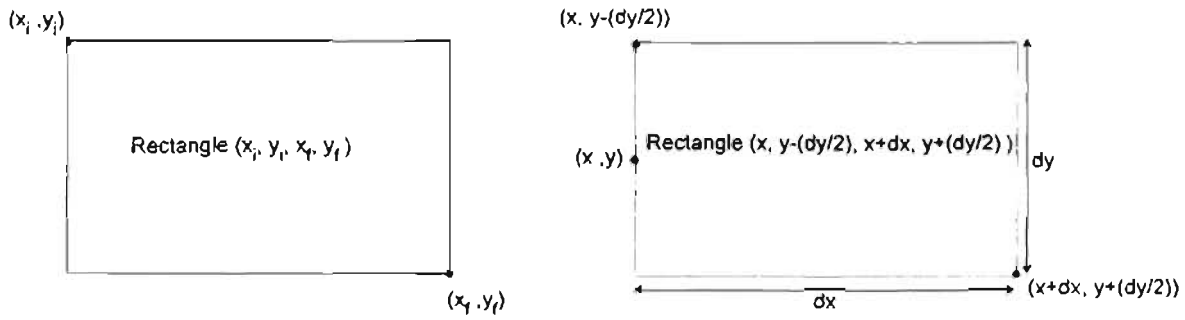


FIGURA 3.3.4

Por lo tanto, nuestro método queda como:

```
procedure tDibujar.Rectang(const x, y, dx, dy:integer);
begin
    if pLienzo<>nil then
  begin
    pLienzo.Rectangle( x, y - round(dy/2), x+dx, y + round(dy/2));
  end;
end;
```

Para dibujar texto, **tCanvas** dispone del método **Textout**; este método únicamente necesita saber el punto donde debe empezar a dibujar y el texto que va a dibujar. Para dibujar texto, utilizamos el mismo método, pero, al igual que con el rectángulo, manejamos una variante. Este método dibuja el texto desde el punto que se le indica hacia la derecha y hacia abajo, como se indica en la figura 3.3.5. Para nosotros el texto se dibuja desde el punto que se le indica hacia la derecha pero la mitad hacia arriba y la mitad hacia abajo. En la figura 3.3.5 vemos la comparación entre el método normal y nuestro método.

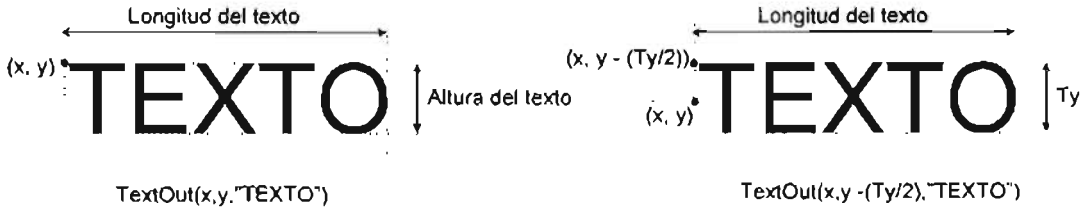


FIGURA 3.3.5

Nuestro método para dibujar texto queda así:

```
procedure tDibujar.Texto(const x, y:integer; xTxt:string);
begin
  if pLienzo<>nil then
  begin
    pLienzo.TextOut(x, y - Tym , xTxt);
  end;
end;
```

Donde **Sy** es igual a la mitad de la altura del texto.

Delphi no dispone de métodos para dibujar flechas, por lo tanto debemos implementar lo nosotros, utilizamos el método más sencillo donde una flecha es un triángulo con relleno. En nuestro método sólo necesitamos indicar el punto hacia donde apunta la flecha, indicar el tamaño de la flecha y el tipo de flecha que deseamos. En la figura 3.3.6 mostramos los cuatro tipos de flechas que manejamos.

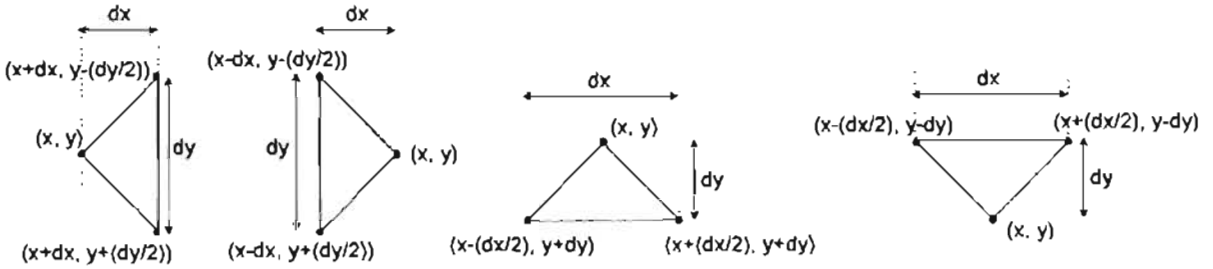


FIGURA 3.3.6

En **tCanvas** el método **Polygon** nos permite dibujar poligonos con relleno, por lo tanto implementamos los siguientes métodos para dibujar flechas:

```
procedure tDibujar.Flechalzq(x, y,  dx, dy :integer);
  var   P1, P2: tPoint;
        vEstilo:tPenStyle;
        vAncho :integer;
begin
  if pLienzo<>nil then
  begin
    vEstilo:=pLienzo.Pen.Style;
    vAncho:=pLienzo.Pen.Width;

    pLienzo.Pen.Style:=psSolid;
    pLienzo.Pen.Width:=1;
```

```
    pLienzo.Brush.Style:=bsSolid;
    pLienzo.Brush.Color:=clBlack;
    pLienzo.Polygon([Point(x+dx, y-round(dy/2)), Point(x+dx, y+round(dy/2)), Point(x, y)]);
    pLienzo.Brush.Style:=bsClear;

    pLienzo.Pen.Style:=vEstilo;
    pLienzo.Pen.Width:=vAncho;
  end;
end;
procedure tDibujar.FlechaDer(x, y,  dx, dy :integer);
  var P1, P2: tPoint;
      vEstilo:tPenStyle;
      vAncho :integer;
begin
  if pLienzo<>nil then
  begin
    vEstilo:=pLienzo.Pen.Style;
    vAncho:=pLienzo.Pen.Width;

    pLienzo.Pen.Style:=psSolid;
    pLienzo.Pen.Width:=1;

    pLienzo.Brush.Style:=bsSolid;
    pLienzo.Brush.Color:=clBlack;
    pLienzo.Polygon([Point(x-dx, y-round(dy/2)), Point(x-dx, y+round(dy/2)), Point(x, y)]);
    pLienzo.Brush.Style:=bsClear;

    pLienzo.Pen.Style:=vEstilo;
    pLienzo.Pen.Width:=vAncho;
  end;
end;
procedure tDibujar.FlechaArr(x, y,  dx, dy :integer);
  var P1, P2: tPoint;
      vEstilo:tPenStyle;
      vAncho :integer;
begin
  if pLienzo<>nil then
  begin
    vEstilo:=pLienzo.Pen.Style;
    vAncho:=pLienzo.Pen.Width;

    pLienzo.Pen.Style:=psSolid;
    pLienzo.Pen.Width:=1;

    pLienzo.Brush.Style:=bsSolid;
    pLienzo.Brush.Color:=clBlack;
    pLienzo.Polygon([Point(x-round(dx/2), y+dy), Point(x+round(dx/2), y+dy), Point(x, y)]);
```

```
      pLienzo.Brush.Style:=bsClear;

      pLienzo.Pen.Style:=vEstilo;
      pLienzo.Pen.Width:=vAncho;
    end;
  end;
  procedure tDibujar.FlechaAba(x, y,  dx, dy :integer);
    var P1, P2: tPoint;
        vEstilo:tPenStyle;
        vAncho :integer;
  begin
    if pLienzo<>nil then
    begin
      vEstilo:=pLienzo.Pen.Style;
      vAncho:=pLienzo.Pen.Width;

      pLienzo.Pen.Style:=psSolid;
      pLienzo.Pen.Width:=1;

      pLienzo.Brush.Style:=bsSolid;
      pLienzo.Brush.Color:=clBlack;
      pLienzo.Polygon([Point(x-round(dx/2), y-dy), Point(x+round(dx/2), y-dy), Point(x, y)]);
      pLienzo.Brush.Style:=bsClear;

      pLienzo.Pen.Style:=vEstilo;
      pLienzo.Pen.Width:=vAncho;
    end;
  end;
```

El ejemplar **Brush** es la brocha que se utiliza para pintar y representa el color de relleno, al indicar que su propiedad **Style** es ***bsSolid*** le estamos pidiendo a **tCanvas** que al dibujar las figuras tengan color de relleno. Al terminar de dibujar la flecha le pedimos que nuevamente las figuras no tengan relleno, dado que nuestras figuras no manejan el concepto de relleno.

Para generar un arco, **tCanvas** dispone del método **Arc**; la descripción de este método es la siguiente:

Arc dibuja una curva elíptica . El arco recorre el perímetro de la elipse acotada por los puntos (X_1,Y_1) y (X_2,Y_2). El arco se dibuja siguiendo el perímetro de la elipse, en el sentido contrario a las manecillas del reloj, desde el punto inicial hasta el punto final. El punto inicial (X_3,Y_3) se define por la intersección de la elipse y una línea desde este punto al centro de la elipse. El punto final (X_4,Y_4) se define de la misma forma. La interpretación gráfica es la

siguiente:

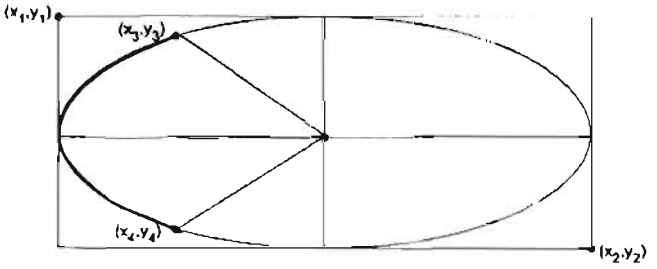
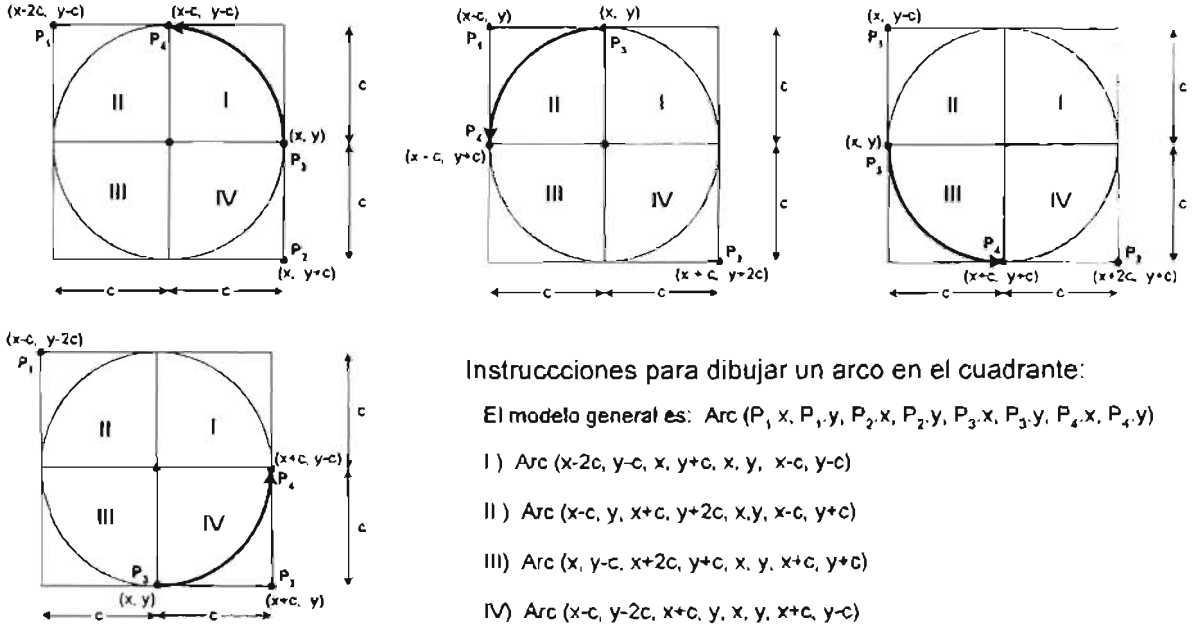


FIGURA 3.3.7

Los arcos que utilizamos para generar nuestras gráficas tienen una característica muy especial; todos están inscritos en un círculo; en la siguiente figura indicamos la forma de estos arcos y las instrucciones que necesitamos para generarlos.



Instruccciones para dibujar un arco en el cuadrante:

El modelo general es: Arc (P_1.x, P_1.y, P_2.x, P_2.y, P_3.x, P_3.y, P_4.x, P_4.y)

I) Arc (x-2c, y-c, x, y+c, x, y, x-c, y-c)

II) Arc (x-c, y, x+c, y+2c, x,y, x-c, y+c)

III) Arc (x, y-c, x+2c, y+c, x, y, x+c, y+c)

IV) Arc (x-c, y-2c, x+c, y, x, y, x+c, y-c)

FIGURA 3.3.8

Por lo tanto, cada vez que generamos un arco, únicamente necesitamos posicionarnos en el punto (x_3, y_3) e indicar el tamaño (**c**) del arco. Por lo tanto tenemos cuatro métodos para generar arcos que son:

```
procedure tDibujar.Arc_I(const x, y, c:integer);
begin
  if pLienzo<>nil then
```

```
  begin
    Lienzo.Arc(x - (2 *c), y-c, x, y +c, x, y, x-c, y-c);
  end;
end;

procedure tDibujar.Arc_II(const x, y, c:integer);
begin
  if pLienzo<>nil then
  begin
      Lienzo.Arc(x - c, y, x+c, y+(2*c), x, y, x-c, y+c);
  end;
end;

procedure tDibujar.Arc_III(const x, y, c:integer);
begin
  if pLienzo<>nil then
  begin
    Lienzo.Arc(x, y-c, x+(2*c), y +c, x, y, x+c, y+c);
  end;
end;

procedure tDibujar.Arc_IV(const x, y, c:integer);
begin
  if pLienzo<>nil then
  begin
    Lienzo.Arc(x - 2 *c, y-c, x, y +c, x, y, x-c, y-c);
  end;
end;
```

Ya sabemos cómo dibujar todas las figuras básicas. Pero, por ejemplo ¿cómo borramos los dibujos que hemos hecho para iniciar uno nuevo? Para eso necesitamos el método **Limpiar**:

```
procedure tDibujar.Limpiar(const dx, dy:integer);
begin
  if pLienzo<>nil then
  begin
    //Borra el contenido del lienzo.
    pLienzo.Brush.Style:=bsSolid;
    pLienzo.FillRect(Rect(0, 0, dx, dy));
    pLienzo.Brush.Style:=bsClear;
```

```
  end;
end;
```

Siempre que dibujamos en el lienzo, la propiedad **Brush**(brocha), se hace cargo del manejo del color del fondo; en nuestro caso, siempre que dibujamos esta propiedad se establece a **bsClear**(sin color de fondo); es decir, nuestras figuras no tienen color de fondo, por lo tanto para borrar lo que hemos dibujado lo que necesitamos hacer es rellenar el rectángulo donde hemos dibujado:

```
(pLienzo.FillRect(Rect(0, 0, dx, dy));)
```

con el color del fondo:

```
pLienzo.Brush.Style:=bsSolid;
```

al finalizar restablecemos la brocha para que no maneje color del fondo.

Para poder dibujar nuestros diagramas necesitamos conocer el espacio que ocupa horizontalmente el texto en nuestro lienzo; esa es otra de las tareas de nuestra clase:

```
function tDibujar.CalcLongTxt(const xTxt:string):integer;
begin
  if pLienzo<>nil then
    result:= pLienzo.TextWidth(xTxt);  // Calculamos la longitud del texto.
end;
```

Es hora de que expliquemos lo relacionado con los parámetros que hemos mencionado para lo cual mostramos el método **Actualizar**:

```
procedure tDibujar.Actualizar;
begin
  pLienzo.Brush.Style:=bsClear;
  Ty:=pLienzo.TextHeight('H');  // Altura del texto
  Tym:=round((Ty/2));  // Mitad de la altura del texto

  Ix:= round(PctIx / 100 * Ty); // Alineación del texto
  Ry:= round(PctRy / 100 * Ty); // Interlineado del texto
  Cx:= round(PctCx / 100 * Ty); // Longitud de la línea horiz. de las cajas
  Fl:= round(PctFl / 100 * Ty); // Longitud de las flechas

  Lx:= round(PctLx / 100 * Ty); // Longitud de las otra líneas horizontales
```

```
    My:= round(PctMy / 100 * Ty); // Margen de las figuras


    SyT:= My + Ry + Ty + Ry + My; // Altura de los símbolos = 24F
    AF:=Max(SyT - round(SyT/2), round(SyT/2));
    Sy:=AF + AF;

    AT:= Ry + Tym;      // Longitud de los arcos de los símbolos terminales
    ATi:= Ry + Ty - AT;

    CAx:= Lx +Sy +Lx;   // Longitud necesaria para dibujar alternativas
    CAmx:= Lx + AF;
    Rpx:= Lx + AF + Lx; // Longitud necesaria para dibujar repeticiones
  end;
```

Podemos ver que todos estos parámetros dependen básicamente del tamaño del texto y de los porcentajes. El tamaño del texto, a su vez, depende del tipo de fuente elegida, de su estilo y de su tamaño. La aplicación permite que el usuario elija estas características para la fuente de los diagramas. Los porcentajes también los puede modificar el usuario. Lo que en resumidas nos deja que el usuario pueda elegir la apariencia de los diagramas que se generan moviendo los valores de los porcentajes y eligiendo las propiedades de la fuente. Ya hemos visto el significado geométrico de estos parámetros, los cuales a propósito se declaran en la unidad ConsTipU.pas, que es la unidad que contiene todas las constantes y variables globales que se utilizan en el programa.

El siguiente método actualiza las propiedades de la fuente utilizada, para posteriormente poder **Actualizar** los parámetros.

```
procedure tDibujar.AsignarFuente(const xFuente:tFont);
begin
  pLienzo.Font.Name:=xFuente.Name;
  pLienzo.Font.Style:=xFuente.Style;
  pLienzo.Font.Size:=xFuente.Size;
end;
```

Por último tenemos la implementación para poder generar los diagramas en formato de Windows Metafile. Los métodos asociados son los siguientes:

```
procedure tDibujar.AbrirMetaArchivo;
```

```
begin
  pMetaFile     := TMetafile.Create;
  pMetafile.Enhanced:=true;
  pMetafileCanvas := TMetafileCanvas.Create(pMetafile, 0);
  pLienzo := pMetaFileCanvas;
end;


procedure tDibujar.MetaArchivoAsignarMedidas(const dx, dy:integer);
begin
  {Ahora ya sabemos cuanto mide}
  pMetafileCanvas.Free;
  pMetafile.Width := dx+1;
  pMetafile.height:= dy+1;
  pMetafileCanvas := TMetafileCanvas.Create(pMetafile, 0);
  pLienzo       := pMetaFileCanvas;
end;


procedure tDibujar.CerrarMetaArchivo(const xOpcionMetaArchivo:
  tOpcionMetaArchivo);
 var Formato            : Word;
    DireccionMemoria    : THandle;
    Paleta              : HPALETTE;
begin
  pMetafileCanvas.Free;
  case xOpcionMetaArchivo of
    maAlPortapapeles:
    begin
      Clipboard.Open;
      pMetafile.SaveToClipboardFormat(Formato, DireccionMemoria, Paleta);
      ClipBoard.SetAsHandle(Formato, DireccionMemoria);
      Clipboard.close;
    end;
    maConfirmarGuardar:
    begin
      dlgGuardarComo:=TSaveDialog.Create(Application);
      dlgGuardarComo.DefaultExt:='*.emf';
      dlgGuardarComo.Filter:='Windows Metafile (*.emf;*.wmf;*.*)|*.emf;*.wmf';
      dlgGuardarComo.Options:=[ofOverwritePrompt,ofHideReadOnly,ofEnableSizing];
      dlgGuardarComo.FileName:=NombreMetaArchivo;
      if dlgGuardarComo.execute then
        pMetafile.SaveToFile(dlgGuardarComo.FileName);
    end;
       maGuardar:pMetafile.SaveToFile(DirectorioMetaArchivos+NombreMetaArchivo+'.
```

```
    emf');
    end;
    pLienzo:=pLienzoPantalla;
    pMetafile.free;
  end;
```

No vamos a entrar en mayores detalles; pero a continuación explicamos brevemente el proceso.

Para generar nuestro diagrama en formato ***Windows Metafile***, procedemos así:

1. Ejecutamos el método **AbrirMetarchivo**.

   Primero creamos un ejemplar de **TMetafile** que es una implementación del formato Win32 Enhanced metafile; además indicamos que el gráfico generado es del tipo .EMF (Win32 Enhanced Metafile).

   ```
   pMetaFile     := TMetafile.Create;
   pMetafile.Enhanced:=true;
   ```

   Después creamos el lienzo en donde va a dibujar este ejemplar y se lo asignamos a **pLienzo** que es siempre nuestro apuntador al lienzo en donde dibuja nuestra clase:

   ```
   pMetafileCanvas := TMetafileCanvas.Create(pMetafile, 0);
   pLienzo       := pMetaFileCanvas;
   ```

   Nótese que aquí perdemos la referencia al lienzo de nuestra ventana en la aplicación principal, pero la podemos restablecer por medio de **pLienzoPantalla**.

2. Medimos la producción.

3. Ejecutamos el método **MetaArchivoAsignarMedidas(const dx, dy)**, donde:

   - ***dx*** = la longitud de la producción deseada.
   - ***dy*** = la altura de la producción deseada.

   Aparentemente para establecer las medidas de **pMetafile**, tenemos que destruir el ejemplar **pMetafileCanvas;** ahora ya podemos establecer las medidas no lo hicimos al inicio porque no sabíamos las medidas:

   ```
   pMetafile.Width := dx+1;
   pMetafile.height:= dy+1;
   ```

tenemos que volver a crear un ejemplar del lienzo donde vamos a dibujar y asignárselo a pLienzo:

```
pMetafileCanvas := TMetafileCanvas.Create(pMetafile, 0);
pLienzo         := pMetaFileCanvas;
```

4. Dibujamos la producción o lo que deseemos dibujar.

5. Ejecutamos el método **CerrarMetaArchivo**.

Lo primero que hay que hacer es:

```
pMetafileCanvas.Free;
```

Pareciera absurdo destruir el lienzo donde dibujamos y luego hacer referencia a pMetafile, que contiene un lienzo que ya no existe, pero si no se destruye el lienzo antes, nada de lo que sigue funciona. Dependiendo de la opción que estemos manejando tenemos:

- **maAlPortapapeles**: las siguientes cuatro líneas son un protocolo establecido que hay que seguir al pie de la letra para que la imagen quede a disposición del portapapeles:

```
Clipboard.Open;
pMetafile.SaveToClipboardFormat(Formato, DireccionMemoria, Paleta);
ClipBoard.SetAsHandle(Formato, DireccionMemoria);
Clipboard.close;
```

- **maConfirmarGuardar**: Se muestra el cuadro de diálogo para que el usuario decida donde guardar el metarchivo, y guardar el archivo:

```
pMetafile.SaveToFile(dlgGuardarComo.FileName);
```

- **maGuardar**: Se guarda el archivo en la carpeta donde el usuario haya elegido:

```
maGuardar:pMetafile.SaveToFile(DirectorioMetaArchivos+NombreMeta
Archivo+'.emf');
```

La variable DirectorioMetaArchivo contiene el nombre del subdirectorio que el usuario eligió para guardar todos los diagramas del archivo de gramática.

El último paso es restablecer el lienzo de la ventana y destruir el ejemplar utilizado para generar el gráfico de Windows Metafile:

```
pLienzo:=pLienzoPantalla;
pMetafile.free;
```

Con esto concluimos la descripción de la clase **tDibujar**.

3.4 Cómo dibujar símbolos

Ahora sí podemos preguntarnos ¿qué necesitamos para generar (dibujar) un diagrama (gráfico)? Primero necesitamos saber qué ***figura*** deseamos dibujar, en segundo lugar necesitamos conocer sus ***dimensiones*** y por último su posición en el ***espacio***. En nuestro caso el ***espacio*** es una ventana de la aplicación, por lo tanto tenemos:

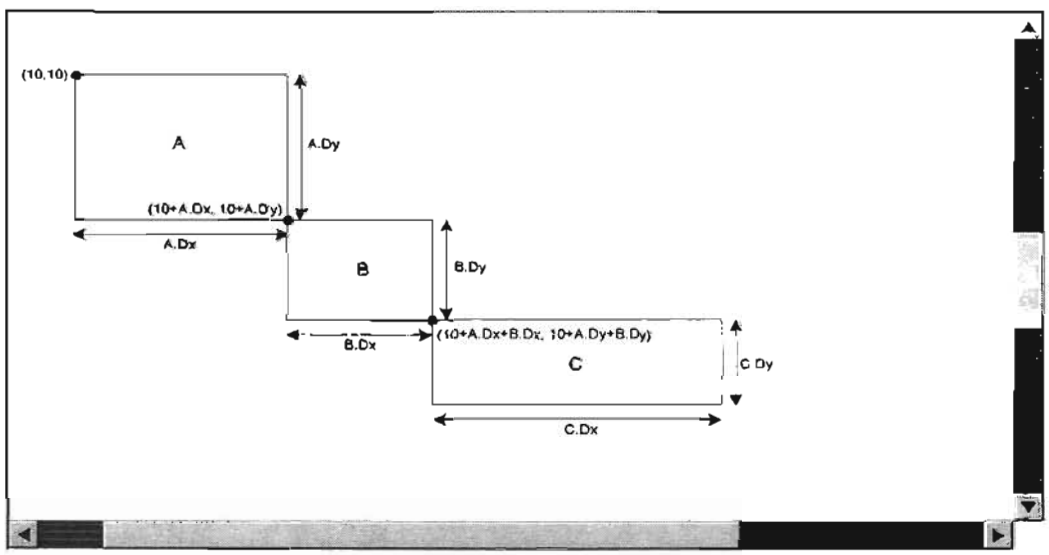


FIGURA 3.4.1

En este ejemplo, tomamos por convención que el inicio de un rectángulo es el punto ubicado en el vértice superior izquierdo, y el final del rectángulo es el punto ubicado en el vértice inferior derecho. Como podemos observar, las tres ***figuras*** son rectángulos, sus ***dimensiones*** las conocemos y su posición queda determinada completamente por la posición inicial de la figura ***A***; esto es así porque la posición inicial de la figura ***B*** coincide con la posición final de la figura ***A***; lo mismo sucede con la figura ***C*** con respecto a la figura ***B***. Estas tres figuras guardan una relación de ***orden*** y de ***posición*** entre sí; podemos deducir que cuando una sucesión de figuras están relacionadas entre sí de esta forma, para dibujarlas necesitamos únicamente la posición de la figura inicial, las dimensiones de cada figura y obviamente la forma.

Por lo tanto necesitamos conocer la ***forma de las figuras*** que deseamos dibujar y determinar las ***relaciones de orden y posición*** que las figuras guardan entre sí; así como las ***dimensiones*** de cada figura basados en estas ***relaciones de orden y posición***; la ***posición inicial*** podemos fijarla en cualquier punto del espacio y será el punto en el que inicia el dibujo de nuestras figuras.

¿Cómo utilizamos la información del árbol sintáctico para generar los diagramas? ¿Podrá su estructura indicarnos cómo dibujar los diagramas?. De antemano sabemos que sí, porque si no, nuestra gramática no sería útil para lo que pretendemos. El problema es saber cómo vamos a realizar el siguiente ejercicio; supongamos que tenemos la regla de producción:

(a |b [c] | d {e}) [g h]

es una regla muy simple pero la elegimos porque involucra a todos los elementos representativos de nuestra gramática G_2. Producto del análisis sintáctico tenemos el árbol de la figura 3.4.3 y su diagrama asociado. Procedemos de la siguiente manera; intentamos relacionar al árbol con la gráfica de acuerdo a la estructura jerárquica de éste, además tratamos de determinar las dimensiones de las figuras y las reglas para dibujar las figuras que conforman el diagrama. No nos interesan los tres primeros nodos, porque no intervienen en el diagrama. Entonces, si tomamos el cuarto nodo tenemos

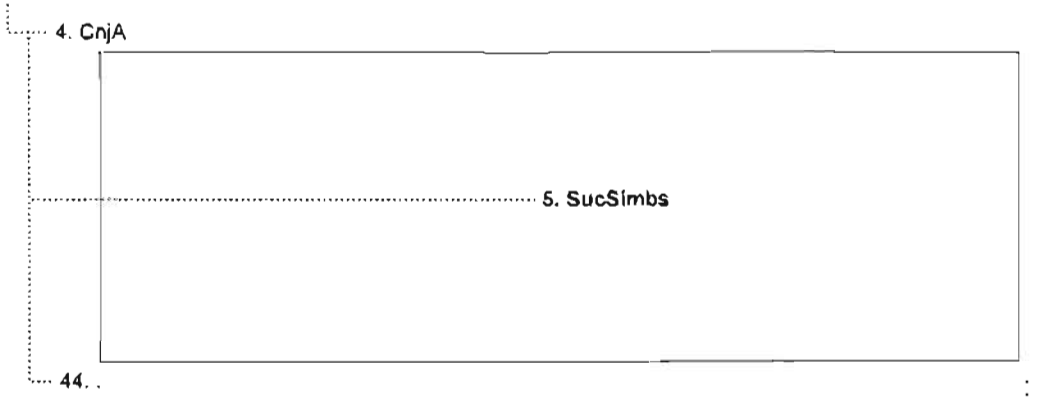
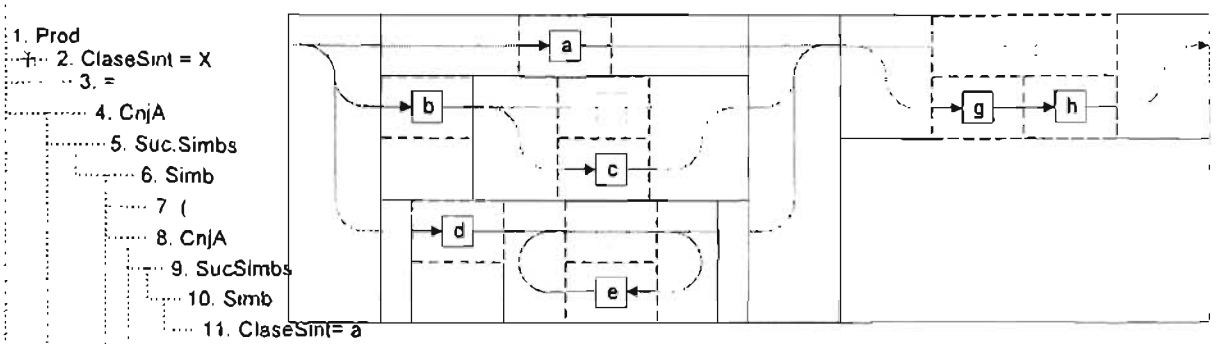


FIGURA 3.4.2



1. Prod
2. ClaseSint = X
3. =
4. CnjA
5. Suc.Simbs
6. Simb
7. (
8. CnjA
9. SucSimbs
10. Simb
11. ClaseSint= a
12. |
13. SucSimbs
14. Simb
15. ClaseSint= b
16. Simb
17. [
18. CnjA
19. SucSimbs
20. Simb
21. ClaseSint= c
22.]
23. |
24. SucSimbs
25. Simb
26. ClaseSint= d
27. Simb
28. {
29. CnjA
30. SucSimbs
31. Simb
32. ClaseSint= e
33. }
34.)
35 Simb
36. [
37. CnjA
38. SucSimbs
39. Simb
40. ClaseSint= g
41. Simb
42. ClaseSint= h
43.]
44. .

FIGURA 3.4.3

Como podemos ver en la gráfica 3.4.2, el nodo 4 contiene únicamente al nodo 5 **SucSimbs** al cual le asociamos el rectángulo que encuadra a todo nuestro diagrama, lo cual es válido porque si observamos el nodo 5 contiene a todos los símbolos asociados con el diagrama.

Por lo tanto podemos decir que

CnjA4.Dx = SucSimbs5.Dx,
CnjA4.Dy = SucSimbs5.Dy,
CnjA4.Dibujar=SucSimbs5.Dibujar;

Tomamos ahora el nodo 5 y continuamos el desglose jerárquico.

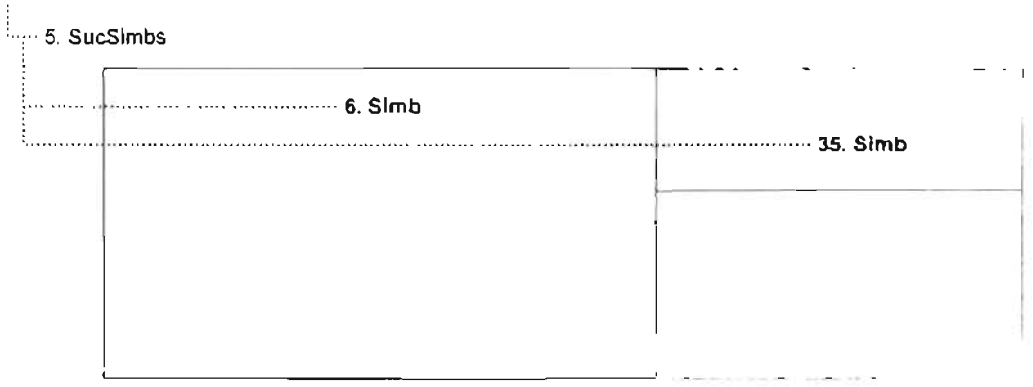


FIGURA 3.4.4

SucSimbs5.Dx = Simb6.Dx + Simb35.Dx,
SucSimbs5.Dy = Max(Simb6.Dy, Simb35.Dy)=Simb6.Dy,

SucSimbs5.Dibujar = 1. Dibujar Simb6;
2. Dibujar Simb35 a la derecha de donde termina Simb6.

Por el momento no nos interesa precisar en dónde termina ***Simb6***, únicamente que ***Simb35*** se dibuja a la derecha de donde termina ***Simb6***.

Continuamos ahora con los nodos 6 y 35 en la figura 3.4.5

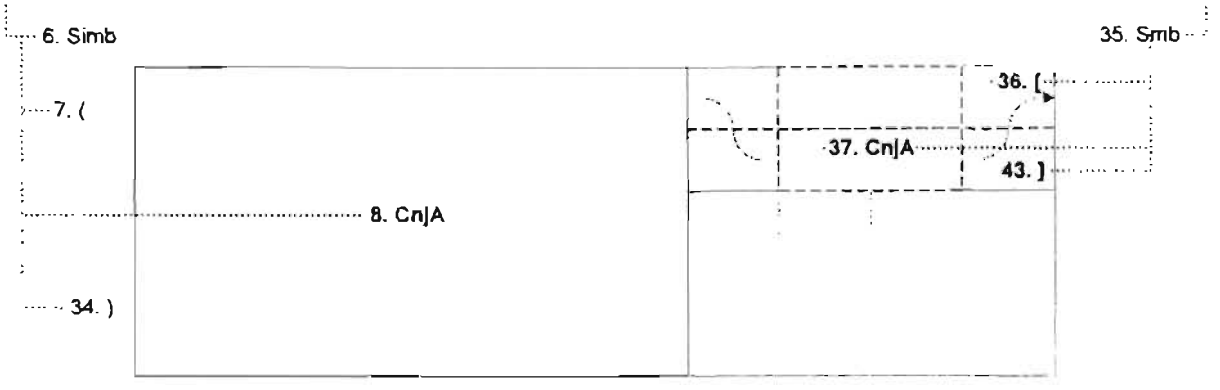


FIGURA 3.4.5

Simb6.Dx = CnjA8.Dx;
Simb6.Dy = CnjA8.Dy;

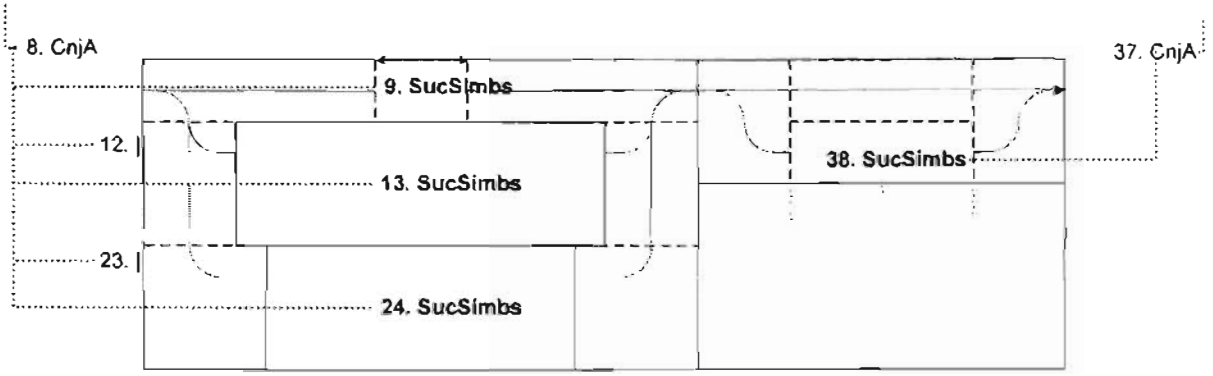
y

Simb35.Dx = CnjA37.Dx + (La longitud de las figuras asociadas con la parte opcional)
Simb35.Dy = CnjA37.Dy + (La Altura de las figuras asociadas con la parte opcional)

Las reglas de dibujo para los símbolos 6 y 35 son:

Simb6.Dibujar = ConjA8.Dibujar;
Simb35.Dibujar = Dibujar figuras asociadas con parte izquierda superior opcional ;
a la derecha dibujar figuras asociadas con parte der. sup. opcional;
abajo dibujar figuras asociadas con parte izq inferior opcional ;
a la derechar CnjA37.Dibujar;
a la derecha dibujar figuras asociadas con parte der. inferior opcional ;

Continuamos con ConjA8 y CnjA37 tenemos:



que corresponde a:

CnjA8.Dx = Max(SucSimbs9.Dx, SucSimbs12.Dx, SucSimbs24.Dx,) + (La longitud de las figuras asociadas con la parte izauierda de alternativas));
CnjA8.Dy = SucSimbs9.Dy + SucSimbs12.Dy + SucSimbs24.Dy;
CnjA37.Dx = SucSimbs38.Dx;
CnjA37.Dy = SucSimbs38.Dy;

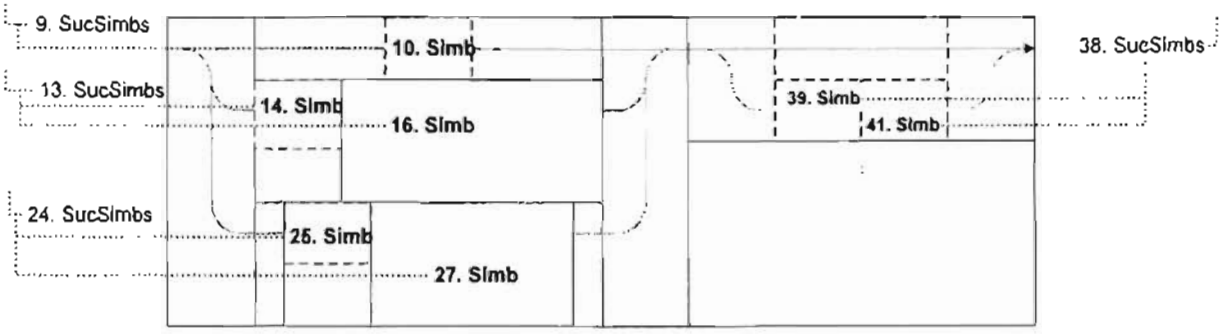
Las reglas de dibujo para el símbolo 8 son:

CnjA8.Dibujar = Dibujar figuras asociadas con parte izq. superior de conjunto alternativas;
a la derecha SucSimbs9.Dibujar;
a la derecha dibujar figuras asociadas con la parte der. superior de cnj. alts.;
abajo dibujar figuras asociadas con parte izq. intermedia de conj. alts.:
a la derecha SucSimbs13.Dibujar;
a la derecha dibujar figuras asociadas con parte der. intermedia de conj. alts.;

abajo dibujar figuras asociadas con parte izq. inferior de conj. alts.;
a la derecha SucSimbs245.Dibujar;
a la derecha dibujar figuras asociadas con parte der. inferior de conj. alts.;

CnjA37.Dibujar = SucSimbs38.Dibujar;

Tomamos los siguientes nodos 9,13,24 y 38:



SucSimbs9.Dx = Simb10.Dx;
SucSimbs9.Dy = Simb10.Dy;
SucSimbs13.Dx = Simb14.Dx + Simb16.Dx;
SucSimbs13.Dy = Max(Simb14.Dy + Simb16.Dy)=Simb16.Dy;
SucSimbs24.Dx = Simb25.Dx + Simb27.Dx;
SucSimbs24.Dy = Max(Simb25.Dy + Simb27.Dy)=Simb27.Dy;
SucSimbs38.Dx = Simb39.Dx + Simb41.Dx;
SucSimbs38.Dy = Max(Simb39.Dy + Simb41.Dy)=Simb39.Dy;

Y las reglas para dibujarlos son:

SucSimbs9.Dibujar = SucSimbs10.Dibujar;
SucSimbs13.Dibujar = Simb14.Dibujar;
a la derecha Simb16.Dibujar;
SucSimbs24.Dibujar = Simb25.Dibujar;
a la derecha Simb27.Dibujar;
SucSimbs38.Dibujar = Simb39.Dibujar;
a la derecha Simb41.Dibujar;

Tomamos los siguientes nodos que corresponden al 10, 14, 25 y 39, por lo que tenemos la figura 3.4.6, de donde tenemos:

Simb10.Dx = ClaseSint11.Dx = La long. de las figuras asociadas + long. del texto;
Simb10.Dy = ClaseSint11.Dy = La altura de las figuras asociadas + alt. del texto;
Simb14.Dx = ClaseSint15.Dx = La long. de las figuras asociadas + long. del texto
Simb14.Dy = ClaseSint15.Dy =La long. de las figuras asociadas + long. del texto

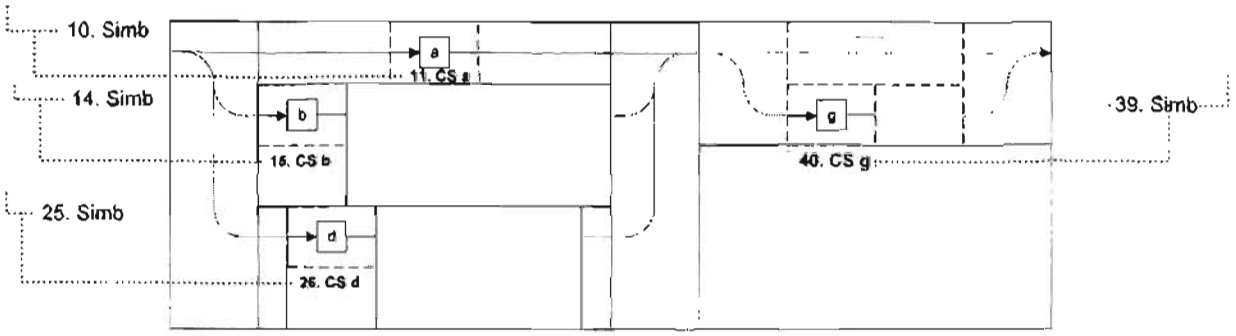


FIGURA 3.4.6

Simb25.Dx = ClaseSint26.Dx=La long. de las figuras asociadas + long. del texto;
Simb25.Dy = ClaseSint26.Dy=La altura de las figuras asociadas + alt. del texto;
Simb39.Dx = ClaseSint40.Dx=La long. de las figuras asociadas + long. del texto;
Simb39.Dy = ClaseSint40.Dy=La altura de las figuras asociadas + alt. del texto;

Y nuestras reglas de dibujo serían:

Simb10.Dibujar = ClaseSint11.Dibujar = Dibujar Linea;
A la derecha dibujar flecha a la derecha;
Dibujar rectángulo para encuadrar texto;
Recorrer a la derecha para identar texto;
dibujar texto= 'a';
A la derecha dibujar linea;

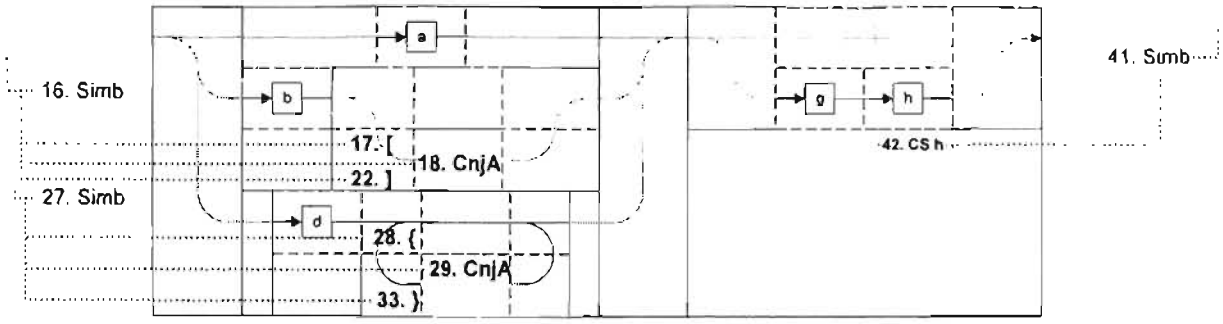
Simb14.Dibujar = ClaseSint15.Dibujar = Dibujar Linea;
A la derecha dibujar flecha a la derecha;
Dibujar rectángulo para encuadrar texto;
Recorrer a la derecha para identar texto;
dibujar texto= 'b';
A la derecha dibujar linea;

Simb25.Dibujar = ClaseSint26.Dibujar = Dibujar Linea;
A la derecha dibujar flecha a la derecha;
Dibujar rectángulo para encuadrar texto;
Recorrer a la derecha para identar texto;
dibujar texto= 'e';
A la derecha dibujar linea;

Simb39.Dibujar = ClaseSint40.Dibujar = Dibujar Linea;
A la derecha dibujar flecha a la derecha;
Dibujar rectángulo para encuadrar texto;
Recorrer a la derecha para identar texto;
dibujar texto= 'g';
A la derecha dibujar linea;

Detallamos un poco más las figuras asociadas con las clases sintácticas pero no fuimos específicos, lo haremos más adelante; nuestro objetivo en este momento es únicamente verificar que la estructura del árbol nos permite medir y dibujar.

Tomamos a continuación los nodos 16, 27 y 41:



Tenemos así las medidas siguentes:

Simb16.Dx = CnjA18.Dx + (La longitud de las figuras asociadas con la parte opcional)
Simb16.Dy = CnjA18.Dy + (La altura de las figuras asociadas con la parte opcional)
Simb27.Dx = CnjA29.Dx + (La longitud de las figuras asociadas con la parte opcional)
Simb27.Dy = CnjA29.Dy + (La altura de las figuras asociadas con la parte opcional)
Simb41.Dx = ClaseSint42.Dx=La long. de las figuras asociadas + long. del texto;
Simb41.Dy = ClaseSint42.Dy=La altura de las figuras asociadas + alt. del texto;

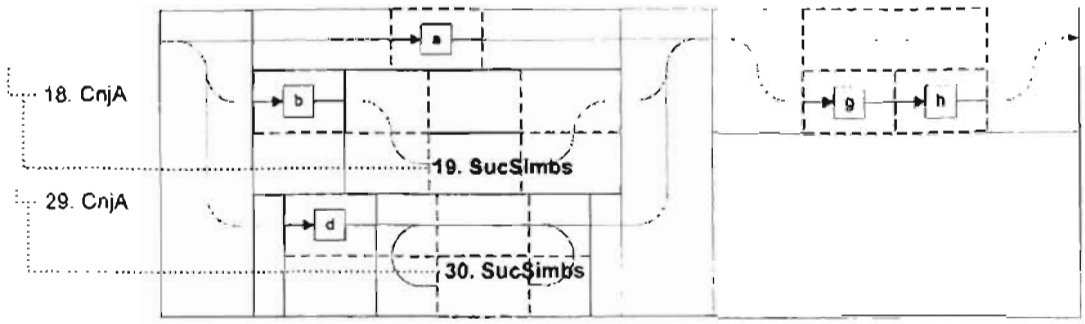
Y nuestras reglas de dibujo serían:

Simb16.Dibujar = Dibujar figuras asociadas con parte izquierda superior opcional ;
a la derecha dibujar figuras asociadas con parte der. sup. opcional;
abajo dibujar figuras asociadas con parte izq inferior opcional ;
a la derechar CnjA18.Dibujar;
a la derecha dibujar figuras asociadas con parte der. inferior opcional ;

Simb27.Dibujar = Dibujar figuras asociadas con parte izquierda superior repetitiva ;
a la derecha dibujar figuras asociadas con parte der. sup. repetitiva;
abajo dibujar figuras asociadas con parte izq inferior repetitiva;
a la derechar CnjA29.Dibujar;
a la derecha dibujar figuras asociadas con parte der. inferior repetitiva;

Simb41.Dibujar = ClaseSint42.Dibujar = Dibujar Linea;
A la derecha dibujar flecha a la derecha;
Dibujar rectángulo para encuadrar texto;
Recorrer a la derecha para identar texto;
dibujar texto= 'h';
A la derecha dibujar linea;

Tomamos a continuación los nodos 18 y 29:



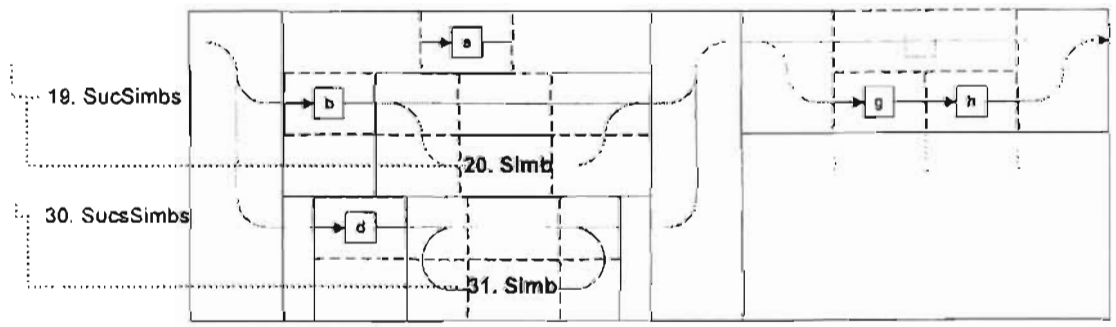
Las medidas son:

CnjA18.Dx = SucSimbs19.Dx;
CnjA18.Dy = SucSimbs19.Dy;
CnjA29.Dx = SucSimbs30.Dx;
CnjA29.Dy = SucSimbs30.Dy;

Y las reglas para dibujar son:

CnjA18.Dibujar = SucSimbs19.Dibujar;
CnjA29.Dibujar = SucSimbs30.Dibujar;

Ahora con los nodos 19 y 30 tenemos:

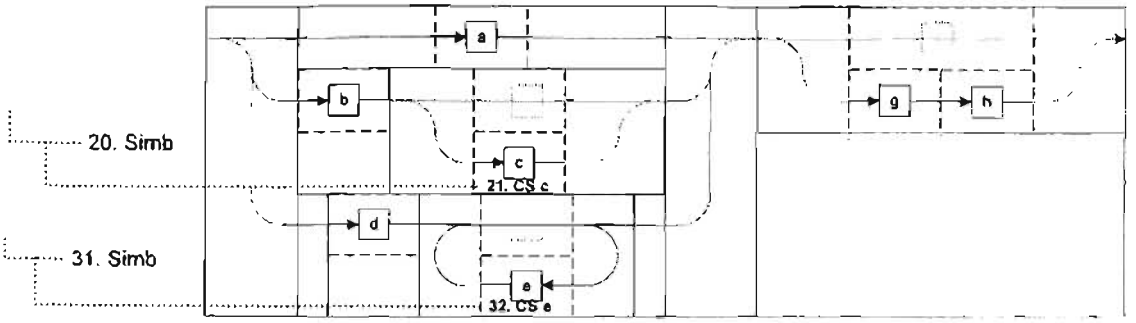


SucSimbs19.Dx = Simb20.Dx;
SucSimbs19.Dy = Simb20.Dy;
SucSimbs30.Dx = Simb31.Dx;
SucSimbs30.Dy = Simb31.Dy;

Y las reglas para dibujar son:

SucSimbs19.Dibujar = Simb20.Dibujar;
SucSimbs30.Dibujar = Simb31.Dibujar;

Por último tomamos los nodos 20 y 31:



Las medidas son:

Simb20.Dx = ClaseSint21.Dx=La long. de las figuras asociadas + long. del texto;
Simb20.Dy = ClaseSint21.Dy=La altura de las figuras asociadas + alt. del texto;
Simb31.Dx = ClaseSint32.Dx=La long. de las figuras asociadas + long. del texto;
Simb31.Dy = ClaseSint32.Dy=La altura de las figuras asociadas + alt. del texto;

Y nuestras reglas de dibujo serían:

Simb20.Dibujar = ClaseSint21.Dibujar = Dibujar Linea;
A la derecha dibujar flecha a la derecha;
Dibujar rectángulo para encuadrar texto;
Recorrer a la derecha para identar texto;
dibujar texto= 'c';
A la derecha dibujar linea;

Simb31.Dibujar = ClaseSint32.Dibujar = Dibujar Linea;
A la derecha dibujar flecha a la derecha;
Dibujar rectángulo para encuadrar texto;
Recorrer a la derecha para identar texto;
dibujar texto= 'e';
A la derecha dibujar linea;

Con esto concluimos exitosamente nuestro ejercicio pragmático; el árbol en efecto nos permite generar nuestros diagramas. Sin embargo, con este ejercicio no hemos generalizado , ni especificado la manera en que podemos generar los diagramas. ¿Cómo podemos lograr esto? Obteniendo estas reglas para cada clase sintáctica de nuestra gramática G_2, ya que cada clase sintáctica tiene asociada una figura determinada. Y como el árbol refleja nuestra gramática; es decir cada nodo representa un símbolo reconocido de nuestra gramática G_2, lo único que necesitamos es asociar a éste su correspondiente símbolo con estas reglas específicas para medir y dibujar. Pero, ¿cómo realizamos esta asociación? En la sección 2.6.4 dejamos pendiente la descripción de un método llamado **AsocSmb**, el cual se encarga

precisamente de esto.

Antes de describir este método tenemos que hacer algunas aclaraciones. Escogimos este componente visual que representa una estructura de árbol porque nos sirve para que el usuario vea el árbol sintáctico generado y porque permite asociar a cada nodo del árbol un ejemplar del tipo **tObject**. La clase **tObject** es la base para toda la estructura jerárquica de las clases que se manejan en ***Delphi***. Esto nos posibilita implementar una clase especializada que podemos asociar a cada nodo del árbol. En nuestro caso vamos a implementar una clase por cada elemento de nuestra gramática. Lo lógico entonces sería que definiéramos cada una de estas clases heredando directamente de **tObject**, pero interponemos entre nuestras clases y **tObject** una clase base a la que llamaremos **tSmb**. ¿Por qué hacemos esto?. La primera razón es que todas nuestras clases tienen una forma y comportamiento básico general a partir del cual podemos empezar a realizar especializaciones; y la segunda porque deseamos beneficiarnos del concepto de ***polimorfismo***.

El concepto de ***polimorfismo*** nos permite manejar a un ejemplar de una clase especializada como si fuera un ejemplar de la clase base. Lo cual es conveniente en nuestro caso porque al realizar el reconocimiento sabemos qué tipo de símbolo debemos asociar con cada nodo, pero luego al recorrer el árbol y querer utilizar el símbolo asociado tendríamos que saber con certeza el tipo símbolo asociado; lo cual no es imposible pero es poco conveniente y haría más complicada la programación. En cambio, utilizando este concepto podemos únicamente hacer referencia explícita a una clase especializada sólo cuando sea necesario y en todos los demás casos manejar a los ejemplares asociados como si fueran de la clase básica **tSmb**. El programa generado resuelve en tiempo de ejecución la referencia adecuada al método o propiedad según la clase a la que pertenece el ejemplar.

Para tal efecto creamos primero la clase base **tSmb** cuya interfaz mostramos a continuación:

```
tSmb = class(tObject)
  private
    //Propiedades-----------------------------------------------------------
    pTipoSimb  : tTipoSimb;
    pDx        :integer;  // Longitud del símbolo
    pDy        :integer;  // Altura del símbolo
  protected
    //Métodos---------------------------------------------------------------
```

```
    function   LeerNombre:string;
    procedure  ActualizarFuenteLinea;
  public
    constructor Create(const xTipoSimb:tTipoSimb);
    //Propiedades----------------------------------------------------------
    property   TipoSimb : tTipoSimb  read  pTipoSimb;
    property   Dx       : integer    read  pDx write pDx;
    property   Dy       : integer    read  pDy write pDy;
    Property   Nombre   : string     read  LeerNombre;
    //Métodos
    procedure  Medir(const xNodo : tTreeNode);virtual;
    procedure  Pintar(const xNodo:tTreeNode;xDir:tDirec;x,y:integer);virtual;
    function   LeerCadAsoc(const xNodo : tTreeNode):string;virtual;
  end;
```

Esta clase representa un modelo general de las relaciones de orden y posición que intentamos representar para poder generar nuestros diagramas. Si analizamos esta clase podemos ver que tiene una propiedad **TipoSimb**, que indica el tipo de simbolo que representa, las propiedades **Dx** y **Dy** en donde guardamos la longitud y altura del símbolo, un método **Medir** que representa las relaciones de orden y posición necesarias para medir la longitud y altura del símbolo para poder después dibujarlo, por último un método **Pintar** que utiliza estas relaciones de orden y posición pero ahora para dibujar la figura adecuada. Finalmente un método **LeerCadAsoc**, que obtiene la cadena asociada a un nodo del árbol el cual utilizamos para resolver la inclusión de un símbolo en la parte superior de una repetición; más adelante explicamos como funciona.

Nótese que los métodos **Medir**, **Pintar** y **LeerCadAsoc**, al final de su declaración incluyen la directiva virtual, esto significa que las clases que heredan de ésta pueden redefinir estos métodos con un comportamiento más especializado.

La implementación de los métodos de esta clase es:

```
constructor tSmb.Create(const xTipoSimb:tTipoSimb);
begin
 inherited Create;
 pTipoSimb:=xTipoSimb;
end;

function tSmb.LeerNombre:string;
```

```
begin
  result:=nombsTipoSimb[ord(pTipoSimb)];
end;

procedure  tSmb.Medir(const xNodo : tTreeNode);
begin
end;

procedure  tSmb.Pintar(const xNodo : tTreeNode; xDir:tDirec; xInicio:tPoint);
begin
end;

function  tSmb.LeerCadAsoc(const xNodo : tTreeNode):string;
begin
  result:=nombsTipoSimb[ord(TipoSimb)];
end;

procedure  tSmb.ActualizarFuenteLinea;
begin
  Dibujar.Lienzo.Font.Color:=arrSimbsFuenteColor[ind];
  Dibujar.Lienzo.Pen.Color:=arrSimbsLineaColor[ind];
  Dibujar.Lienzo.Pen.Width:=arrSimbsLineaAncho[ind];
  Dibujar.Lienzo.Pen.Style:=arrSimbsLineaEstilo[ind];
end;
```

Nótese que el método **Create** fija el tipo de símbolo creado e inicializa las medidas a ceros. El método **LeerNombre** lee de la tabla de nombres de tipos de símbolos el nombre indicado. Y los métodos **Medir** y **Pintar** están vacíos. ¿Por qué hicimos esto? Total, si **tSmb** es un clase base de la cual no vamos a crear ningún ejemplar pudimos haber declarado a estos métodos además de ***virtuales*** como ***abstractos***. Entonces no tendríamos de implementarlos de esta manera. Pero lo hicimos así porque hay varias clases para las que no se requiere implementar absolutamente nada y para las que vamos a invocar estos métodos; para las que tendríamos que estar definiendo de esta misma manera estos métodos. Mejor hacerlo sólo una vez y no varias. Éstas clasess nos convienen para ser congruentes con la idea de que cada nodo tiene un símbolo asociado. El método **LeerCadAsoc**, toma el nombre del tipo de símbolo asociado, este método nos va a servir para todos los símbolos que son terminales.

Por último tenemos el método **ActualizarFuenteLinea** que establece las características de las líneas y la fuente que se utilizan para dibujar los diferentes tipos de símbolos; ***Ind*** es

una variable global que apunta al tipo de símbolo y que cada tipo de símbolo establece antes de dibujar para que apunte a sus características.

Derivada de esta clase **tSmb** creamos una clase para cada tipo de símbolo involucrado en nuestra gramática G_2. A cada nodo de nuestro árbol sintáctico le asociamos un símbolo de este tipo que representa el tipo de símbolo reconocido; el método **AsocSimb,** cuya definición habíamos dejado pendiente, crea un símbolo del tipo mencionado anteriormente y se lo asocia al nodo correspondiente cada vez que se reconoce un símbolo. El código correspondiente a **AsocSimb** es el siguiente:

```
procedure tASintactico.AsocSmb(n:tTreeNode; const xTipoSimb:tTipoSimb);
  var S1, S2:string;
   SimbAntHer   :tSimb;
   SimbAntHerNodo:tTreeNode;
   SimbIni      : tTreeNode;
   SimbFin      : tTreeNode;
   X:tTreeNode;
begin
  case xTipoSimb of
    tsSepProduc   :n.Data:=tSepProduc.Create(xTipoSimb);
    tsSepAlts     :n.Data:=tSepAlts.Create(xTipoSimb);
    tsIzqConjAlts :n.Data:=tIzqConjAlts.Create(xTipoSimb);
    tsDerConjAlts :n.Data:=tDerConjAlts.Create(xTipoSimb);
    tsIzqPOpc     :n.Data:=tIzqPOpc.Create(xTipoSimb);
    tsDerPOpc     :n.Data:=tDerPOpc.Create(xTipoSimb);
    tsIzqPRepeti  :
    begin
      n.Data:=tIzqPRepeti.Create(xTipoSimb);
      //Obtenemos al hermano de  nuestro padre
      If (SimbAntNodo <> nil) and (not SimbAntNodo.IsFirstNode)then
      begin
        if  not SimbAntNodo.IsFirstNode  then
        begin
          // Obtenemos símbolo anterior que puede entrar en parte superior
          SimbIni:=SimbAntNodo.GetPrevSibling;
              if ((SimbIni<>nil) and (tSmb(SimbIni.data).TipoSimb<>tsSaltoDiagr)) then
  begin
            pila.Push(SimbIni);
            tIzqPRepeti(n.Data).SimbAsoc:=SimbIni;
            tSimb(SimbIni.Data).EsRepSupIni:=false;
          end else begin
            pila.Push(nil);
            tIzqPRepeti(n.Data).SimbAsoc:=nil;
          end;
```

```
        end;
      end else begin
        pila.Push(nil);
        tIzqPRepeti(n.Data).SimbAsoc:=nil;
      end;
    end;
    tsDerPRepeti  :
    begin
      n.Data:=tDerPRepeti.Create(xTipoSimb);
      SimbFin:=n.getPrevSibling;
      SimbFin:=SimbFin.getFirstChild;
      SimbFin:=SimbFin.getLastChild;
      SimbIni:=pila.Pop;
      if SimbIni<>nil then begin
                  if tSmb(SimbIni.Data).LeerCadAsoc(SimbIni)=tSmb(SimbFin.Data).
LeerCadAsoc(SimbFin) then
        begin
          tSimb(SimbIni.Data).EsRepSupIni:=true;
          tSimb(SimbFin.Data).EsRepSupIni:=true;
        end else begin
          x:=n.getPrevSibling;
          x:=x.getPrevSibling;
          tIzqPRepeti(x.Data).SimbAsoc:=nil;
        end;
      end;
    end;
    tsFinProduc   :n.Data:=tFinProduc.Create(xTipoSimb);
    tsFinCadena   :n.Data:=tFinCadena.Create(xTipoSimb);
    tsComentario  :;
    tsIgnorados   :;
    tsSimbTerm    :n.Data:=tSimbTerm.Create(xTipoSimb, indEnTabla);
    tsClaseSint   :n.Data:=tClaseSint.Create(xTipoSimb, indEnTabla);
    tsGramat      :;
    tsReglaProduc :;
    tsPDerRegla   :;
    tsMetaSimbPDer :;
    tsProduc      :n.Data:=tProduc.Create(xTipoSimb);
    tsConjAltsSimbs:n.Data:=tConjAltsSimbs.Create(xTipoSimb);
    tsSuceSimbs   :n.Data:=tSuceSimbs.Create(xTipoSimb);
    tsSimb        :
    begin
      n.Data:=tSimb.Create(xTipoSimb);
      // Creamos un apuntador al símbolo inmediato anterior
      SimbAntNodo:=n;
    end;
    tsSimbInvalido :begin
      n.Data:=tSimbInvalido.Create(xTipoSimb, 'ERROR');
```

```
      end;
      //tsSaltoDiagr  :n.Data:=tSaltoDiagrm.Create(xTipoSimb); ORIGINAL AAA
    end;
  end;
```

El parámetro ***n*** nos indica el nodo que estamos generando producto del reconocimiento de un símbolo de la gramática. La propiedad **Data** nos permite asociar a este nodo del árbol (**tTreeView**) un ejemplar del tipo **tObject**. Es por eso que nuestra clase base hereda de la clase **tObject**.

Ahora sí podemos empezar a crear cada clase que hereda de **tSmb** analizando cada elemento de nuestra gramática y determinando las reglas que utiliza para medir y pintar.

Si recordamos las reglas de G_2, que es la gramática que nos interesa dibujar, tenemos:

1. Produccion = **ClaseSintactica** "=" [ConjuntoAlternativasSimbolos] "."
2. ConjuntoAlternativasSimbolos = SucesionSimbolos { '|' SucesionSimbolos }
3. SucesionSimbolos = Simbolo { Simbolo } .
4. Simbolo = '#' |
   SimboloTerminal |
   ClaseSintactica |
   '(' ConjuntoAlternativasSimbolos ')' |
   '[' ConjuntoAlternativasSimbolos ']' |
   '{' ConjuntoAlternativasSimbolos '}' | .

Tomamos primero **ConjuntoAlternativasSímbolos**, y tenemos:

Aquí podemos ver las figuras asociadas al diagrama de esta regla. A partir de este dibujo agregamos la siguiente convención:

- Una línea punteada significa que la línea es opcional y que su existencia depende básicamente de un ajuste que se hace para que no existan líneas sin conectar o para centrar ciertas figuras.

El dibujo muestra como se dibujarían **n** alternativas, y las dimensiones asociadas con cada parte del dibujo. Sacamos las siguiente conclusiones:

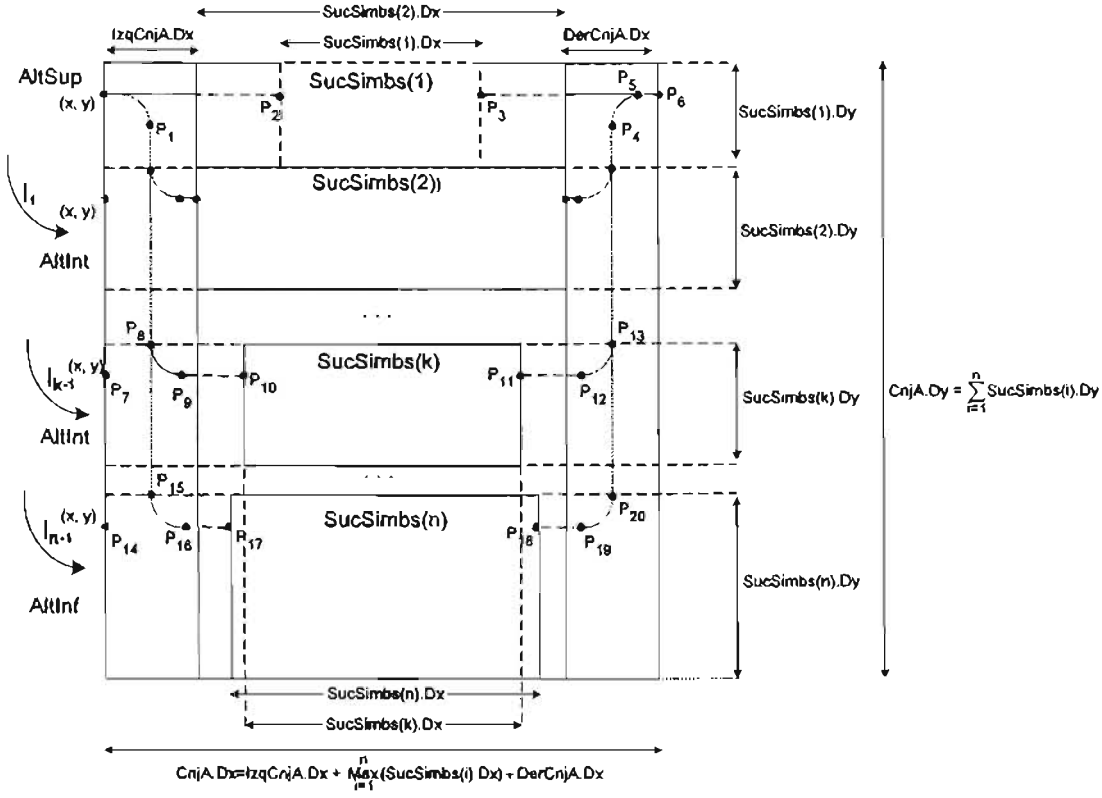
- Cuando sólo tenemos una sucesión no hay nada que dibujar excepto la figura que pueda generar la sucesión por sí misma de acuerdo a sus características.

Para esta clase sintáctica nuestra interfaz es:

```
tConjAltsSimbs = class(tSmb)
  private
    pNumSuce:byte;
    pMaxX:integer;
  public
    property   NumSuce:byte read pNumSuce write pNumSuce;
    procedure  Pintar(const xNodo : tTreeNode; xDir:tDirec; xInicio:tPoint);override;
    procedure  Medir(const xNodo : tTreeNode);override;
    function   LeerCadAsoc(const xNodo : tTreeNode):string;override;
end;
```

CnjA

Caso 1) Con ***n*** alternativas



Caso 2) Con ***1*** alternativa

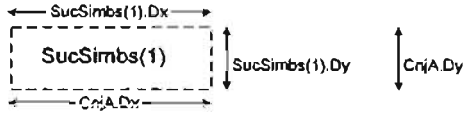


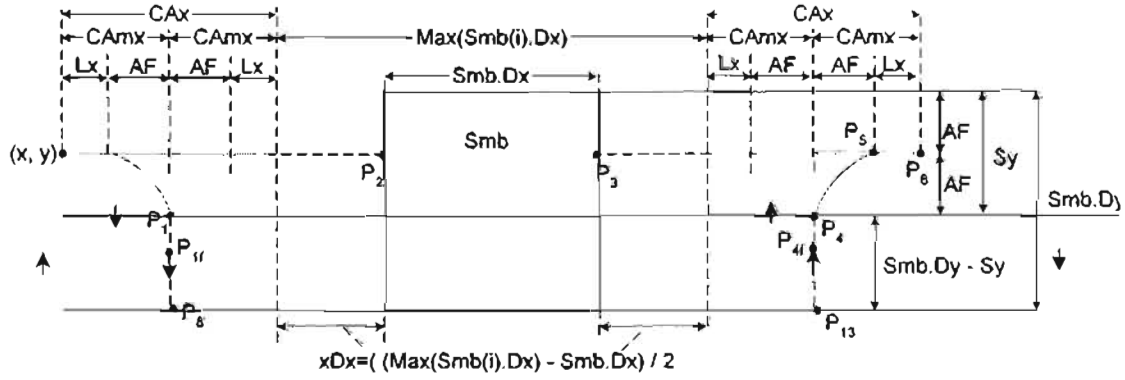
FIGURA 3.4.7

La propiedad **NumSuce**, la agregamos para contar el número de sucesiones que contiene el conjunto de alternativas, **pMaxX** para saber cuál es la longitud más grande de las sucesiones que contiene el conjunto y poder realizar los ajustes a la hora de dibujar.

En la figura 3.4.7 vemos marcadas **AltSup**, **AltInt** y **AltiNf**, que son las figuras asociadas con una alternativa superior, alternativas intermedias y una alternativa inferior. Antes de implementar los métodos de **Medir** y **Pintar**, vamos a ver a detalle qué necesitamos para poder dibujar estas figuras alternativas.

Vamos a ver ahora cómo generamos estas alternativas

AltSup



Instruccciones para dibujar

Linea desde (x,y) hasta P_2 : LinHor(x, y, Cax + xDx)

Arco I desde P1 : Arcol(x + CAmx, y + AF, AF);

Linea desde P_3 hasta P_8 : LinHor(x + Dx - Cax - xDx, ((Max(Smb(i).Dx) - Smb.Dx)/2) + 2LX +2AF)

Arco II desde P5 : Arcol(x + Dx - Lx, y, AF);

Si Smb.Dy>Sy

Linea desde P_1 hasta P_8 : LinVer(x + CAmx, y + AF, Smb.Dy - 2AF

Linea desde P_4 hasta P_{13} : LinVer(x + Dx - CAmx, y + AF, Smb.Dy - 2AF

Dirección hacia la derecha

FlechaAba en el punto P_{11} : FlechaAba(x+CAmx, y+AF+((Smb.Dy - Sy)/2), Fl, Fl)

FlechaArr en el punto P_{41} : FlechaArr(x+Dx-CAmx, y+AF+((Smb.Dy - Sy)/2), Fl, Fl)

Dirección hacia la izquierda

FlechaArr en el punto P_{11} : FlechaArr(x+CAmx, y+AF+((Smb.Dy - Sy)/2), Fl, Fl)

FlechaAba en el punto P_{41} : FlechaAba(x+Dx-CAmx, y+AF+((Smb.Dy - Sy)/2), Fl, Fl)

Si Smb.Dy>Sy

Dirección hacia la derecha

FlechaArr en el punto P_1 : FlechaAba(x+CAmx, y+AF+(Fl/2), Fl, Fl)

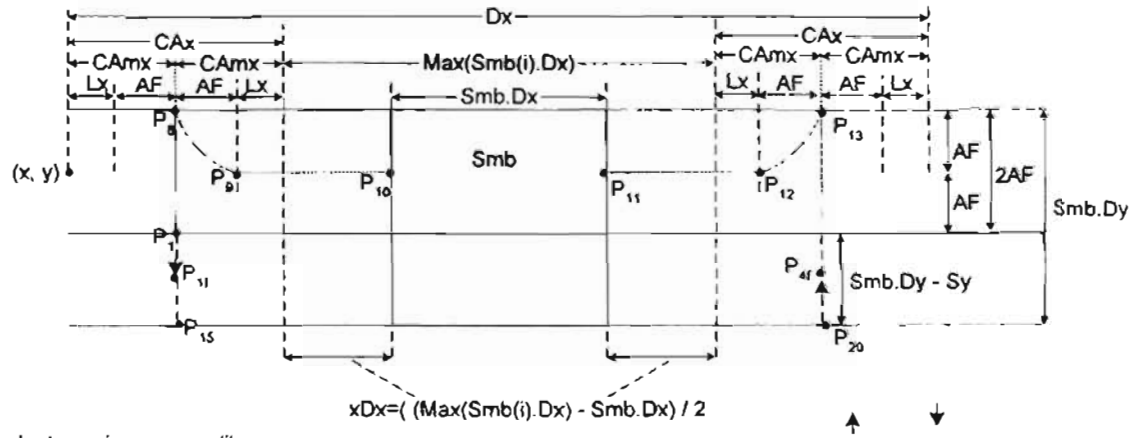
FlechaAba en el punto P_4 : FlechaArr(x+Dx-CAmx, y+AF-Fl, Fl, Fl)

Dirección hacia la izquierda

FlechaAba en el punto P_1 : FlechaArr(x+CAmx, y+AF-Fl, Fl, Fl)

FlechaArr en el punto P_4 : FlechaAba(x+Dx-CAmx, y+AF+(Fl/2), Fl, Fl)

AltIntInf



Instrucciones para dibujar

Linea desde P_9 hasta P_{10} : LinHor(x + Camx + AF, y, Lx + xDx)

Arco III desde P8 : ArcoI(x + CAmx, y - AF, AF);

Linea desde P_{11} hasta P_{12} : LinHor(x + Dx - Cax - xDx, y, xDx + Lx)

Arco IV desde P12 : ArcoIV(x + Dx - Camx -AF, y, AF);

Si es Int

Linea desde P_8 hasta P_{15} : LinVer(x + CAmx, y - AF, Smb.Dy)

Linea desde P_{13} hasta P_{20} : LinVer(x + Dx - CAmx, y - AF, Smb.Dy)

Dirección hacia la derecha

FlechaAba en el punto P_{11} : FlechaAba(x+CAmx, y-AF+(Smb.Dy/2) + (Fl/2), Fl, Fl)

FlechaArr en el punto P_{41} : FlechaArr(x+Dx-CAmx, y-AF+(Smb.Dy/2) -Fl/2, Fl, Fl)

Dirección hacia la izquierda

FlechaAr en el punto P_{11} : FlechaArr(x+CAmx, y-AF+(Smb.Dy/2)-(Fl/2), Fl, Fl)

FlechaAba en el punto P_{41} : FlechaAba(x+Dx-CAmx, y-AF+(Smb.Dy /2)+(Fl/2), Fl, Fl)

FIGURA 3.4.8

Aquí tenemos el detalle para generar estas tres alternativas; ahora sí podemos implementar la clase **tConjAltsSimbs**. Primero el método **Medir**:

```
procedure tConjAltsSimbs.Medir(const xNodo : tTreeNode);
  var  nodoHijo : tTreeNode;
begin
  Dy:=0;
  Dx:=0;
  pMaxX:=0;
  nodoHijo:=xNodo.GetFirstChild;  //Empezamos el recorrido
  if NumSuce>1 then begin //¿Se trata de un conjunto de alternativas?
    while nodoHijo<>nil do //Recorremos hasta que no tenga hijos
```

```
    begin
     //¿Hay una sucesión por medir?
      if tSmb(nodoHijo.data).tipoSimb=tsSuceSimbs then
      begin
       // Medimos el símbolo
        tSmb(nodoHijo.data).Medir(nodoHijo);
        // Actualizamos la longitud con lo que hemos medido
        pMaxX:=max(pMaxX , tSmb(nodoHijo.data).Dx);
        // Incrementamos la altura del conjunto
        Dy:=Dy + tSmb(nodoHijo.data).Dy;
      end;
      // Avanzamos al siguiente nodo hijo
      nodoHijo:=nodoHijo.GetNextSibling;
    end;
    {Finalmente la longitud del conjunto es igual a esta longitud más la
    longitud que necesitamos para dibujar las alternativas}
    Dx:=pMaxX + (2*CAx);
  end else begin
    {Si solo contiene una sucesión, mandamos que se mida y sus medidas serán
    tambien las medidas del conjunto no hay necesidad de dibujar figuras
    alternativas}
    tSmb(nodoHijo.data).Medir(nodoHijo);
    Dx:=tSmb(nodoHijo.data).Dx;
    Dy:=tSmb(nodoHijo.data).Dy;
    pMaxX:=0;
  end;
end;
```

Antes de analizar este método en particular, establecemos las siguientes convenciones:

- En todos nuestros métodos recorremos los nodos del árbol en el orden en que se generaron éstos. En nuestro caso necesitamos visitar el árbol de la siguiente manera:

```
nodoHijo:=xNodo.GetFirstChild;  //Empezamos el recorrido
while nodoHijo<>nil do //Recorremos hasta que no tenga hijos
begin
        //Aquí va el proceso que necesitemos
        nodoHijo:=nodoHijo.GetNextSibling;
 end;
```

- Todos nuestros métodos se aplican a un nodo, de ahí el parámetro **xNodo**; esto es así porque el nodo es el que nos permite recorrer el árbol y conociendo el nodo conocemos el tipo de símbolo. El cual además podemos manejar con flexibilidad debido al polimorfismo.

Aquí vemos ya la utilidad del contador **numSuce**, que es un contador del número de sucesiones que integran al conjunto. Es muy importante saber si se trata de más de una sucesión al medir, ya que necesitamos considerar el espacio que requerimos para dibujar las figuras asociadas a un conjunto de alternativas, que en este caso sería igual a **2 * CAx** como podemos ver en la figura 3.4.8; si sólo se trata de una sucesión no hay necesidad de hacer nada. En el caso de un conjunto, la longitud de un conjunto de alternativas es igual a la longitud máxima de las sucesiones que contiene más **(2*CAx)** y la altura es la suma de la altura de dichas sucesiones. Aprovechamos además para guardar la longitud máxima en la variable **pMaxX**, ya que es útil para dibujar lineas de ajuste para cada sucesión. Cuando el conjunto se reduce a una sola sucesión la longitud y la altura coinciden con los de la sucesión contenida.

```
procedure tConjAltsSimbs.Pintar(const xNodo:tTreeNode;xDir:tDirec; x, y:integer);
  var  nodoHijo     : tTreeNode;
       Yv, Dxa,Dxb  : integer;
       contSuce     : byte; //Solo 255 sucesiones ¿suficiente, no?}
begin
  nodoHijo:=xNodo.GetFirstChild;//Empezamos el recorrido
  Yv:=y;
  if NumSuce>1 then
  begin
    contSuce:=1;
    while nodoHijo<>nil do
    begin
      //¿Hay una sucesión por dibujar?
      if tSmb(nodoHijo.data).tipoSimb=tsSuceSimbs then begin
        // Calculamos el ajuste horizontal
        if pMaxX > tSmb(nodoHijo.data).Dx then
          Dxa :=Round((pMaxX- tSmb(nodoHijo.data).Dx)/2)
        else
          Dxa:=0;

        // Dibujamos las figuras asociadas con la alternativa
        if contSuce=1 then
          OpcSup(xDir, x, Yv, Dxa, tSmb(nodoHijo.data).Dy)
        else begin
          if contSuce=NumSuce then
            OpcIntInf(xDir, x, Yv, Dxa, tSmb(nodoHijo.data).Dy, true)
          else
            OpcIntInf(xDir, x, Yv, Dxa, tSmb(nodoHijo.data).Dy, false);
        end;
```

```
        // Dibujamos la sucesión de símbolos que contiene
        tSmb(nodoHijo.data).Pintar(nodoHijo, xDir, x + CAx + Dxa, Yv);

        //Dibujamos la siguiente sucesión justo debajo de ésta
        Yv:=Yv + tSmb(nodoHijo.data).Dy;

        inc(contSuce);
      end;

      // Buscamos la siguiente sucesión
      nodoHijo:=nodoHijo.GetNextSibling;
    end;
  end else begin
    //Sólo hay una sucesión, que ésta se pinte
    tSmb(nodoHijo.data).Pintar(nodoHijo, xDir, x, y);
  end;
end;
```

Nuevamente **NumSuce** juega un rol importante, ya que nos indica el tipo de alternativa que hay que dibujar asociada a cada sucesión. **pMaxX** nos sirve para dibujar líneas horizontales que nos permiten conectar las figuras asociadas a cada alternativa con el simbolo correspondiente y además permite centrar a éste, de acuerdo a su longitud. También como podemos ver, el punto de inicio lo vamos moviendo de acuerdo a las necesidades del dibujo; por ejemplo cada símbolo empieza justo debajo del anterior; esto lo logramos tomando la misma coordenada de inicio **x**, pero incrementando la coordenada **y** por la altura de cada símbolo involucrado. También, en el caso de una sola sucesión lo único que necesitamos es solicitarle a dicha sucesión que se pinte. A continuación mostramos los métodos asociados con **OpcSup** y **OpcIntInf**, los cuales nos permiten dibujar las figuras asociadas con cada alternativa.

```
procedure tConjAltsSimbs.OpcSup(const xDir:tDirec; x, y, xDx, xDy:integer);
begin
  //Dibujamos la parte izquierda----------------------------------------------
  Dibujar.LinHor(x, y, CAx + xDx);  // Dibujamos línea hirzontal con ajuste
  Dibujar.Arco_I(x + CAmx, y + AF, AF);

  //Dibujamos la parte derecha-----------------------------------------------
  Dibujar.LinHor(x+Dx-CAx-xDx-1, y, CAx + xDx+1); // Dibujamos línea hor.con ajuste
  Dibujar.Arco_II(x + Dx - Lx, y, AF);

  If xDy>Sy then
```

```
  begin
    Dibujar.LinVer(x + CAmx-1, y + AF, xDy -Sy);//Línea izq. vert.con ajuste
    Dibujar.LinVer(x+Dx-CAmx, y+AF, xDy-Sy); //Línea der. vert. con ajuste

    If (xDir=dDer) then begin
      Dibujar.FlechaAba(x + CAmx-1, y + AF + round((xDy-Sy)/2), Fl, Fl);
      Dibujar.FlechaArr(x+Dx-CAmx, y + AF + round((xDy-Sy)/2), Fl, Fl);
    end else begin
      Dibujar.FlechaArr(x + CAmx-1, y + AF + round((xDy-Sy)/2), Fl, Fl);
      Dibujar.FlechaAba(x+Dx-CAmx, y + AF + round((xDy-Sy)/2), Fl, Fl);
    end;
  end else begin
    if (xDir = dDer) then begin
      Dibujar.FlechaAba(x + CAmx-1, y + AF+round(Fl/2), Fl, Fl);
      Dibujar.FlechaArr(x+Dx-CAmx, y+AF-Fl, Fl, Fl);
    end else begin
      Dibujar.FlechaArr(x + CAmx-1, y + AF-Fl, Fl, Fl);
      Dibujar.FlechaAba(x+Dx-CAmx, y+AF+round(Fl/2), Fl, Fl);
    end;
  end;
end;

procedure tConjAltsSimbs.OpcIntInf(const xDir:tDirec; x, y, xDx, xDy:integer;
                                                                    xEsInf:boolean);
begin
  //Dibujamos la parte izquierda----------------------------------------
  Dibujar.LinHor(x+CAmx+AF, y, Lx + xDx);// Dibujamos línea horiz.con ajuste
  Dibujar.Arco_III(x + CAmx-1, y - AF, AF+1);


  //Dibujamos la parte derecha----------------------------------------
  Dibujar.LinHor(x+Dx-CAx-xDx, y, xDx + Lx);// Dibujamos línea vert. con ajuste
  Dibujar.Arco_IV(x + Dx - CAmx - AF, y+1, AF+1);


  If not xEsInf then
  begin
    // La altura de la línea vertical siempre es igual a la alt. del símbolo
    Dibujar.LinVer(x + CAmx-1, y - AF, xDy);
    Dibujar.LinVer(x + Dx - CAmx, y - AF, xDy);

    If xDy>Sy then begin
      If (xDir=dDer) then begin
        Dibujar.FlechaAba(x + CAmx-1, y - AF + round(xDy/2)+round(Fl/2), Fl, Fl);
```

```
        Dibujar.FlechaArr(x + Dx - CAmx, y - AF + round(xDy/2)-round(Fl/2), Fl, Fl);
      end else begin
        Dibujar.FlechaArr(x + CAmx-1, y - AF + round(xDy/2)-round(Fl/2), Fl, Fl);
        Dibujar.FlechaAba(x + Dx - CAmx, y - AF + round(xDy/2)+round(Fl/2), Fl, Fl);
      end;
    end;
  end;
end;
```

En la figura 3.4.8 podemos ver la interpretación geométrica de cada dibujo, sin embargo vale la pena hacer las siguientes aclaraciones:

- El método **OpcIntInf** cubre el dibujo de una sucesión de símbolos intermedia y de una sucesión de símbolos inferior. Lo hicimos así porque la única diferencia entre una y otra son las líneas verticales adicionales de la opción intermedia. Por eso utilizamos la bandera **xEsInf** que nos indica si se trata de la última sucesión.
- Hasta ahora no hemos hablado de la dirección del dibujo en las figuras, la cual se indica por medio de flechas. No lo habíamos hecho porque la dirección no influye en el tamaño de las figuras sólo influye en el orden en que se dibujan, cuando se trata de sucesiones de símbolos. La dirección por defecto es hacia la derecha, cuando la dirección es hacia la izquierda el dibujo de las flechas se invierte; en algunas ocasiones además cambia la posición.

En la figura 3.4.8 podemos ver la interpretación geométrica, además las posiciones y medidas que implementamos en estos métodos.

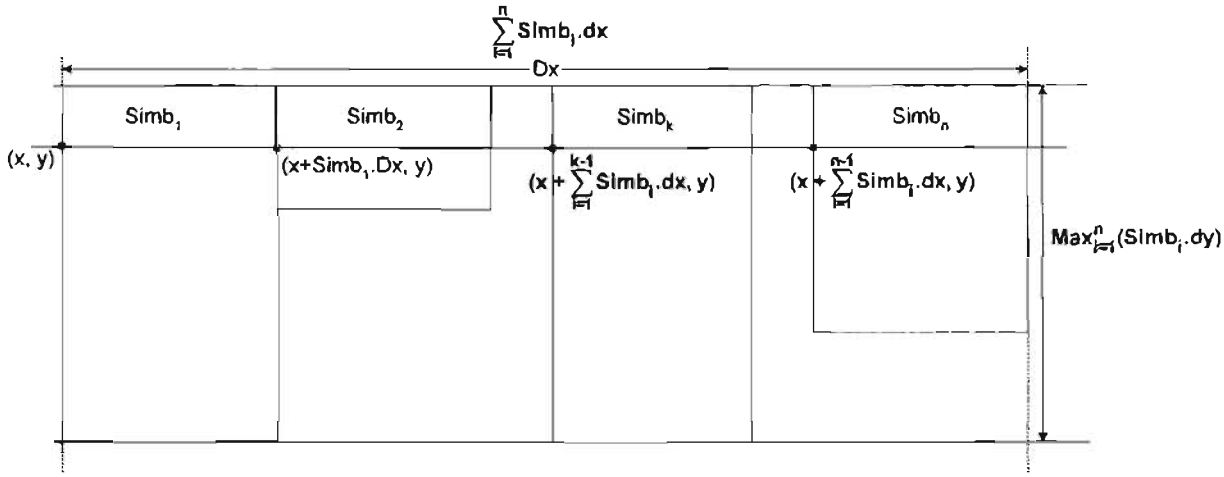
Tenemos por último el método

```
function tConjAltsSimbs.LeerCadAsoc(const xNodo : tTreeNode):string;
  var  nodoHijo : tTreeNode;
begin
  result:='';
  nodoHijo:=xNodo.GetFirstChild;
  while nodoHijo<>nil do
  begin
    result:=result + tSmb(nodoHijo.data).LeerCadAsoc(NodoHijo);
    nodoHijo:=nodoHijo.GetNextSibling;
  end;
end;
```

Construye la cadena asociada con este conjunto de alternativas, nótese que el recorrido del árbol sigue siendo el mismo.

Analizamos ahora **SuceSimbs**, veamos cómo se dibujan una sucesión de símbolos:

SuceSimbs



Ésta sería la manera convencional de dibujar sucesiones de símbolos; sin embargo hemos incluido un concepto al que hemos llamado salto de diagrama que va a permitir al usuario del sistema marcar el lugar en el que desea que el diagrama dé un salto hacia abajo, para los casos en los que los diagramas no quepan porque su dibujo se extiende mucho hacia la derecha. Esta clase al ser una sucesión de símbolos, es la encargada de interceptar los saltos y efectuar el salto en el dibujo como se muestra en la figura 3.4.9.

La interfaz para nuestra clase es:

```
tSuceSimbs = class(tSmb)
  private
    procedure SaltoDiagrIni(const xDir:tDirec; x, y,xDx, xDy:integer;
                                                    xEsPrimer:boolean);
    procedure SaltoDiagrFin(const xDir:tDirec; x, yi, yf, xDx, xDy:integer);
  public
    HaySaltos : boolean;
    procedure Pintar(const xNodo:tTreeNode;xDir:tDirec;x, y:integer);override;
    procedure Medir(const xNodo : tTreeNode);override;
    function  LeerCadAsoc(const xNodo : tTreeNode):string;override;
end;
```

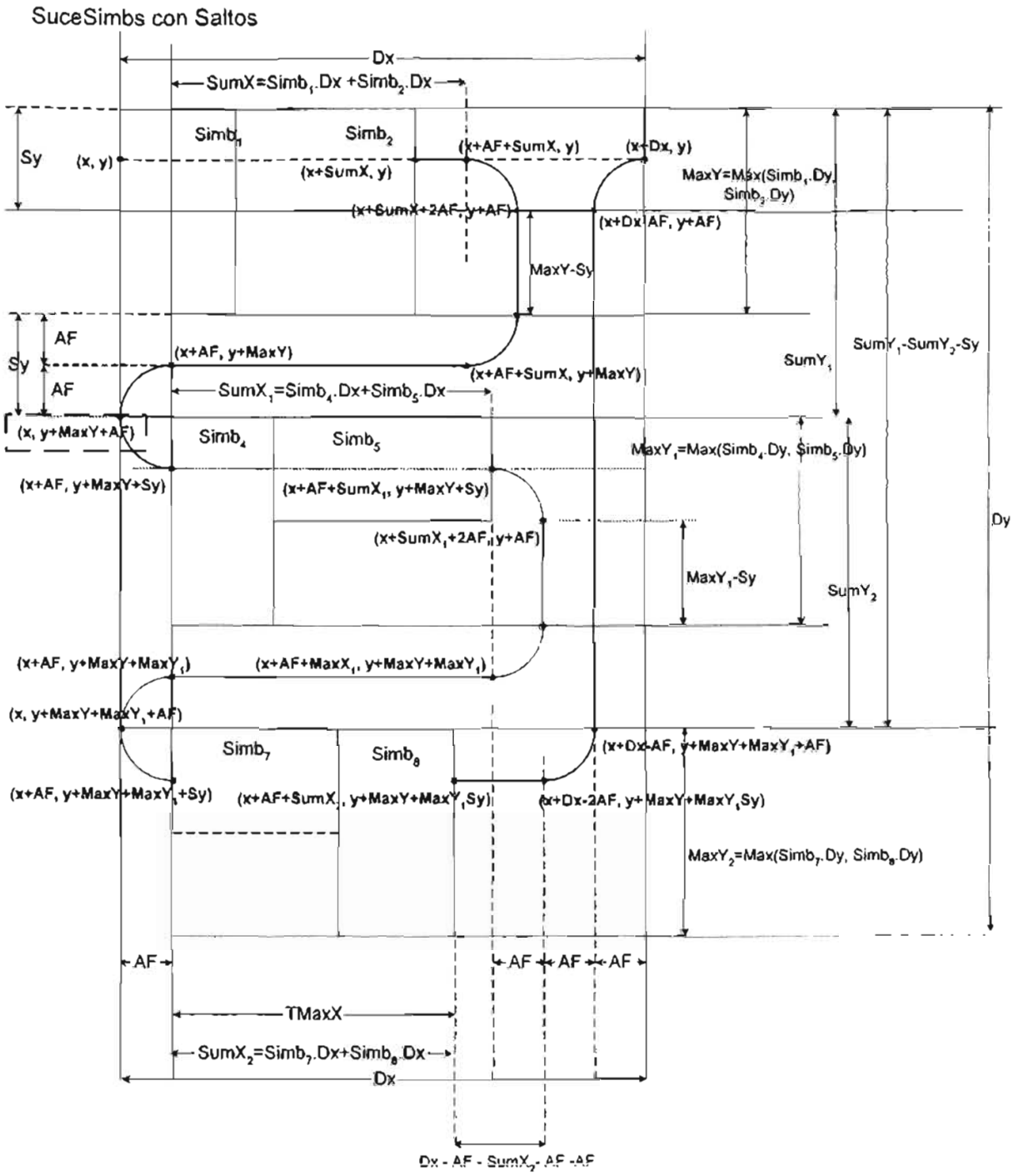


FIGURA 3.4.9

La propiedad **HaySaltos** es una bandera que nos indica si en una sucesión hay saltos.

Analizamos primero el método **Medir**:

```
procedure tSuceSimbs.Medir(const xNodo : tTreeNode);
  var  nodoHijo : tTreeNode;
       tMaxX, tSumY, sumX, maxY:integer;
begin
  MostrarNodoSimbolo('SuceSimbs.Medir:', xNodo);
  nodoHijo:=xNodo.GetFirstChild; //Empezamos el recorrido
  Dx:=0;
  Dy:=0;
  tMaxX:=0;  // Longitud total
  tSumY:=0;  // Altura total
  sumX:=0;   // Longitud parcial:Longitud de símbolos que entran en un salto
  maxY:=0;   // Altura parcial:Altura de símbolos que entran en  un salto
  HaySaltos:=false;//Bandera para saber si hay saltos
  while ((nodoHijo<>nil) and (nodoHijo.data<>nil)) do
  begin
   //¿Es un símbolo que indica un salto en el diagrama?
    if tSmb(nodoHijo.data).TipoSimb = tsSaltoDiagr then
    begin
     // Encendemos la bandera de saltos
      haySaltos:=true;
     { Acumulamos la altura total con la altura parcial que se haya
      calculado con los símbolos que entran en este salto}
      tSumY:=tSumY + maxY + Sy;

     {Comparamos la longitud total máxima calculada contra la longitud
     de los símbolos que entraron en este salto y tomamos la más larga}
      tMaxX:=max(tMaxX, sumX);
     {Inicializamos la longitud y la altura parcial para empezar a
     * calcularlas con el siguiente grupo de símbolos}
      sumX:=0;
      maxY:=0;
    end else begin
     // Nótese que los cálculos parciales coinciden con una sucesión normal
     // ¿Es un símbolo que entra en la parte superior de una repetición?
      if not tSimb(nodoHijo.data).EsRepSupIni then begin
       // Si no es así, medimos al símbolo y :
        tSmb(nodoHijo.data).Medir(nodoHijo);
       // Acumulamos  la longitud parcial con este símbolo
        sumX:=sumX + tSmb(nodoHijo.data).Dx;
       { Comparamos la longitud del símbolo contra la de los otros símbolos
       * que entran en este salto}
        maxY:=max(maxY , tSmb(nodoHijo.data).Dy);
```

```
      end;
    end;
    // Avanzamos al siguiente símbolo
    nodoHijo:=nodoHijo.GetNextSibling;
  end;

  tSumY:=tSumY + maxY ; // Culminamos la acumulación de alturas de los símbolos
  {Tomamos la longitud máxima de todos los grupos formados}
  tMaxX:=max(tMaxX, sumX);

  if haySaltos then begin
    {Si hay saltos agregamos la longitud necesaria para dibujar las vueltas de
    los saltos de diagrama}
    Dx:=tMaxX +(2 * Sy);
  end else
    Dx:=tMaxX ;   //No incrementamos nada pues es una sucesión normal
  // A la altura no le hacemos nada tomamos la que esté calculada
  Dy:=tSumY;
end;
```

Al inicio del recorrido para medir suponemos que no hay saltos, **tMaxX** representa la longitud máxima de la sucesión y se calcula tomando en cada ocasión la longitud máxima de cada grupo de símbolos que vamos obteniendo; mientras que la longitud parcial de un grupo a la cual denotamos por **sumX**, es igual a la suma de las longitudes de los símbolos que contiene . Por otra parte **tSumY** es la suma de las alturas parciales de cada grupo de símbolos que entran en un salto, la suma parcial de cada grupo es igual a la altura máxima de los símbolos que entran en este grupo. Al finalizar de medir, si hubo saltos tenemos que sumar a la longitud total la longitud necesaria para dibujar los saltos.

Para pintar utilizamos el método:

```
procedure tSuceSimbs.Pintar(const xNodo : tTreeNode; xDir:tDirec; x, y:integer);
  var nodoHijo   : tTreeNode;
    Xv, Yv, sumX, maxY, tSumY  :integer;
    esPrimer:boolean;
begin
  { Es en uno de los dos aspectos en los que influye la dirección, cuando
  * la dirección es hacia la derecha el recorrido es normal, cuando es hacia
  * la izquierda el recorrido es del último al primero}
  if xDir = dDer then
    nodoHijo:=xNodo.GetFirstChild
  else
```

```
  nodoHijo:=xNodo.GetLastChild;

sumX:=0;     // Longitud parcial
maxY:=0;     // Altura parcial
tSumY:=0;  // Altura total
Xv:=x;
Yv:=y;

esPrimer:=true;
while ((nodoHijo<>nil) and (nodoHijo.data<>nil)) do
begin
 //¿Es un símbolo que indica un salto en el diagrama?
  if tSmb(nodoHijo.data).TipoSimb = tsSaltoDiagr then
  begin

   // Dibujamos el salto
    SaltoDiagrIni(xDir, x, Yv, SumX, maxY, EsPrimer );

   { Acumulamos la altura total con la altura parcial que se haya
    calculado con los símbolos que entran en este salto}
    tSumY:=tSumY + maxY + Sy;

   // El siguiente grupo de símbolos inicia debajo del que acaba de pasar.
    Xv:=x+AF;
    Yv:=Yv + maxY + Sy;
    maxY:=0;      // Inicializamos los parciales del nuevo grupo de símbolos
    sumX:=0;
    EsPrimer:=false;
  end else begin
   // ¿Es un símbolo que entra en la parte superior de una repetición?
    if not tSimb(nodoHijo.data).EsRepSupIni then begin
     // No, que el símbolo se dibuje
      tSmb(nodoHijo.data).Pintar(nodoHijo, xDir, Xv, Yv);
     //El siguiente se Pinta a la derecha del que acaba de pasar
      Xv:=Xv + tSmb(nodoHijo.data).Dx;
     //Acumulamos la longitud de este grupo de símbolos
      sumX:=sumX + tSmb(nodoHijo.data).Dx;
     //Calculamos la altura máxima de este grupo para poder dar la vuelta
      maxY:=max(maxY , tSmb(nodoHijo.data).Dy);
    end;
  end;

 // Continuamos el recorrido según lo indique la dirección
  if xDir = dDer then
```

```
      nodoHijo:=nodoHijo.GetNextSibling
    else
      nodoHijo:=nodoHijo.GetPrevSibling;
  end;
  {Ya finalizamos el recorrido, si hubo saltos dibujamos el final para que la
  sucesión continúe en donde debe}
  if haySaltos then begin
    // Nótese que tSumY no contienen la altura de la última sucesión
    SaltoDiagrFin(xDir, x, y, Yv, sumX, tSumY);
  end;
end;

procedure tSuceSimbs.SaltoDiagrIni(const xDir:tDirec; x, y, xDx, xDy:integer;
                                   xEsPrimer:boolean);
begin
  ind:=6;
  ActualizarFuenteLinea;

  // Dibujamos la vuelta de la derecha
    If xEsPrimer then
      Dibujar.LinHor(x + xDx, y, AF+1);
    Dibujar.Arco_I(x + xDx + AF + AF +1, y+AF, AF);
    Dibujar.LinVer(x + xDx + AF + AF, y+AF, xDy - Sy);
    Dibujar.Arco_IV(x + xDx + AF, y+xDy+1, AF+1);
  // Dibujamos la línea horizontal
    Dibujar.LinHor(x + AF, y+ xDy, xDx);

  // Dibujamos la dirección
    if xDir=dDer then
      Dibujar.FlechaIzq(x+AF+round(xDx/2), y+xDy, fl, fl)
    else
      Dibujar.FlechaDer(x+AF+round(xDx/2), y+xDy, fl, fl);

  // Dibujamos la vuelta de la izquierda
    Dibujar.Arco_II(x +AF, y+xDy, AF);
    //DibujarArco_III(x, y+xDy+AF+1, AF+1);
    Dibujar.Arco_III(x, y+xDy+AF+1, AF);
end;

procedure tSuceSimbs.SaltoDiagrFin(const xDir:tDirec;x,yi,yf,xDx,xDy:integer);
begin
  ind:=6;
  ActualizarFuenteLinea;
```

```
// Dibujamos la línea de ajuste de la derecha del ultimo grupo de salto
Dibujar.LinHor(x + xDx +AF, yf, Dx - AF - xDx - AF -AF);
// Dibujamos la vuelta de la derecha
Dibujar.Arco_IV(x + Dx -Sy, yf+1, AF+1);
Dibujar.LinVer(x +Dx-AF, yi+AF, xDy - Sy);
// Dibujamos la dirección
if xDir=dDer then
   Dibujar.FlechaArr(x+Dx-AF,yi+AF+round((xDy-Sy)/2), fl, fl)
 else
   Dibujar.FlechaAba(x+Dx-AF,yi+AF+round((xDy-Sy)/2), fl, fl);

 Dibujar.Arco_II(x + Dx, yi, AF);
end;
```

El recorrido de los nodos al dibujar es el mismo que seguimos para medir; cuando encontramos un salto debemos dibujar el salto hasta el siguiente grupo de símbolos, lo que identificamos como **SaltoDiagrIni**. Si el símbolo no representa un salto debemos proceder de manera normal dibujando la sucesión de símbolos, la excepción se encunetra cuando el símbolo en cuestión es un símbolo que entra en la parte superior de una repetición; lo cual sabemos si la propiedad **esRepSupIni**. La mayor influencia del parámetro dirección, que nos indica la dirección del flujo del diagrama, se presenta dibujamos sucesiones de símbolos, ya que la dirección determina el orden en que visitamos los nodos asociados a estos símbolos; cuando la dirección es hacia la derecha recorremos los nodos de manera convencional de izquierda a derecha (desde el primero hacia el siguiente); pero, si la dirección es hacia la izquierda el orden de visita de los nodos se invierte y los visitamos de derecha a izquierda (desde el último hacia al anterior). El único otro momento en que la dirección influye se presenta al dibujar las flechas. Por último si hay saltos, dibujamos el final de un grupo de saltos al cual identificamos por **SaltoDiagrFin**.

El método LeerCadenaAsoc para las sucesiones es:

```
function tSuceSimbs.LeerCadAsoc(const xNodo : tTreeNode):string;
  var  nodoHijo : tTreeNode;
begin
  result:='';
  nodoHijo:=xNodo.GetFirstChild;
  while ((nodoHijo<>nil) and (nodoHijo.data<>nil)) do
  begin
    result:=result + tSmb(nodoHijo.data).LeerCadAsoc(NodoHijo);
    nodoHijo:=nodoHijo.GetNextSibling;
  end;
```

```
end;
```

Ahora analizamos la clase **tSimb**, su interfaz es:

```
tSimb = class(tSmb)
  private
    procedure OpcSup(const xDir:tDirec; x, y:integer);
    procedure OpcInf(const xDir:tDirec; x, y:integer);
    procedure RepSupSin(const xDir:tDirec; x, y:integer);
    procedure RepSupCon(const xDir:tDirec; x, y, xDx, xDy:integer);
    procedure RepInf(const xDir:tDirec; x, y, xDx, xDy:integer);
  public
    EsRepSupIni : boolean;
    procedure Pintar(const xNodo:tTreeNode;xDir:tDirec;x, y:integer);override;
    procedure Medir(const xNodo : tTreeNode);override;
    function  LeerCadAsoc(const xNodo : tTreeNode):string;override;
end;
```

Si recordamos la definición de **Simbolo** tenemos:

```
Simbolo =  '#'                                          |
           SimboloTerminal                              |
           ClaseSintactica                              |
           '(' ConjuntoAlternativasSimbolos ')'         |
                '[' ConjuntoAlternativasSimbolos ']'    |
           '{' ConjuntoAlternativasSimbolos '}'         .
```

Esta clase se encarga del manejo de las partes opcionales y las partes repetitivas, por eso tenemos los métodos **OpcSup**, **OpcInf**, **RepSup**, **RepSupCon** y **RepSupSin**. Para esta clase no tenemos asociada una única manera de dibujar ni de medir; sino tantas maneras de medir y dibujar como las que representan **SimboloTerminal**, ***ClaseSintáctica***, "(" , "[" parte opcional y "{" parte repetitiva. Por lo tanto el método para medir es:

```
procedure tSimb.Medir(const xNodo : tTreeNode);
  var nodoHijo : tTreeNode;
  var SmbAso : tTreeNode;
begin
  MostrarNodoSimbolo('Simb.Medir:', xNodo);
  Dx:=0;
  Dy:=0;
```

```
// Iniciamos el recorrido
nodoHijo:=xNodo.GetFirstChild;
if ((nodoHijo<>nil) and (nodoHijo.data<>nil)) then begin
  case tSmb(nodoHijo.data).tipoSimb of
    tsIzqConjAlts:
    begin
     // El siguiente es el símbolo que hay que medir
      nodoHijo:=nodoHijo.GetNextSibling;
      if ((nodoHijo<>nil)and (nodoHijo.data<>nil)) then
      begin
        MostrarNodoSimbolo('Simb.IzqConjAlts.Medir:', xNodo);
        tSmb(nodoHijo.data).Medir(nodoHijo);
        Dx:=tSmb(nodoHijo.data).Dx;
        Dy:=tSmb(nodoHijo.data).Dy;
      end;
     //No nos interesa visitar el siguiente porque es ")"
    end;
    tsIzqPOpc:
    begin
      nodoHijo:=nodoHijo.GetNextSibling;
      if ((nodoHijo<>nil)and (nodoHijo.data<>nil)) then
      begin
        MostrarNodoSimbolo('Simb.IzqPOpc.Medir:', xNodo);
        tSmb(nodoHijo.data).Medir(nodoHijo);
       { Obviamente aparte del símbolo necesitamos el espacio para dibujar las
       * figuras correspondientes a una parte opcional}
        Dx:=CAx + tSmb(nodoHijo.data).Dx + CAx;
        Dy:=Sy + tSmb(nodoHijo.data).Dy;
       //No nos interesa visitar el siguiente porque es "]"
      end;
    end;
    tsIzqPRepeti:
    begin
     // Tomamos el símbolo asociado a esta repetición
      SmbAso:=tIzqPRepeti(nodoHijo.data).SimbAsoc;
      MostrarNodoSimbolo('Simb.Aso.IzqPRepeti.Medir:', xNodo);
     //¿Hay símbolo asociado a la parte superior de la repetición?
      if ((SmbAso<>nil) and (SmbAso.data<>nil))  then
      begin
        if tSimb(SmbAso.Data).EsRepSupIni then
        begin
          {Si, hay símbolo asociado a la parte superior de la repetición,
         * por lo tanto la parte superior contiene un símbolo que hay que
         pintar}
```

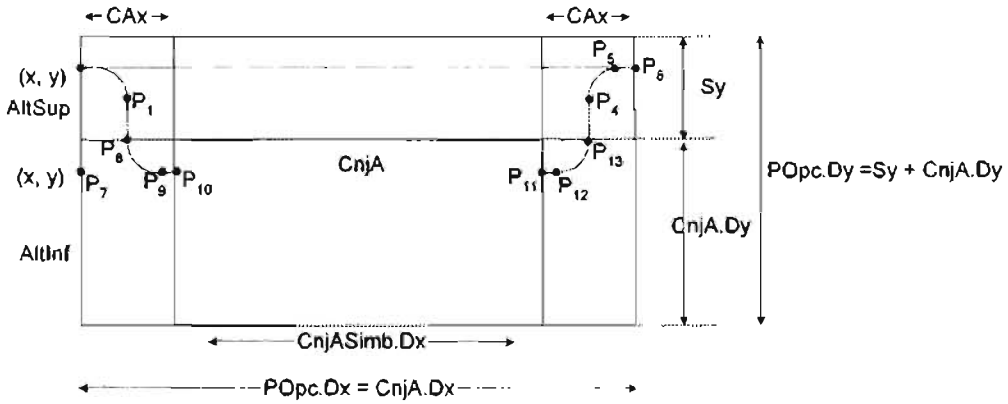
```
    //Para poder medirlo apagamos la bandera de que es asociado
    tSimb(SmbAso.Data).EsRepSupIni:=false;
    //Medimos el símbolo asociado con la parte superior
    tSimb(SmbAso.Data).Medir(SmbAso);
    //Apagamos la bandera para no medirlo en aquí y en otro lado
    tSimb(SmbAso.Data).EsRepSupIni:=true;

    { Inicializamos la longitud y la altura con las medidas de este
   * símbolo}
    Dx:=tSimb(SmbAso.Data).Dx ;
    Dy:=tSimb(SmbAso.Data).Dy ;

    // Ahora sí tomamos el símbolo de la parte inferior
    nodoHijo:=nodoHijo.GetNextSibling;
    if ((nodoHijo<>nil)and (nodoHijo.data<>nil)) then
    begin
      MostrarNodoSimbolo('Simb.Aso.IzqPRepeti.Inferior.Medir:', xNodo);

      // Lo medimos
      tSmb(nodoHijo.data).Medir(nodoHijo);
      //La longitud obviamente es la mayor entre el sup. y el inf.
      Dx:=max(Dx, tSmb(nodoHijo.data).Dx);
      //La altura se acumula
      if tSmb(nodoHijo.data).Dy<>0 then
        Dy:=Dy+tSmb(nodoHijo.data).Dy
      else
        Dy:=Dy+Sy;
      { Agregamos el espacio para dibujar las figuras que corresponden
     * a una parte repetitiva}
      Dx:=Dx + (2*RPx);
    end;
  end;
end else begin
  { Es una repetición normal sin símbolo asociado a la parte
 * superior por lo tanto tomamos el símbolo que corresponde a la
 * parte inferior}
  nodoHijo:=nodoHijo.GetNextSibling;
  if ((nodoHijo<>nil)and (nodoHijo.data<>nil)) then
  begin
    MostrarNodoSimbolo('Simb.IzqPRepeti.Inferior.Medir:', xNodo);
   //Medimos el símbolo
    tSmb(nodoHijo.data).Medir(nodoHijo);
    Dx:=tSmb(nodoHijo.data).Dx + (2*RPx);
    Dy:=tSmb(nodoHijo.data).Dy + Sy;
```

```
          end;
        end;
      end;
      else begin
        if ((nodoHijo<>nil)and (nodoHijo.data<>nil)) then
        begin
          MostrarNodoSimbolo('Simb.Otro.Medir:', xNodo);
         // Cualquier otro símbolo lo medimos
          tSmb(nodoHijo.data).Medir(nodoHijo);
         //Y asignamos las medidas
          Dx:=tSmb(nodoHijo.data).Dx;
          Dy:=tSmb(nodoHijo.data).Dy;
        end;
      end;
    end;
  end;
end;
```

Como podemos ver el método procede según el tipo de símbolo; cuando encuentra *"("* no hace nada especial, simplemente le asigna las medidas del siguiente conjunto de alternativas contenidas entre *"("* y *")"*.

Si encuentra el inicio de una parte opcional, obviamente a las medidas del conjunto de alternativas le agrega el espacio necesario para dibujar el inicio y final de la parte opcional.

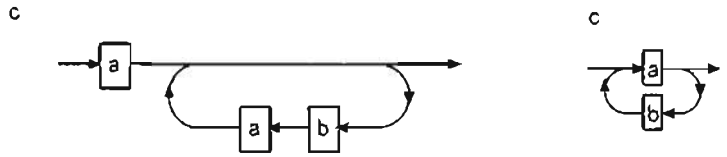
POpc



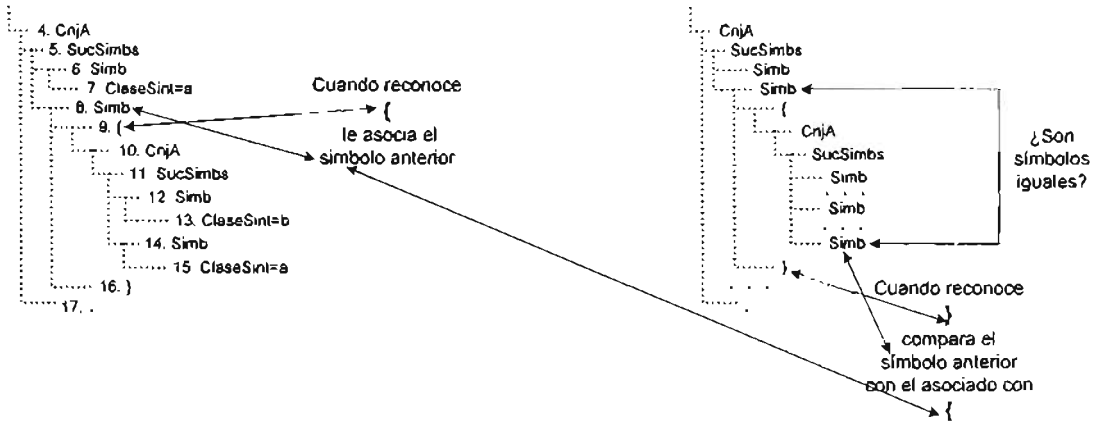
Llegó el momento de que aclaremos a detalle cómo resolvemos los casos de las repeticiones cuando el símbolo anterior a la misma coincide con el último símbolo que se

encuentra dentro de la repetición. En estos casos deseamos que este símbolo formeparte de la parte superior de la repetición.

Supongamos que tenemos la siguiente regla de producción: *a* { *c a*}. Podemos dibujarla como el dibujo de la izquierda sin considerar el hecho de que el símbolo antes de iniciar la repetición es el mismo que el último que contiene la repetición. O más apropiadamente podemos dibujarla como aparece en la parte de la derecha.



Obviamente el reconocimiento y el dibujo se dificultan pero esta consideración simplifica y hace más elegantes nuestros diagramas. Pero ¿cómo resolvemos este detalle? La solución es muy sencilla. Si analizamos el árbol sintáctico de esta producción tenemos:



Por lo tanto cuando reconocemos simbolos en el método AsocSimb de **tAnalizadorSintactico**, tenemos:

```
tsSimb      :
begin
  n.Data:=tSimb.Create(xTipoSimb);
 // Creamos un apuntador al símbolo inmediato anterior
  SimbAntNodo:=n;
end;
```

En el mismo método cuando reconocemos:

```
tsIzqPRepeti :
begin
  n.Data:=tIzqPRepeti.Create(xTipoSimb);
 //Obtenemos al hermano de nuestro padre
  If (SimbAntNodo <> nil) and (not SimbAntNodo.IsFirstNode)then
  begin
    if not SimbAntNodo.IsFirstNode then
    begin
     // Obtenemos símbolo anterior que puede entrar en parte superior
      SimbIni:=SimbAntNodo.GetPrevSibling;
      if ((SimbIni<>nil) and (tSmb(SimbIni.data).TipoSimb<>tsSaltoDiagr)) then
     begin
        pila.Push(SimbIni);
        tIzqPRepeti(n.Data).SimbAsoc:=SimbIni;
        tSimb(SimbIni.Data).EsRepSupIni:=false;
      end else begin
        pila.Push(nil);
        tIzqPRepeti(n.Data).SimbAsoc:=nil;
      end;
    end;
  end else begin
    pila.Push(nil);
    tIzqPRepeti(n.Data).SimbAsoc:=nil;
  end;
end;
```

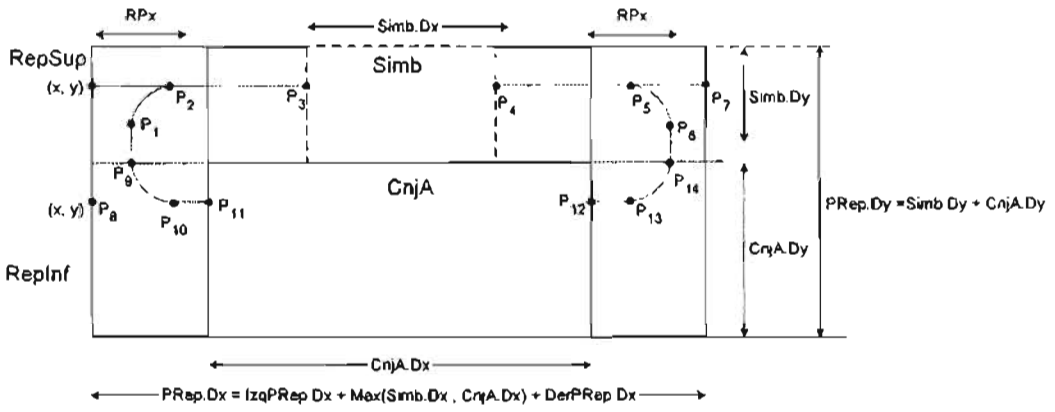
Y cuando reconocemos:

```
tsDerPRepeti :
begin
  n.Data:=tDerPRepeti.Create(xTipoSimb);
  //Obtenemos el CnjA asociado con la repetición
  SimbFin:=n.getPrevSibling;
  //Obtenemos la sucesión de símbolos asociada con la repetición
  SimbFin:=SimbFin.getFirstChild;
  //Obtenemos el último símbolo asociado con la sucesión
  SimbFin:=SimbFin.getLastChild;
  {Sacamos de la pila el símbolo asociado con el que tenemos que
```

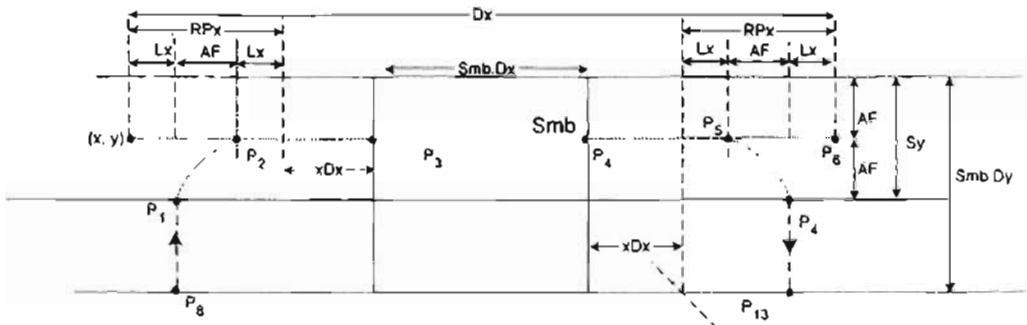
```
    * compararlo. Necesitamos la pila porque el proceso es recursivo, con
    * la pila resolvemos el apareamiento de símbolos}
    SimbIni:=pila.Pop;
    if SimbIni<>nil then begin
      {Leemos las cadenas asociadas a estos símbolos. El símbolo puede
      * contener a su vez cadenas complicadas de símbolos}
      if tSmb(SimbIni.Data).LeerCadAsoc(SimbIni)=tSmb(SimbFin.Data).
                                                    LeerCadAsoc(SimbFin) then
      begin
        {Marcamos ambos símbolos para saber que forman parte de la
        parte superior de una repetición}
        tSimb(SimbIni.Data).EsRepSupIni:=true;
        tSimb(SimbFin.Data).EsRepSupIni:=true;
      end else begin
        // Desmarcamos al símbolo como asociado a una repetición
        x:=n.getPrevSibling;
        x:=x.getPrevSibling;
        tIzqPRepeti(x.Data).SimbAsoc:=nil;
      end;
    end;
  end;
```

Con esto queda claro porque necesitamos del método **LeerCadAsoc**, que construye las cadenas asociadas con los símbolos; con ello podemos comparar si dos símbolos son iguales no importando la complejidad que puedan tener. Para completar el método pintar incluimos la interpretación gráfica de como medimos y pintamos repeticiones:

PRep



PRepSup



Instruccciones para dibujar

Con símbolo asociado:

xDx=((Dx -Simb.Dx - Rpx -RPx)/2);

Linea desde (x,y) hasta P_3 : LinHor(x, y, RPx + xDx)

Arco II desde P2 : Arcol(x + Lx + AF, y , AF);

Linea desde P_1 hasta P_8 : LinVer(x + Lx, y + AF, Smb.Dy - Sy)

Linea desde P_4 hasta P_6 : LinHor(x + Dx - RPx - xDx,y, RPx + xDx)

Arco I desde P4 : Arcol(x + Dx - Lx , y + AF, AF);

Linea desde P_4 hasta P_{13} : LinVer(x + Dx - Lx, y + AF, Smb.Dy - Sy)

Dirección hacia la derecha

FlechaArr : FlechaArr(x+Lx, y + AF + (Smb.Dy-Sy)/2) - Fl, Fl, Fl)

FlechaAba : FlechaAba(x+Dx-CAmx, y + AF + ((Smb.Dy-Sy)/2)+ (Fl/2), Fl, Fl)

Dirección hacia la izquierda

FlechaAba : FlechaAba(x+CAmx, y + AF + ((Smb.Dy-Sy)/2)+ (Fl/2), Fl, Fl)

FlechaArr : FlechaArr(x+Dx-CAmx, y + AF + ((Smb.Dy-Sy)/2)- Fl, Fl, Fl)

Sin símbolo asociado:

Linea desde (x,y) hasta P_6 : LinHor(x, y, Dx)

Arco II desde P2 : Arcol(x + Lx + AF, y , AF);

Arco I desde P4 : Arcol(x + Dx - Lx , y + AF, AF);

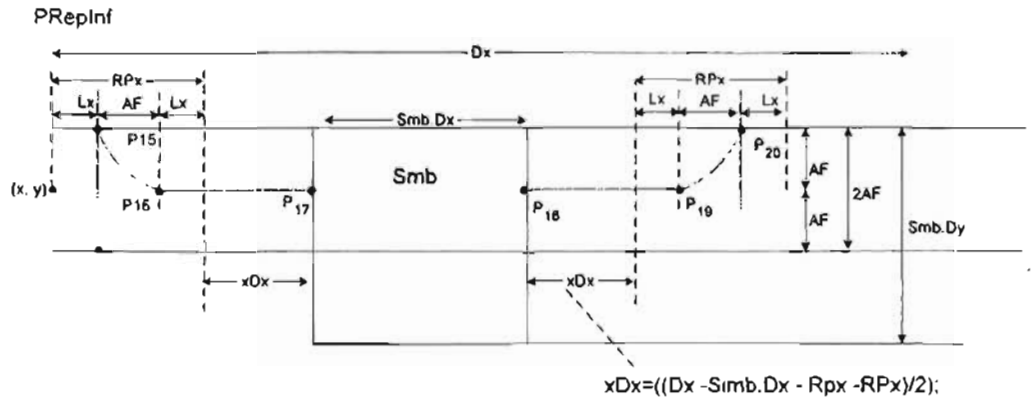
Dirección hacia la derecha

FlechaDer(x+(Dx/2),y,Fl,Fl)

Dirección hacia la izquierda

Flechalzq(x+(Dx/2),y,Fl,Fl);

FIGURA 3.4.10



Instruccciones para dibujar

Arco III desde P15 : ArcoI(x + Lx + AF, y - AF, AF);

Linea desde P_{16} hasta P_{17} : LinHor(x + Lx +AF, y, xDx+ Lx)

Linea desde P_{18} hasta P_{19} : LinHor(x +Dx-RPx-xDx, y, xDx+Lx)

Arco IV desde P19 : ArcoIV(x + Dx -Lx -AF, y, AF);

FIGURA 3.4.11

Por último incluimos el método **pintar** que con las explicaciones dadas y las figuras proporcionadas no merece mayores comentarios.

```
procedure tSimb.Pintar(const xNodo : tTreeNode; xDir:tDirec; x, y:integer);
  var nodoHijo : tTreeNode;
     Xv, Yv, Dxa, Dxb : integer;
     SmbAso:tTreeNode;
begin
  MostrarNodoSimbolo('Simb.Pintar:', xNodo);
  //Iniciamos el recorrido
  nodoHijo:=xNodo.GetFirstChild;
  Xv:=x;
  Yv:=y;
  if ((nodoHijo<>nil)and (nodoHijo.data<>nil)) then begin
  case tSmb(nodoHijo.data).tipoSimb of
    tsIzqConjAlts:
    begin
     // Obtenemos el conjunto de alternativas
      nodoHijo:=nodoHijo.GetNextSibling;
      if ((nodoHijo<>nil)and (nodoHijo.data<>nil)) then
```

```
  begin
    MostrarNodoSimbolo('Simb.IzqConjAlts.Pintar:', xNodo);
   // Dibujamos el conjunto de alternativa
    tSmb(nodoHijo.data).Pintar(nodoHijo, xDir, x, y);
  end;
end;
tsIzqPOpc:
begin
 // Obtenemos el símbolo asociado a la opción
  nodoHijo:=nodoHijo.GetNextSibling;
  if ((nodoHijo<>nil)and (nodoHijo.data<>nil)) then begin
    MostrarNodoSimbolo('Simb.IzqPOpc.Pintar:', xNodo);

    Dxa :=Round(tSmb(nodoHijo.data).Dx/2);
    Dxb :=tSmb(nodoHijo.data).Dx - Dxa;

   // Dibujamos la parte superior de una opción
    OpcSup(xDir, x, Yv);

   // Dibujamos el símbolo que contiene la opción
    Yv:=Yv + Sy;
    tSmb(nodoHijo.data).Pintar(nodoHijo, xDir, x+CAx, Yv);

   // Dibujamos la parte inferior de una opción
    OpcInf(xDir, x, Yv);

   // Obtenemos el símbolo asociado a la opción
    nodoHijo:=nodoHijo.GetNextSibling;
    if ((nodoHijo<>nil)and (nodoHijo.data<>nil)) then
      tSmb(nodoHijo.Data).Pintar(nodoHijo, xDir, x+CAx, Yv);
  end;
end;
tsIzqPRepeti:
begin
 // Tomamos el símbolo asociado a esta repetición
  SmbAso:=tIzqPRepeti(nodoHijo.data).SimbAsoc;
  MostrarNodoSimbolo('Simb.Aso.IzqPRepeti.Pintar:', xNodo);

 //¿Hay símbolo asociado a la parte superior de la repetición?
  if (SmbAso<>nil) and (SmbAso.data<>nil)then
  begin
    if tSimb(SmbAso.Data).EsRepSupIni then
    begin
      {Si, hay símbolo asociado a la parte superior de la repetición,
```

```
    * por lo tanto la parte superior contiene un símbolo que hay que
    pintar}

    // Dibujamos el inicio de la parte superior de una repetición
     Dxa:=Round((Dx -tSimb(SmbAso.Data).Dx - (2*Rpx))/2);
     Dxb:=(Dx -tSimb(SmbAso.Data).Dx) -  (2*Rpx) - Dxa;

     RepSupCon(xDir, x, Yv , Dxa, tSimb(SmbAso.Data).Dy);

     tSimb(SmbAso.Data).EsRepSupIni:=false;
     tSimb(SmbAso.Data).Pintar(SmbAso, xDir, x + RPx + Dxa, Yv);
     tSimb(SmbAso.Data).EsRepSupIni:=true;

     if xDir=dDer then
       xDir:=dIzq
     else
       xDir:=dDer;

     nodoHijo:=nodoHijo.GetNextSibling;
     if (nodoHijo<>nil) and (nodoHijo.Data<>nil) then
     begin
       MostrarNodoSimbolo('Simb.Aso.IzqPRepeti.Inferior.Pintar:', xNodo);

       Dxa:=Round((Dx -tSimb(nodoHijo.Data).Dx - (2*RPx))/2);
       Dxb:=(Dx -tSimb(nodoHijo.Data).Dx - (2*RPx)) - Dxa;

       Xv :=x;
       Yv := Yv + tSimb(SmbAso.Data).Dy;

       RepInf(xDir, x, Yv, Dxa, tSimb(SmbAso.Data).Dy);

       Xv :=x + RPx +Dxa;

       tSmb(nodoHijo.data).Pintar(nodoHijo, xDir, Xv, Yv);
     end;
   end;
 end else begin
  // Obtenemos el símbolo asociado a la repetición
   nodoHijo:=nodoHijo.GetNextSibling;
   if ((nodoHijo<>nil)and (nodoHijo.data<>nil)) then
   begin
     MostrarNodoSimbolo('Simb.IzqPRepeti.Inferior.Pintar:', xNodo);

     RepSupSin(xDir, x, y);
```

```
          if xDir=dDer then
            xDir:=dIzq
          else
          xDir:=dDer;

        // Dibujamos el símbolo que contiene la repetición
          Xv :=x + RPx;
          Yv:=y + Sy;
          tSmb(nodoHijo.data).Pintar(nodoHijo, xDir, Xv, Yv);
          RepInf(xDir, x, Yv, 0, 0);
        end;
      end;
    end
    else begin
      MostrarNodoSimbolo('Simb.Otro.Pintar:', xNodo);
      tSmb(nodoHijo.data).Pintar(nodoHijo, xDir, x, y);
    end;
  end;
  end;
end;

procedure tSimb.OpcSup(const xDir:tDirec; x, y:integer);
begin
  ind:=2;
  ActualizarFuenteLinea;

  //Dibujamos la parte izquierda----------------------------------------------
  Dibujar.LinHor(x, y, Dx);
  if xDir=dDer then
    Dibujar.FlechaDer(x+round(Dx/2),y,Fl,Fl)
  else
    Dibujar.FlechaIzq(x+round(Dx/2),y,Fl,Fl);

  Dibujar.Arco_I(x + CAmx, y + AF, AF);

  //Dibujamos la parte derecha-----------------------------------------------
  Dibujar.Arco_II(x + Dx - Lx, y, AF);
end;

procedure tSimb.OpcInf(const xDir:tDirec; x, y:integer);
begin
  ind:=2;
  ActualizarFuenteLinea;
  //Dibujamos la parte izquierda----------------------------------------------
```

```
  Dibujar.LinHor(x+CAmx+AF, y, Lx);
  Dibujar.Arco_III(x + CAmx-1, y - AF, AF+1);

  //Dibujamos la parte derecha----------------------------------------------
  Dibujar.LinHor(x + Dx - CAx, y, Lx);
  Dibujar.Arco_IV(x + Dx - CAmx - AF, y+1, AF+1);

end;

procedure tSimb.RepSupSin(const xDir:tDirec; x, y:integer);
begin
  ind:=3;
  ActualizarFuenteLinea;

  //Dibujamos la parte izquierda--------------------------------------------
  Dibujar.LinHor(x, y, Dx);      // Dibujamos la línea con ajuste
  Dibujar.Arco_II(x + Lx +AF, y, AF);

  //Dibujamos la parte derecha----------------------------------------------
  Dibujar.Arco_I(x + Dx - Lx, y+AF, AF);

  if xDir=dDer then
    Dibujar.FlechaDer(x+round(Dx/2),y,Fl,Fl)
  else
    Dibujar.FlechaIzq(x+round(Dx/2),y,Fl,Fl);
end;

procedure tSimb.RepSupCon(const xDir:tDirec; x, y, xDx, xDy:integer);
begin
  ind:=3;
  ActualizarFuenteLinea;
  //Dibujamos la parte izquierda--------------------------------------------
  Dibujar.LinHor(x, y, RPx + xDx);      // Dibujamos la línea con ajuste
  Dibujar.Arco_II(x + Lx +AF, y, AF);
  Dibujar.LinVer(x + Lx, y + AF, xDy - Sy);    //Dibujamos la línea vert.con ajuste

  //Dibujamos la parte derecha----------------------------------------------
  Dibujar.LinHor(x+Dx-RPx-xDx, y, RPx + xDx);  // Dibujamos línea Hor. con ajuste
  Dibujar.Arco_I(x + Dx - Lx, y+AF, AF);
  Dibujar.LinVer(x+Dx-Lx-1, y+AF, xDy-Sy+1);  // Dibujamos línea vert. con ajuste

  If (xDir=dDer) then begin
      Dibujar.FlechaArr(x + Lx, y + AF + round((xDy-Sy)/2) - Fl, Fl, Fl);
      Dibujar.FlechaAba(x + Dx - Lx-1, y + AF + round((xDy-Sy)/2)+ round(Fl/2), Fl, Fl);
  end else begin
```

```
      Dibujar.FlechaAba(x + Lx, y + AF + round((xDy-Sy)/2)+ round(Fl/2), Fl, Fl);
      Dibujar.FlechaArr(x + Dx - Lx-1, y + AF + round((xDy-Sy)/2)- Fl, Fl, Fl);
  end;
end;

procedure tSimb.RepInf(const xDir:tDirec; x, y, xDx, xDy:integer);
begin
  ind:=3;
  ActualizarFuenteLinea;
  //Dibujamos la parte izquierda----------------------------------------------
  Dibujar.LinHor(x+Lx+AF, y, Lx + xDx); // Dibujamos línea horiz. con ajuste
  Dibujar.Arco_III(x + Lx , y - AF, AF+1);

  //Dibujamos la parte derecha-----------------------------------------------
  Dibujar.LinHor(x+Dx-RPx-xDx, y, xDx+Lx); // Dibujamos línea horiz. con ajuste
  Dibujar.Arco_IV(x + Dx - Lx - AF , y+1, AF);
end;

function tSimb.LeerCadAsoc(const xNodo : tTreeNode):string;
  var  nodoHijo : tTreeNode;
begin
  result:='';
  nodoHijo:=xNodo.GetFirstChild;
  while ((nodoHijo<>nil)and (nodoHijo.data<>nil)) do
  begin
    result:=result + tSmb(nodoHijo.data).LeerCadAsoc(NodoHijo);
    nodoHijo:=nodoHijo.GetNextSibling;
  end;
end;
```

Ya casi terminamos; la siguiente clase se llama **tSmbTxt**, la cual implementa el comportamiento general correspondiente a las clases **SimTerm** y **ClaseSint**, ya que ambas manejan texto asociado. La interfaz de la clase es:

```
tSmbTxt = class(tSmb)
  private
    Tx              : integer; // Longitud del texto
    pIndEnTabla     : integer;
  public
    property    IndEnTabla : integer read pIndEnTabla;
```

```
    constructor  Create(const xTipoSimb:tTipoSimb; xIndEnTabla:integer);
    function   LeerCadAsoc(const xNodo : tTreeNode):string;override;
end;
```

La implementación de ambos métodos no merece mayores comentarios.

```
constructor tSmbTxt.Create(const xTipoSimb:tTipoSimb; xIndEnTabla:integer);
begin
  inherited Create(xTipoSimb);
  pIndEnTabla :=xIndEnTabla;
end;

function tSmbTxt.LeerCadAsoc(const xNodo : tTreeNode):string;
begin
  result:=ASintactico.LeerSimbTermTabla(indEnTabla);
end;
```

Con esto pasamos a los dos últimos tipos de símbolos que tenemos pendientes. El primero son los símbolos terminales, cuya interfaz es:

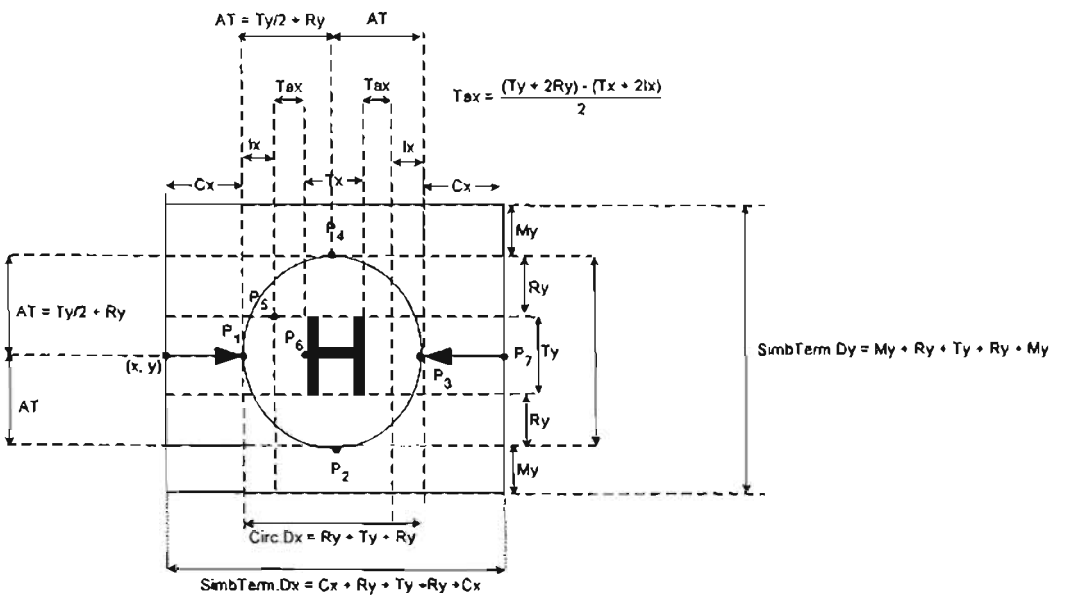
```
tSimbTerm = class(tSmbTxt)
  private
    Tax  : integer; // Ajuste a la longitud del texto
  public
    procedure Medir(const xNodo:tTreeNode);override;
    procedure Pintar(const xNodo:tTreeNode;xDir:tDirec;x, y:integer);override;
end;
```

En las figuras 3.4.12 y 3.4.13 vemos como podemos dibujar símbolos terminales, tenemos dos modalidades porque cuando la longitud del texto que vamos a dibujar más su alineación es menor que la altura del texto más su interlineado (que es igual al diámetro del círculo con el que encerramos al símbolo terminal) tenemos que hacer un ajuste en la longitud del texto para compensar por esta diferencia. A continuación tenemos el método medir:

```
procedure tSimbTerm.Medir(const xNodo : tTreeNode);
begin
  MostrarNodoSimbolo('SimbTerm.Medir:', xNodo);
  Tax:=0; // Por defecto suponemos que no hay ajuste
  // Primero calculamos la longitud del texto asociado a este símbolo
  Tx:=Dibujar.CalcLongTxt(ASintactico.LeerSimbTermTabla(indEnTabla));
```

```
if (Ry + Ty + Ry) > (Ix + Tx + Ix) then
begin
  {Si la altura del círculo es mayor a la longitud del texto necesitamos
  ajustar la longitud del texto para que éste quede centrado en el círculo}
  Tax:= round( (((Ry + Ty + Ry) - (Ix + Tx + Ix))/2));

  Dx:=Cx + Ry + Ty + Ry + Cx;
end else
  // La longitud del texto es mayor a la altura del círculo
  Dx:=Cx + Ix + Tx + Ix + Cx;

Dy:=My + Ry + Ty + Ry + My; // O lo que es lo mismo 2AF
end;
```

SimbTerm Caso a) (Ty +2Ry) mayor o igual a (Tx+2Ix)



Instruccciones para dibujar

Linea desde (x,y) hasta P_1 : LineaHor(x, y, Cx)

Arco I desde P3 : ArcoI(x+Dx - Cx, y, AT);

Arco II desde P4 : ArcoII(x + Cx + AT, y - AT, AT);

Arco III desde P1 : ArcoIII(x + Cx, y, AT);

Arco IV desde P2 : ArcoIV(x + Dx - Cx - AT, y + AT, AT);

Linea desde P_3 hasta P_7 : LineaHor(x + Dx- Cx - AT, y, Cx)

Texto desde P_6 : Texto(x + Cx + Ix + Tax, y, "Texto")

Dirección hacia la derecha

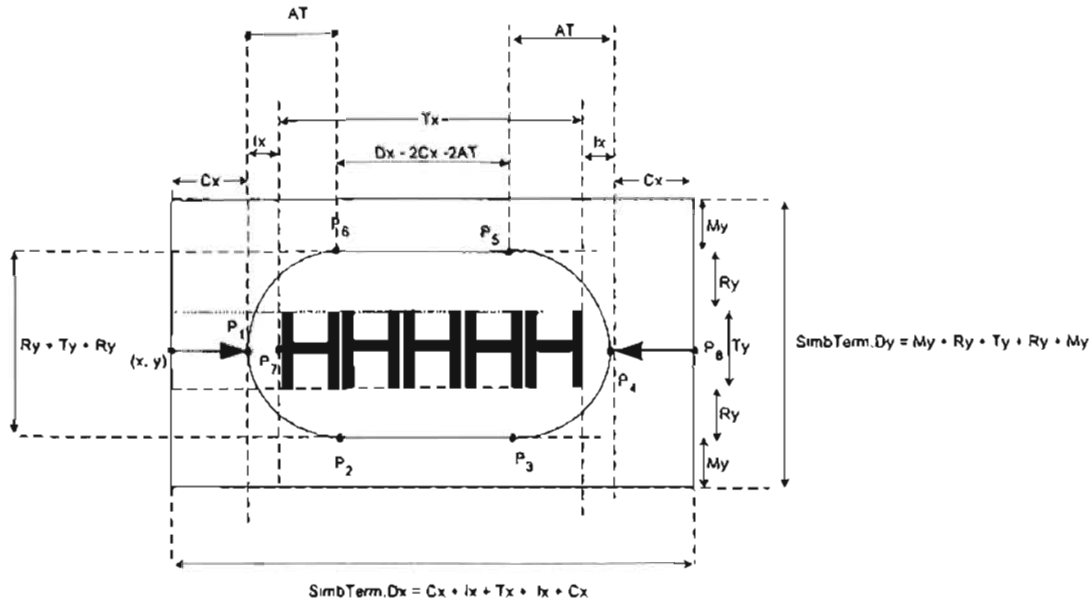
FlechaDer en el punto P_1 . FlechaDer(x+Cx, y, FL, FI)

Dirección hacia la izquierda

Flechalzq en el punto P_3 : Flechalzq(x+Dx-Cx, y, FI, FI)

FIGURA 3.4.12

SimbTerm Caso b) (Ty +2Ry) menor a (Tx+2Ix)



Instruccciones para dibujar

Linea desde (x,y) hasta P_1 : LineaHor(x, y, Cx)

Arco I desde P3 : ArcoI(x+Dx - Cx, y, AT);

Arco II desde P4 . ArcoII(x + Cx + AT, y - AT, AT);

Arco III desde P1 : ArcoIII(x + Cx, y, AT);

Arco IV desde P2 . ArcoIV(x + Dx - Cx - AT, y + AT, AT);

Linea desde P_3 hasta P_7 : LineaHor(x + Dx- Cx - AT, y, Cx)

Texto desde P_6 : Texto(x + Cx + Ix + Tax, y, "Texto")

Linea desde P_8 hasta P_5 : LineaHor(x + Cx + AT, y - AT, Dx - 2AT -2Cx)

Linea desde P_2 hasta P_3: LineaHor(x + Cx + AT, y + AT, Dx - 2AT -2Cx)

Dirección hacia la derecha

FlechaDer en el punto P_1 : FlechaDer(x+Cx, y, FL, Fl)

Dirección hacia la izquierda

FlechaIzq en el punto P_3 : FlechaIzq(x+Dx-Cx, y, Fl, Fl)

FIGURA 3.4.13

Por último presentamos el método **pintar**:

```
procedure tSimbTerm.Pintar(const xNodo : tTreeNode; xDir:tDirec; x, y:integer);
begin
  MostrarNodoSimbolo('SimbTerm.Pintar:', xNodo);
  ind:=4;
  ActualizarFuenteLinea;
```

```
    Dibujar.LinHor(x-1, y, Cx+1); //Línea a la izquierda de la caja de texto
    Dibujar.Arco_I(x + Dx - Cx, y, AT);      {Dibujamos los 4 arcos}
    Dibujar.Arco_II(x + Cx + AT, y-AT, AT);
    Dibujar.Arco_III(x + Cx, y, AT);
    Dibujar.Arco_IV(x + Dx - Cx - AT, y + AT, AT);
    Dibujar.LinHor(x + Dx - Cx-1, y, Cx+1);   //Línea a la derecha de la caja de texto
    Dibujar.Texto(x+Cx+Ix+Tax , y, ASintactico.LeerSimbTermTabla(indEnTabla));

    if xDir = dDer then
      Dibujar.FlechaDer(x + Cx, y,Fl, Fl)
    else
      Dibujar.FlechaIzq(x + Dx - Cx, y, Fl, Fl);

    if Tax=0 then
    begin
     // Línea superior de la caja de texto
      Dibujar.LinHor(x + Cx + AT, y - AT, Dx - (2*AT) -(2*Cx) );
     // Línea inferior de la caja de texto
      Dibujar.LinHor(x + Cx + AT, y + AT - 1, Dx - (2*AT) -(2*Cx));
    end;

  end;
```

Ya sólo nos queda pendiente el símbolo correspondiente a la clase sintáctica, cuya interfaz es:

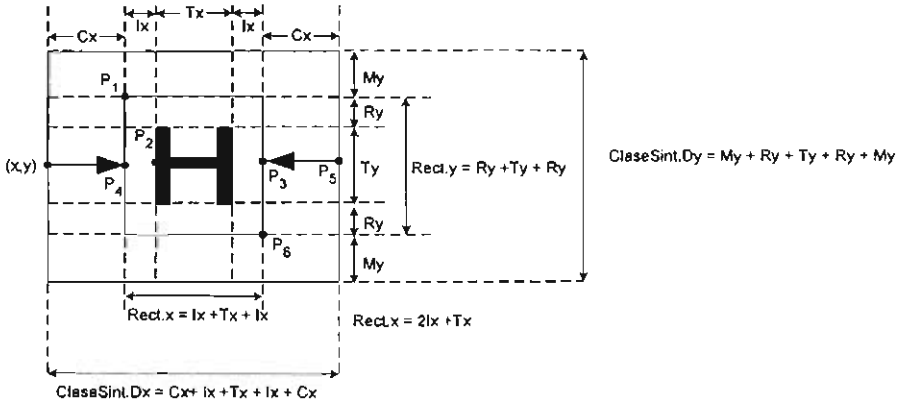
```
  tClaseSint = class(tSmbTxt)
    public
      procedure  Medir(const xNodo:tTreeNode);override;
      procedure Pintar(const xNodo:tTreeNode;xDir:tDirec;x, y:integer);override;
      procedure PintarTitulo(const xNodo:tTreeNode; x, y:integer);
  end;
```

En la figura 3.4.14, presentamos la manera de medir y pintar clases sintácticas, los métodos no merecen mayores comentarios.

```
  procedure tClaseSint.Medir(const xNodo : tTreeNode);
  begin
    MostrarNodoSimbolo('ClaseSint.Medir:', xNodo);
    // Primero calculamos la longitud del texto asociado a este símbolo
    Tx:=Dibujar.CalcLongTxt(ASintactico.LeerSimbTermTabla(indEnTabla));
    Dx:=Cx + Ix + Tx + Ix + Cx;
```

```
  Dy:=My + Ry + Ty + Ry + My; // O lo que es lo mismo 2AF
end;
```

ClaseSint



Instruccciones para dibujar

Linea desde (x,y) hasta P_4 : LinHor(x, y, Cx)

Rectángulo desde P_4 hasta P_3 : Rect(x+Cx, y, 2Ix+Tx, 2Ry+Ty)

Texto desde P_2 : Texto(x + Cx + Ix , y, "Texto")

Linea desde P_3 hasta P_5 : LinHor(x +Dx - Cx, y, Cx)

Dirección hacia la derecha

FlechaDer en el punto P_4 : FlechaDer(x+Cx, y, Fl, Fl)

Dirección hacia la izquierda

FlechaDer en el punto P_4 : FlechaIzq(x+Dx-Cx, y, FL, Fl)

FIGURA 3.4.14

```
procedure tClaseSint.Pintar(const xNodo : tTreeNode; xDir:tDirec; x, y:integer);
begin
  MostrarNodoSimbolo('ClaseSint.Pintar:', xNodo);
  ind:=5;
  ActualizarFuenteLinea;
  Dibujar.LinHor(x-1, y, Cx+1); //Línea a la izquierda de la caja de texto
  Dibujar.Rectang(x +Cx, y, Ix +Tx +Ix, Ry + Ty + Ry);
  Dibujar.Texto( x + Cx + Ix, y, ASintactico.LeerSimbTermTabla(indEnTabla));
  Dibujar.LinHor(x + Dx - Cx-1, y, Cx+1); //Línea a la derecha de la caja de texto
  if xDir = dDer then
    Dibujar.FlechaDer(x + Cx, y,Fl, Fl)
  else
    Dibujar.FlechaIzq(x + Dx - Cx, y, Fl, Fl);
end;
```

3.5 La interfaz gráfica de la aplicación

La última parte de nuestro manual de programador tiene que ver con la parte con la que interactúa el usuario que es a través de la cual manipula la aplicación. Antes de iniciar nuestra descripción tenemos que recordar que Windows es un sistema operativo basado en eventos y mensajes; los principales eventos que nos interesan son los relacionados con el ratón y el teclado, porque es a través de estos dispositivos por medio de los cuales se comunica principalmente el usuario con la computadora.

Por lo tanto en esta parte estamos interesados en proporcionar elementos gráficos que le permitan al usuario comunicarle al sistema qué es lo que desea hacer. ¿Qué son estos elementos de los que hablamos? Pues básicamente los elementos visuales principales de los que se compone una aplicación típica de Windows: ventanas, cuadros de diálogo, menús, botones, barras de desplazamiento, etc..

Delphi ofrece para su programación en Windows una gran variedad de estos elementos a los que les da el nombre de componentes; estos componentes de acuerdo a su naturaleza ofrecen la posibilidad de controlar los eventos y mensajes que son la base del funcionamiento de Windows. El componente principal de una aplicación de Windows para Delphi es la ***forma*** que no es otra cosa que una ventana típica de Windows. Esta forma puede contener varios componentes o ninguno. Obviamente una aplicación que haga algo interesante contiene varios componentes como es el caso de la nuestra.

En la figura 3.5.1 podemos ver la forma principal de nuestra aplicación. En Delphi cada forma se define como una clase que hereda de la clase **tForm**, por lo tanto la interfaz para la clase de nuestra forma es:

```
{* Estados posibles del archivo en la aplicación:----------------------------
* eaNuevo       = Nuevo que no ha sido modificado.
* eaAbierto     = Abierto que no ha sido modificado.
* eaNuevoModif  = Nuevo que fue modificado
* eaAbiertoModif = Abierto y modificado.                          }
tEdoArch = (eaNuevo, eaAbierto, eaNuevoModif, eaAbiertoModif);

//Opciones asociadas a los eventos click de los controles para los archivos
tOpcsArchivo=(ocCrear, ocAbrir, ocGuardar, ocGuardarComo, ocSalir);
```

```
// Forma principal de aplicación
TfrmEbnf2Dt = class(TForm)
  // CONTROLES VISIBLES DIRECTAMENTE EN LA APLICACIÓN---------
  panSuperior: TPanel;
    panHerramientas: TPanel;
      tbrPrincipal: TToolBar;
        tbtPrincipalNuevo: TToolButton;
        tbtPrincipalAbrir: TToolButton;
        tbtPrincipalGuardar: TToolButton;
        tbtPrincipalGuardarComo: TToolButton;
        tbtPrincipalSalir: TToolButton;
        tbtSep01: TToolButton;
      tbrListaClases: TToolBar;
        listaClases: TComboBox;
        tbrListaClasesOrdenar: TToolButton;
        tbrListaClasesNueva: TToolButton;
        tbrListaClasesEliminar: TToolButton;
        tbrListaClasesModificar: TToolButton;
        tbrBuscarClase: TToolButton;
        tbtSep04: TToolButton;
        tbrCopiarEnPortapapeles: TToolButton;
        tbrGrabarEnMetarchivo: TToolButton;
        tbrGrabarMetarchivos: TToolButton;
        edEditorFuente: TEdit;
    Produc: TSynEdit;
  pgcPrincipal  : TPageControl;
    pagPrincipal01 : TTabSheet;
      panConfig: TPanel;
        tbrConfig: TToolBar;
          edDiagramaFuente: TEdit;
          tbtSep02: TToolButton;
          cbDiagramaParamNombre: TComboBox;
          slDiagramaParamValor: TJvxSlider;
          edDiagramaParamValor: TEdit;
          tbtSep03: TToolButton;
          cbSimbsNomb: TComboBox;
          cbSimbsLineaColor: TJvColorComboBox;
          cbSimbsLineaEstilo: Tpenstylecombo;
          cbSimbsLineaAncho: Tpenwidthcombo;
          cbSimbsFuenteColor: TJvColorComboBox;
      panLienzoBase: TPanel;
        Lienzo: TPaintBox;
      scbDiagramaBarraVer: TScrollBar;
      scbDiagramaBarraHor: TScrollBar;
```

```
    pagPrincipal02 : TTabSheet;
      panReconocimiento: TPanel;
        edArbolFuente: TEdit;
        edTablaFuente: TEdit;
      arbSintac: TTreeView;
      tablaSimbs: TMemo;

  // Menu Principal de la aplicación.
  mnuPrincipal        : TMainMenu;
    mnuArchivo: TMenuItem;
      mnuArchivoNuevo: TMenuItem;
      mnuArchivoAbrir: TMenuItem;
      mnuArchivoGuardar: TMenuItem;
      mnuArchivoGuardarComo: TMenuItem;
      mnuArchivoSalir: TMenuItem;
    mnuClase: TMenuItem;
      mnuClaseOrdenar: TMenuItem;
      mnuClaseInsertar: TMenuItem;
      mnuClaseEliminar: TMenuItem;
      mnuClaseModificar: TMenuItem;
      mnuClaseBuscar: TMenuItem;
    mnuDiagrama: TMenuItem;
      mnuDiagramaPortapapeles: TMenuItem;
      mnuDiagramaArchivo: TMenuItem;
      mnuDiagramaTodos: TMenuItem;
    mnuVer: TMenuItem;
      mnuVerLista: TMenuItem;
      mnuVerEditor: TMenuItem;
      mnuVerDiagrama: TMenuItem;
      mnuVerDiagBarraHor: TMenuItem;
      mnuVerDiagBarraVert: TMenuItem;
      mnuVerArbol: TMenuItem;
      mnuVerTabla: TMenuItem;
    mnuConfigurar: TMenuItem;
      mnuConfigurarEditor: TMenuItem;
      mnuConfigurarDiagrama: TMenuItem;
        mnuDiagramaFuente: TMenuItem;
        mnuDiagramaNombres: TMenuItem;
        mnuDiagramaValores: TMenuItem;
      mnuConfigurarSimbolos: TMenuItem;
        mnuSimbolosNomb: TMenuItem;
        mnuSimbolosColorLineas: TMenuItem;
        mnuSimbolosTipoLineas: TMenuItem;
        mnuSimbolosAnchoLineas: TMenuItem;
```

```
      mnuSimbolosColorTexto: TMenuItem;
    mnuConfigurarArbol: TMenuItem;
    mnuConfigurarTabla: TMenuItem;

//COMPONENTES NO VISIBLES DIRECTAMENTE EN LA APLICACIÓN----------
  // Cajas de Diálogos.
    dlgGuardarComo: TSaveDialog;
    dlgAbrir: TOpenDialog;
    dlgDiagramaFuente: TFontDialog;
    dlgBuscarDir: TJvBrowseForFolderDialog;
  // Contiene todos los iconos de la aplicación
    imlIconos: TImageList;
  { Muestra las pistas(hints)asociadas con los controles con un estilo
  * más llamativo y elegante.                              }
    bhnPistas: TJvBalloonHint;
  { Para marcar los metasímbolos de nuestra sintaxis trabaja en
    conjunción con el control visible Produc, que es el editor de clases
    sintácticas.}
    susMarcadorTexto: TSynUniSyn;

{ Control que separa la ventana de aplicación en 2 partes:superior e
* inferior y permite al arrastarlo con el mouse modificar su tamaño.    }
  splPrincipal: TJvNetscapeSplitter;
  splSecundario: TJvNetscapeSplitter;


//EVENTOS ASOCIADOS CON LA APLICACIÓN-----------------------------------
  // Con la forma principal
   procedure FormCreate(Sender: TObject);
   procedure FormClose(Sender: TObject; var Action: TCloseAction);

  // Con el editor de clases sintácticas
   procedure ProducDblClick(Sender: TObject);

  // Con la pag principal 01 que es la de los diagramas
   procedure pagPrincipal01Show(Sender: TObject);

   // Con el Click del mouse sobre los controles
     procedure OpcionesArchivo(Sender: TObject);
     procedure tbrListaClasesOrdenarClick(Sender: TObject);
     procedure tbrListaClasesNuevaClick(Sender: TObject);
     procedure tbrListaClasesEliminarClick(Sender: TObject);
     procedure tbrListaClasesModificarClick(Sender: TObject);
     procedure tbrBuscarClaseClick(Sender: TObject);
```

```
    procedure tbrCopiarEnPortapapelesClick(Sender: TObject);
    procedure tbrGrabarEnMetarchivoClick(Sender: TObject);
    procedure tbrGrabarMetarchivosClick(Sender: TObject);
    procedure edEditorFuenteClick(Sender: TObject);
    procedure edDiagramaFuenteClick(Sender: TObject);
    procedure edArbolFuenteClick(Sender: TObject);
    procedure edTablaFuenteClick(Sender: TObject);

    procedure mnuVerListaClick(Sender: TObject);
    procedure mnuVerEditorClick(Sender: TObject);
    procedure mnuVerArbolClick(Sender: TObject);
    procedure mnuVerDiagramaClick(Sender: TObject);
    procedure mnuVerDiagBarraHorClick(Sender: TObject);
    procedure mnuVerDiagBarraVertClick(Sender: TObject);
    procedure mnuVerTablaClick(Sender: TObject);
    procedure mnuDiagramaNombresClick(Sender: TObject);
    procedure mnuDiagramaValoresClick(Sender: TObject);
    procedure mnuSimbolosNombClick(Sender: TObject);
    procedure mnuSimbolosColorLineasClick(Sender: TObject);
    procedure mnuSimbolosTipoLineasClick(Sender: TObject);
    procedure mnuSimbolosAnchoLineasClick(Sender: TObject);
    procedure mnuSimbolosColorTextoClick(Sender: TObject);

  // Cuando los controles sufren modificaciones
    procedure ProducSeModifico(Sender: TObject);
    procedure listaClasesChange(Sender: TObject);
    procedure cbDiagramaParamNombreChange(Sender: TObject);
    procedure slDiagramaParamValorChange(Sender: TObject);
    procedure edDiagramaParamValorChange(Sender: TObject);
    procedure cbSimbsNombChange(Sender: TObject);
    procedure cbSimbsLineaColorChange(Sender: TObject);
    procedure cbSimbsLineaEstiloChange(Sender: TObject);
    procedure cbSimbsLineaAnchoChange(Sender: TObject);
    procedure cbSimbsFuenteColorChange(Sender: TObject);
    procedure scbDiagramaBarraHorChange(Sender: TObject);
    procedure scbDiagramaBarraVerChange(Sender: TObject);
  // Para que el diagrama siempre se vea en la forma
    procedure LienzoPaint(Sender: TObject);
  private
   { Variables relacionadas con el comportamiento del archivo: }
    edoArch : tEdoArch;
    nombArch : string;
```

```
    {Bandera que utilizamos para saber que al cerrar la aplicación en el
    * evento onPaint de la forma no debemos hacer nada}
    FinApl  : boolean;
    {Bandera que utilizamos para al entrar al evento onPäint de la
    * forma, si estamos grabando metarchivos, evento onPaint de la forma
      no debemos hacer nada}
    GrabandoMetarchivos:boolean;

    {Métodos asociados con el proceso de archivos}
    procedure  ProcesarOpcsArch(const xOpc:byte);
      function CrearArch:boolean;
      function AbrirArch:boolean;
      function GuardarArch(const xNombArch:string):boolean;
      function GuardarArchComo:boolean;
      function DeboEjecOpcArch(const xEdoArch:tEdoArch):boolean;
      procedure ActualizarMsjsUsuario;
      procedure ArchSeModif;

    {Métodos asociados con clase sintáctica}
      procedure CargarClase(xNoRenglon:integer);
      procedure BorrarListaClases;
      function EstaVaciaLaLista:boolean;
      function LeeIndActivoLista:integer;
      procedure EscIndActivoLista(xInd:integer);
      procedure ModifReglaProduc(const xInd:integer; xProduc:String);
      function InsertReglaProduc(const xNombre:string):integer;
      procedure GuardarListaClases(const xNombArch:string);

    // Métodos asociado con árbol sintáctico
      procedure BorrarArbSintac;
      procedure BorrarArbSintacNodo(const xNodo:tTreeNode);

    // Métodos asociados con producción
      procedure DibujarProduccion;
  procedure CopiarProdEnMetaArch(const xOpcionMetaArchivo:tOpcionMetaArchivo;
xInd:integer);

      property indActivoLista : integer  read LeeIndActivoLista
                                write EscIndActivoLista;

 public
end;
```

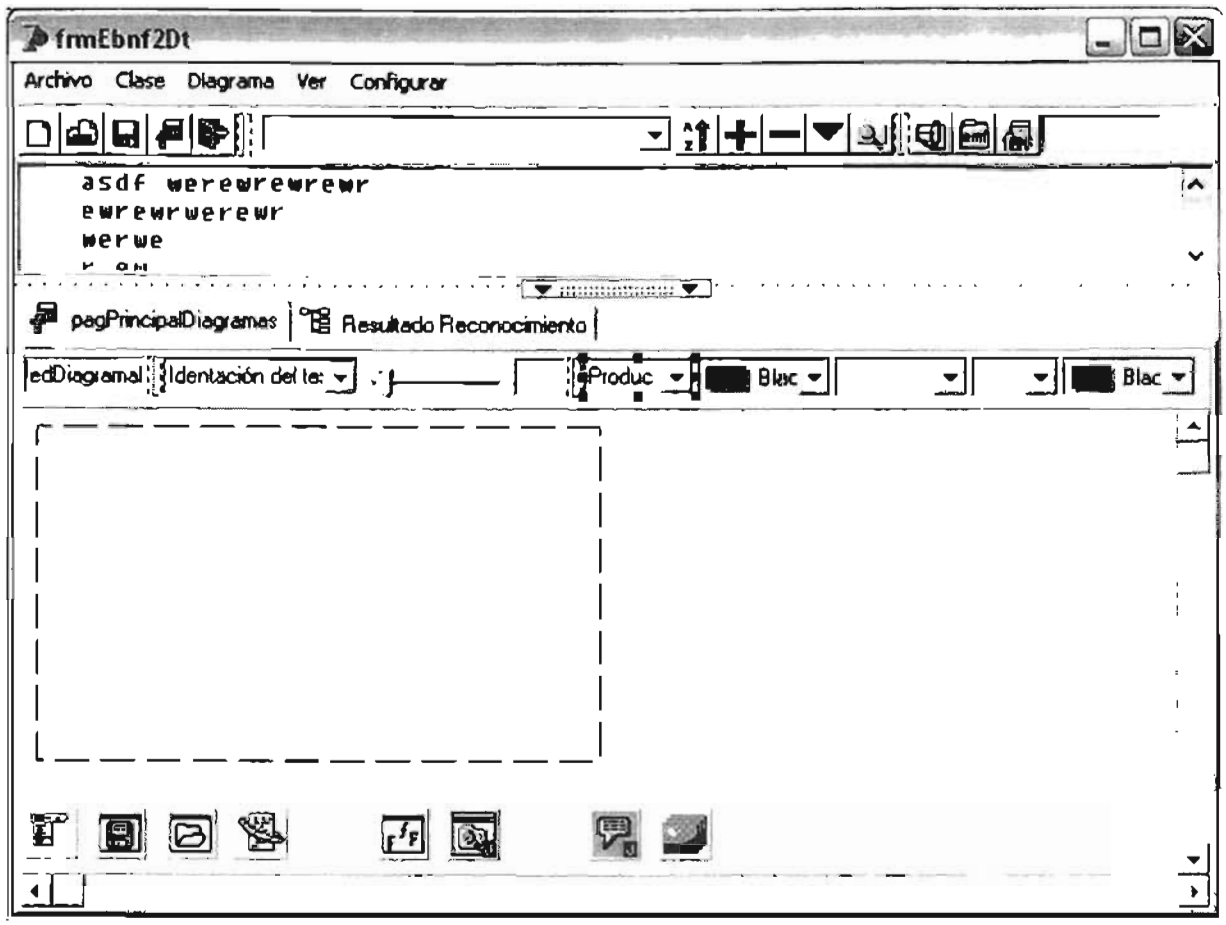


FIGURA 3.5.1

Parece que es mucho y algo complicado pero se trata de algo muy simple; en realidad ésta es la parte más simple del sistema salvo algunos detalles relacionados con la manera de funcionar de los eventos y mensajes. Lo que vamos a hacer es ir explicando qué componentes o controles integran la forma, cuál es su función y como se relacionan con el resto del sistema.

El primer componente que analizamos es la ***forma*** que no es otra que la ventana que aparece en la figura 3.5.1 . Como podemos observar la forma es además un contenedor donde se encuentran todos los demás controles que integran la aplicación.

Antes de continuar con los controles que contiene, tenemos que considerar que la forma es un elemento visual de Windows y por lo tanto tiene eventos y mensajes asociados a ella. Existen varios eventos asociados a la existencia de una forma o ventana pero a nosotros

nos interesan dos. El primero está relacionado con la creación de la forma, deseamos inicializar nuestra aplicación en ese momento para que cuando la forma esté visible podamos manipular nuestra aplicación. Es por eso que asociamos al evento OnCreate de nuestra forma el método:

```
procedure TfrmEbnf2Dt.FormCreate(Sender: TObject);
  var i:byte;
begin
  GrabandoMetarchivos:=false;//Bandera que utilizamos cuando grabamos todos

  {A los controles relacionados con los archivos les asignamos un identificador
  en su propiedad tag}
  mnuArchivoNuevo.tag     :=ord(ocCrear);
  mnuArchivoAbrir.tag     :=ord(ocAbrir);
  mnuArchivoGuardar.tag   :=ord(ocGuardar);
  mnuArchivoGuardarComo.tag:=ord(ocGuardarComo);
  mnuArchivoSalir.tag     :=ord(ocSalir);

  tbtPrincipalNuevo.tag     :=ord(ocCrear);
  tbtPrincipalAbrir.tag     :=ord(ocAbrir);
  tbtPrincipalGuardar.tag   :=ord(ocGuardar);
  tbtPrincipalGuardarComo.tag:=ord(ocGuardarComo);
  tbtPrincipalSalir.tag     :=ord(ocSalir);

  // Creamos los objetos que utiliza la aplicación
    ASintactico:=tASintactico.Create(arbSintac);
    Dibujar := tDibujar.create(Lienzo.Canvas);

  { Inicializamos los valores de los parámetros relacionados con los tipos de
  * símbolos}
 for i:=0 to 6 do
 begin
  arrSimbsLineaColor[i]:=clBlack;
  arrSimbsLineaEstilo[i]:=psSolid;
  arrSimbsLineaAncho[i]:=1;
  arrSimbsFuenteColor[i]:=clBlack;
 end;

 {Forzamos a que las listas de paráemtros se posicionen en el primer elemento
 * y en el texto muestren éste valor}
 cbSimbsNomb.ItemIndex:=-1;
 cbSimbsNomb.ItemIndex:=0;
 cbSimbsLineaColor.ColorValue:=arrSimbsLineaColor[0];
```

```
    cbSimbsLineaEstilo.Selection:=arrSimbsLineaEstilo[0];
    cbSimbsLineaAncho.Selection:=arrSimbsLineaAncho[0];
    cbSimbsFuenteColor.ColorValue:=arrSimbsFuenteColor[0];

      PctIx:=15;
      PctRy:=15;
      PctMy:=15;
      PctCx:=30;
      PctLx:=30;
      PctFl:=35;

      cbDiagramaParamNombre.ItemIndex:=0;
      slDiagramaParamValor.value:=PctIx;
      edDiagramaParamValor.text:=IntToStr(PctIx);

      edDiagramaFuente.Text:=dlgDiagramaFuente.Font.Name;
      edEditorFuente.Text:=Produc.Font.Name;
      edArbolFuente.text:=ArbSintac.Font.Name;
      edTablaFuente.text:=TablaSimbs.Font.Name;
      Dibujar.Actualizar;

      //Al iniciar la aplicación creamos un Arch nuevo.
      CrearArch;
  end;
```

Esto nos garantiza que cuando la forma se crea y antes de que sea visible para el usuario se realizan las tareas que indica el método anterior. Básicamente son tareas que preparan al sistema para que pueda funcionar e inicializan la aplicación.

Otro evento al que nos interesa asociar un método es el relacionado con la destrucción de la forma; es decir cuando el usuario decide cerrar la aplicación:

```
procedure TfrmEbnf2Dt.FormClose(Sender: TObject; var Action: TCloseAction);
  case frmEbnf2Dt.tag of
    0:
    begin
      OpcionesArchivo(tbtPrincipalSalir);
      Action:=caNone;
    end;
    1:
    begin
      //Salida autorizada
```

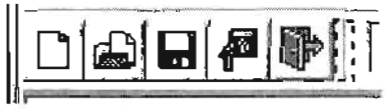
```
        // Destruimos los objetos que utiliza la aplicación
        // Antes destruimos los objetos asociados con la lista
        BorrarListaClases;
        BorrarArbSintac;

        ASintactico.free;
        Dibujar.free;
      end;
    end;
  end;
```

Este método trabaja en conjunción con el método **OpcionesArchivo** porque cuando hacemos click en el ícono  es invocado el método FormClose; en donde forzamos la ejecución del método

```
OpcionesArchivo(tbtPrincipalSalir);
```

Esto es porque deseamos no perder la lógica en el manejo de nuestros archivos, de la que se encarga el método OpcionesArchivo como veremos más adelante. Si al entrar al a este método no cancelamos la acción de salida, el método cambia el valor de la propiedad tag de la forma principal a 1, y vuelve a invocar el método **FormClose**; en esta segunda llamada como el valor de **frmEbnf2Dt.tag** es 1, se llevan a cabo las tareas de limpieza necesarias cuando deseamos salir de la aplicación. Cuando cancelamos la salida, **OpcionesArchivo** restablece el valor de **frmEbnf2Dt.tag** a 0.

Incluimos ahora nuestros primer grupo de controles visibles para el manejo de archivos.



A continuación presentamos cada botón e indicamos a que opciones de archivo corresponden cuando se hace click en ellos:

 Creación de archivo nuevo.

 Abrir un archivo bnf existente.

 Guardar el archivo que se está trabajando.

Guardar con otro nombre el archivo que se está trabajando.

Salir de la aplicación.

Este grupo de opciones se manejan en conjunto, ya que son opciones que se encuentran interrelacionadas entre sí como podemos ver en la figura 3.5.2, donde se muestra el proceso involucrado en estas opciones. Este proceso representa el manejo de un archivo a la vez; es decir su lógica no es para más de un archivo; sin embargo cuida los detalles de una implementación para manejo de archivos. Como estamos manejando estos cinco botones como un grupo, a cada uno de ellos en el evento **OnClick** le asociamos el mismo método que es **ProcesarOpcsArch**.

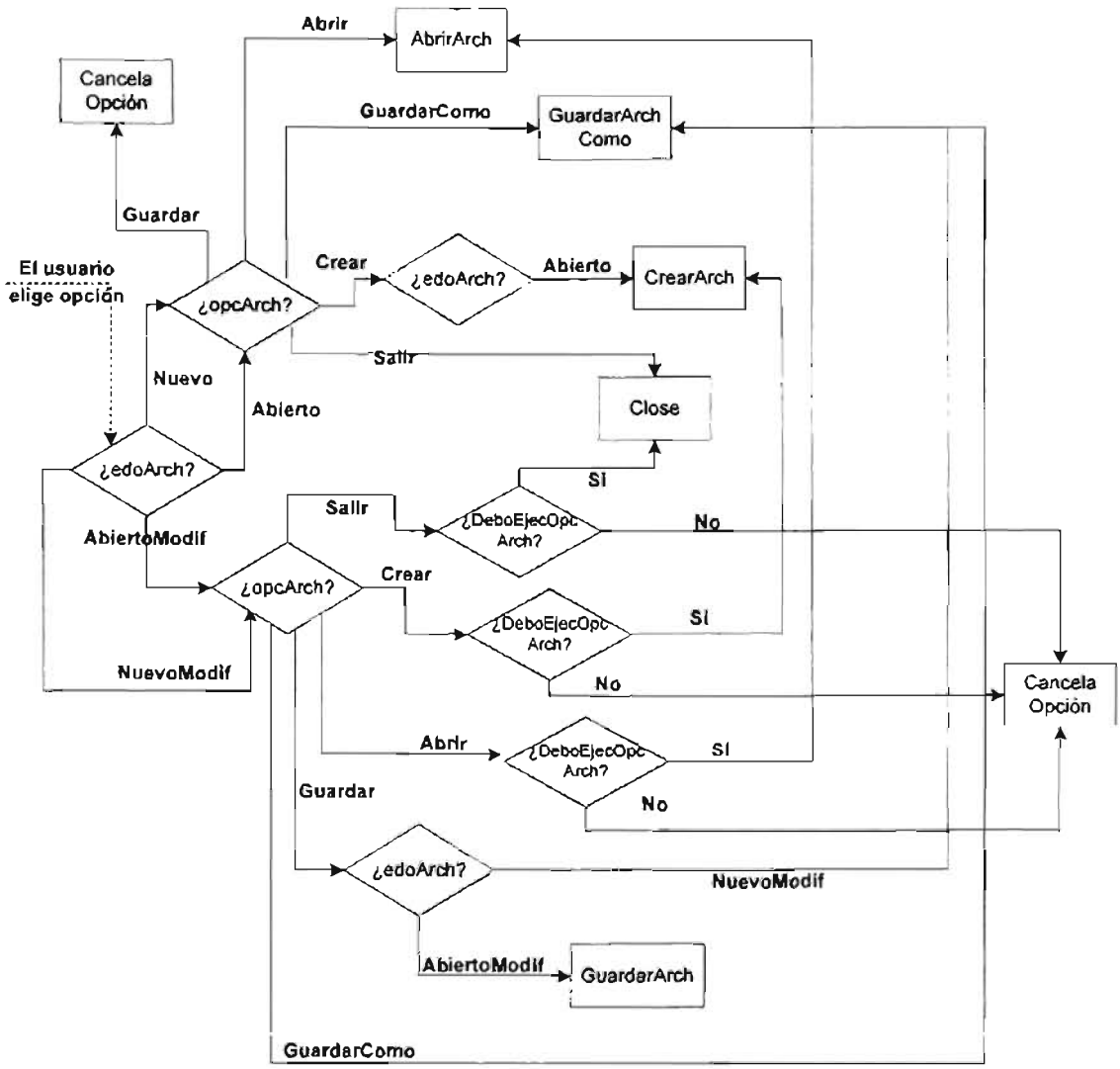


FIGURA 3.5.2

```
procedure TfrmEbnf2Dt.OpcionesArchivo(Sender: TObject);
begin
  case edoArch of
   //Cuando no hay modificaciones en el Arch.
   'eaNuevo..eaAbierto:
     case tControl(Sender).tag of
       ord(ocCrear)       :if edoArch= eaAbierto
                          then CrearArch;
       ord(ocAbrir)       :AbrirArch;
       ord(ocGuardar)     :;
       ord(ocGuardarComo) :GuardarArchComo;
       ord(ocSalir)       :
       begin
         frmEbnf2Dt.tag:=1; // Autorizamos la salida
         Close;
       end;
     end;
   //Cuando hay modificaciones en el Arch.
    eaNuevoModif..eaAbiertoModif:
     case tControl(Sender).tag of
       ord(ocCrear)      :if DeboEjecOpcArch(edoArch)
                          then CrearArch;
       ord(ocAbrir)      :if DeboEjecOpcArch(edoArch)
                          then AbrirArch;
       ord(ocGuardar)    :if edoArch= eaNuevoModif
                          then GuardarArchComo
                          else GuardarArch(nombArch);
       ord(ocGuardarComo) :GuardarArchComo;
       ord(ocSalir)      :if DeboEjecOpcArch(edoArch)
                          then
                            begin
                              frmEbnf2Dt.tag:=1;//Autorizamos salida
                              Close;
                            end else
                              frmEbnf2Dt.tag:=0;//Cancelamos salida
     end;
  end;
end;

{-----------------------------------------------------------------------
* Le ofrece al usuario la oportunidad de:
*   a) Cancelar el proceso(la opción que originó esta llamada).
*   b) Desechar las modificaciones efectuadas y continuar con el proceso.
*   c) Guardar las modificaciones: en el caso de Archs nuevos, si el
```

```
*      usuario decide no guardar las modificaciones el proceso se aborta.   }
function TfrmEbnf2Dt.DeboEjecOpcArch(const xedoArch:tEdoArch):boolean;
begin
  // De entrada suponemos que no se debe ejecutar la opción.
  result:=false;
  case MessageDlg('El Arch '+nombArch+
                  ' ha sido modificaciones, ¿desea conservalas?',
                        mtConfirmation, [mbYes, mbNo, mbCancel], 0) of
    // El usuario desecha las modificaciones y continúa con el proceso.
     mrNo:result:=True;
    //El usuario desea guardar las modificaciones.
     mrYes:
       if xedoArch=eaNuevoModif then
       begin
        //Si el usuario guardó las modificaciones, continuamos.
          if GuardarArchComo then result:=true;
       end else
       begin
        //Guardamos las modificaciones y continuamos con el proceso.
          if GuardarArch(nombArch)then result:=true;
       end;
  end;
end;
```

Como podemos ver, el método **ProcesarOpciones** procede de acuerdo a la combinación del tipo de opción que se desea ejecutar y al estado del archivo. El siguiente paso es analizar los métodos asociados al proceso de archivos.

Tomamos primero el proceso de crear un archivo  :

```
function TfrmEbnf2Dt.CrearArch:boolean;
begin
  // De entrada suponemos que no se puede crear
  result:=false;
  try
    // Creamos el Archivo de gramática nuevo
     nombArch := 'Nuevo.bnf';
     edoArch := eaNuevo;
     ActualizarMsjsUsuario;
    // Lista de producciones
      // Borramos la lista de  reglas de producción existentes
        BorrarListaClases;
        BorrarArbSintac;
```

```
    // Inicializamos a la clase cargada
     Produc.Lines.Clear;
    // Indicamos que no hay ninguna clase cargada
     Produc.tag:=-1;
     Dibujar.Limpiar(Lienzo.Width, Lienzo.Height);
     result:=true;
   except
   end;
end;
```

Nótese que, por omisión, a un archivo nuevo le damos el nombre de **Nuevo.bnf**; en seguida actualizamos el estado del archivo indicando que se trata de un archivo nuevo.

```
{ Mostramos el nombre del Arch y actualizamos su estado cada vez que es
* necesario, para que el usuario sepa en todo momento el estado del Arch. }
procedure TfrmEbnf2Dt.ActualizarMsjsUsuario;
begin
 frmEBnf2Dt.Caption:='Ebnf2dt:' + nombArch;
 case edoArch of
  eaNuevo..eaAbierto:frmEBnf2Dt.Caption:=frmEBnf2Dt.Caption;
  eaNuevoModif..eaAbiertoModif:frmEBnf2Dt.Caption:=frmEBnf2Dt.Caption+
                                                  ' <Modif>';
 end;
end;
```

ActualizarMsjsUsuario le informa al usuario del estado del archivo.

```
procedure TfrmEbnf2Dt.BorrarListaClases;
  var i:integer;
begin
  // Primero borramos las partes ders. asociadas con las partes izqs.
  for i:=0 to ListaClases.Items.Count-1 do
    // Liberamos las cadenas asociadas con las partes ders. de cada regla
    if ListaClases.Items.Objects[i]<>nil then
      tRegla(ListaClases.Items.Objects[i]).free;

  // Finalmente borramos la lista de partes izquierdas
  ListaClases.clear;
end;

procedure TfrmEbnf2Dt.BorrarArbSintac;
begin
  if ArbSintac.Items.count>0 then begin
    BorrarArbSintacNodo(ArbSintac.Items[ 0 ]); //Borramos los datos asociados
```

```
    ArbSintac.Items.clear;//Finalmente borramos el árbol.
  end;
end;

procedure TfrmEbnf2Dt.BorrarArbSintacNodo(const xNodo:tTreeNode);
  var nodoHijo:tTreeNode;
begin
  if xNodo.data<>nil then
    tSmb(xNodo.data).Free;
  if xNodo<>nil then begin
    if( xNodo.GetFirstChild <> nil ) then
      BorrarArbSintacNodo(xNodo.GetFirstChild);
    if( xNodo.GetNextSibling <> nil ) then
      BorrarArbSintacNodo(xNodo.GetNextSibling);
  end;
end;
```

El método **BorrarListaClases** se encarga de borrar primero la regla de producción asociada a la clase sintáctica y al final borra todas las clases sintácticas de la lista. Por su parte **BorrarArbSintac** se encarga primero de recorrer todo el árbol e ir borrando todos los datos asociados a cada nodo que son ejemplares del tipo **tSmb**; una vez concluido el recorrido borra todos los nodos del árbol sintáctico. También se borra el contenido de la caja de edición de reglas de produccion **Produc** y se limpia el área de dibujo. Nótese que además a **Produc.tag** se le asigna el valor de -1; este es un truco para sincronizar la lista de clases con la caja de edición, ya que cuando hay una clase cargada en la caja de edición **Produc** el valor de la propiedad **tag** apunta al índice de la clase que se encuentra cargada.

A continuación analizamos la opción AbrirArch  :

```
function TfrmEbnf2Dt.AbrirArch:boolean;
begin
  // De entrada suponemos que no se puede crear.
  result:=false;
  try
    if dlgAbrir.execute then
    begin
      //Pudimos abrir el Arch, actualizamos nombre y estado del Arch.
      edoArch:=eaAbierto;
      nombArch :=dlgAbrir.FileName;
      ActualizarMsjsUsuario;

      // Antes de abrir borramos la lista de clases
```

```
      BorrarListaClases;
      BorrarArbSintac;
      ASintactico.ReconocerArch(dlgAbrir.FileName, ListaClases);
      ASintactico.VaciarTabla;


     // Inicializamos la clase cargada
      Produc.Lines.Clear;
      if ListaClases.Items.count=0 then
        Produc.Tag:=-1  // No hay ninguna clase por cargar
      else
      begin
        CargarClase(0);//Cargamos la clase del renglón cero de la lista.
        ListaClases.ItemIndex:=0;// Nos posicionamos en ese renglón
        ListaClasesChange(nil);
      end;
      result:=true;
    end;
  except
   //
  end;
end;
```

Aquí incluimos al componente **dlgAbrir**  , que es un control visible; los cinco botones anteriores siempre están visibles pero **dlgAbrir** es visible cuando ejecutamos el método **execute**, en ese momento el control muestra una ventana de diálogo que permite elegir un archivo para abrirlo. El método execute está implementado como una función que regresa el valor de verdadero cuando el usuario elige un archivo para apertura. En cuyo caso actualizamos el estado del archivo para indicar que es un archivo abierto; actualizamos el mensaje al usuario con esta información, nuevamente borramos la lista de clases sintácticas y el árbol. Si la lista no contiene elementos, lo cual no suena lógico pero sí posible, indicamos que la ventana de edición **Produc** no apunta a ninguna clase. Si la lista no está vacía, cargamos la clase que se encuentra en el primer elemento de la lista:

```
      CargarClase(0);//Cargamos la clase del renglón cero de la lista.

procedure TfrmEbnf2Dt.CargarClase(xNoRenglon:integer);
begin
  // Cargamos el texto de la parte izquierda que está en el renglón indicado.
   Produc.Text:=
            tRegla(ListaClases.Items.Objects[xNoRenglon]).Produccion;
   Produc.tag:=xNoRenglon;
```

```
end;
```

Para ser congruentes activamos a ese renglón en la lista

```
ListaClases.ItemIndex:=0;      // Nos posicionamos en ese renglón
```

y finalmente invocamos el método

```
ListaClasesChange(nil);
```

que es el método asociado al componente **ListaClases** para el evento **onChange**.

En este caso estamos forzándolo, ya que el hecho de cambiar de renglón por medio de una instrucción como la anterior, no genera o activa el evento **OnChange**; éste se activa únicamente cuando cambiamos de renglón a través del componente visual ya sea por medio del ratón o del teclado.

Claro, porque deseamos que cada que se elija una clase diferente de la lista, se cargue una regla de producción diferente en **Produc** lista para modificarse y que se dibuje un diagrama diferente, por lo que dicho método queda así:

```
procedure TfrmEbnf2Dt.listaClasesChange(Sender: TObject);
  var noRenglon : integer; parteDerecha:string;
begin
  //¿La lista contiene elementos?
  if ListaClases.itemIndex<>-1 then
  begin
    if Produc.tag<>-1 then
    begin
      if Produc.text<>
        tRegla(ListaClases.Items.Objects[Produc.tag]).Produccion then
              tRegla(ListaClases.Items.Objects[Produc.tag]).Produccion
                                                          :=Produc.text;
    end;
    Produc.text:=tRegla(ListaClases.Items.Objects[indActivoLista]).Produccion;
    Produc.tag:=indActivoLista;
    ListaClases.text:=ListaClases.Items[indActivoLista];
    DibujarProduccion;
  end else
  begin
    Produc.Text:='';
    Produc.tag:=-1;
  end;
```

```
end;
```

Cuando la lista contiene elementos antes de efectuar el cambio de renglón verificamos si la regla de producción que estaba cargada en **Produc** sufrió cambios

```
if Produc.text<>
  tRegla(ListaClases.Items.Objects[Produc.tag]).Produccion then
```

para actualizarlos antes de proceder al nuevo renglón.

```
tRegla(ListaClases.Items.Objects[Produc.tag]).Produccion:=Produc.text;
```

Una vez hecha esta verificación, cargamos a **Produc** con el contenido de la nueva regla asociada

```
Produc.text:=tRegla(ListaClases.Items.
                    Objects[indActivoLista]).Produccion;
```

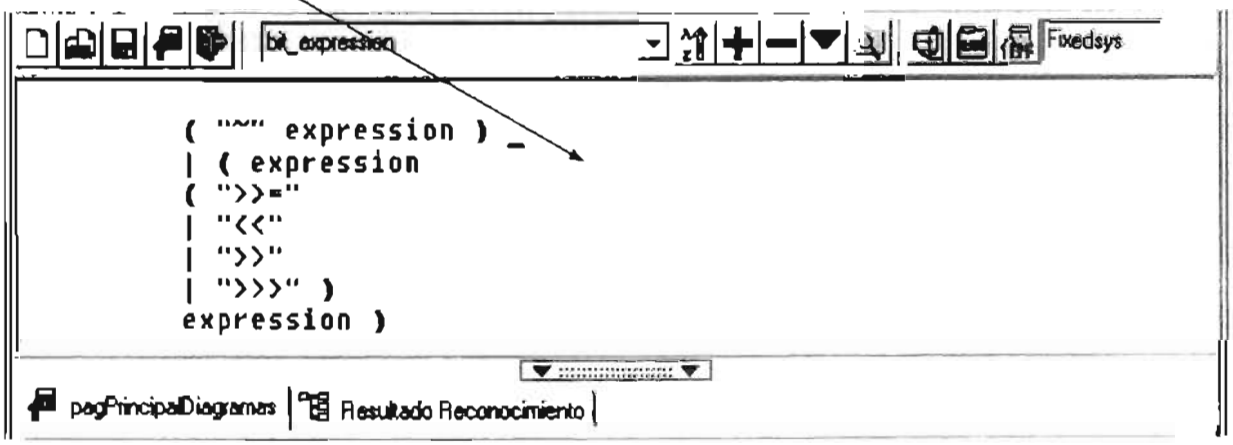
En la propiedad tag de **Produc** guardamos el índice activo en la lista de clases sintácticas

```
Produc.tag:=indActivoLista;
```

y dibujamos el nuevo diagrama asociado

```
DibujarProduccion;
```

Antes de analizar este último método, redondeamos las ideas expuestas con la inclusión del componente **Produc** que no es más que un componente visual que permite editar texto.



Este componente/editor permite, en combinación con el componente no visual

susMarcadorTexto, establecer propiedades para marcar con diferente color y fondo los carácteres que pertenecen al conjunto de metasímbolos de nuestra gramática. Todo esto se hace de manera muy sencilla simplemente asignándole al editor **Produc** en su propiedad **Highlighter** un apuntador al componente **susMarcadorTexto** y estableciendo en él todas las propiedades que deseamos para nuestros metasímbolos. **Produc** y **susMarcadorTexto** son componentes que no forman parte de los nativos de Delphi; en la parte de mapas de programas se listan todos los componentes no nativos de Delphi indicando a qué paquete de componentes pertenecen y en dónde pueden obtenerse, ya que todos son componentes gratuitos que se encuentran disponibles en Internet.

Volvemos a nuestra discusión del método dibujar .

```
procedure TfrmEbnf2Dt.DibujarProduccion;
begin
  if ListaClases.Items.Count>0 then begin
    TablaSimbs.Clear;
    BorrarArbSintac;
    Dibujar.Limpiar(Lienzo.Width, Lienzo.Height);

    ASintactico.ReconocerClaseSint(ListaClases.Items.Strings[Produc.tag]
        +cSepProduc+ Produc.text+cFinProduc); // Efectuamos el reconocimiento
    arbSintac.Items[0].Expand(true);
    ASintactico.VaciarTablaEn(tablaSimbs); // LLenamos la tabla de símbolos

    // Actualizamos parámetros y medimos
    Dibujar.AsignarFuente(dlgDiagramaFuente.Font);
    Dibujar.Actualizar;
    // Medimos los símbolos
    tSmb(arbSintac.Items[0].getFirstChild.Data).Medir(arbSintac.Items[0].
                                                      getFirstChild);

    // Guardamos las medidas
    Lienzo.Width:=tSmb(arbSintac.Items[0].GetFirstChild.Data).Dx;
    Lienzo.Height:=tSmb(arbSintac.Items[0].GetFirstChild.Data).Dy;

    // Dibujamos
    tSmb(arbSintac.Items[0].GetFirstChild.Data).Pintar(arbSintac.Items[0].GetFirstChild,
  dDer, 0, 0);

    scbDiagramaBarraHor.Max:=Lienzo.Width;
    scbDiagramaBarraVer.Max:=Lienzo.Height;
  end;
end;
```

Este método se ejecuta si la lista de clases contiene elementos, en cuyo caso se borran previamente la tabla de símbolos, el árbol sintáctico y el área de dibujo. Después utilizamos nuestro ejemplar **ASintactico** para realizar el reconocimiento de la regla de producción que se encuentra cargada en el editor ***Produc***.

```
ASintactico.ReconocerClaseSint(ListaClases.Items.Strings[Produc.tag]
    +cSepProduc+ Produc.text+cFinProduc); // Efectuamos el reconocimiento
```

Ya vimos la implementación de la clase **ASintactico**, que se encarga de todos los detalles del proceso de reconocimiento y además construye los nodos del árbol sintáctico junto con los símbolos que permiten dibujar el diagrama. Después expandemos todos los nodos del árbol generado y llenamos la tabla de símbolos.

El siguiente paso es asegurarnos de actualizar los parámetros de dibujo para efectuar las mediciones.

```
tSmb(arbSintac.Items[0].getFirstChild.Data).Medir(arbSintac.Items[0].getFirstChild);
```

Tomamos el nodo raiz de nuestro árbol y ordenamos el reconocimiento, lo cual desencadena el reconocimiento recursivo de los nodos del árbol que tienen asociado un ejemplar del tipo **tSmb**; he aquí la ventaja de utilizar polimorfismo ya que únicamente utilizamos un tipo de símbolo base para hacer referencia a los diferentes tipos de símbolos asociados a cada nodo. Al finalizar cada nodo sabe cuánto mide cada símbolo asociado. Una vez que sabemos las medidas del diagrama las utilizamos para establecer las dimensiones del área de dibujo.

```
Lienzo.Width:=tSmb(arbSintac.Items[0].GetFirstChild.Data).Dx;
Lienzo.Height:=tSmb(arbSintac.Items[0].GetFirstChild.Data).Dy;
```

Y lo único que nos resta es dibujar el diagrama

```
tSmb(arbSintac.Items[0].GetFirstChild.Data).
        Pintar(arbSintac.Items[0].GetFirstChild, dDer, 0, 0);
```

Cada símbolo asociado a los nodos del árbol sabe como dibujar la parte que le corresponde y nuevamente este método desencadena el proceso recursivo para dibujar todos los símbolos.

Al final establecemos los valores de las barras de desplazamiento del área de dibujo.

Analizamos ahora la opción para guardar el archivo que se está trabajando  .

```
function TfrmEbnf2Dt.GuardarArch(const xNombArch:string):boolean;
  var noRenglon : integer; parteDerecha:string;
begin
  //De entrada suponemos que no podemos Guardar al Arch.
  result:=false;
  try
    { Antes de guardar la lista en el archivo, verificamos si hay clase
    * cargada y si sufrió cambios                                    }
    noRenglon:=Produc.tag;
    if (noRenglon<>-1) then  begin
      if Produc.text<>
          tRegla(ListaClases.Items.Objects[noRenglon]).Produccion then
      begin
        tRegla(ListaClases.Items.Objects[noRenglon]).Produccion
                                          :=Produc.text;
      end;
    end;
    GuardarListaClases(xNombArch);
    nombArch:=xnombArch;
    edoArch:=eaAbierto;
    ActualizarMsjsUsuario;
    result:=true;
  except
    //
  end;
end;
```

Vemos si la regla de producción cargada en el editor sufrió cambios para reflejar los cambios en la lista de clases. Después guardamos la lista en el nombre de archivo indicado:

```
procedure TfrmEbnf2Dt.GuardarListaClases(const xNombArch:string);
  var i:integer; gram:tStringList; reng:String;
begin
  gram:=tStringList.create;
  gram.Text:='';
  for i:=0 to ListaClases.Items.Count-1 do  begin
    reng:=//Tomamos el nombre de la clase
        ListaClases.Items.Strings[i]
        // Agregamos el separador
        + ' ' + cSepProduc
        // Tomamos su regla de producción
```

```
        + tRegla(ListaClases.Items.Objects[i]).Produccion;

      if CopyRange(reng, length(reng)-1, length(reng))<>cNuevaLinea then
        reng:= reng + cNuevaLinea + cFinProduc
      else
        reng:= reng + cFinProduc;

      gram.Text:=gram.text+reng;
    end;
    reng:=CopyRange(Gram.Text, 1, length(Gram.Text)-2);
    Gram.text:=reng;

    // Guardamos la lista
    gram.SaveToFile(xNombArch);
    gram.free;
  end;
```

Para tal efecto creamos un ejemplar de **tStringList** que es una clase que nos permite crear una lista de cadenas y guardarla en un archivo. En nuestro editor de reglas de producción el usuario nunca tiene que preocuparse por el separador de producción ("=") ni por el fin de producción("."). Cuando guardamos nuestro archivo agregamos estos dos carácteres:

```
reng:= reng + cNuevaLinea + cFinProduc
```

De esta manera vamos construyendo todas las clases sintácticas y sus reglas asociadas hasta que podemos salvar la totalidad del archivo

```
gram.SaveToFile(xNombArch);
```

Pasamos a la opción de archivo  GuardarComo:

```
{ En la propiedad "Options" del control "dlgGuardarComo", activamos la opción
* "ofOverwritePrompt", para que cuando el Arch elegido ya exista, despliegue
* un mensaje de advertencia al respecto y nos de la oportunidad de escoger:
* si lo reemplazamos o escogemos otro nombre.                    }
function TfrmEbnf2Dt.GuardarArchComo:boolean;
begin
  {De entrada suponemos que el usuario no desea guardar con otro nombre.    }
  result:=false;
  try
    // Mostramos el diálogo para guardar con otro nombre.
    dlgGuardarComo.FileName:=nombArch;
    // ¿El usuario eligió un nombre válido de Arch?.
```

```
      if dlgGuardarComo.execute then
      begin
        // Si podemos guardarlo con ese nombre, indicamos todo fue un éxito.
        if GuardarArch(dlgGuardarComo.FileName) then result:=true;
      end;
    except
    end;
  end;
```

El procedimiento es bastante obvio; lo nuevo es el componente  **dlgGuaradarComo** que es simplemente un cuadro de diálogo que le permite al usuario elegir la carpeta y un nombre de archivo con el que quiere guardar el archivo que está trabajando.

La última opción de archivo que analizamos es  Salir de la aplicación. Ya vimos que este botón tiene asociado en el evento **OnClick** la instrucción **Close**; la cual provoca que la forma (ventana de la aplicación) se cierre.

Con esto terminamos nuestro análisis de las opciones de archivo y sus componentes asociados. Ahora describiremos los eventos y métodos asociados a los botones que nos permiten administrar la lista de clases sintácticas.

Tomamos primero el botón  que nos sirve para ordenar alfabéticamente la lista de clases sintácticas. Al ordenar la lista se pierde el orden en que se crearon los elementos de la misma. A este botón le asociamos en el evento **OnClick** el siguiente método:

```
procedure TfrmEbnf2Dt.tbrListaClasesOrdenarClick(Sender: TObject);
begin
  // ¿La lista está vacía?
  if not EstaVaciaLaLista then begin
    // NO, ¿La producción cargada sufrió cambios?
    if Produc.text<>ListaClases.Items[indActivoLista] then
      // Actualizamos los cambios antes de insertar nueva.
      ModifReglaProduc(indActivoLista, Produc.text);
    ListaClases.Sorted:=true;
    ListaClases.Sorted:=false;
    Produc.tag:=-1;
```

```
    ListaClasesChange(nil);
    ArchSeModif;
  end;
end;
```

Antes de ordenar vemos si la regla que encuentra cargada en el editor **Produc** sufrió cambios para actualizarlos antes de ordenar.

```
    if Produc.text<>ListaClases.Items[indActivoLista] then
      ModifReglaProduc(indActivoLista, Produc.text);

procedure tfrmEbnf2Dt.ModifReglaProduc(const xInd:integer; xProduc:String);
begin
  tRegla(ListaClases.items.objects[xInd]).Produccion:=xProduc;
end;
```

Inmediatamente después ordenamos la lista

```
  ListaClases.Sorted:=true;
```

Una vez ordenada la lista desactivamos la propiedad **sorted** ya que por oimisión solo ordenamos la lista a petición del usuario; de esta manera los elementos de la lista se siguen agregando en el orden en que se crean.

```
  ListaClases.Sorted:=false;
```

Después forzamos la actualización del diagrama ya que al ordenar el renglón activo es el primero de la lista.

```
    Produc.tag:=-1;
    ListaClasesChange(nil);
```

Por último indicamos que el archivo sufrió modificaciones.

```
procedure TfrmEbnf2Dt.ArchSeModif;
begin
  case edoArch of
    eaNuevo:edoArch:=eaNuevoModif;
    eaAbierto:edoArch:=eaAbiertoModif;
  end;
  // Reportamos el cambio de estado del Arch.
  ActualizarMsjsUsuario;
```

```
end;
```

Tomamos ahora el botón  que nos permite al hacer click en él crear una clase sintáctica nueva. A este botón le asociamos el método:

```
procedure TfrmEbnf2Dt.tbrListaClasesNuevaClick(Sender: TObject);
  var vCad:string;
     i:integer;
begin
  frmClaseSint:=TfrmClaseSint.Create(Application);
  frmClaseSint.Caption:='ALTA de Reglas de Producción';
  frmClaseSint.edtNombre.text:='';
  if frmClaseSint.ShowModal = mrOk then
  begin
   //¿El usuario capturó algún nombre?
    if frmClaseSint.edtNombre.text<>'' then
    begin
     // ¿La lista está vacía?
      if not EstaVaciaLaLista then
     // NO, ¿La producción cargada sufrió cambios?
     //if Produc.text<>ListaClases.Items[indActivoLista] then
     // Actualizamos los cambios antes de insertar nueva.
        ModifReglaProduc(indActivoLista, Produc.text);
     // Insertamos la nueva regla
      vCad:='';
      for i:=1 to length(frmClaseSint.edtNombre.text) do
      begin
        if frmClaseSint.edtNombre.text[i]=' ' then
          vCad:=vCad + '_'
        else
        begin
          if frmClaseSint.edtNombre.text[i] in cIdentificador then
            vCad:=vCad + frmClaseSint.edtNombre.text[i];
        end;
      end;
      indActivoLista:=InsertReglaProduc(vCad);
      Produc.tag:=-1;
      ListaClasesChange(nil);
      ArchSeModif;
    end;
  end;
end;
```

```
function tfrmEbnf2Dt.InsertReglaProduc(const xNombre:string):integer;
  var Reg:tRegla;
begin
  Reg:=tRegla.Create('');
  result:=ListaClases.Items.AddObject(xNombre, Reg);
end;
```

Para capturar los nombres de clase sintácticas creamos una forma **frmClaseSint**. Nótese que al insertar una clase se crea el ejemplar asociado a la regla de producción. No hay mucho más que comentar ya que el método asegura que la nueva clase quede cargada en el editor **Produc** para que el usuario defina su regla.

Tomamos el siguiente botón  que nos sirve para eliminar clases sintácticas de la lista con todo y la regla asociada. A este botón le asociamos en el evento **OnClik** el siguiente método:

```
procedure TfrmEbnf2Dt.tbrListaClasesEliminarClick(Sender: TObject);
  var reng:integer;
begin
  //¿Hay reglas por borrar?
  if ListaClases.Items.count>0 then
  begin
    if MessageDlg('Desea eliminar la regla : '+
      ListaClases.Items[indActivoLista] +'?', mtConfirmation,
      [mbYes, mbNo], 0) = mrYes then
    begin
      if MessageDlg('Está seguro que desea eliminar '+
        'la regla : '+ ListaClases.Items[indActivoLista] +'?',
        mtConfirmation, [mbYes, mbNo], 0) = mrYes then
      begin
       // Guardamos el indice activo
        reng:=indActivoLista;
       // Eliminamos la regla deseada
        ListaClases.Items.Delete(indActivoLista);
        if ListaClases.Items.Count= 0 then
          indActivoLista:=-1
        else
          indActivoLista:=min(reng, ListaClases.Items.Count-1);
        Produc.tag:=-1;
        ListaClasesChange(nil);
        ArchSeModif;
```

```
      end;
    end;
  end;
end;
```

Tampoco hay mucho por comentar; el método se encarga de asegurarse que el usuario desea eliminar la clase sintáctica y antes de eliminar la clase de la lista elimina el ejemplar **tRegla** asociado, después se asegura que se dibuje el diagrama correspondiente a la clase a la que la lista está apuntando después de la eliminación.

El siguiente botón  nos permite modificar el nombre de la clase sintáctica: a éste le asignamos en el evento **OnClick** el método:

```
procedure TfrmEbnf2Dt.tbrListaClasesModificarClick(Sender: TObject);
  var reng, i:integer;
    vCad:string;
begin
  //¿Hay reglas por modificar?
  if ListaClases.Items.count>0 then
  begin
    reng:=indActivoLista;
    frmClaseSint:=TfrmClaseSint.Create(Application);
    frmClaseSint.Caption:='MODIFICACIÓN de Reglas de Producción';
    frmClaseSint.edtNombre.text:=ListaClases.Items[indActivoLista];
    if frmClaseSint.ShowModal = mrOk then begin
      vCad:='';
      for i:=1 to length(frmClaseSint.edtNombre.text) do
      begin
        if frmClaseSint.edtNombre.text[i]=' ' then
          vCad:=vCad + '_'
        else
        begin
          if  frmClaseSint.edtNombre.text[i] in cIdentificador then
            vCad:=vCad + frmClaseSint.edtNombre.text[i];
        end;
      end;
      ListaClases.Items[reng]:=vCad;
      ArchSeModif;
    end;
    indActivoLista:=reng;
  end;
end;
```

Nuestro último botón de este grupo es  que al hacer click en él nos permite buscar una clase en la lista sin necesidad de recorrer la lista. A este botón en el evento **OnClick** le asociamos el evento:

```
procedure TfrmEbnf2Dt.tbrBuscarClaseClick(Sender: TObject);
  var reng:integer;
begin
  reng:=ListaClases.Items.IndexOf(InputBox('Búsqueda de clases', '¿Qué clase desea
                                                                    buscar?', ''));
  if reng<>-1 then begin
    ListaClases.ItemIndex:=reng;
    ListaClasesChange(nil);
  end;
end;
```

Busca la clase indicada por el usuario; si la encuentra fuerza el cambio de renglón en lista para que se dibuje el diagrama correspondiente.

Pasamos a analizar los botones asociados con la generación de metarchivos.

Tomamos el primer botón  que nos permite copiar el diagrama activo al portapapeles. A este botón le asignamos en el evento **OnClick** el método:

```
procedure TfrmEbnf2Dt.tbrCopiarEnPortapapelesClick(Sender: TObject);
begin
  CopiarProdEnMetaArch(maAlPortapapeles, ListaClases.ItemIndex);
end;
```

El botón  nos permite enviar el diagrama activo a una archivo del tipo ***emf*** (Enhanced Metafile); le asignamos en el evento **OnClick** el método:

```
procedure TfrmEbnf2Dt.tbrGrabarEnMetarchivoClick(Sender: TObject);
begin
  CopiarProdEnMetaArch(maConfirmarGuardar, ListaClases.ItemIndex);
end;
```

El último botón de este grupo es  , el cual nos permite enviar todos los diagramas de

la lista como archivos ***emf*** a un subdirectorio que el usuario elige, a este botón le asignamos en el evento **OnClick** el método:

```
procedure TfrmEbnf2Dt.tbrGrabarMetarchivosClick(Sender: TObject);
  var i,j:integer;
     vProducText:string;
begin
  j:=ListaClases.ItemIndex;
  GrabandoMetarchivos:=true;
  if dlgBuscarDir.execute then
  begin
    DirectorioMetaArchivos:=dlgBuscarDir.Directory+'\';
    for i:=0 to ListaClases.Items.Count-1 do
      CopiarProdEnMetaArch(maGuardar, i);
  end;
  GrabandoMetarchivos:=False;
  ListaClases.ItemIndex:=j;
end;
```

Este método es muy sencillo y le permite al usuario (por medio del componente ) elegir el subdirectorio en donde desea guardar todos los archivos generados; para tal efecto recorre toda la lista y va generando los diagramas como un archivo ***emf***. La bandera **GrabandoMetaarchivos** nos sirve para que cuando estemos generando todos los archivos no se generen los diagramas visualmente; su uso quedará claro cuando analicemos el método **OnPaint** asociado al componente **Lienzo**. En los tres últimos métodos aparece invocado el método **CopiarprodEnMetaArch** con parámetros diferentes, este método es el siguiente:

```
procedure TfrmEbnf2Dt.CopiarProdEnMetaArch(const xOpcionMetaArchivo:
  tOpcionMetaArchivo; xInd:integer);
  var vProducText:string;
begin
  NombreMetaArchivo:=ListaClases.Items.Strings[xInd];
  vProducText:=tRegla(ListaClases.Items.Objects[xInd]).Produccion;

  TablaSimbs.Clear;
  BorrarArbSintac; // Borramos el árbol sintáctico
  ASintactico.ReconocerClaseSint(NombreMetaArchivo
                +cSepProduc+ vProducText+cFinProduc);  // Efectuamos el reconocimiento
  Dibujar.AbrirMetaArchivo;
  //Actualizamos parámetros
  Dibujar.AsignarFuente(dlgDiagramaFuente.Font);
```

```
    Dibujar.Actualizar;

    // Medimos los símbolos
    tSmb(arbSintac.Items[0].getFirstChild.Data).Medir(arbSintac.Items[0].getFirstChild);

    Dibujar.MetaArchivoAsignarMedidas(
                                        tSmb(arbSintac.Items[0].GetFirstChild.Data).Dx,
                                        tSmb(arbSintac.Items[0].GetFirstChild.Data).Dy);

    //Actualizamos parámetros
    Dibujar.AsignarFuente(dlgDiagramaFuente.Font);
    Dibujar.Actualizar;

    // Dibujamos
    tSmb(arbSintac.Items[0].GetFirstChild.Data).Pintar(arbSintac.Items[0].GetFirstChild,
                                                                        dDer, 0, 0);
    Dibujar.CerrarMetaArchivo(xOpcionMetaArchivo);
  end;
```

El método es prácticamente igual al de **DibujarProducción**, únicamente que la producción que se dibuja se toma de la lista de clases; además se involucra el manejo de metarchivos cuyo funcionamiento se explicó en la sección anterior; con esto le indicamos a la clase **Dibujar** que debe dibujar en un metarchivo y no en el componente **Lienzo** asociado a la pantalla. Por último, el parámetro **xOpcionMetaArchivo** le indica a la clase **Dibujar** si antes de cerrar el metarchivo debe copiarlo en el portapapeles o en un archivo cuyo destino debe elegir el usuario o guardarlo en un subdirectorio fijo que previamente eligió el usuario.

El componente **edEditorFuente**  , nos sirve poder elegir la fuente utilizada en el componente **Produc** que es nuestro editor reglas de producción. Simplemente le asociamos en el evento **OnClick** el método:

```
procedure TfrmEbnf2Dt.edEditorFuenteClick(Sender: TObject);
begin
  begin
    dlgDiagramaFuente.Font.Name:=edEditorFuente.text;
    if dlgDiagramaFuente.Execute then begin
      Produc.Font:=dlgDiagramaFuente.Font;
      edEditorFuente.text:=dlgDiagramaFuente.Font.Name;
    end;
  end;
```

```
end;
```

No hay comentarios adicionales por lo que pasamos ahora al componente **Produc** que es nuestro editor de reglas de producción:

De este componente nos interesan dos eventos. El primero es cuando el componente sufre cambios; por lo cual asociamos al evento **OnChange** el método:

```
procedure TfrmEbnf2Dt.ProducSeModifico(Sender: tObject);
begin
  ArchSeModif;
  DibujarProduccion;
end;
```

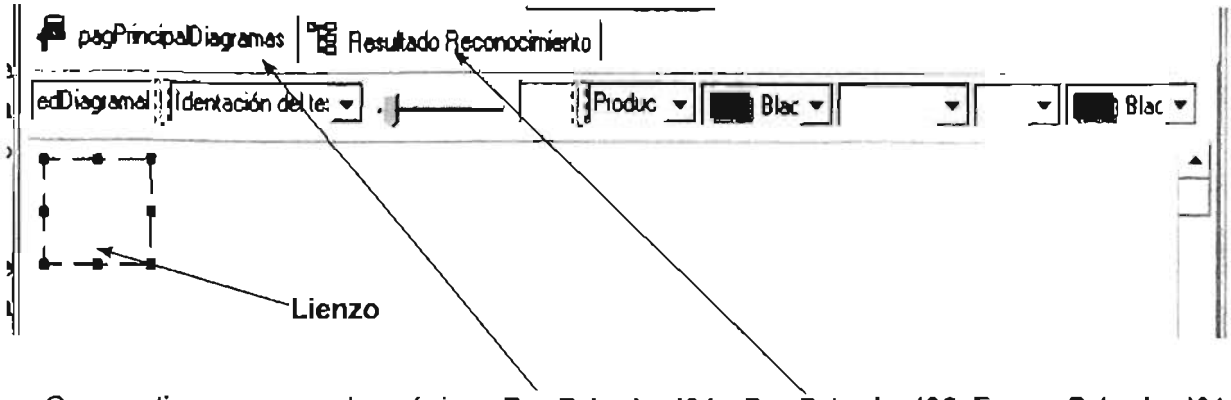
Con esto logramos que el usuario vea en tiempo real las modificaciones que le hace a la regla y por otro lado reflejamos que el estado del archivo se modificó.

El otro evento que nos interesa se presenta cuando el usuario hace Doble-Click, por eso asociamos al evento **OnDblClick** el método :

```
procedure TfrmEbnf2Dt.ProducDblClick(Sender: TObject);
begin
  ListaClases.ItemIndex:=ListaClases.Items.IndexOf(Produc.SelText);
  ListaClasesChange(nil);
end;
```

Esto le permite al usuario otra manera de navegar a través de la lista de clases sintácticas yq que cuando el usuario hace doble click en el texto de **Produc**, se selecciona la parte del texto en donde se hizo doble click si este texto corresponde a una clase sintáctica esta se carga inmediatamente como si se hubiera elegido de la lista.

Analizamos ahora el componente **pagPrincipal**

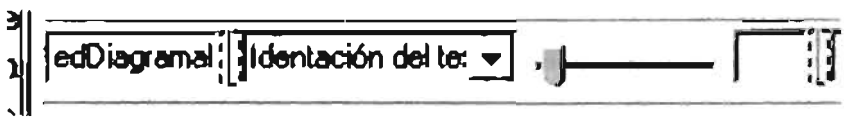


Que contiene a su vez dos páginas **PagPrincipal01** y **PagPrincipal02**. En **pagPrincipal01** se encuentran todas las opciones para configurar la forma en que se dibujan los diagramas y además contiene al componente **Lienzo** que es el área donde se dibuja el diagrama. En particular de este componente nos interesa sólo el evento **OnPaint** al cual le asociamos el método:

```
procedure TfrmEbnf2Dt.LienzoPaint(Sender: TObject);
begin
  if (not GrabandoMetarchivos) then
    DibujarProduccion;
end;
```

El evento **OnPaint** se dispara cada que el componente necesita repintarse en la pantalla. Al asociar a este evento nuestro método para dibujar nos aseguramos que en todo momento, sin importar cómo estemos manipulando nuestro sistema (ya sea sólo o en combinación con otras aplicaciones), el dibujo del diagrama siempre va está visible para el usuario. Es aquí donde utilizamos la bandera **GrabandoMetarchivos,** ya que cuando estamos generando todos no nos sirve que el dibujo se refleje en pantalla, pues esto alentaría la generación de los archivos.

Tomamos ahora el conjunto de botones asociados con la configuración de los tamaños de los diagramas:



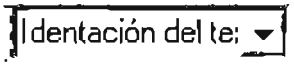
Tomamos primero la caja de texto 

, cuando se hace click en ella le

asociamos:

```
procedure TfrmEbnf2Dt.edDiagramaFuenteClick(Sender: TObject);
begin
  begin
    dlgDiagramaFuente.Font.Name:=edDiagramaFuente.text;
    if dlgDiagramaFuente.Execute then begin
      Dibujar.AsignarFuente(dlgDiagramaFuente.Font);
      Dibujar.Actualizar;
      DibujarProduccion;
      edDiagramaFuente.text:=dlgDiagramaFuente.Font.Name;
      scbDiagramaBarraVer.SetFocus;
    end;
  end;
end;
```

Este método utiliza el componente  , que muestra una caja de diálogo para que el usuario pueda elegir la fuente, su estilo y tamaño; el color aunque se muestra aquí no pinta porque se le permite al usuario elegirlo para la producción, los símbolos terminales y las clases sintácticas en otro grupo de controles.

La lista de parámetros 

le permiten elegir al usuario el parámetro que desea configurar, ya que la lista contiene todos los parámetros que modifican el tamaño de los diagramas como son:

- Alineación del texto
- Interlineado del texto
- Margen entre figuras
- Longitud línea del texto
- Longitud línea de las figuras
- Longitud flechas

A este componente le asociamos en el evento **OnChange** el método:

```
procedure TfrmEbnf2Dt.cbDiagramaParamNombreChange(Sender: TObject);
begin
  case cbDiagramaParamNombre.ItemIndex of
    0:
    begin
      slDiagramaParamValor.value:=PctIx;
```

```
      edDiagramaParamValor.text:=IntToStr(PctIx);
    end;
    1:
    begin
      slDiagramaParamValor.value:=PctRy;
      edDiagramaParamValor.text:=IntToStr(PctRy);
    end;
    2:
    begin
      slDiagramaParamValor.value:=PctMy;
      edDiagramaParamValor.text:=IntToStr(PctMy);
    end;
    3:
    begin
      slDiagramaParamValor.value:=PctCx;
      edDiagramaParamValor.text:=IntToStr(PctCx);
    end;
    4:
    begin
      slDiagramaParamValor.value:=PctLx;
      edDiagramaParamValor.text:=IntToStr(PctLx);
    end;
    5:
    begin
      slDiagramaParamValor.value:=PctFl;
      edDiagramaParamValor.text:=IntToStr(PctFl);
    end;
  end;
end;
```

Lo único que hace es cargar a los componentes

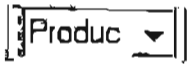
con los valores de los porcentajes asociados al parámetro elegido; para que el usuario al manipular cualquiera de los dos actualice los valores de los parámetros. Esto lo logramos asociando a estos dos componentes en el evento **OnChange** los siguientes métodos correspondientes:

```
procedure TfrmEbnf2Dt.slDiagramaParamValorChange(Sender: TObject);
begin
```

```
    edDiagramaParamValor.Text:=IntToStr(slDiagramaParamValor.value);
    case cbDiagramaParamNombre.ItemIndex of
      0:PctIx:=slDiagramaParamValor.value;
      1:PctRy:=slDiagramaParamValor.value;
      2:PctMy:=slDiagramaParamValor.value;
      3:PctCx:=slDiagramaParamValor.value;
      4:PctLx:=slDiagramaParamValor.value;
      5:PctFl:=slDiagramaParamValor.value;
    end;
    Dibujar.Actualizar;
    DibujarProduccion;
  end;

  procedure TfrmEbnf2Dt.edDiagramaParamValorChange(Sender: TObject);
  begin
    slDiagramaParamValor.value:=StrToInt(edDiagramaParamValor.Text);
    case cbDiagramaParamNombre.ItemIndex of
      0:PctIx:=slDiagramaParamValor.value;
      1:PctRy:=slDiagramaParamValor.value;
      2:PctMy:=slDiagramaParamValor.value;
      3:PctCx:=slDiagramaParamValor.value;
      4:PctLx:=slDiagramaParamValor.value;
      5:PctFl:=slDiagramaParamValor.value;
    end;
    Dibujar.Actualizar;
    DibujarProduccion;
  end;
```

Ambos métodos toman los valores que el usuario haya elegido y se actualizan de acuerdo al parámetro elegido, actualizan las medidas de los valores elegidos y dibujan la producción; esto permite que el usuario vea de manera inmediata los cambios que está realizando; esto es así para todos los parámetros y características que se pueden configurar de los diagramas.

Tomamos el siguiente grupo de botones que nos permiten configurar características no del tamaño, pero sí del aspecto de nuestros diagramas; de acuerdo al tipo de símbolo que elijamos.

La lista 

nos permite elegir el tipo de símbolo que deseamos configurar, los símbolos considerados son:

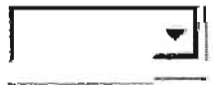
- Produc
- CnjAlts
- PartOpc
- PartRepet
- SmbTerm
- ClasSint
- Saltos

De esta manera podemos asignar diferente color a la fuente por ejemplo, para la producción, que es en este contexto, el nombre de la clase sintáctica, los colores de las líneas y el estilo y ancho de las líneas. Por lo tanto a esta lista le asignamos en el evento **OnClick** el método

```
procedure TfrmEbnf2Dt.cbSimbsNombChange(Sender: TObject);
begin
  cbSimbsLineaColor.ColorValue:=arrSimbsLineaColor[cbSimbsNomb.ItemIndex];
  cbSimbsLineaEstilo.Selection:=arrSimbsLineaEstilo[cbSimbsNomb.ItemIndex];
  cbSimbsLineaAncho.Selection:=arrSimbsLineaAncho[cbSimbsNomb.ItemIndex];
  if cbSimbsNomb.ItemIndex in[0,4,5] then
  begin
    cbSimbsFuenteColor.ColorValue:=arrSimbsFuenteColor[cbSimbsNomb.ItemIndex];
    cbSimbsFuenteColor.visible:=true;
  end else
    cbSimbsFuenteColor.visible:=false;
end;
```

El método asigna a los botones asociados los valores del tipo de símbolo que deseamos configurar.

es un lista de colores de línea para elegir


es una lista con los estilos de línea para elegir


es una lista con los anchos de línea a elegir

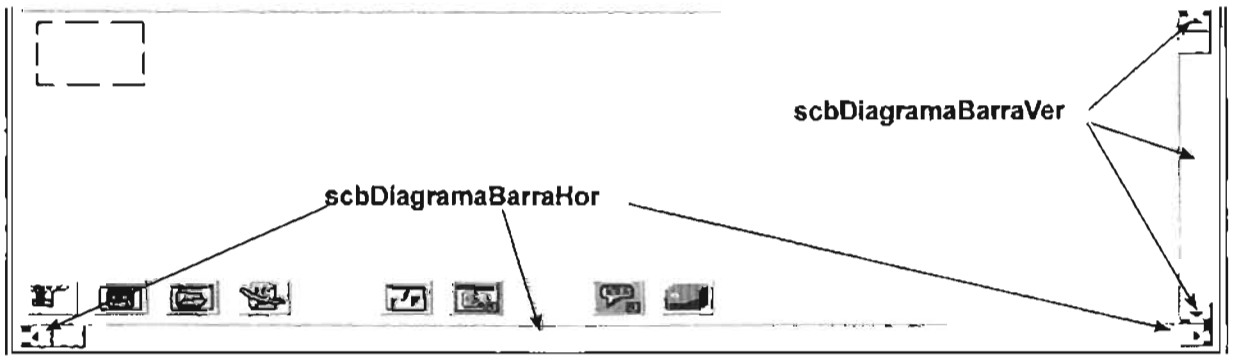
es una lista con los colores de fuente para elegir

A los eventos **OnChange** de cada uno de los cuatro componentes anteriores les asignamos correspondientemente los siguiente métodos:

```
procedure TfrmEbnf2Dt.cbSimbsLineaColorChange(Sender: TObject);
begin
  arrSimbsLineaColor[cbSimbsNomb.ItemIndex]:=cbSimbsLineaColor.ColorValue;
  DibujarProduccion;
end;

procedure TfrmEbnf2Dt.cbSimbsLineaEstiloChange(Sender: TObject);
begin
  arrSimbsLineaEstilo[cbSimbsNomb.ItemIndex]:=cbSimbsLineaEstilo.Selection;
  DibujarProduccion;
end;

procedure TfrmEbnf2Dt.cbSimbsLineaAnchoChange(Sender: TObject);
begin
  arrSimbsLineaAncho[cbSimbsNomb.ItemIndex]:=cbSimbsLineaAncho.Selection;
  DibujarProduccion;
end;

procedure TfrmEbnf2Dt.cbSimbsFuenteColorChange(Sender: TObject);
begin
  arrSimbsFuenteColor[cbSimbsNomb.ItemIndex]:=cbSimbsFuenteColor.ColorValue;
  DibujarProduccion;
end;
```

Bastante obvio por lo que no merece más comentarios.

Por último de pagPrincipal01 nos quedan los componentes

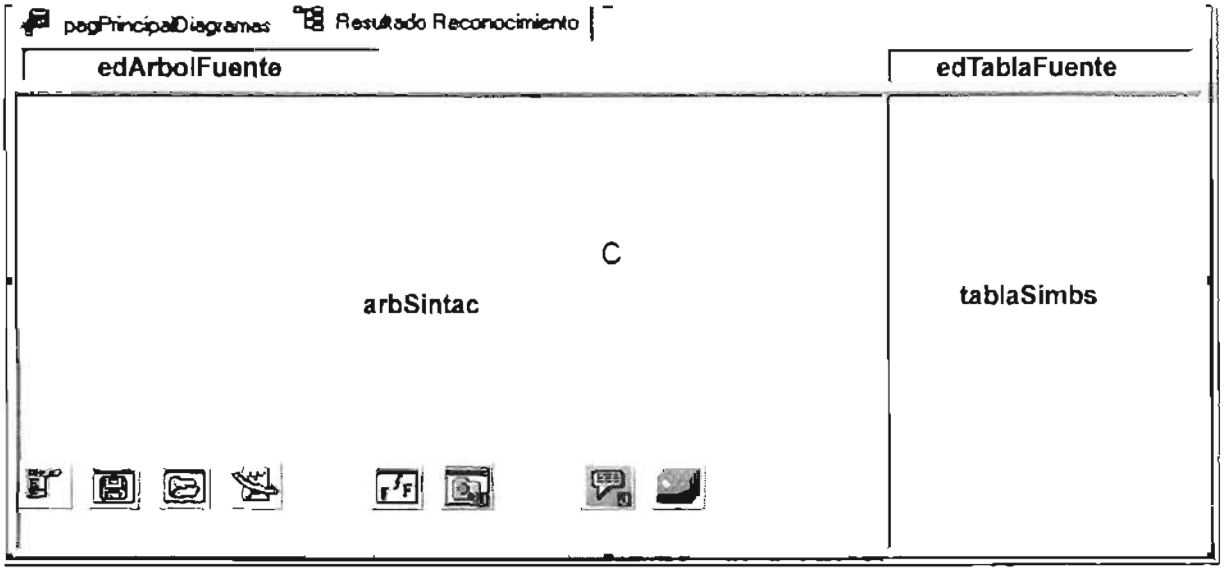


A las dos barras de desplazamiento les asignamos en el evento **OnChange** los siguientes métodos correspondientes:

```
procedure TfrmEbnf2Dt.scbDiagramaBarraHorChange(Sender: TObject);
begin
  Lienzo.left:=-scbDiagramaBarraHor.Position;
  If (Lienzo.left=0) then
    Lienzo.Left:=5;
  DibujarProduccion;
end;

procedure TfrmEbnf2Dt.scbDiagramaBarraVerChange(Sender: TObject);
begin
  Lienzo.Top:=-scbDiagramaBarraVer.Position;
  If (Lienzo.Top=0) then
    Lienzo.Top:=5;
  DibujarProduccion;
end;
```

Mueve al lienzo hacia la izquierda o hacia arriba y dibuja a la producción para reflejar el desplazamiento; útil cuando el diagrama no cabe en la pantalla.

Pasamos ahora a **pagPrincipal02**



Como podemos ver contiene cuatro componentes **arbSintac** que es un ejemplar del tipo **tTreeView** y que representa a nuestro árbol sintáctico, **tablaSimbs** que es nuestra tabla de

símbolos. La caja de texto **edArbolFuente** nos permite escoger el tipo de fuente del texto del árbol; esto lo logramos asociando a dicho componente en el evento **OnClick** el método:

```
procedure TfrmEbnf2Dt.edTablaFuenteClick(Sender: TObject);
begin
  pgcPrincipal.ActivePageIndex:=1;
  dlgDiagramaFuente.Font.Name:=edTablaFuente.text;
  if dlgDiagramaFuente.Execute then begin
    TablaSimbs.Font:=dlgDiagramaFuente.Font;
    edTablaFuente.text:=dlgDiagramaFuente.Font.Name;
  end;
end;
```

Hacemos los mismo para el componente edTablaFuente:

```
procedure TfrmEbnf2Dt.edTablaFuenteClick(Sender: TObject);
begin
  pgcPrincipal.ActivePageIndex:=1;
  dlgDiagramaFuente.Font.Name:=edTablaFuente.text;
  if dlgDiagramaFuente.Execute then begin
    TablaSimbs.Font:=dlgDiagramaFuente.Font;
    edTablaFuente.text:=dlgDiagramaFuente.Font.Name;
  end;
end;
```

El último componente que nos falta es  que representa el menú principal de la aplicación. A cada uno de sus elementos le asignamos en la propiedad ***OnClick*** los siguientes métodos que no vamos a analizar porque no aportan nada nuevo; son solo el complemento perfecto para que el usuario manipule el sistema como mejor la parezca, ya se por medio de los botones o del menú.

```
procedure TfrmEbnf2Dt.mnuVerListaClick(Sender: TObject);
begin
  ListaClases.SetFocus;
end;
procedure TfrmEbnf2Dt.mnuVerEditorClick(Sender: TObject);
begin
  Produc.SetFocus;
end;
procedure TfrmEbnf2Dt.mnuVerArbolClick(Sender: TObject);
```

```
begin
  pgcPrincipal.ActivePageIndex:=1;
  arbSintac.SetFocus;
end;

procedure TfrmEbnf2Dt.mnuVerDiagramaClick(Sender: TObject);
begin
  pgcPrincipal.ActivePageIndex:=0;
  scbDiagramaBarraHor.SetFocus;
end;

procedure TfrmEbnf2Dt.mnuVerDiagBarraHorClick(Sender: TObject);
begin
  scbDiagramaBarraHor.SetFocus;
end;

procedure TfrmEbnf2Dt.mnuVerDiagBarraVertClick(Sender: TObject);
begin
  scbDiagramaBarraVer.SetFocus;
end;

procedure TfrmEbnf2Dt.mnuVerTablaClick(Sender: TObject);
begin
  pgcPrincipal.ActivePageIndex:=1;
  tablaSimbs.SetFocus;
end;

procedure TfrmEbnf2Dt.mnuDiagramaNombresClick(Sender: TObject);
begin
  pgcPrincipal.ActivePageIndex:=0;
  cbDiagramaParamNombre.SetFocus;
end;

procedure TfrmEbnf2Dt.mnuDiagramaValoresClick(Sender: TObject);
begin
  pgcPrincipal.ActivePageIndex:=0;
  slDiagramaParamValor.SetFocus;
end;

procedure TfrmEbnf2Dt.mnuSimbolosNombClick(Sender: TObject);
begin
  pgcPrincipal.ActivePageIndex:=0;
  cbSimbsNomb.SetFocus;
end;

procedure TfrmEbnf2Dt.mnuSimbolosColorLineasClick(Sender: TObject);
begin
  pgcPrincipal.ActivePageIndex:=0;
```

```
  cbSimbsLineaColor.SetFocus;
end;

procedure TfrmEbnf2Dt.mnuSimbolosTipoLineasClick(Sender: TObject);
begin
  pgcPrincipal.ActivePageIndex:=0;
  cbSimbsLineaEstilo.SetFocus;
end;

procedure TfrmEbnf2Dt.mnuSimbolosAnchoLineasClick(Sender: TObject);
begin
  pgcPrincipal.ActivePageIndex:=0;
  cbSimbsLineaAncho.SetFocus;
end;

procedure TfrmEbnf2Dt.mnuSimbolosColorTextoClick(Sender: TObject);
begin
  pgcPrincipal.ActivePageIndex:=0;
  cbSimbsFuenteColor.SetFocus;
end;
```

Únicamente nos resta mostrar el mapa de programas y cómo se interrelacionan entre sí cada una de las unidades que integran el sistema.

3.6 Mapa de Programas

El sistema consta de las siguiente unidades, cada una de cuales define una o más clases

La unidad **EBnf2DtP.dpr** corresponde a nuestro proyecto en Delphi todas las unidades en Delphi utilizan una cláusula **use** para indicar las unidades de las cuales depende su implementación.

La cláusula ***use*** de esta unidad de EBnf2DtP.dpr es:

```
uses
  Forms,
  Ebnf2DtU in 'Ebnf2DtU.pas' {frmEbnf2Dt};
```

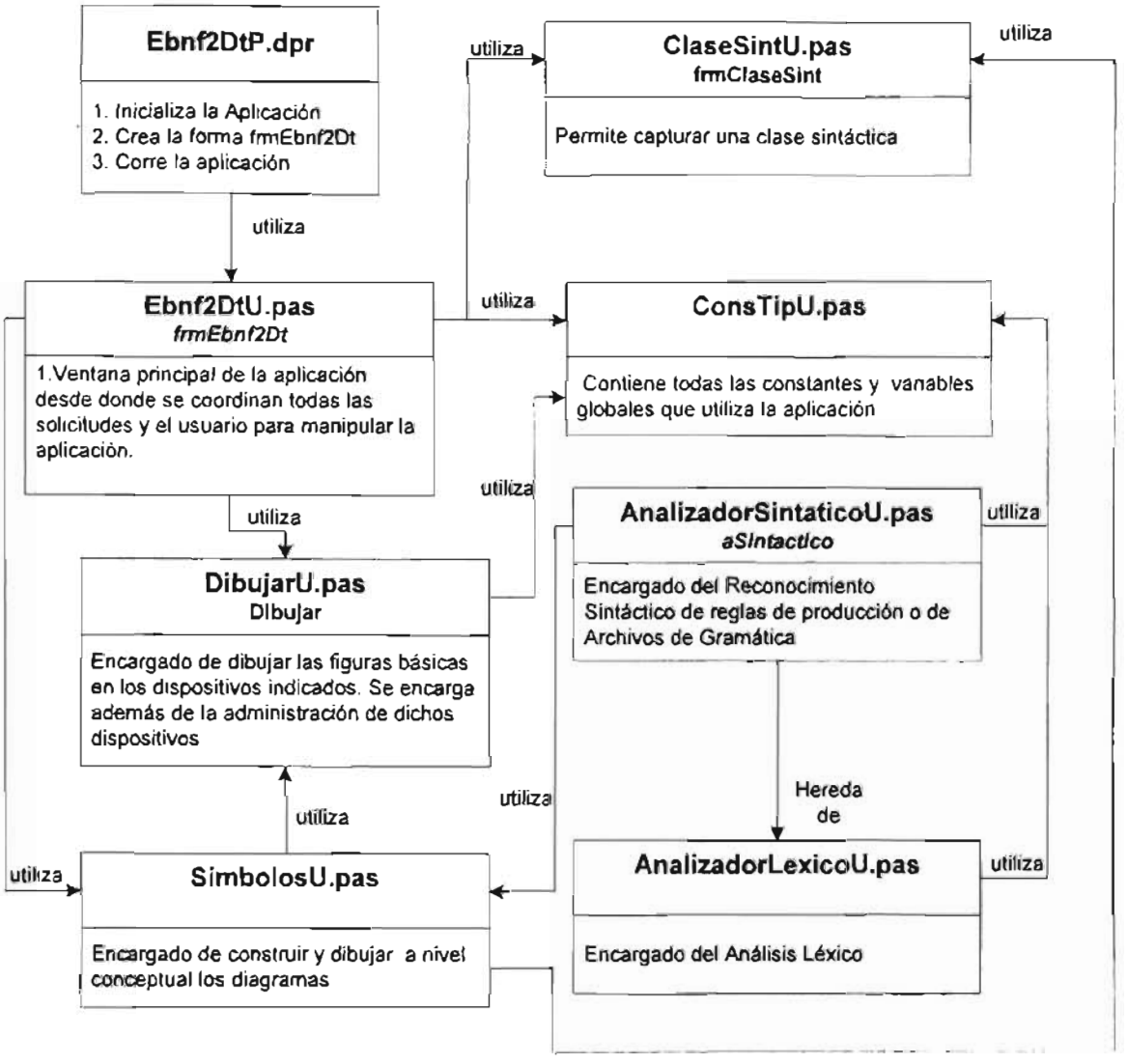


FIGURA 2.8.1

La unidad Ebnf2DtU implementa la clase **TfrmEbnf2Dt** y su cláusula ***use*** es la siguiente:

```
USES
  //Componentes Estandar de Delphi:
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls, ComCtrls, ImgList, ToolWin, StdCtrls, Menus,

  //Componentes de JEDI VCL:
  JvExExtCtrls, JvComponent, JvCaptionPanel, JvBalloonHint, JvgCaption,
```

```
JvDSADialogs, JvExControls, JvLED, JvSplit, JvSplitter, JvSyncSplitter,
JvNetscapeSplitter, JvDrawImage, JvPrvwDoc, JvPrvwRender, JvExStdCtrls,
JvCombobox, JvColorCombo, JvxSlider, JvBaseDlg, JvBrowseFolder,
JvFindReplace, PJMessageDialog,

//SynEdit: Editor para marcar la sintáxis de lenguajes de programación.
SynEditHighlighter,
SynEdit, SynEditTypes,

{UniHighLighter en combinación con SynEdit permite establecer cualquier
* Keyword.                                                    }
SynUniHighlighter, Clipbrd,

penstylecombo, penwidthcombo,// Controles para el estilo y ancho de la pluma

//Clases propias de la aplicación.
ConstipU,  AnalizadorSintacticoU, SimbolosU, DibujarU;
```

Nótese que esta unidad hace uso de todas las unidades específicas de la aplicación. Hace uso de la unidad **AnalizadorSintáctico.pas** a través del ejemplar **AnalizadorSintactico** declarado como variable global de la unidad **AnalizadorSintatico.pas**. Hace uso de la unidad **SimbolosU.pas** a través de ejemplares de las clases definidas para cada tipo de símbolo que va creando a medida que efectúa el reconocimiento sintáctico. Hace uso de la unidad **DibujarU.pas** a través del ejemplar **Dibujar** definido como variable global en la unidad.

La unidad **ClaseSintU.pas** implementa la clase **TfrmClaseSint**, y su cláusula ***use*** es:

```
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls, StdCtrls, Buttons;
```

La unidad **ConsTipU.pas** no implementa ninguna clase sino que sirve únicamente para declarar las constantes, tipos y variables globales que utilizan las demás clases. Sin embargo si tiene una cláusula ***use*** que es:

```
uses Graphics;
```

Esto es porque utiliza tipos definidos en esa clase, como por ejemplo **tColor**;

La unidad **DibujarU.pas** implementa la clase **tDibujar** que es la encargada de dibujar todas las figuras básicas y de administrar todos los dispositivos de dibujo, como son la

ventana y el portapapeles. Su cláusula ***use*** es:

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  ConstipU, Dialogs, ExtCtrls, Math, Types;
```

La unidad **AnalizadorLexicoU.pas** implementa la clase **tAnalizadorLexico** que es la encargada del análisis léxico tanto de reglas de producción como de archivos de gramática. Su cláusula **use** es:

```
USES Windows, Messages, SysUtils, Classes, Dialogs, ComCtrls, StdCtrls,
   ConstipU;
```

La unidad **AnalizadorSintáticoU.pas** implementa la clase **tAnalizadorSintactico** heredando de la clase **tAnalizadorLexico**; es la encargada del análisis sintáctico tanto de reglas de producción como de archivos de gramática. Su cláusula ***use*** es:

```
USES Windows, Messages, SysUtils, Classes, Dialogs,
   ComCtrls, StdCtrls, Contnrs, AnalizadorLexicoU, ConstipU;
```

La unidad **SimbolosU.pas** implementa una clase para cada tipo de símbolo asociado a cada nodo del árbol sintáctico. Este conjunto de clases de símbolos son las encargadas de dibujar los diagramas de trenes. Su cláusula ***use*** es:

```
uses Windows, Classes, Forms, sysutils, Graphics, Controls, ComCtrls, Math,

  // Clases Programadas para la Tesis
    ConstipU,   AnalizadorSintacticoU, DibujarU;
```

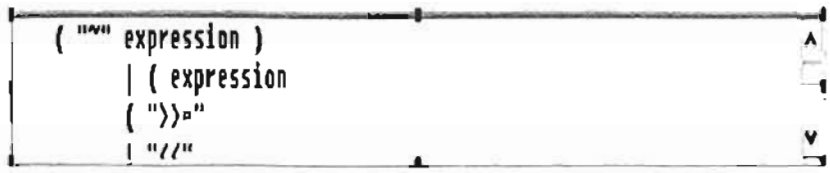
A su vez en esta sección indicamos los paquetes adicionales de componentes o bibliotecas de funciones que se utilizaron para construir la aplicación; todos son gratuitos, no tienen restricciones en su uso y son los siguientes:

Paquete de Componentes JVCL

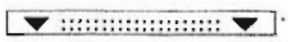
tSynUniSyn

tJvBrowseForFolderDialog

tJvBallonhint



tSynEdit



tJvNetscapeSplitter

tJvxSlider

tJvColorComboBox

disponibles en internet en la dirección:

http://prdownloads.sourceforge.net/jcl/

Los componentes

tPenStyleCombo y tPenWidthCombo

los puede encontrar en Internet en el sitio de ***Torry's Delphi Pages***.

Adicionalmente se hace uso de la unidad **cStrings.pas** para dos o tres manejos con cadenas; esta unidad es parte de un conjunto de bibliotecas de funciones llamada "Delphi Fundamentals" la cual se puede encontrar en :***http://fundementals.sourceforge.net/*** .

De cualquier manera anexo a este trabajo hay un CD en el que incluyo todos estos componentes para que la persona que desee hacer modificaciones a este sistema cuente con todos los elementos necesarios para ello.

Manual del usuario

Capítulo 4

4.1 Características de la aplicación

La mayoría de los lenguajes de programación utilizan alguna variante de ***EBFN*** para definir la gramática de su lenguaje. Éste es un requisito técnico para la construcción del compilador pero además proporciona un marco de referencia que no deja lugar a dudas acerca de la estructura del lenguaje; por lo tanto cuando queremos aprender un lenguaje un tema obligado es el estudio de esta gramática.

BNF es un acrónimo para "***Backus Naur Form***". John Backus y Peter Naur introdujeron por primera vez la notación formal para describir la sintaxis de ***ALGOL***. A los metasímbolos de ***BNF*** son:

Metasímbolo	Significado
::=	**Se define como:**
|	**"o"**
<>	**Representan clases sintácticas o nombres de reglas**
;	**Para terminar la descripción de una regla**

Los paréntesis angulares distinguen a las clases sintáticas de los símbolos terminales que se escriben exactamente como se representan. Un ejemplo de una regla de producción escrita en ***BNF*** podría ser:

```
<program> ::= program
                <declaration_sequence>
              begin
```

```
        <statements_sequence>
    end ;
```

Con el tiempo se han agregado algunas extensiones a **BNF** para facilitar su uso como son:

- Los elementos opcionales se encierran entre los metasímbolos **[** y **]**, por ejemplo:

```
<if_statement> ::= if <boolean_expression> then
                       <statement_sequence>
                       [ else
                            <statement_sequence> ]
                       end if ;
```

- Los elementos repetitivos se encierran entre los metasímbolos **{** y **}**, por ejemplo:

```
<identifier> ::= <letter> { <letter> | <digit> }
```

cuya regla es equivalente a la regla recursiva:

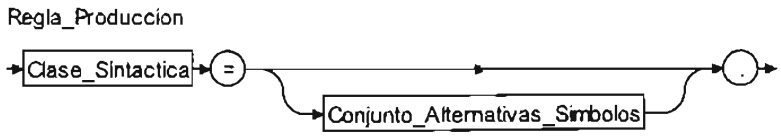
```
<identifier> ::= <letter> | <identifier> [ <letter> |
<digit> ]
```

Nuestra aplicación utiliza esta misma definición de ***BNF*** y sus extensiones con las siguientes modificaciones:

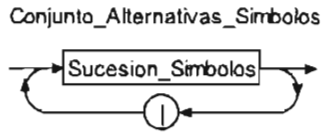
- Las clases sintácticas no se encierran entre los metasímbolos **<** y **>**.
- Son los símbolos terminales los que se encierran entre comillas simples o dobles.
- En lugar del símbolo **::=** se utiliza únicamente **=**.
- Para terminar una regla de producción en lugar de utilizar **;** se utiliza . (punto) .

Podemos aprovechar a nuestro ***BNF*** extendido para describirse a sí mismo, mostramos también el diagrama de tren asociado (estos diagramas fueron generados por nuestra aplicación):

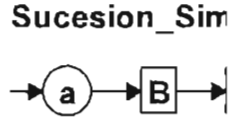
Regla_Produccion = Clase_Sintactica "=" [Conjunto_Alternativas_Simbolos] "." .



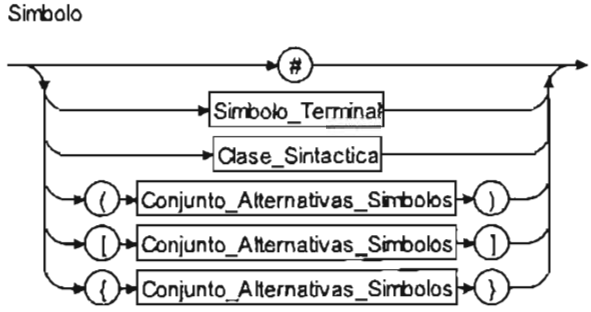
Conjunto_Alternativas_Simbolos = Sucesion_Simbolos { '|' Sucesion_Simbolos } .



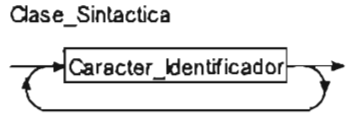
Sucesion_Simbolos = Simbolo { Simbolo } .



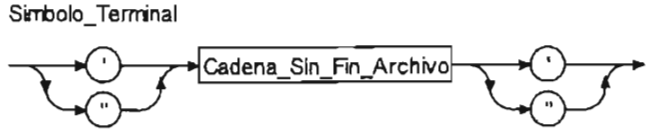
Simbolo = '#' |
Simbolo_Terminal |
Clase_Sintactica |
'(' Conjunto_Alternativas_Simbolos ')' |
'[' Conjunto_Alternativas_Simbolos ']' |
'{' Conjunto_Alternativas_Simbolos '}' .



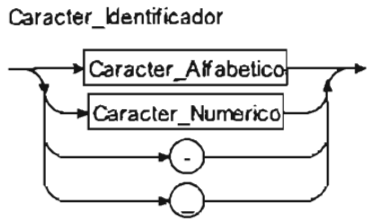
Clase_Sintactica = Caracter_Identificador { Caracter_Identificador} .



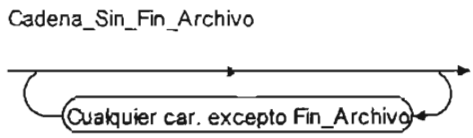
Simbolo_Terminal = ("'" | '"') Cadena_Sin_Fin_Archivo ("'" | '"') .



Caracter_Identificador = Caracter_Alfabetico | Caracter_Numerico | "-" | "_"

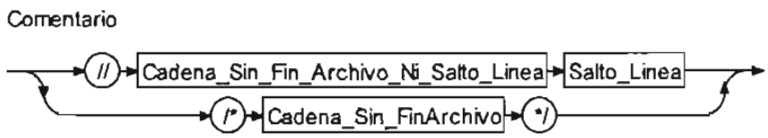


Cadena_Sin_Fin_Archivo = {Cualquier car. excepto Fin_Archivo}.



Aunque no es parte natural de EBNF, nosotros hemos incluido la posibilidad de poder incluir comentarios dentro de las reglas de producción, por lo tanto también definimos:

Comentario = ("//" CadenaSinFinArchivoNiSaltoLinea SaltoLinea) |



Obviamente los comentarios al no formar parte de la descripción se ignoran al momento del reconocimiento para generar el diagrama.

Este traductor sirve para visualizar las reglas de producción escritas en este ***BNF*** modificado como diagramas de tren. El usuario puede proceder de dos maneras para manejar las reglas que desea diagramar:

- Puede crear un archivo en cualquier procesador de texto y capturar las clases sintácticas seguidas de sus reglas asociadas, de acuerdo a las reglas del ***BNF*** modificado que el sistema maneja el cual describimos más adelante. En seguida ejecutar la aplicación, abrir el archivo creado y visualizar los diagramas.
- Ejecutar la aplicación y construir las reglas sintácticas dentro de la aplicación, ya que permite guardar el archivo generado; el cual se guarda en formato de texto; en teoría

sería igual al archivo del párrafo anterior.

Construir el archivo dentro de la aplicación ofrece varias ventajas como son:

- El usuario dispone de una lista de clases sintácticas, que puede ordenar alfabéticamente.
- Puede buscar una clase sintáctica en particular.
- El editor de reglas de producción además marca a los metasímbolos con distintos colores para su mejor identificación.
- El usuario ve inmediatamente el resultado en pantalla; es decir conforme se teclea la regla el diagrama se va dibujando .

Adicionalmente:

- La aplicación genera :
  - El árbol sintáctico asociado con la regla de producción identifcando los nodos y el tipo de símbolo reconocido.
  - La tabla de símbolos asociada con la regla.
  - Mensajes del tipo de error identificado cuando la regla es incorrecta. En ningún momento la aplicación se detiene; en todo momento genera el diagrama hasta el punto en que le fue posible efectuar un reconocimiento exitoso.
- La aplicación genera en formato de Enhanced Windows Metafile.:
  - El diagrama con el objeto de copiarlo en el portapapeles para pegarlo en cualquier aplicación que permita el manejo de gráficos del tipo Enhanced Windows Metafile.
  - En un archivo cuya localización puede elegir el usuario.
  - Con un solo click del ratón todos los diagramas del archivo cargado como archivos de EMF. En la computadora en donde se desarrolló este sistema tarda aproximadamente 1 segundo en generar alrededor de 60 diagramas.
- Se puede configurar el tamaño de las figuras que integran los diagramas, así como el tamaño, color y estilo de la fuente utilizada en el texto y las líneas de los diagramas.

4.2 Instalación

La aplicación requiere como mínimo del sistema operativo Windows 95, de un espacio en disco de aproximadamente 1.2 MB, más el necesario para los archivos de diagramas, no requiere de un ratón pero contar con uno facilita su uso.

La instalación es muy sencilla, motivo por el cual no se incluye ningún programa de instalación, únicamente es necesario copiar en el subdirectorio que se desee el archivo **Ebnf2DtP.exe** que se incluye en el CD. No se requiere de nada extra, la aplicación está lista para correr. Se sugiere crear un subdiretorio que identitique a esta aplicación y copiar ahí este ejecutable y guardar ahí también todos los archivos con los que se vaya a trabajar.

Dado su tamaño, la aplicación puede correrse desde un diskette, aunque claro la ejecución es más lenta.

No hay programa para desintalar la aplicación, ya que únicamente tendríamos que borrar el ejecutable del disco.

4..3 ¿Cómo utilizar la aplicación?

Al ejecutar la aplicación el usuario verá una ventana como la de la figura 4.3.1. La aplicación está lista para empezar a trabajar. Cada vez que el usuario mueve el ratón y se detiene sobre un control, la aplicación muestra un mensaje que indica para qué sirve dicho control y cómo utilizarlo. Como la aplicación inicia no hay nada con qué trabajar, todo está limpio o vacío.

Anallizamos primero los 4 botones que nos permiten manipular archivos, como podemos ver en la figura 4.3.1, los botones son:

Creación de archivo nuevo.

Abrir un archivo bnf existente.

Guardar el archivo que se está trabajando.

Guardar con otro nombre el archivo que se está trabajando.

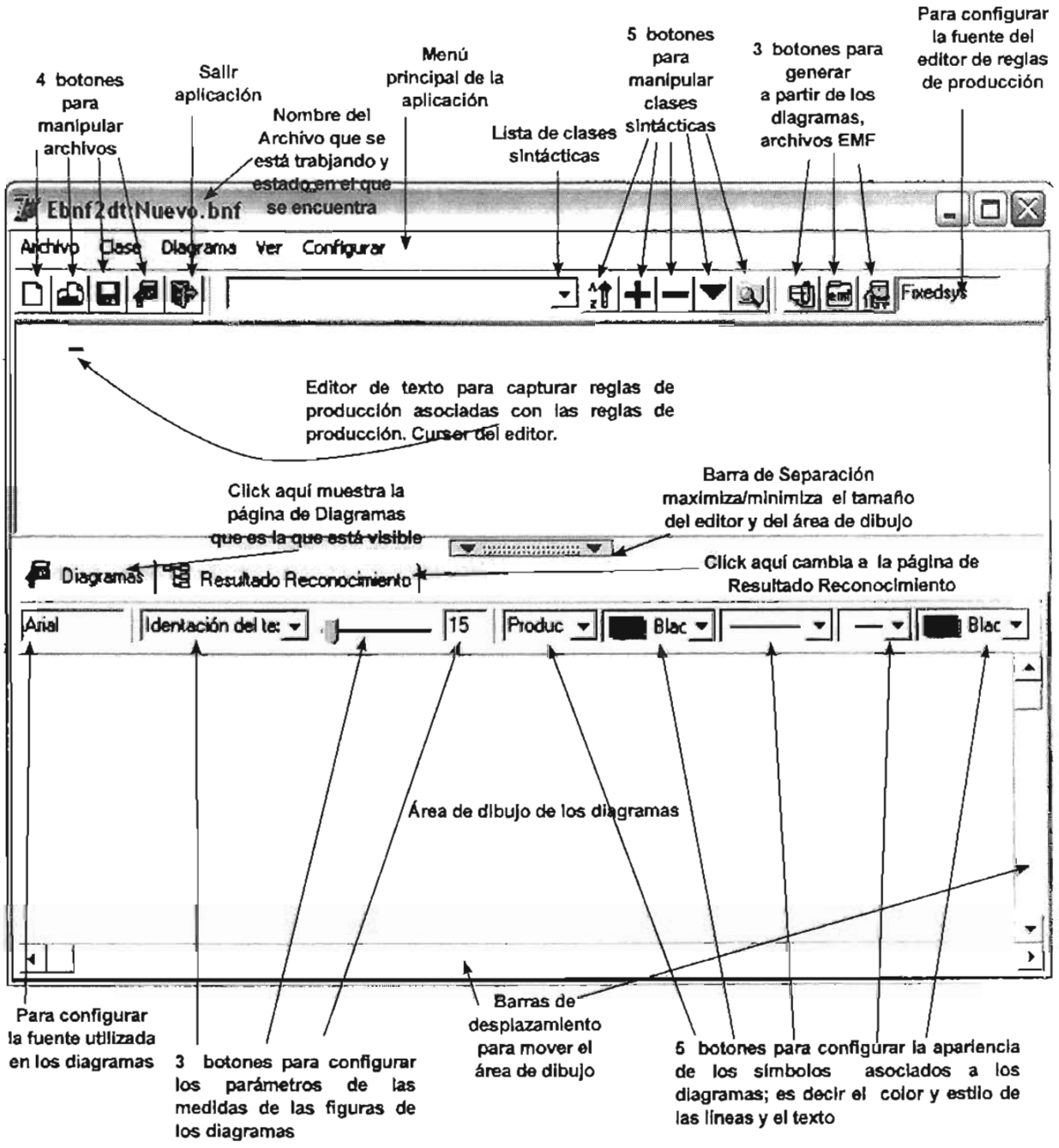
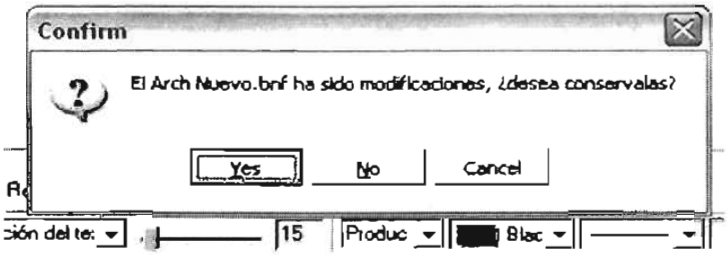


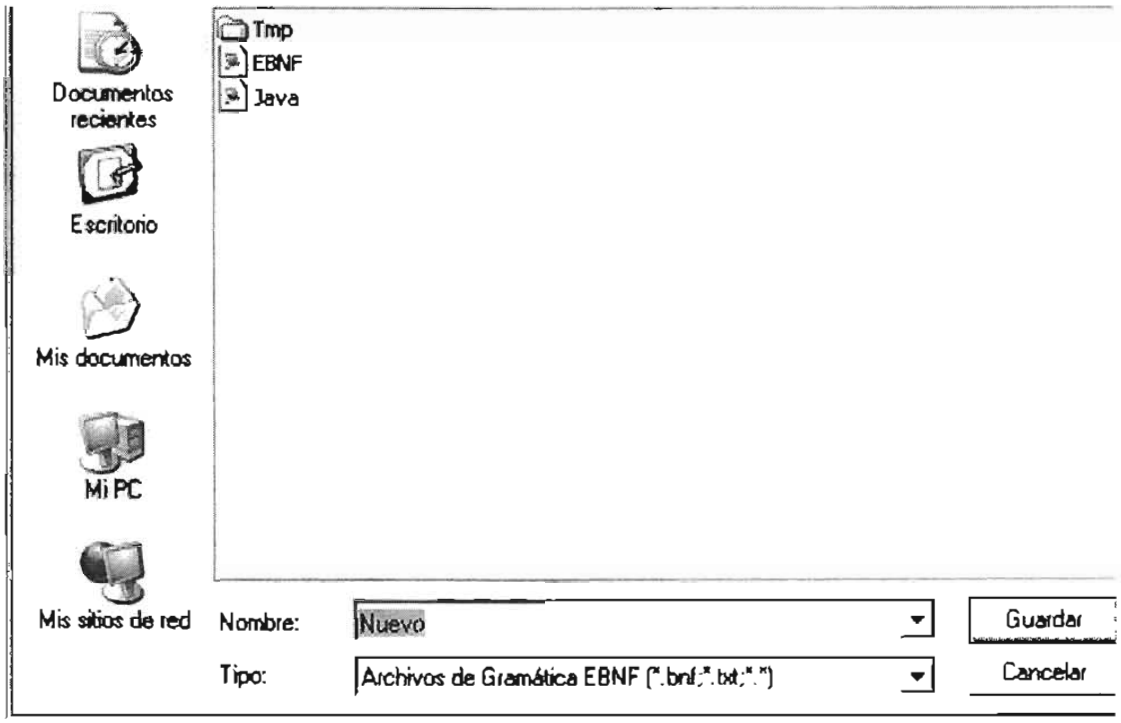
FIGURA 4.3.1

Hacemos click en  , cuando queremos crear un archivo nuevo; la aplicación procede de la siguiente manera:

- Si no hay ningún archivo cargado como cuando iniciamos la aplicación, no hace nada.
- Si hay algún archivo cargado pero no ha sufrido modificaciones, como cuando abrimos un archivo y no lo modificamos, la aplicación limpia todos los controles como cuando iniciamos la aplicación; como se muestran en la figura 4.3.1.
- Si hay algún archivo cargado que ha sufrido modificaciones, la aplicación muestra el siguiente cuadro de diálogo para que el usuario decida la acción a seguir:



- Si elegimos la opción ***Yes***, la aplicación abre el siguiente cuadro de diálogo:

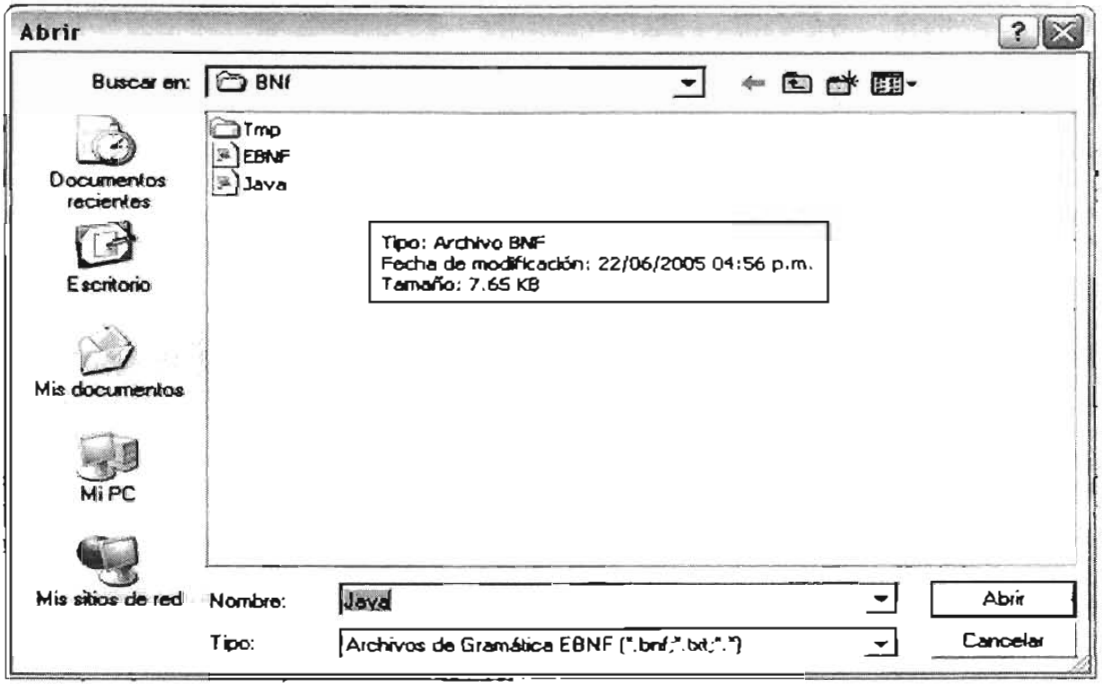


- Si elegimos la opción ***Cancelar***, el proceso se cancela por completo, es como si nunca hubiéramos hecho click en el boton ***Crear*** archivo nuevo.
- Si elegimos la opción ***Guardar*** después de elegir el subdirectorio y nombre de archivo deseado la aplicación guarda el archivo y la aplicación se inicializa.

- Si elegimos la opción ***No***, la aplicación ignora los cambios efectuados y simplemente se inicializa.
- Si elegimos la opción ***Cancel***, el proceso se cancela como si no hubiéramos hecho click en el boton ***Crear*** archivo nuevo.

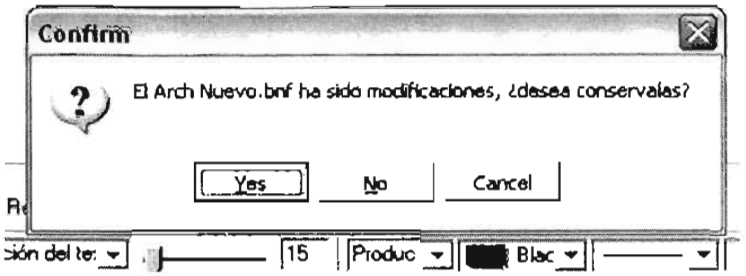
Hacemos click en  , cuando queremos abrir un archivo de gramática; la aplicación procede de la siguiente manera:

- Si no hay ningún archivo cargado como cuando iniciamos la aplicación, la aplicación muestra el siguiente cuadro de diálogo:

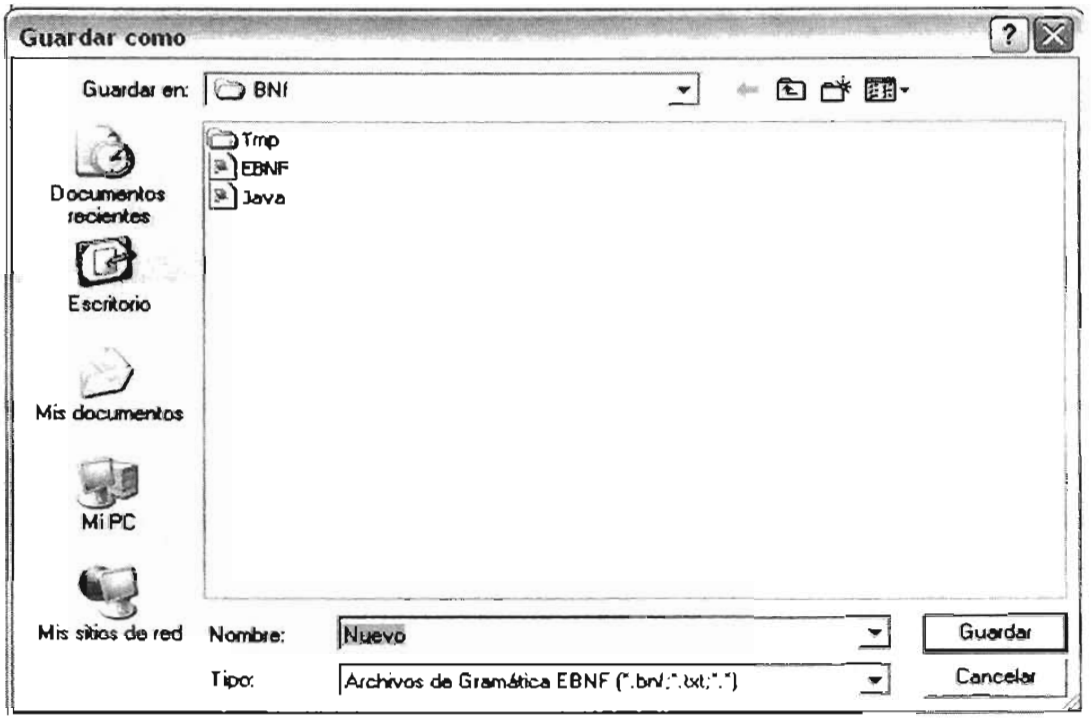


- Si elegimos la opción ***Cancelar***, es como si no hubiéramos hecho click en ***Abrir***.
- Si elegimos la opción ***Abrir*** después de elegir el subdirectorio y nombre de archivo deseado, la aplicación abre el archivo elegido y carga las clases sintácticas contenidas en el archivo.

- Si hay algún archivo cargado pero no ha sufrido modificaciones, como cuando abrimos un archivo y no lo modificamos, la aplicación procede igual que en el paso anterior.
- Si hay algún archivo cargado que ha sufrido modificaciones, la aplicación muestra el siguiente cuadro de diálogo para que el usuario decida la acción a seguir:



- Si elegimos la opción ***Yes***, la aplicación abre el siguiente cuadro de diálogo:



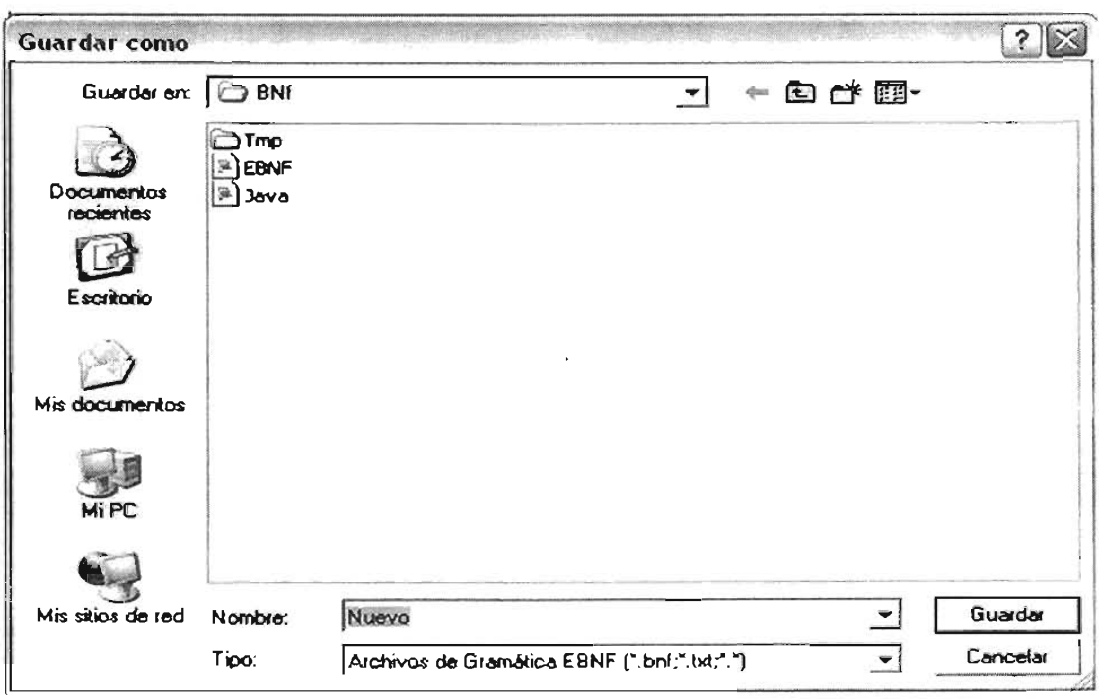
- Si elegimos la opción ***Cancelar***, el proceso se cancela por completo, es como si nunca hubiésemos hecho click en el boton ***Abrir*** archivo nuevo.
- Si elegimos la opción ***Guardar*** después de elegir el subdirectorio y nombre de archivo deseado, la aplicación guarda el archivo y muestra el cuadro de diálogo abrir; nuevamente si elegimos la opción ***Cancelar***, es como si no hubiéramos hecho click en el botón ***Abrir*** archivo. Si elegimos la opción ***Abrir*** después de elegir el subdirectorio y nombre de archivo deseado la

aplicación abre el archivo elegido y carga las clases sintácticas contenidas en el archivo.

- Si elegimos la opción ***No***, la aplicación ignora los cambios efectuados y muestra el cuadro de diálogo abrir; nuevamente si elegimos la opción ***Cancelar***, es como si no hubiéramos hecho click en el botón ***Abrir*** archivo. Si elegimos la opción ***Abrir*** después de elegir el subdirectorio y nombre de archivo deseado, la aplicación abre el archivo elegido y carga las clases sintácticas contenidas en el archivo.
- Si elegimos la opción ***Cancel***, el proceso se cancela como si no hubiéramos hecho click en el boton ***Abrir*** archivo .

Hacemos click en  , cuando queremos guardar el archivo que se está trabajando; la aplicación procede de la siguiente manera:

- Si no hay ningún archivo cargado, como cuando iniciamos la aplicación, no hace nada.
- Si hay archivo cargado pero no ha sufrido modificaciones, no hace nada.
- Si el archivo sufrió modificaciones procede de la siguente manera:
  - Si el archivo es nuevo, muestra el siguiente cuadro de diálogo:

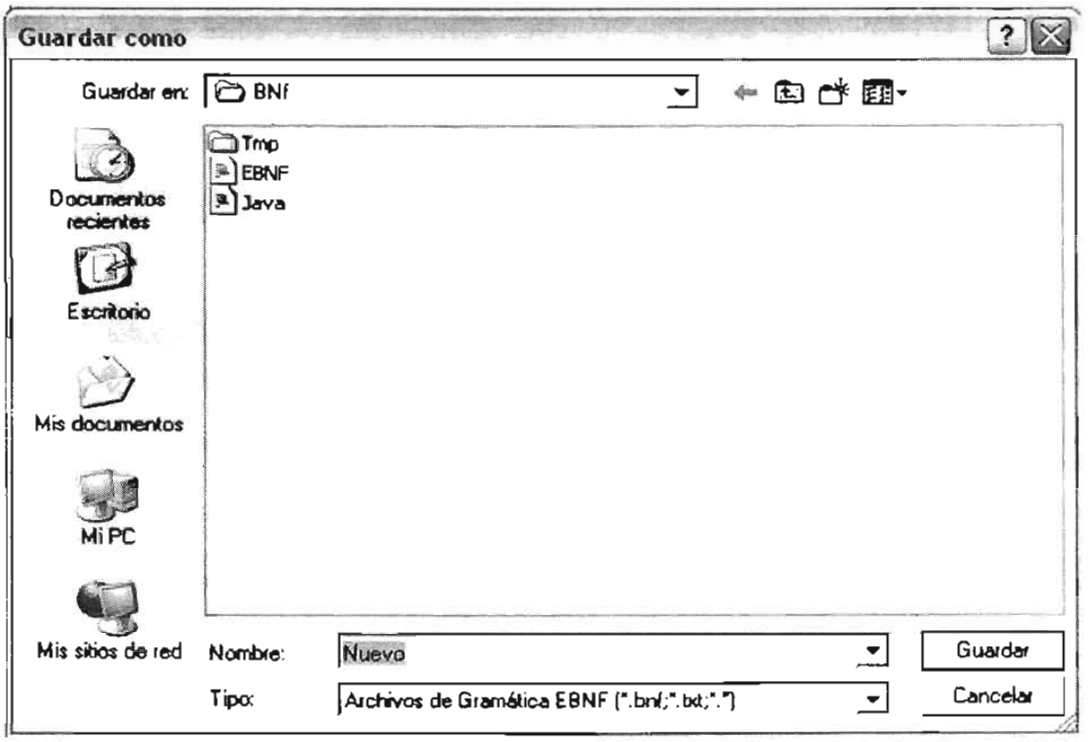


- Si elegimos cancelar el proceso se cancela como si no hubiéramos hecho click en el botón ***Guardar***.
- Si elegimos la opción ***Guardar*** después de elegir el subdirectorio y nombre de archivo deseado la aplicación guarda el archivo.

- Si el archivo no es nuevo, sencillamente guarda al archivo con el nombre de archivo con el que ya existe.

Hacemos click en  cuando queremos cambiar el nombre del archivo que estamos trabajando, la aplicación procede de la siguiente manera:

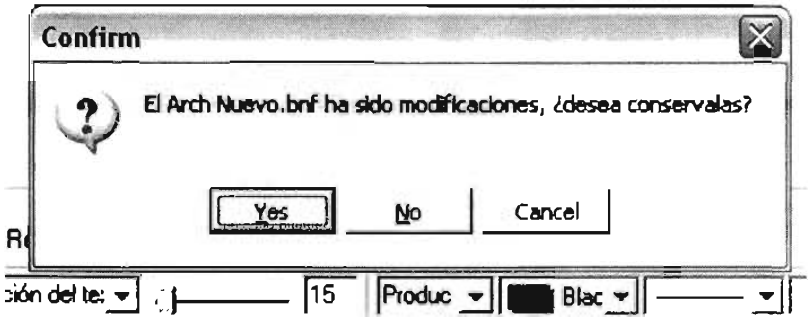
- Si no hay ningún archivo cargado como cuando iniciamos la aplicación, no hace nada.
- Si hay algún archivo cargado, muestra el cuadro de diálogo:



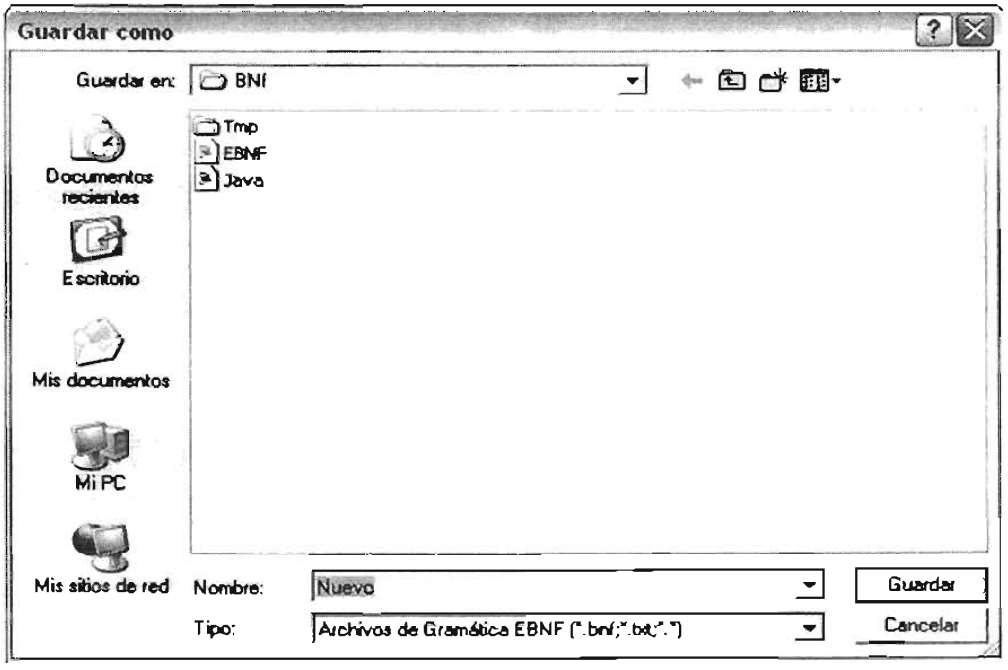
- Si elegimos cancelar el proceso se cancela como si no hubiéramos hecho click en el botón ***Guardar***.
- Si elegimos la opción ***Guardar*** después de elegir el subdirectorio y nombre de archivo deseado la aplicación guarda el archivo con el nombre deseado.

Hacemos click en el botón  cuando queremos salir de la aplicación, la cual procede así:

- Si no hay ningún archivo cargado, como cuando iniciamos la aplicación, ésta se cierra.
- Si hay algún archivo cargado pero no ha sufrido modificaciones, la aplicación se cierra.
- Si el archivo cargado sufrió modificaciones, se muestra el siguiente cuadro de diálogo:



- Si elegimos la opción ***Yes***, la aplicación abre el siguiente cuadro de diálogo:

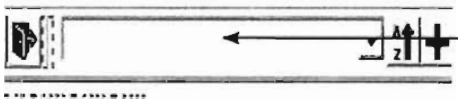


- Si elegimos la opción ***Cancelar***, el proceso se cancela por completo, es como si nunca hubiéramos hecho click en el botón de ***Salir*** archivo nuevo.

- Si elegimos la opción ***Guardar*** después de elegir el subdirectorio y nombre de archivo deseado la aplicación guarda el archivo y la aplicación se cierra.
- Si elegimos la opción ***No***, la aplicación ignora los cambios efectuados y simplemente se cierra.

- Si elegimos la opción ***Cancel***, el proceso se cancela como si no hubiéramos hecho click en el boton ***Salir*** archivo nuevo.

Ya sabemos cómo manejar archivos; ahora vamos a ver el siguiente grupo de botones que nos permiten manipular a las clases sintácticas y sus reglas de producción asociadas que están contenidas en los archivos de gramática.



 botón para ordenar la lista de clases.

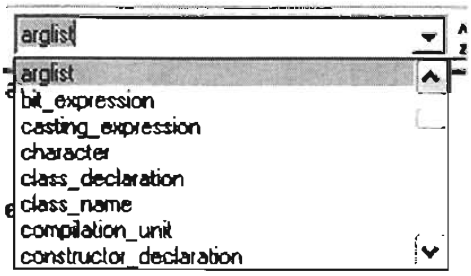
 botón para agregar una clase sintáctica y su regla de producción asociada la lista de clases.

 botón para eliminar la clase sintáctica elegida en la lista y su regla asociada de la lista.

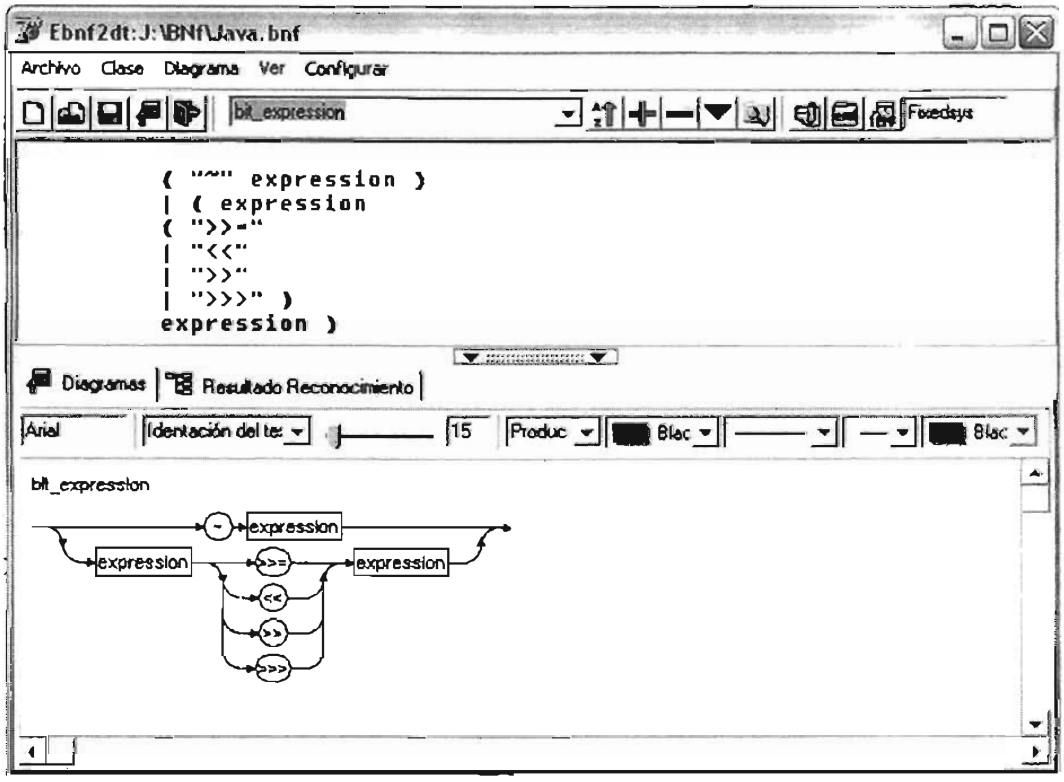
 botón para modificar el nombre de la clase sintáctica elegida en la lista.

 botón para buscar una clase sintáctica en la lista.

Cuando hacemos click en la lista de clases, ésta despliega los elementos que contiene y nos permite elegir el que deseemos:



Si por ejemplo elejimos la clase sintáctica **bit_expression**, la ventana de la aplicación mostraría lo siguiente:

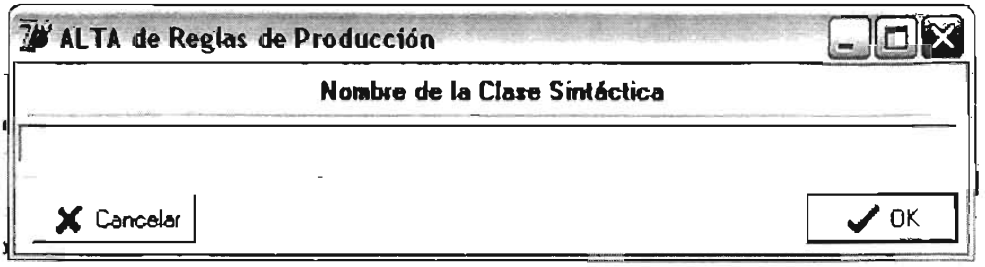


La actualización es automática, sólo hay que elegir una nueva clase sintática de la lista para que tanto el editor de reglas de producción como el área de dibujo muestren los datos y el diagrama indicado.

Hacemos click en  cuando deseamos ordenar la lista de clases sintácticas en orden alfabético ascendente.

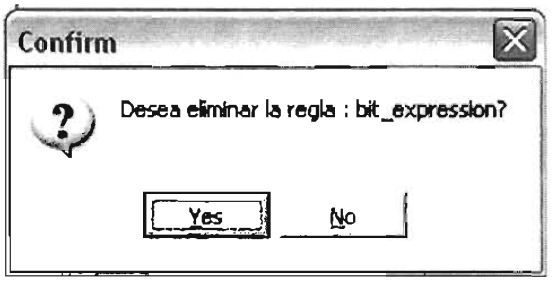
Hacemos click en  cuando deseamos agregar una clase sintáctica a la lista,

Inmediatamante la aplicación abre la siguiente ventana.



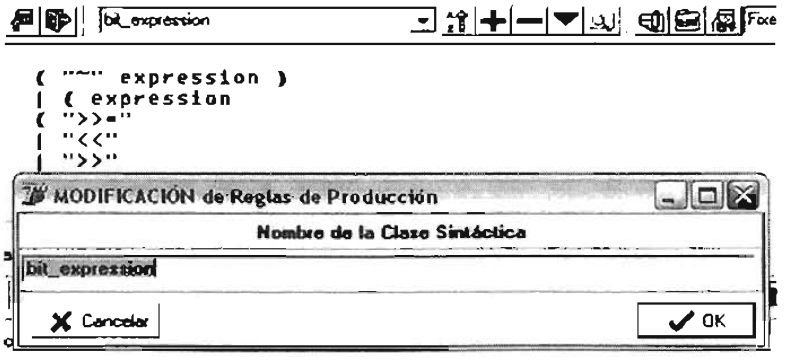
Si elegimos la opción ***Cancelar***, es como si no hubiéramos hecho click en el botón ***Insertar*** clase sintáctica. Si capturamos un nombre de clase sintáctica y elegimos la opción ***OK***, la clase sintáctica se inserta en la lista y el editor de reglas de producción se muestra en blanco, listo para capturar la correspondiente regla de producción.

Hacemos click en  , cuando deseamos eliminar la clase sintáctica que se encuentra elegida en la lista de clases sintácticas. Inmediatamante aparece el siguiente cuadro de diálogo:



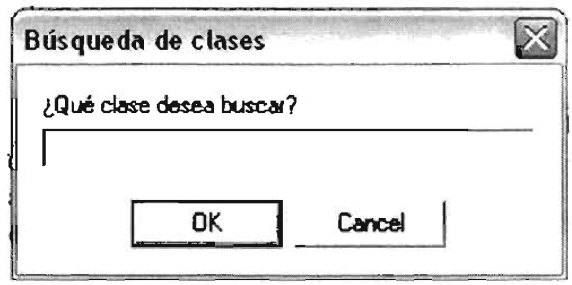
Si elegimos la opción ***Yes***, el cuadro de diálogo vuelve a aparecer; es decir, para eliminar una clase de la lista debemos confirmar dos veces que deseamos hacerlo.

Hacemos click en  cuando deseamos modificar el nombre de la clase que se encuentra elegida en la lista, inmediatamante aparece el siguiente cuadro de diálogo:

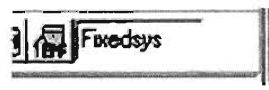


Si elegimos la opción ***Cancelar*** el nombre de la clase sintáctica no se modifica, pero si cambiamos el nombre de la clase sintáctica y elegimos la opción ***Ok***, el nombre de la clase cambia y este cambio se refleja tanto en la lista como en el diagrama generado.

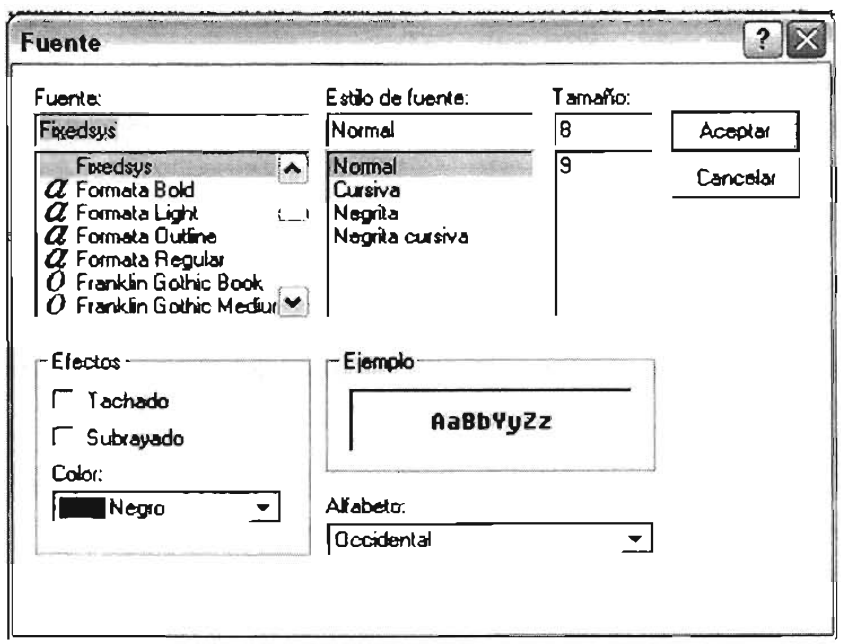
Hacemos click en el botón  , cuando deseamos buscar el nombre de una clase sintáctica en la lista, inmediatamente aparece el siguiente cuadro de diálogo:



Si tecleamos un nombre de clase y elegimos la opción ***Ok***, la clase se busca en la lista, si se encuentra la aplicación posiciona a la lista en el renglón correspondiente a esa clase y su regla correspondiente y su diagrama se muestran en el editor y en el área de dibujo; si la búsqueda fracasa o elegimos la opción ***Cancel***, no pasa nada.

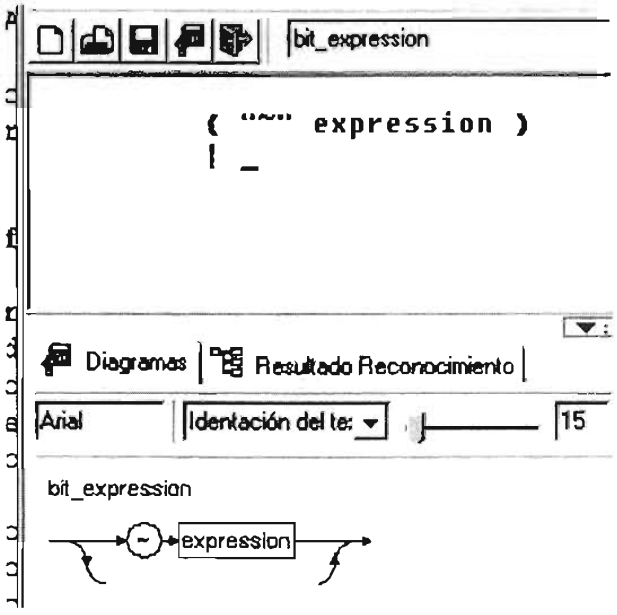
Hacemos click en la caja de texto 

cuando queremos modificar el tipo de fuente que utiliza el editor de reglas de producción, inmediatamante aparece el cuadro:



Si elegimos una fuente y estilo deseados, veremos el cambio inmediatamente reflejado en nuestros diagramas; en la aplicación todos los cambios que efectuamos son instantáneos. Del cuadro de diálogo anterior la única característica que se ignora es el color de la fuente porque como veremos más adelante, cada metasímbolo tiene asociado un color específico.

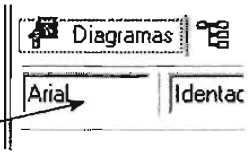
Cuando tecleamos en el editor de texto de las reglas de producción y existe una clase en la lista, los cambios correspondientes se muestran de manera inmediata en el área de dibujo; es decir el diagrama empieza a dibujarse interactivamente a medida que vamos tecleando la regla de producción, como podemos ver en la siguiente figura.



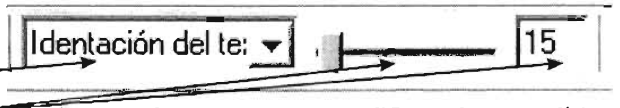
Aun cuando no hemos terminado de teclear toda la producción, el diagrama se dibuja hasta donde es posible; esta interactividad tal vez no sería posible si el reconocimiento fuera más complicado o el tamaño de nuestras reglas fuera considerablemente "grande", pero en realidad para el tamaño promedio de las reglas de producción que vamos a manejar funciona bastante bien y nos permite ver cómo se va construyendo el diagrama en tiempo real. En general, cualquier cambio que realizamos, ya sea a la regla de producción o a la configuración de los diagramas, se refleja inmediatamente. Aquí no es posible verlo, pero el color de cada metasímbolo es diferente en el editor; los colores correspondientes a cada metásímbolo son:

- ***(*** y ***)*** , que son los que agrupan conjuntos de alternativas: en color ***rojo***.
- ***(*** y ***]*** , que son los que identifican partes opcionales: en color ***azul***.• ***(*** y ***]***.

- **{** y **}** , que son los que identifican partes repetitivas: en color ***azul verde oscuro***.
- **|** , que separa conjuntos de altenativas: en color ***magenta***..
- Los símbolos terminales encerrados entre doble comilla en color verde pistache.
- Los símbolos terminales encerrados entre comilla sencilla en color ***azul verde claro***.
- Los comentarios ya sean los seguidos por doble *//* o encerrados entre */** y ***/* en color ***crema***.

Hacemos click en la caja de texto 

cuando queremos modificar la fuente del texto que se utiliza para dibujar en los diagramas, tanto los nombres de las reglas de producción, como los símbolos terminales y las clases sintácticas. Inmediatamente aparece un cuadro de diálogo para elegir la fuente y el estilo deseado.También aquí se ignora el color de la fuente, ya que podemos elegir un color diferente tanto para los nombres de las reglas, como para los símbolos terminales y las clases sintácticas. Aquí es muy importante tanto la fuente elegida como el tamaño, ya que su elección determina básicamente el tamaño de los diagramas.

Hacemos click en 

cuando queremos modificar el valor de alguno de los parámetros que modifican las medidas de los diagramas. Los parámetros que podemos modificar son:

- La alineación del texto.
- El interlineado del texto.
- El margen entre las figuras.
- La longitud de las líneas asociadas al texto.
- La longitud de las líneas de las figuras
- La longitud de las flechas.

La interpretación geométrica de estos parámetros en los diagramas se muestra en la figura 4.3.2 los valores asociados a cada parámetro representan un porcentaje del tamaño de la fuente; esto es así para guardar una armonía en las figuras, de manera que cuando modificamos el tamaño

de la fuente, el tamaño del diagrama se modifica proporcionalmente.

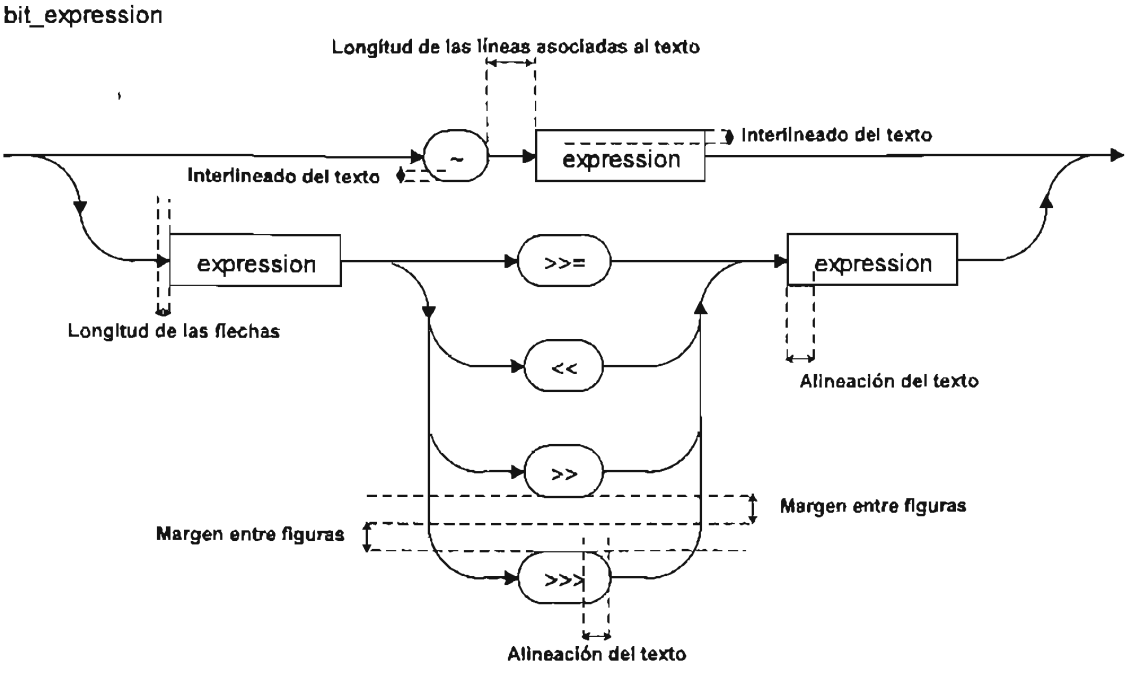
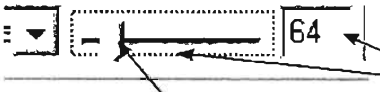


FIGURA 4.3.2

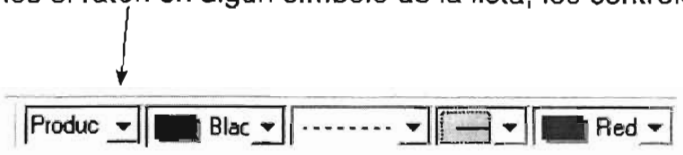
Los controles  muestran los valores correspondientes al parámetro elegido de la lista, lo que no permite, en el caso de la barra de deslizamiento modificar el valor desplazando la barra, y en el caso de la caja de texto, tecleando un nuevo valor. Los cambios se reflejan inmediatamente en el dibujo del diagrama.

Hacemos click en la lista  cuando queremos modificar las características de los colores y los estilos de líneas asociados a los símbolos que se dibujan en los diagramas. La lista es:

- Nombre de la regla de producción.
- Conjuntos de alternativas.
- Partes opcionales.
- Partes repetitivas.
- Símbolos terminales.
- Clases sintácticas

- Saltos en los diagramas.

Cada vez oprimimos el ratón en algún símbolo de la lista, los controles:



muestran las características seleccionadas en ese momento para dicho símbolo y nos permiten modificarlas.

No hay mucho por explicar al respecto; si presionamos el ratón en el control de color de las líneas la aplicación nos permite elegir el color y así respectivamente para el estilo, el ancho y el color de la fuente del texto asociado al símbolo. No todos los símbolos tienen asociado texto, por ejemplo las partes opcionales no tienen asociado texto y por lo tanto este último control no se muestra cuando elegimos un símbolo de esa naturaleza. Al igual que con los valores de los parámetros para las medidas, la actualización es inmediata en el dibujo del diagrama.

En la lista de símbolos mencionamos uno que dice ***saltos en los diagramas***; no hemos aclarado cuál es su función o cómo puede aparecer en nuestros diagramas, de hecho en ninguna descripción de BNF vamos a encontrar este concepto ya que es algo particular de esta aplicación; es un metasímbolo que introducimos para tener la opción de partir la longitud de una gráfica y que dé un salto hacia abajo para permitirnos dibujar diagramas que se extienden mucho hacia la derecha y que de otra manera podrían no caber en el área que tenemos destinada para nuestro dibujo, ya sea en pantalla o como un archivo. Su aparición es por lo tanto a solicitud y a conveniencia; para funcionar correctamente deben aparecer entre dos sucesiones de símbolos pero esto no es necesario; nos podemos dar cuenta inmediatamente cuando está mal colocado porque el diagrama no se va a dibujar como deseamos. Vamos a poner un ejemplo de su uso. Supongamos que tenemos cargada la clase sintáctica **float_literal**, la cual se muestra en la figura 4.3.3; como podemos ver, el dibujo del diagrama no alcanza a salir en la pantalla; si nuestra pantalla es más grande que la ventana de nuestra aplicación, podríamos maximizar la aplicación y tal vez cabría o podríamos utilizar las barras de desplazamiento para ver la parte del diagrama que está oculto. Pero supongamos que deseamos, si es posible, ver el diagrama completo, utilizando el concepto de salto, que sabemos nos permite el flujo del diagrama, dando un salto hacia abajo pero respetando la lógica de dibujo del diagrama. Si observamos el diagrama podemos

ver que aparentemente nuestro problema es que la parte opcional **float_type_suffix**, es la que está provocando el desbordamiento del diagrama. ¿Por qué no poner un salto de diagrama ahí?

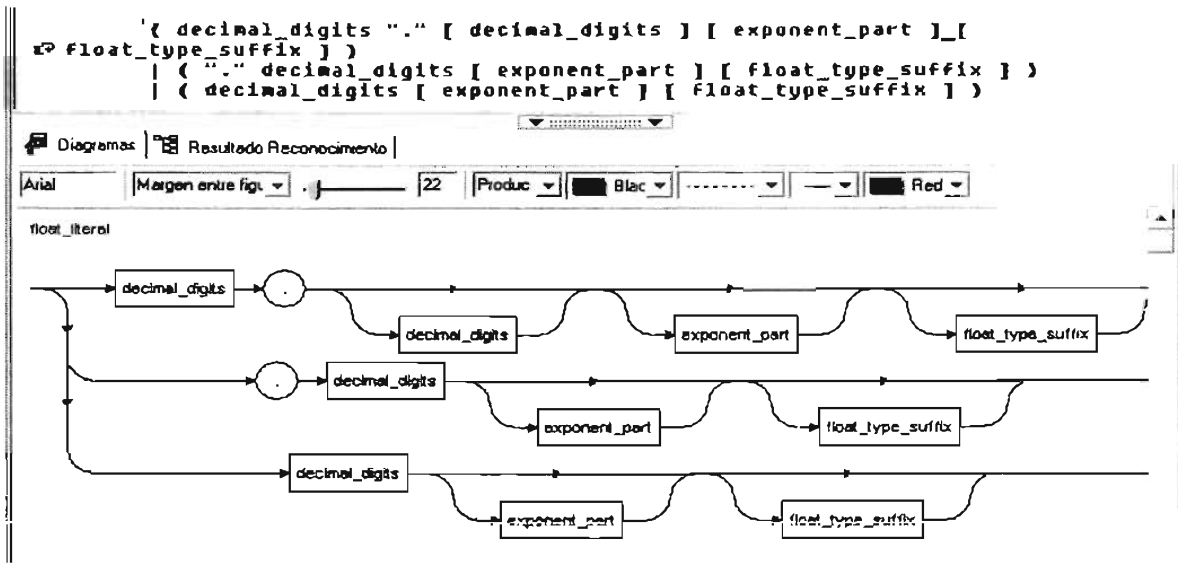


FIGURA 4.3.3

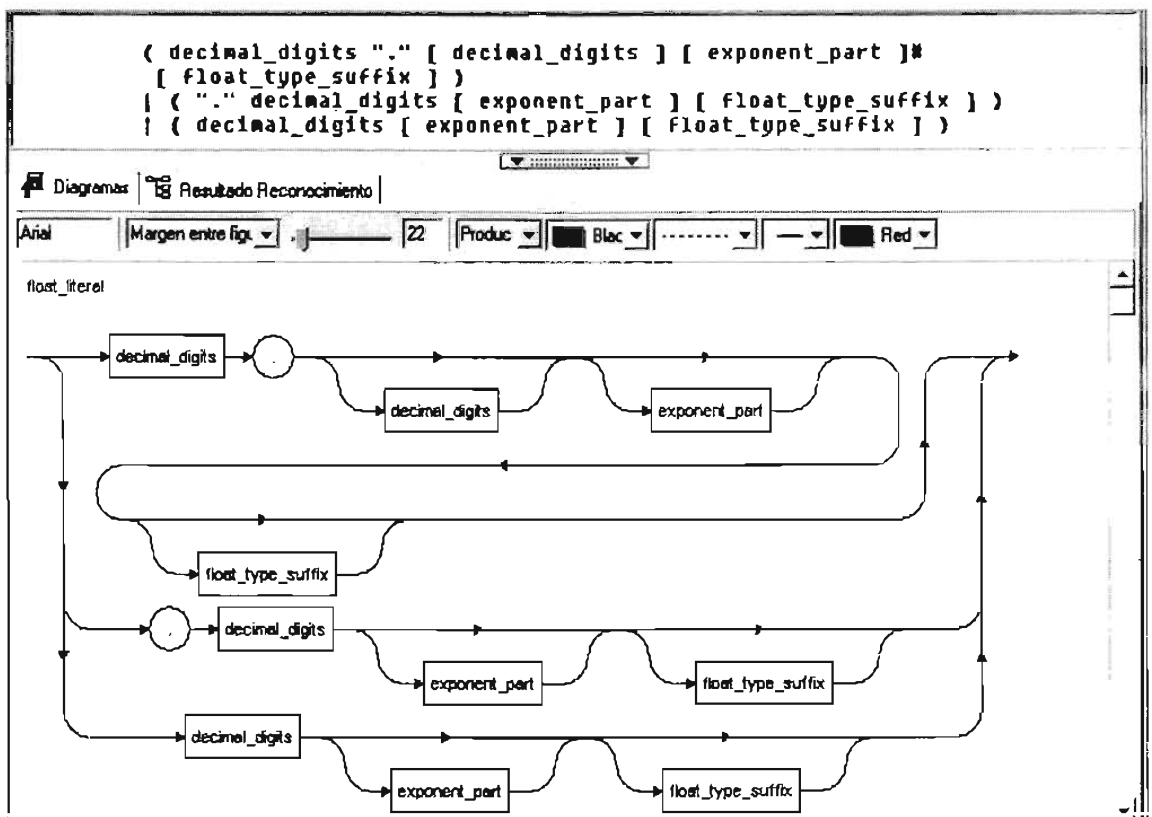


FIGURA 4.3.4

Lo hacemos así y obtenemos el resultado deseado; ahora nuestro diagrama ya cabe en la pantalla. Es así como debemos utilizar estos ***saltos de diagrama***.

Analizamos ahora la página de ***Resultado Reconocimiento***.

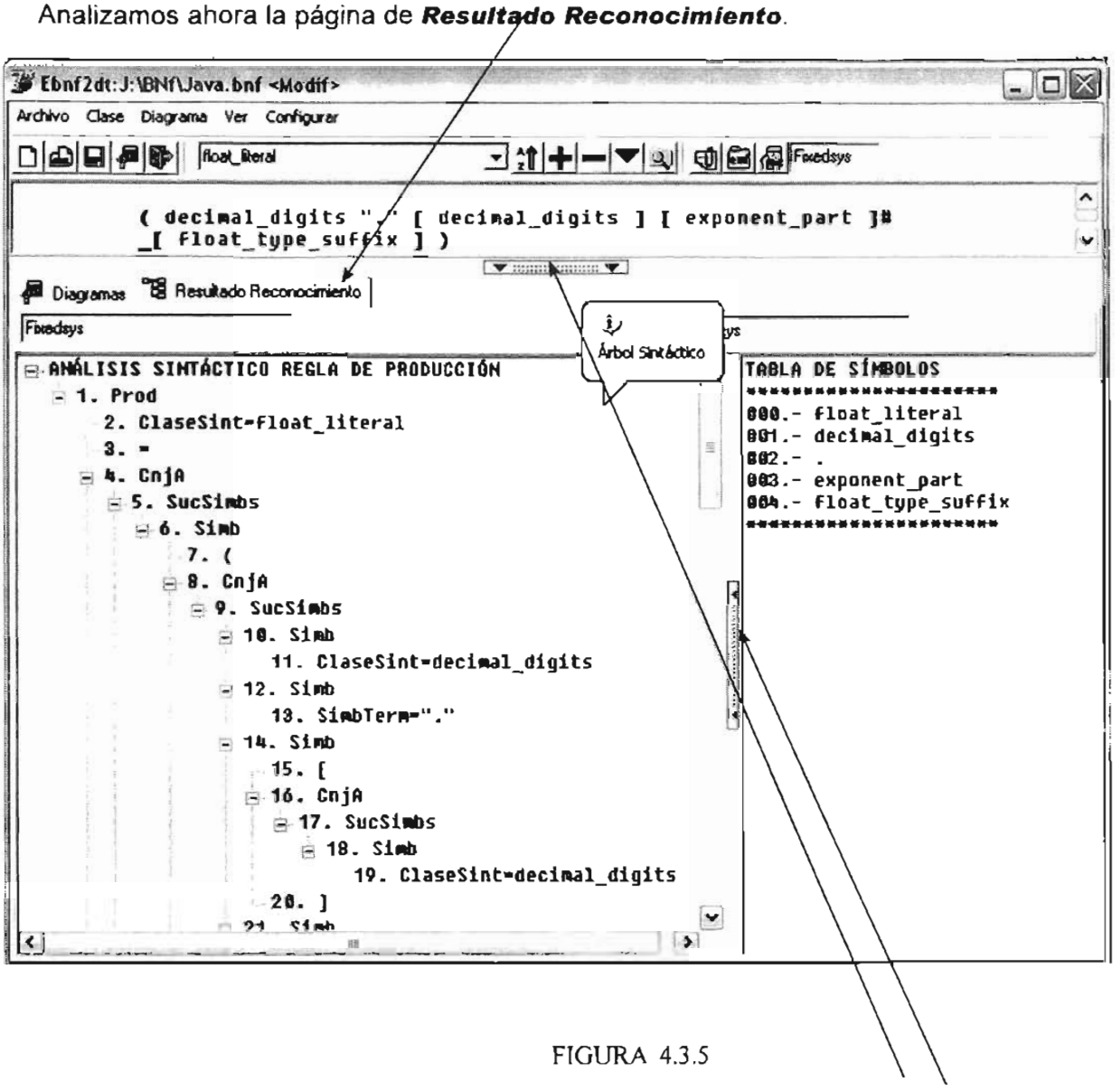


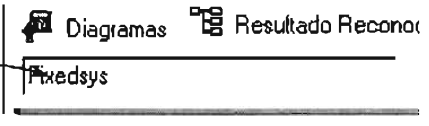
FIGURA 4.3.5

Antes abrimos un paréntesis para mencionar cómo funciona la barra de separación; si recordamos la figura 4.3.1, podemos ver que la altura del editor es más grande que en esta figura; eso es porque las barras de separación nos permiten agrandar o reducir el área que se encuentra hacia abajo/arriba o hacia su derecha/izquierda; todo lo que tenemos que hacer es arrastrar la barra en el sentido que deseemos. Por ejemplo, si arrastramos la barra de separación horizontal hacia abajo la altura del editor se hará más grande mientras que

la altura de la página de ***Resultado Reconocimiento*** se hará más chica. Si procedemos de manera inversa, es decir si la desplazamos hacia arriba, la altura del editor se hará más pequeña y la altura de la página ***Resultado Reconocimiento*** será más grande.

De la misma forma, si arrastramos la barra de separación vertical hacia la izquierda la anchura de la tabla de símbolos se hará más grande, mientras que la anchura del ***árbol sintáctico*** se hará más pequeña e inversamente, si arrastramos la barra de separación vertical hacia la derecha la anchura de la ***tabla de símbolos*** se hara más pequeña mientras que la anchura del ***árbol sintáctico*** se hará más grande.

La página ***Resultado Reconocimiento***, nos permite ver el resultado del reconocimiento sintáctico, por una parte el árbol sintáctico, donde cada nodo indica el tipo de simbolo reconocido y en la tabla de símbolos podemos ver tanto la tabla de símbolos generada como los mensajes de error emitidos por la aplicación. En la tabla de símbolos vemos sólo eso, nombres de símbolos, cuando el reconocimiento fue exitoso.

Presionamos el ratón en 

cuando queremos elegir las características de la fuente utilizada para desplegar el texto del árbol sintáctico. Inmediatamente aparace la caja de diálogo que nos permite elegir las características de la fuente.

Y por último presionamos el ratón en 

cuando queremos elegir las características de la fuente utilizado para desplegar el texto de la tabla de símbolos. Inmediatamente aparace la caja de diálogo que nos permite elegir las características de la fuente.

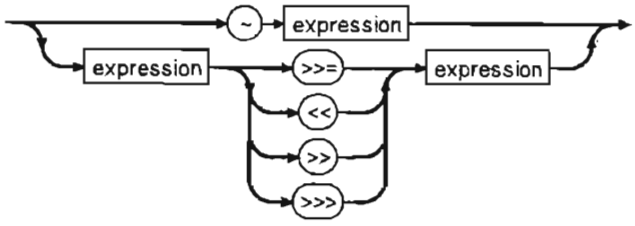
Con esto damos por terminado este manual, en la siguiente sección, mostramos algunos de los ejemplos más elaborados de reglas de producción pertencientes al lenguaje Java y sus respectivos diagramas de tren.

4.4 Diagramas generados con la aplicación

bit_expression =

```
("~" expression)
|
( expression ( ">>=" | "<<" | ">>" | ">>>" ) expression ) .
```

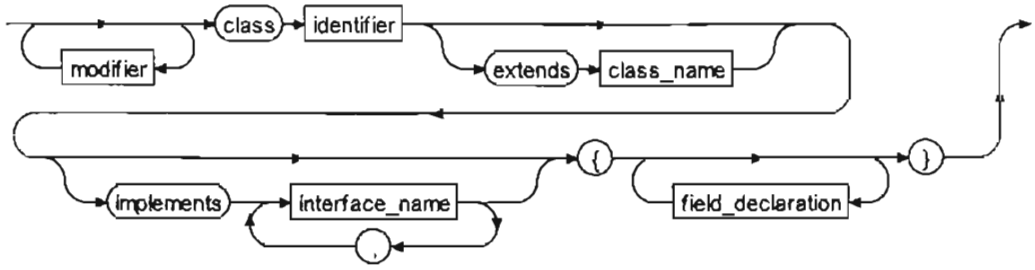
bit_expression



class_declaration =

```
{ modifier } "class" identifier [ "extends" class_name ] #
[ "implements" interface_name { "," interface_name } ]
"{" { field_declaration } "}" .
```

class_declaration

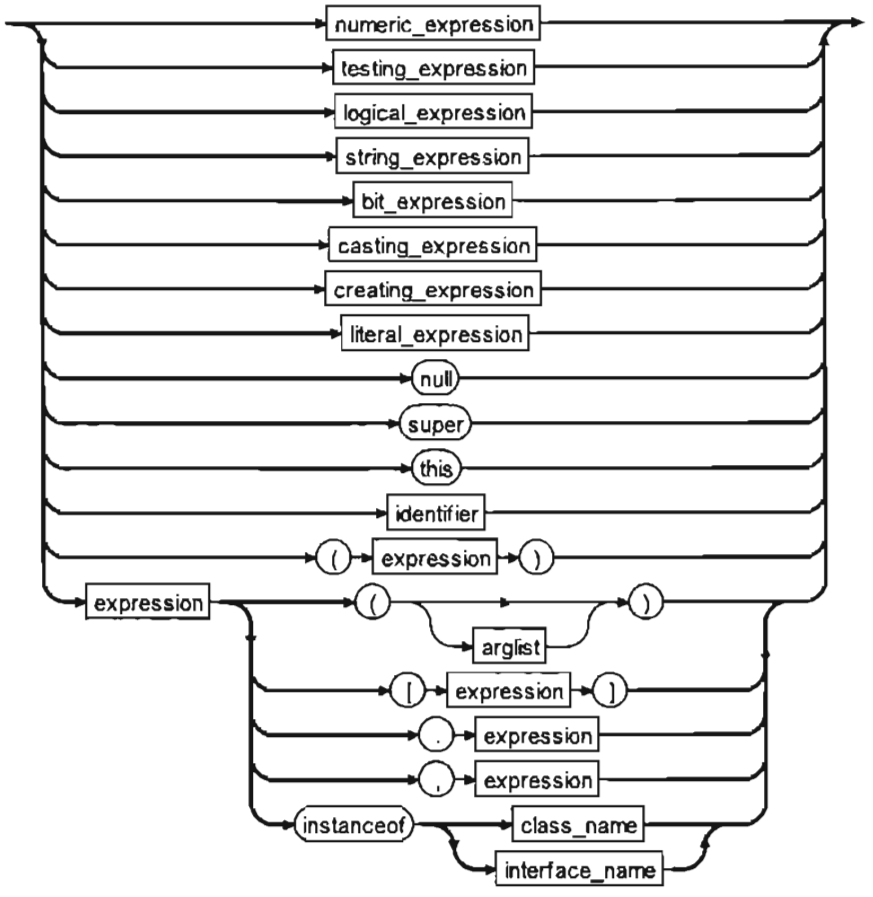


expression =

```
numeric_expression      | testing_expression
| logical_expression    | string_expression
| bit_expression        | casting_expression
| creating_expression   | literal_expression
| "null"                | "super"
| "this"                | identifier
| ( "(" expression ")" )   |
```

```
( expression ( ( "(" [ arglist ] ")" ) | ( "[" expression "]" )
                    | ( "." expression )  | ( "," expression )
               | ( "instanceof" ( class_name | interface_name ) )
               ) ) .
```

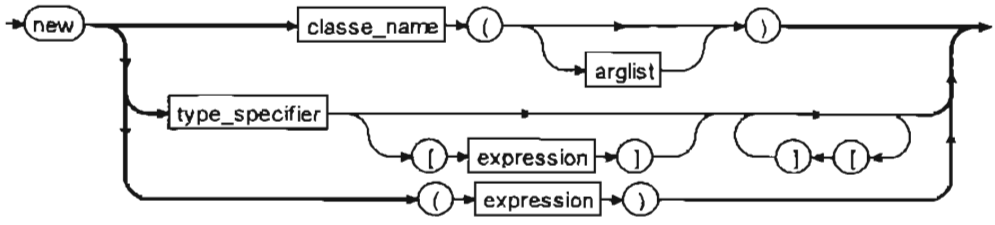
expression



creating_expression =

```
"new" ( ( classe_name "(" [ arglist ] ")" )
      | ( type_specifier [ "[" expression "]" ] { "[" "]" } )
      | ( "(" expression ")" ) ) .
```

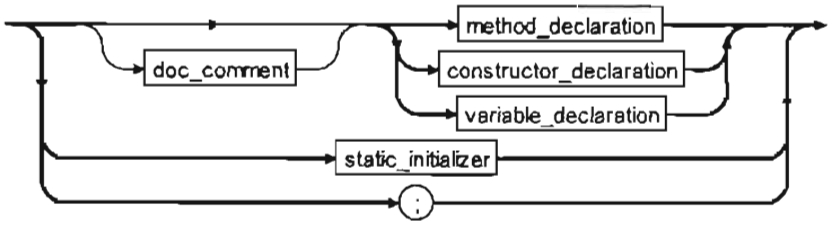
creating_expression



field_declaration =

```
( [ doc_comment ] ( method_declaration
   | constructor_declaration
   | variable_declaration ) )
   | static_initializer
   | ";" .
```

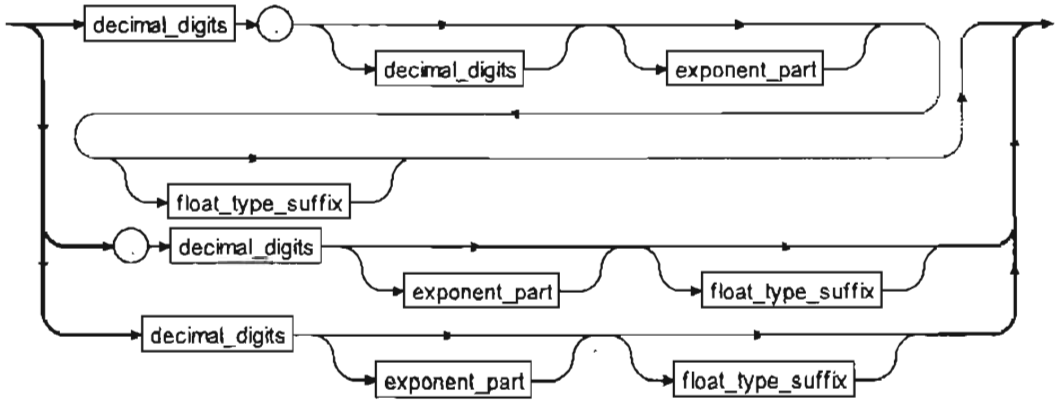
field_declaration



float_literal =

```
( decimal_digits "." [decimal_digits ) [ exponent_part ]#
 [ float_type_suffix ] )
| ( "." decimal_digits [ exponent_part ] [ float_type_suffix ] )
| ( decimal_digits [ exponent_part ] [ float_type_suffix ] ) .
```

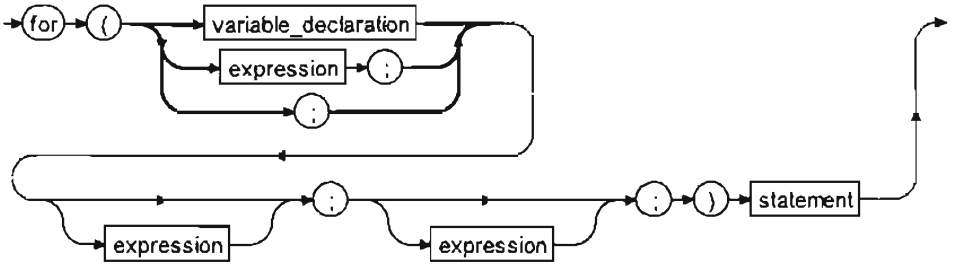
float_literal



for_statement =

```
"for" "(" ( variable_declaration | ( expression ";" ) | ";" ) #
      [ expression ] ";" [ expression ] ";"  ")" statement .
```

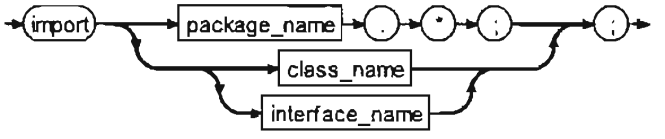
for_statement



import_statement =

```
"import" ( ( package_name "." "*" ";" ) |
( class_name | interface_name ) ) ";"
```

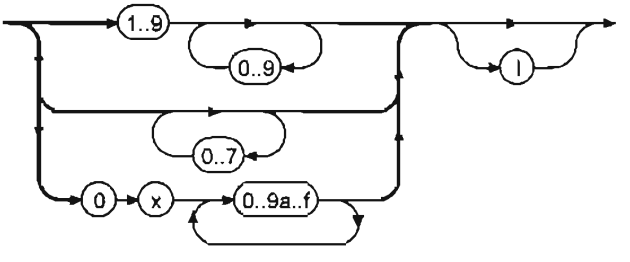
import_statement



integer_literal=

```
( ( "1..9" { "0..9" } )
   | { "0..7" }
   | ( "0" "x" "0..9a..f" { "0..9a..f" } ) )
 [ "l" ]
```

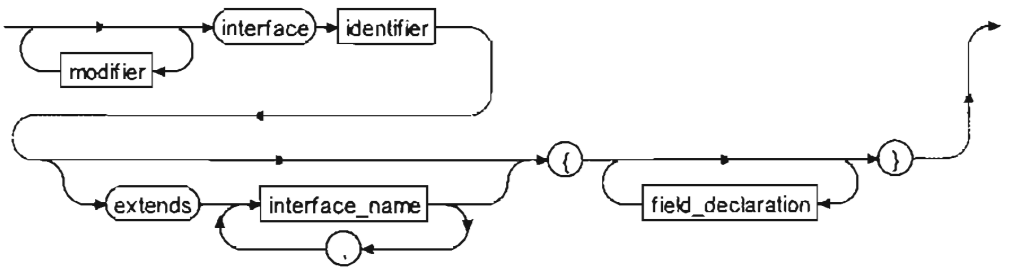
integer_literal



interface_declaration =

```
{ modifier } "interface" identifier #
    [ "extends" interface_name { "," interface_name } ]
    "{" { field_declaration } "}"
```

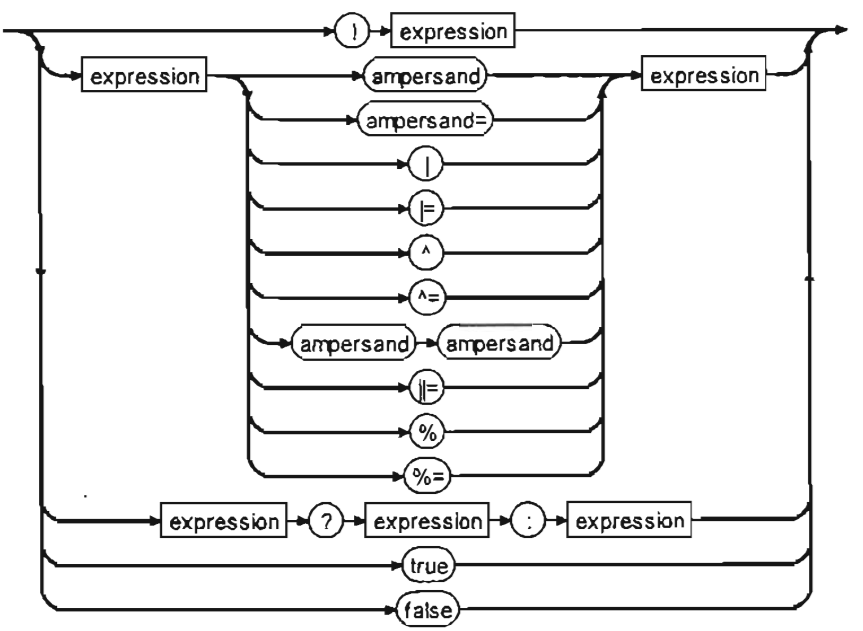
interface_declaration



logical_expression =

```
( "!" expression )
| ( expression  ( "ampersand"  | "ampersand="  | "|"   | "|="
                | "^"  | "^=" | ( "ampersand" "ampersand" )
                | "||=" | "%" | "%=" )  expression )
|( expression "?" expression ":" expression )| "true" | "false"
```

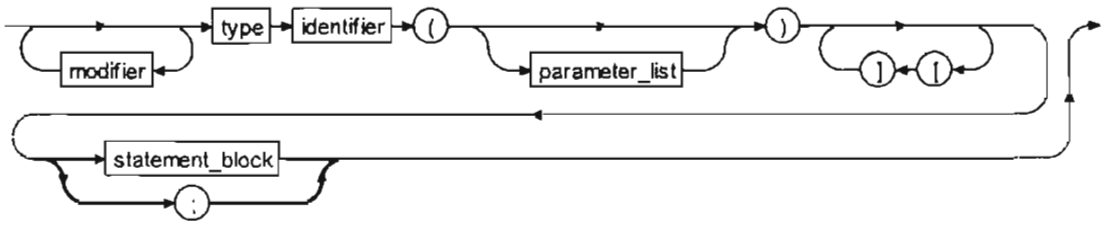
logical_expression



method_declaration =

```
{ modifier } type identifier "(" [ parameter_list ] ")" { "[" "]" }
        # ( statement_block | ";" )
```

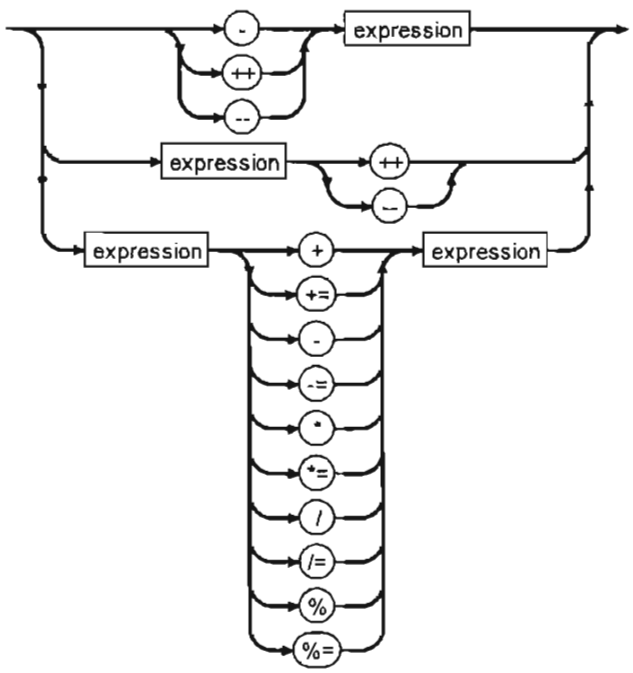
method_declaration



numeric_expression =

```
( ( "-" | "++" | "--" ) expression )
| ( expression ( "++" | "--" ) )
| ( expression ( "+" | "+=" | "-" | "-=" | "*" | "*=" | "/"
                | "/="| "%"| "%=" ) expression )
```

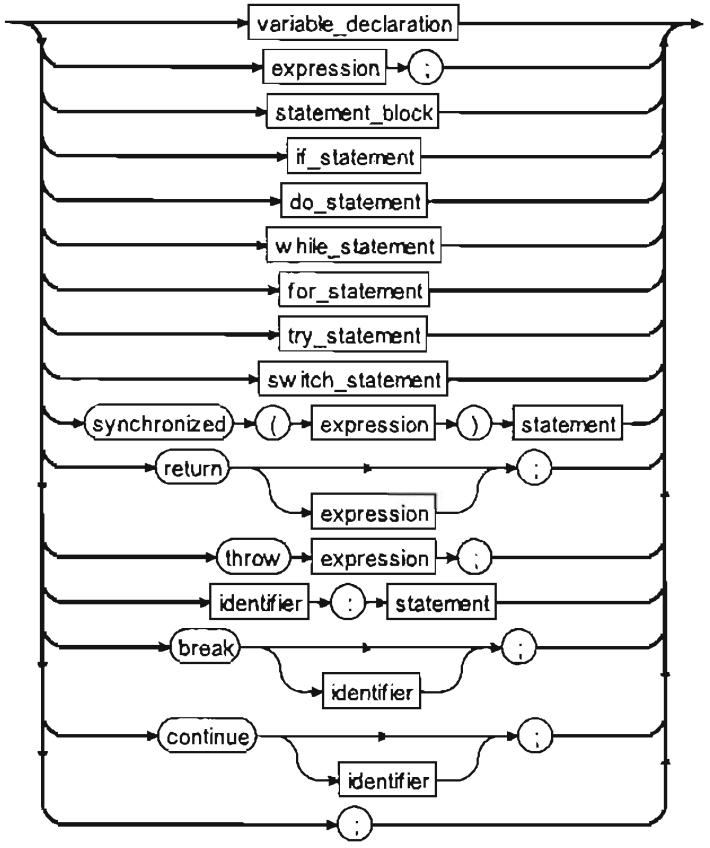
numeric_expression



statement =

```
    variable_declaration | ( expression ";" )  | ( statement_block
)
    | ( if_statement )   | ( do_statement )   | ( while_statement )
    | ( for_statement )  | ( try_statement )  | ( switch_statement )
    | ( "synchronized" "(" expression ")" statement )
    | ( "return" [ expression ] ";" )  | ( "throw" expression ";" )
    | ( identifier ":" statement )     | ( "break" [ identifier ]
";" )
    | ( "continue" [ identifier ] ";" ) | ( ";" )
```

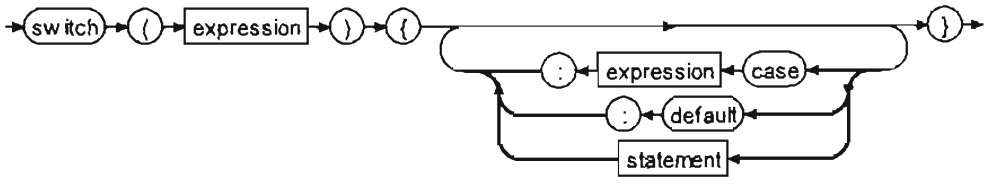
statement



switch_statement =

```
  "switch" "(" expression ")" "{"
  { ( "case" expression ":" )| ( "default" ":" ) | statement }
```

switch_statement

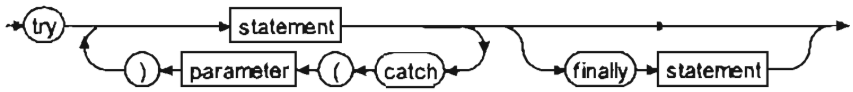


"}"

try_statement =

```
"try" statement { "catch" "(" parameter ")" statement }
 [ "finally" statement ]
```

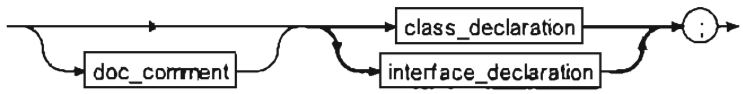
try_statement



type_declaration =

```
[ doc_comment ] ( class_declaration ) interface_declaration ) ";"
```

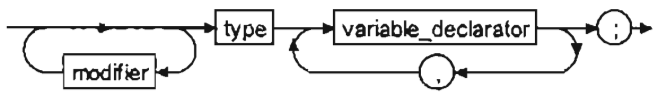
type_declaration



variable_declaration =

```
{ modifier }type variable_declarator { "," variable_declarator }
";"
```

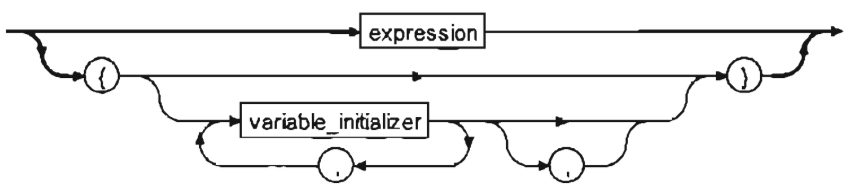
variable_declaration



variable_initializer =

```
expression
| ( "{" [ variable_initializer
{ "," variable_initializer } [ "," ] ] "}" )
```

variable_initializer

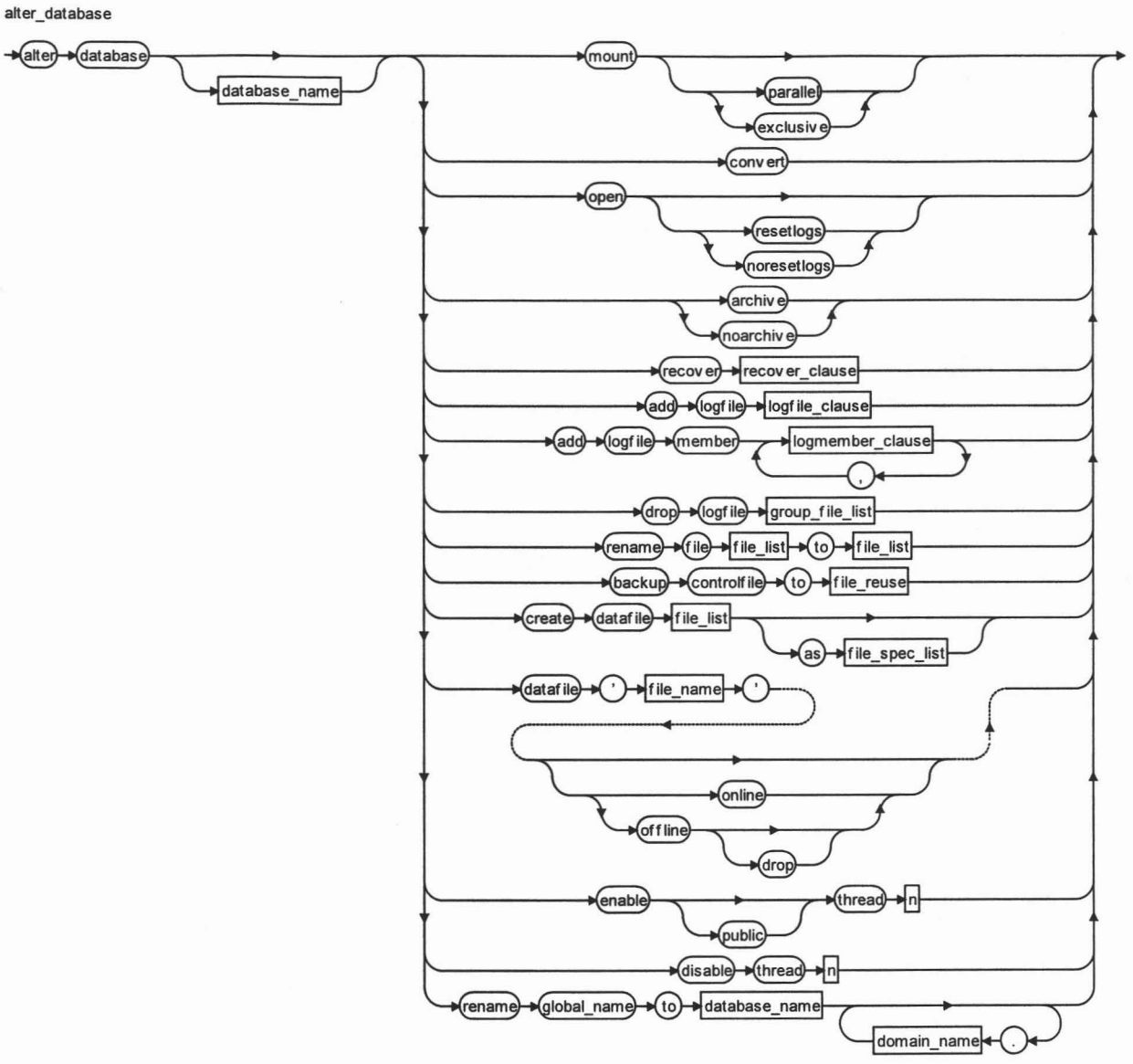


Estos ejemplos se tomaron de la descripción para el lenguaje Java, no es una muestra exhaustiva sino más bien representativa de los ejemplos de las reglas de producción más elaboradas y complicadas.

Conclusiones

La aplicación incluida en este trabajo es la principal aportación. La mayoría de los lenguajes de programación describen su sintaxis por medio de BNF o EBNF; por lo tanto, una aplicación que traduzca las reglas que integran esta sintaxis a diagramas de trenes, es muy útil desde el punto de vista didáctico; la mayoría de las veces, es más sencillo entender un diagrama de tren que una regla de producción escrita en EBNF. Por ejemplo la siguiente regla de producción se tomó de la descripción de SQL de Oracle(tm) DBMS versión 7.

```
alter_database =
      "alter" "database" [ database_name ]
      ( ( "mount" [ "parallel" | "exclusive" ] )
      | ( "convert" )
      | ( "open" [ "resetlogs" | "noresetlogs" ] )
      | ( "archive" | "noarchive" )
      | ( "recover" recover_clause )
      | ( "add" "logfile" logfile_clause )
      | ( "add" "logfile" "member" logmember_clause
      { "," logmember_clause } )
      | ( "drop" "logfile" group_file_list )
      | ( "rename" "file" file_list
      "to" file_list )
      | ( "backup" "controlfile" "to" file_reuse )
      | ( "create" "datafile" file_list [ "as" file_spec_list ] )
      | ( "datafile" "'" file_name "'"
      [ "online" | ( "offline" [ "drop" ] ) ] )
      | ( "enable" [ "public" ] "thread" n )
      | ( "disable" "thread" n )
      | ( "rename" "global_name" "to" database_name
      { "." domain_name } ) )
```



Como podemos ver, en este ejemplo complicado, resulta más fácil echarle un vistazo al diagrama para entender cómo se integra una instrucción de ***alter database*** en ***SQL***, que descifrar la sintaxis ***EBNF*** involucrada.

Este trabajo también puede servir:

- Como una ligera introducción a la teoría de compiladores y su aplicación con un ejemplo real; que aunque limitado cubre las principales partes de un compilador como son: el análisis léxico, sintáctico y la generación de código (en este caso para dibujar) a partir

del árbol sintáctico.

- Para ejemplificar algunos aspectos básicos e intermedios de la programación orientada a objetos; ya que la aplicación se ubica en un nivel intermedio de complejidad y el trabajo incluye una explicación a detalle del código generado.

La aplicación es amigable y sencilla; contempla los aspectos básicos para generar los diagramas, sin embargo quedaron pendientes de implementar las siguientes opciones:

- Imprimir los diagramas. No la incluí porque los diagramas se pueden cargar en el portapapeles y de ahí se pueden pegar a cualquier aplicación que maneje archivos del tipo de Windows Metafile (lo cual hacen la mayoría de las aplicaciones más utilizadas en Windows, como son: Excel, Word, Wordpad, Photoshop, Visio, e Illustrator entre otros).
- Incluir en los diagramas los comentarios asociados a las reglas de producción.
- Generar todos los diagramas, dibujando cada diagrama debajo del anterior; con la condición: se genera un archivo, cada vez que ya no caben más diagramas, en un área de dibujo cuyo tamaño define el usuario.

Configurar algunas características de los diagramas, como por ejemplo:

- Poder utilizar una fuente distinta para los nombres de las reglas, de las clases sintácticas y de los símbolos terminales
- Manejar un color para el fondo de los símbolos.

Estas opciones son fáciles de agregar, pero exceden el ámbito de este trabajo,por lo que se dejan para una extensión posterior.

Bibliografía

CUEVA Lovelle Juan Manuel (Noviembre 2000). ***Análisis Léxico en Procesadores de Lenguaje***, Universidad de Oviedo. Disponible en Web:<http://di002.edv.uniovi.es/procesadores/apuntes/Lexico.pdf>.

CUEVA Lovelle Juan Manuel. ***Análisis Sintáctico Descendente***, Universidad de Oviedo. Tema 5. Disponible en Web:<http://di002.edv.uniovi.es/procesadores/apuntes/Tema5.pdf>.

CUEVA Lovelle Juan Manuel et al. (Enero 2005). ***Análisis Sintáctico en Procesadores de Lenguaje***, Universidad de Oviedo. Temas 4 y 5. Disponible en Web:<http://di002.edv.uniovi.es/procesadores/apuntes/Sintactico.pdf>.

CUEVA Lovelle Juan Manuel (Diciembre 1998). ***Conceptos Básicos de Procesadores de Lenguaje***, Universidad de Oviedo. Disponible en Web:<http://di002.edv.uniovi.es/procesadores/apuntes/10_Conceptos_Basicos_Procesadores_Lenguaje.pdf>.

CUEVA Lovelle Juan Manuel (Noviembre 2001). ***Lenguajes Gramáticas y Autómatas***, Universidad de Oviedo. Disponible en Web:<http://di002.edv.uniovi.es/procesadores/apuntes/AUTOMATA.pdf>.

GARSHOL Lars Marius (2003). ***BNF and EBNF: What are they and how do they work?***. Disponible en Web: <http://www.garshol.priv.no/download/text/bnf.html>.

GRIES David (1971), ***Compiler Construction for Digital Computers***, John Wiley & Sons, Inc. . Capítulos 1, 2 3 y 4

GRUNE Dick; JACOBS Ceriel J.H.(1998), ***Parsing Techniques - A practical guide***, Amstelveen/Amsterdam. Disponible en Web: <http://www.cs.vu.nl/~dick/PTAPG.html>.

ISO/IEC(1996) ***ISO/IEC 14977 : 1996(E) Extended EBNF***. Disponible en Web: <http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf>.

MARIN Morales Roque et al. (Enero 2005) , ***Teoría de Autómatas y Lenguajes Formales***, Departamento de Ingeniería de la Información y las Comunicaciones, Universidad de Murcia. Disponible en Web: <http://perseo.dif.um.es/~roque/talf/Material/apuntes.pdf>.

MÖSSENBÖCK Hanspeter , ***Compiler Construction***, University of Linz. Disponible en Web: <http://www.ssw.uni-linz.ac.at/Misc/CC/>.

TERRY, P.D. (1996) ***Compilers and Compiler Generators an Introduction with C++,*** Rhodes University, 1996. Capítulos 5, 8, 9 y 10. Disponible en Web: <http://www.scifac.ru.ac.za/compilers/>.

TREMBLAY Jean-Paul; SORENSON Paul G (1985), ***The Theory and Practice of Compiler Writing***, McGraw-Hill International Editions Computer Science Series. Capítulos 1, 2, 4 y 6.

Índice

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z