UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

HERRAMIENTAS PARA LA CONSTRUCCION DE COMPILADORES

TESIS QUE PRESENTA

ELKE CAPELLA KORT

Para obtener el título de

MATEMATICA

México, D.F.

TESIS CON
FALLA DE ORIGEN

Abril 1991

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

DERECHOS RESERVADOS © PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis está protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

I N D I C E

I. INTRODUCCION 2

II. GRAMATICAS Y LENGUAJES FORMALES

- Generalidades 6
- Conceptos preliminares 6
- Definiciones 7
- Clasificación de gramáticas 8
- Gramáticas libres de contexto 9
- Arboles de derivación 10
- Derivaciones izquierda y derecha. Ambigüedad 14

III. ANALISIS SINTACTICO

- La función del reconocedor sintáctico de un compilador y tipos de reconocedores 17
- Reconocimiento de arriba hacia abajo 18
- Reconocimiento no recursivo por predicción 19

IV. TRANSFORMACION DE GRAMATICAS LIBRES DE CONTEXTO

- Generalidades 21
- Eliminación de símbolos inactivos 21
- Eliminación de símbolos inalcanzables 23
- Eliminación de producciones vacías 26
- Eliminación de producciones unitarias y ciclos 30
- Recursividad izquierda 33
- Formas Normales 34
- Factorización izquierda 43

V. CONSTRUCCION DE UN RECONOCEDOR NO RECURSIVO POR PREDICCION

- Gramáticas LL(1) 47
- Aspectos generales 49
- Construcción de la tabla de acción del autómata 50
- Construcción de los conjuntos FIRST y FOLLOW 50
- Comportamiento del autómata 52
- Algoritmo de reconocimiento 53
- Ejemplo 54

VI. INSTRUMENTACION DEL SISTEMA

- Generalidades 60
- Programa 1: Transformación de gramáticas 60
- Programa 2: Construcción automática del analizador sintáctico 65

APENDICE I 70

APENDICE II 73

APENDICE III

BIBLIOGRAFIA

I. INTRODUCCION

La labor de un compilador ha dejado de ser una labor de arte o artesanía. Con los resultados teóricos en la teoría de autómatas y lenguajes formales, esta labor se ha convertido, en sus partes más tediosas y laboriosas, en la aplicación adecuada de teoremas cuyas demostraciones constructivas, proporcionan el algoritmo que construye tal o cual objeto, o que transforma una gramática de un tipo poco manejable a otro tipo que posee las características necesarias para poderlo procesar automáticamente.

Es desafortunado, en términos de confiabilidad del compilador, que algunos de los compiladores más usados hoy en día utilicen métodos "ad hoc" de análisis sintáctico. (Por métodos "ad hoc" entendemos métodos no formalizados). Estos métodos son populares porque permiten una interface eficiente, directa y conceptualmente simple entre el análisis sintáctico y la generación de código.

La importancia de la utilización de métodos formalizados de análisis sintáctico es que éstos pueden ser demostrados matemáticamente respecto a que son correctos, mientras que la construcción de un reconocedor "ad hoc" depende totalmente del cuidado que hayan tenido sus diseñadores y programadores para asegurar que son correctos. Así mismo deben tener un conocimiento y entender extraordinarios de los detalles del lenguaje que se está instrumentando y deben ser especialmente cuidadosos al verificar su desempeño.

Creemos que un curso de compiladores o de autómatas y lenguajes formales que no instruya sobre el uso de estas herramientas formales, se ubica en cuanto actualidad en un ambiente computacional de cuando menos hace diez años. En el contexto de las ciencias de la computación es importante, frente a los teoremas y sus demostraciones, mostrar los programas que instancian esas demostraciones, así como las aplicaciones prácticas que se consiguen a través de ellos.

El presente trabajo consiste, por un lado, en la exposición de los fundamentos teóricos que permiten la utilización de métodos formalizados para transformar lenguajes propuestos a través de gramáticas libres de contexto y para la construcción automática de un analizador sintáctico no recursivo por predicción. Por el otro, en un sistema que instrumenta estos métodos y cuyo objetivo es poder ser utilizado, en el contexto de un curso de compiladores o de autómatas y lenguajes formales, como auxiliar didáctico.

En el capítulo II. **Gramáticas y Lenguaje Formales**, se presentan los conceptos y definiciones a las que se hará referencia

a lo largo del trabajo, la clasificación de las gramáticas, haciéndo énfasis en las llamadas gramáticas libres de contexto y los árboles de derivación como representación de las derivaciones de una gramática.

En el capitulo III. **Análisis Sintáctico**, se define la función de un reconocedor sintáctico, se mencionan los tipos de reconocedores que existen y las propiedades que debe satisfacer una gramática para que la construcción de un analizador sintáctico no recursivo por predicción sea posible. Cuando una gramática no satisface estas propiedades es posible someterla a un proceso de transformación a través del cual obtenemos una gramática equivalente que si las satisface.

En el capitulo IV. **Transformación de gramáticas libres de contexto**, se presenta, para cada uno de los pasos en el proceso de transformación de gramáticas, el lema o teorema que nos garantiza que la gramática resultante es equivalente a la original, su demostración y el algoritmo a través del cual podemos obtener dicha gramática.

Después de que una cierta gramática ha sido sometida a este proceso de transformación, debe verificarse que la gramática sea LL(1), pues es para éste tipo de gramáticas para las cuales la construcción del reconocedor sintáctico no recursivo por predicción es posible. Para esta construcción utilizamos un autómata de stack, el cual reconoce a las cadenas por stack vacío. En el capitulo V. **Construcción de un reconocedor no recursivo por predicción**, se definen las gramáticas LL(1), la función de reconocimiento a través de la cual construimos la tabla de acción del autómata (la cual estará multidefinida cuando la gramática no sea LL(1)), el comportamiento del autómata y se presenta el algoritmo de reconocimiento del lenguaje que genera la gramática.

El sistema instrumentado está formado por dos programas. El primero de ellos realiza el proceso de transformación de una gramática formal de un lenguaje libre de contexto sin restricciones, y nos permite ver la gramática resultante de cada paso de la transformación. El segundo verifica, en primer lugar, que la gramática sea LL(1). En caso de que no lo sea, nos reporta para cual entrada y con cuales producciones la tabla de acción del autómata está multidefinida. En caso de que la gramática si sea LL(1), construye la tabla de acción del autómata y determina la pertenencia de cadenas al lenguaje generado por la gramática, reportando las producciones aplicadas en la construcción del árbol de derivación de la cadena.

En el capítulo VI. **Instrumentación del Sistema** se presentan, para cada uno de los programas, la entrada, la salida, las estructuras de datos y el método utilizado para objetivizar las estructuras formales o matemáticas de los algoritmos expuestos en los capítulos anteriores.

En el **apéndice** I, se presentan algunas definiciones y el algoritmo de Warshall, a los que se hace referencia en los capítulos IV, V y VI. En el **apéndice II**, están los programas fuente del sistema, y el **apéndice III** contiene la impresión de las distintas pantallas que despliega el sistema.

II. GRAMATICAS Y LENGUAJES FORMALES

Generalidades

Un lenguaje consiste de un conjunto finito o infinito de enunciados. Los lenguajes finitos pueden ser especificados enumerando exhaustivamente todos sus enunciados. Sin embargo, para los lenguajes infinitos, tal enumeración no es posible. Por otro lado, cualquier especificación de un lenguaje, para ser útil en el ámbito computacional, debe ser finita.

La descripción de los lenguajes de programación debe ser precisa y sin ambigüedades. Un <u>lenguaje formal</u> se caracteriza por ser un lengauje para el que se crea una notación especial para los símbolos del alfabeto y se describe el conjunto de reglas para operar con esta notación o para transformar los símbolos. Este tipo de lenguaje nos permite describir formalmente su sintaxis.

Una <u>gramática</u> es un mecanismo que satisface este requerimiento y a través del cual podemos dar una representación precisa del lenguaje formal. Las gramáticas permiten el estudio de la estructura de los lenguajes que consisten de un número infinito de enunciados.

Una gramática consiste de un conjunto finito no vacío de <u>reglas</u> o <u>producciones</u> a través de las cuales podemos generar los enunciados válidos del lenguaje que describe dicha gramática.

Conceptos preliminares

Un <u>alfabeto</u> es un conjunto finito de símbolos que pueden ser utilizados para construir los enunciados correctos de un lenguaje.

Una <u>cadena</u> sobre algún alfabeto es una secuencia finita de símbolos del alfabeto. Los términos <u>frase</u> y <u>enunciado</u> serán usados como sinónimos de cadena.

La <u>longitud</u> de una cadena **w**, denotada por $|\mathbf{w}|$ es el número de símbolos que componen la cadena.

La <u>cadena vacía</u>, denotada por λ es la que consta de cero símbolos, es decir $|\lambda|=0$.

La <u>concatenación</u> de dos o más cadenas es una nueva cadena, formada por las dos o más cadenas, escritas una después de otra sin espacio entre ellas. Denotaremos al operador de concatenación por $\circ$. Así, por ejemplo, la concatenación de las cadenas 'a' y 'bc' la representamos por medio de 'a'$\circ$'bc' y el resultado de dicha operación es la cadena 'abc'.

Sea **A** un alfabeto. $A \circ A = A^2$ denota a todas las cadenas de longitud 2 sobre el alfabeto **A**, $A \circ A \circ A = A^2 \circ A = A^3$ denota a todas las cadenas de longitud 3, y en general $A \circ A \circ \ldots \circ A = A^n$ denota a todas las cadenas de longitud n sobre el alfabeto **A**.

La <u>cerradura positiva</u> de **A**, denotada por A^+, se define como:

$$A^+ = A \cup A^2 \cup A^3 \cup \ldots.$$

es decir, el conjunto de todas las cadenas sobre el alfabeto **A**.

Si combinamos a la cadena vacia λ con el conjunto A^+ obtenemos el conjunto cerradura $A^* = \{\lambda\} \cup A^+$, esto es, el conjunto de todas las cadenas sobre el alfabeto **A** incluyendo a la cadena vacia.

Definiciones

Definición 1 Una <u>gramática</u> se denota como $G=(V_N,V_T,P,S)$ donde V_N es el conjunto de simbolos no terminales, V_T es el conjunto de simbolos terminales, **S** es un elemento distinguido del conjunto de simbolos no terminales llamado simbolo inicial y **P** es un subconjunto finito no vacio de la relación de $(V_N \cup V_T)^* V_N (V_N \cup V_T)^*$ a $(V_N \cup V_T)^*$. En general, un elemento (α,β) de la relación se escribe como $\alpha \rightarrow \beta$ y se llama una producción o regla de derivación.

Definición 2 Sea $G=(V_N,V_T,P,S)$ una gramática. Sea $V = V_N \cup V_T$. Para $\sigma,\tau \in V^*$, se dice que σ es una <u>derivación directa</u> de τ, denotado por $\tau \rightarrow \sigma$ si existen cadenas Φ_1 y Φ_2 (posiblemente cadenas vacias) tales que $\tau = \Phi_1\alpha\Phi_2$, $\sigma = \Phi_1\beta\Phi_2$ y $\alpha \rightarrow \beta$ es una producción de G.

Definición 3 Sea $G=(V_N,V_T,P,S)$ una gramática. La cadena τ produce σ (σ es una derivación de τ), denotado por $\tau \rightarrow^+ \sigma$ si existen cadenas $\Phi_0, \Phi_1, \ldots, \Phi_n$ (n>0) tales que $\tau = \Phi_0 \rightarrow \Phi_1$, $\Phi_1 \rightarrow \Phi_2, \ldots, \Phi_{n-1} \rightarrow \Phi_n = \sigma$. La relación $\rightarrow^+$ es la <u>cerradura transitiva</u> de la relación $\rightarrow$.

Si n=0, definimos la <u>cerradura transitiva reflexiva</u> de $\rightarrow$ como:

$$\tau \rightarrow^* \sigma \quad \text{si y sólo si } \tau \rightarrow^+ \sigma \quad \text{ó} \quad \tau = \sigma$$

Definición 4 Un <u>enunciado</u> o <u>frase</u> es cualquier derivación del simbolo no terminal distinguido **S**. El <u>lenguaje L generado por una gramática G</u> es el conjunto de todos los enunciados o frases cuyos simbolos son terminales, es decir, $L(G)=\{\sigma \mid S \rightarrow^* \sigma \text{ y } \sigma \in V_T^*\}$.

Así tenemos que, una cadena está en L(G) si:

1. La cadena consiste únicamente de simbolos terminales.
2. La cadena puede ser derivada de **S**.

Clasificación de Gramáticas

La siguiente clasificación se debe a Noam Chomsky quien divide a las gramáticas en cuatro clases distintas. Cada clase se caracteriza por el tipo de restricciones que se aplican a las producciones de la gramática.

Una gramática sin restricciones o tipo 0 es aquella en la cual no se imponen restricciones a la forma de sus producciones.

Los lenguajes generados por este tipo de gramáticas se llaman lenguajes recursivos.

Una gramática sensible al contexto o tipo 1 es aquella que únicamente contiene producciones de la forma $\alpha \rightarrow \beta$, donde $|\alpha| \leq |\beta|$. Otra manera de describir las producciones de una gramática de esta clase, de tal forma que que el significado de sensible al contexto se hace más claro, es la siguiente: Si expresamos a α y β como $\alpha = \Phi_1 A \Phi_2$ y $\beta = \Phi_1 \tau \Phi_2$ (donde Φ_1 y/o Φ_2 pudieran ser la cadena vacía) y τ debe no ser la cadena vacía, entonces la aplicación de la producción $\Phi_1 A \Phi_2 \rightarrow \Phi_1 \tau \Phi_2$ significa que **A** se reescribe como τ en el contexto entre Φ_1 y Φ_2.

Estas dos formas de describir las restricciones para las producciones son equivalentes, en el sentido de que un lenguaje definido por una gramática con producciones de la primera forma es siempre definible por alguna gramática (con frecuencia diferente a la primera) con reglas de la segunda forma.

Se dice que las gramáticas sensibles al contexto generan lenguajes sensibles al contexto.

Una gramática libre de contexto o tipo 2 es aquella que únicamente contiene producciones de la forma $\alpha \rightarrow \beta$, donde $|\alpha| \leq |\beta|$, y α es un elemento del conjunto V_N. Con estas gramáticas, al aplicar alguna producción, un símbolo no terminal es reemplazado sin tomar en cuenta a los símbolos en su vecindad o contexto.

Se dice que las gramáticas libres de contexto generan lenguaje libres de contexto.

Una gramática regular o tipo 3 es aquella que únicamente contiene producciones de la forma $\alpha \rightarrow \beta$, donde $|\alpha| \leq |\beta|$, $\alpha \in V_N$ y β es de la forma aB ó a, donde $a \in V_T$ y $B \in V_N$.

Los lenguajes generados por estas gramáticas se llaman lenguajes regulares.

Gramáticas libres de contexto

Aún cuando la motivación original de las gramáticas libres de contexto fue la descripción de los lenguajes naturales, estas gramáticas no tiene el poder para representar adecuadamente las partes significativas de estos lenguajes, ya que la dependencia del contexto se requiere frecuentemente para poder analizar apropiadamente la estructura de una oración. Sin embargo, las gramáticas libres de contexto juegan un papel importante en la lingüística computacional.

Los lenguajes libres de contexto, son de gran importancia práctica, particularmente en la definición de los lenguajes de programación, en la formalización de la noción de reconocimiento, en la simplificación de traducción de lenguajes de programación, y en otras aplicaciones de procesamiento de cadenas.

El uso de las gramáticas libres de contexto ha simplificado enormemente la definición de los lenguajes de programación y la construcción de compiladores. La razón de este éxito se debe, en parte, a que los lenguajes de programación pueden ser descritos en forma natural por este tipo de gramáticas.

Definición Una gramática libre de contexto se denota con $G=(V_N,V_T,P,S)$, donde:

V_N = conjunto de símbolos no terminales.
V_T = conjunto de símbolos terminales.

$V_N \cap V_T = \phi$

S = símbolo no terminal distinguido llamado símbolo inicial.

P = conjunto finito de producciones, donde cada producción es de la forma:

$$A \rightarrow \alpha \quad \text{con } A \in V_N \quad y \quad \alpha \in (V_N \cup V_T)^*$$

Ejemplo

La siguiente gramática $G=(V_N,V_T,P,S)$, en la que

$V_N = \{S,A,B\}$, $V_T = \{a,b,c\}$,

P: $S \rightarrow Aa \mid B$

$A \rightarrow Bc \mid \lambda$

$B \rightarrow ba \mid Ac \mid b$

donde con el símbolo | separamos las distintas producciones de cada símbolo no terminal, es una gramática libre de contexto.

Para efectos de simplificar la exposición convendremos, siempre y cuando no se indique otra cosa, la siguiente notación:

A,B,C,...	para símbolos no terminales.
S	para el símbolo inicial.
a,b,c,...	para símbolos terminales.
X,Y,Z	para símbolos ya sean terminales o no terminales.
$\alpha,\beta,\sigma,\ldots$	para cadenas de símbolos.

Arboles de derivación

Es útil representar las derivaciones de una gramática por medio de árboles. Estos árboles, llamados <u>árboles de derivación</u>, imponen cierta estructura a la derivación de los enunciados del lenguaje, que es útil en aplicaciones tales como la compilación de lenguajes de programación.

El análisis sintáctico o reconocimiento de un enunciado de algún lenguaje de programación, es precisamente la construcción de su árbol de derivación.

Los vértices de un árbol de derivación están etiquetados con símbolos de la gramática, ya sean éstos terminales o no terminales, o con la cadena vacía. El nodo raíz del árbol está etiquetado con el símbolo inicial de la gramática. Las hojas del árbol representan a los símbolos terminales del enunciado que ha sido reconocido. Si un vértice interior **n** está etiquetado con **A** y los hijos de **n** están etiquetados con X_1, X_2, ..., X_k de izquierda a derecha, entonces $A \rightarrow X_1X_2\ldots X_k$ debe ser una producción de la gramática.

Formalizando estos conceptos tenemos la siguiente definición:

Definición Sea $G=(V_N,V_T,P,S)$ una gramática libre de contexto. Un árbol es un árbol de derivación si:

1. Todo vértice tiene una etiqueta, que es un símbolo de $V_N \cup V_T \cup \{\lambda\}$.

2. La etiqueta de la raíz es **S**.

3. Si un vértice es interior y tiene etiqueta **A**, entonces **A** deb estar en V_N.

4. Si **n** tiene etiqueta **A** y los vértices $n_1,n_2,\ldots,n_k$ son los hijos del vértice **n**, de izquierda a derecha etiquetados con $X_1,X_2,\ldots,X_k$ respectivamente, entonces $A \rightarrow X_1X_2\ldots X_k$ debe ser una producción de P.

5. Si un vértice **n** tiene etiqueta λ, entonces **n** es una hoja y es el único hijo de su padre.

Ejemplo

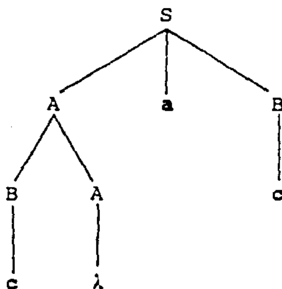
Consideremos la siguiente gramática:

$S \rightarrow AaB$	$A \rightarrow \lambda$
$S \rightarrow bA$	$B \rightarrow c$
$A \rightarrow BA$	$B \rightarrow Aab$

La derivación de la cadena 'cac' se obtiene aplicando las producciones $S \rightarrow AaB$, $A \rightarrow BA$, $B \rightarrow c$, $A \rightarrow \lambda$ y $B \rightarrow c$.

$$S \Rightarrow AaB \Rightarrow BAaB \Rightarrow cAaB \Rightarrow c\lambda aB \Rightarrow c\lambda ac = cac$$

El árbol de derivación de la cadena 'cac' es



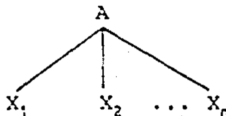
Definición Decimos que una cadena es un producto de un árbol de derivación, si y sólo si está formada por la concatenación de izquierda a derecha de las hojas de éste y contiene exclusivamente símbolos terminales.

Hemos mencionado que el reconocimiento de un enunciado de algún lenguaje es precisamente la construcción de su árbol de derivación. Tenemos así el siguiente teorema:

Teorema Sea $G=(V_N,V_T,P,S)$ una gramática libre de contexto. Entonces $S \Rightarrow^* \alpha$ si y sólo si existe un árbol de derivación de la gramática G con producto α.

Demostración Probaremos que para cualquier **A** en V_N, $A \Rightarrow^* \alpha$ si y sólo si existe un árbol con raíz **A** y producto α.

Supongamos que α es el producto de un árbol con raíz **A**. Probaremos por inducción sobre la profundidad del árbol, que $A \Rightarrow^* \alpha$. Si la profundidad del árbol es 1, el árbol se ve de la siguiente forma:



En ese caso, $X_1X_2 \ldots X_n$ debe ser α y, por definición de un árbol de derivación, $A \rightarrow \alpha$ debe ser una producción de P.

Supongamos ahora que el resultado es cierto para árboles de profundidad k-1 a lo más. Supongamos también que α es el producto de un árbol con raíz **A** y de profundidad k para alguna k>1. Consideremos a los hijos de la raíz. Estos pueden no ser todos hojas. Sean las etiquetas de los hijos $X_1, X_2, \ldots, X_n$ en orden de izquierda a derecha. Entonces $A \rightarrow X_1X_2 \ldots X_n$ es una producción de P. Nótese que **n** puede ser cualquier entero mayor o igual al del argumento que sigue.

Si el i-ésimo hijo no es una hoja, es la raíz de un subárbol, y X_i es un símbolo no terminal. Este subárbol tiene raíz X_i y tiene algún producto α_i. Si el vértice i es una hoja, sea $\alpha_i = X_i$. Es fácil ver que si j<i, el vértice j y todos sus descendientes están a la izquierda del vértice i y de todos sus descendientes. Entonces $\alpha = \alpha_1\alpha_2 \ldots \alpha_n$. Un subárbol debe tener menos vértices que su árbol, a menos que el subárbol sea el árbol completo. Por la hipótesis de inducción, para cada vértice i que no es una hoja $X_i \Rightarrow^* \alpha_i$, ya que el subárbol con raíz X_i no es el árbol completo. Si $X_i = \alpha_i$, entonces $X_i \Rightarrow^* \alpha_i$. Podemos poner todas estas derivaciones parciales juntas, y vemos que

$$A \Rightarrow X_1X_2 \ldots X_n \Rightarrow^* \alpha_1X_2 \ldots X_n \Rightarrow^* \alpha_1\alpha_2X_3 \ldots X_n \Rightarrow^* \ldots \Rightarrow^* \alpha_1\alpha_2 \ldots \alpha_n = \alpha.$$

Por lo tanto, $A \Rightarrow^* \alpha$.

Supongamos ahora que $A \Rightarrow^* \alpha$. Probaremos ahora que existe un árbol con raíz **A** y con producto α. Si $A \Rightarrow^* \alpha$ en un solo paso, entonces $A \rightarrow \alpha$ es una producción de P, y existe un árbol con producto α, de la forma mostrada antes.

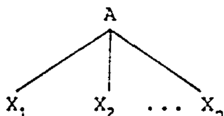
Asumamos ahora que para cualquier símbolo no terminal **A** si $A \Rightarrow^* \alpha$ por una derivación de menos de **k** pasos, entonces existe un árbol con raíz **A** y producto α. Supongamos que $A \Rightarrow^* \alpha$ por una derivación de **k** pasos. Sea el primer paso $A \rightarrow X_1X_2 \ldots X_n$. Es claro que cualquier símbolo de α debe ser alguno de los $X_1, X_2, \ldots, X_n$ o ser derivable de alguno de ellos. También, la porción de α derivada de X_i debe quedar a la izquierda de los símbolos derivados de X_j si

$i<j$. Por lo tanto, podemos escribir α como $\alpha_1\alpha_2\ldots\alpha_n$, donde **para** cada **i** entre 1 y **n**,

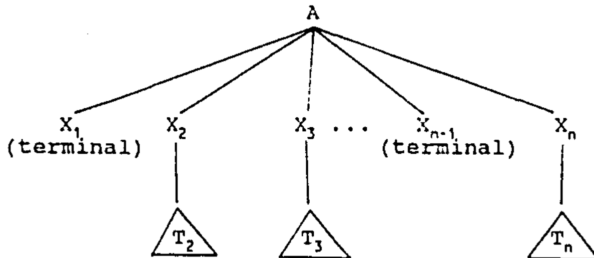
1) $\alpha_i = X_i$ si X_i es un símbolo terminal, y
2) $X_i \Rightarrow^* \alpha_i$ si X_i es un símbolo no terminal.

Si X_i es un símbolo no terminal, entonces la derivación α_i de X_i debe ser de menos de **k** pasos, ya que la derivación completa $A \Rightarrow^* \alpha$ es de **k** pasos, y el primer paso no es parte de la derivación $X_i \Rightarrow^* \alpha_i$. Así que, por la hipótesis de inducción, para cada X_i que es un símbolo no terminal, existe un árbol con raiz X_i y producto α_i. Sea este árbol T_i.

Comenzamos construyendo un árbol con raiz **A**, con **n** hojas etiquetadas con $X_1, X_2, \ldots, X_n$ y sin más vértices:



Cada vértice con etiqueta X_i, donde X_i es un símbolo no terminal, se reemplaza por el árbol T_i. Si X_i es un símbolo terminal, no hacemos ningún reemplazo. Por ejemplo:



El producto de este árbol es α. ■

LLamaremos una <u>forma sentencial</u> de una derivación a la cadena formada por la concatenación de los símbolos que estén en las hojas del árbol en algún paso de una derivación.

Derivaciones izquierda y derecha. Ambigüedad

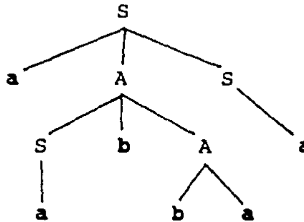
Si en cada paso de una derivación se aplica una producción al símbolo no terminal más a la izquierda en el árbol, entonces la derivación se llama derivación izquierda. Análogamente si la producción se aplica al símbolo no terminal más a la derecha en el árbol, entonces la derivación se llama derivación derecha. Si **w** está en L(G) para una gramática libre de contexto G, entonces **w** tiene por lo menos un árbol de reconocimiento asociado, y correspondientes a un árbol de reconocimiento particular, **w** tiene una derivación izquierda y una derecha únicas.

Ejemplo

Consideremos la siguiente gramática:

$$S \rightarrow aAS \qquad A \rightarrow SS$$
$$S \rightarrow a \qquad A \rightarrow ba$$
$$A \rightarrow SbA$$

El árbol de reconocimiento de la cadena 'aabbaa' es:



y sus respectivas derivaciones son:

Derivación izquierda:

$$S \Rightarrow aAS \Rightarrow aSbAS \Rightarrow aabAS \Rightarrow aabbaS \Rightarrow aabbaa$$

Derivación derecha:

$$S \Rightarrow aAS \Rightarrow aAa \Rightarrow aSbAa \Rightarrow aSbbaa \Rightarrow aabbaa$$

Por supuesto, una cadena **w** podría tener varias derivaciones izquierdas y derechas si hubiera más de un árbol de reconocimiento para **w**. Sin embargo, es fácil probar que para cada árbol de reconocimiento, sólo pueden ser obtenidas una derivación izquierda y una derecha.

Una gramática libre de contexto G en la cual algún enunciado tiene dos árboles de reconocimiento se llama <u>ambigua</u>. Una definición equivalente de ambigüedad es que algún enunciado tenga más de una derivación izquierda ó más de una derivación derecha. Cuando un enunciado tiene más de un árbol de reconocimiento, usualmente tiene más de un significado.

<u>Ejemplo</u>

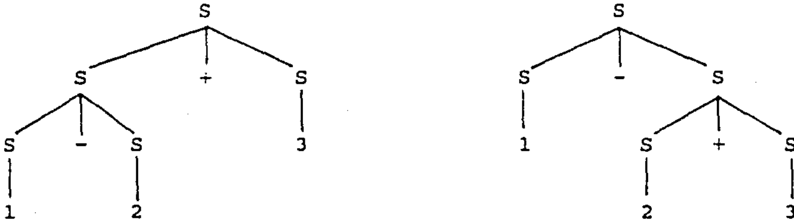
Consideremos la siguiente gramática

$S \rightarrow S+S \qquad S \rightarrow S-S$

$S \rightarrow 1 \qquad S \rightarrow 2$

$S \rightarrow 3$

Esta gramática es ambigua pues tiene dos árboles de reconocimiento distintos para el enunciado 1-2+3, y dos significados distintos.



El primer árbol representa al enunciado (1-2)+3 y el segundo al enunciado 1-(2+3), cuyos significados no son el mismo.

En aplicaciones de compiladores, debemos diseñar gramáticas que no sean ambiguas, o usar gramáticas ambiguas con reglas adicionales para resolver la ambigüedad, cuando ésto sea posible.

Un lenguaje libre de contexto para el cual toda gramática libre de contexto es ambigua, es un lenguaje libre de contexto inherentemente ambiguo.

III. ANALISIS SINTACTICO

La función del reconocedor sintáctico de un compilador y tipos de reconocedores.

En el proceso de un compilador el reconocedor o analizador sintáctico obtiene una cadena de símbolos del analizador léxico y verifica que la cadena pueda ser generada por la gramática que define al lenguaje fuente. Se espera que el reconocedor reporte cualquier error sintáctico en una forma intelegible. Así mismo, debe ser capaz de recuperarse de los errores que ocurren comúnmente, de tal forma que pueda continuar procesando el resto de la entrada.

Al discutir este problema, es útil pensar en la construcción de un árbol de reconocimiento, aún cuando un compilador no construye en realidad tal árbol. Sin embargo, un reconocedor debe ser capaz de construirlo, pues de lo contrario no puede garantizarse que el reconocimiento sea correcto.

Existen distintos tipos de reconocedores sintácticos, que determinan si una cierta cadena **w**, pertenece o no al lenguaje generado por una gramática libre de contexto **G**.

El más simple de los algoritmos que construye un reconocedor para cualquier gramática es el que se expone a continuación. En primer lugar debemos transformar la gramática a forma normal de Greibach (ver formas normales, en el capitulo IV). Como cada producción de una gramática en esta forma agrega exactamente un símbolo terminal a la cadena que está siendo generada, sabemos que si **w** tiene una derivación en la gramática **G**, esta derivación tiene exactamente tantos pasos como la longitud de **w**. Ahora, si todo símbolo no terminal de la gramática tiene a lo más **k** producciones, entonces hay a lo más $k^{|w|}$ derivaciones izquierdas de cadenas de longitud $|w|$. Debemos, sistemáticamente, ir generando todas las cadenas de longitud $|w|$, hasta derivar a la cadena **w**. Si derivamos a todas las cadenas y ninguna de ellas resulta ser **w**, entonces la cadena no pertenece al lenguaje.

Este algoritmo, aunque simple, es sumamente ineficiente pues toma un tiempo de orden exponencial en función de la longitud de la cadena de entrada.

Los <u>métodos universales de reconocimiento</u> tales como la técnica de programación dinámica descubierta independientemente por J. Cocke, Younger (1967) y Kasami (1965), y el algoritmo de Earley(1970), permiten también el reconocimiento de cualquier gramática libre de contexto.

El algoritmo de Cocke-Younger-Kasami parte de una gramática libre de contexto en forma normal de Chomsky (ver formas normales, en el capitulo IV) y una cadena de entrada $w = a_1 a_2 \ldots a_n$. Usando programación dinámica, se construye una tabla T de **n** x **n**, en la cual $T[i,j] = \{ A \mid A \rightarrow^* a_i a_{i+1} \ldots a_{i+j-1} \}$. Esto es, **A** deriva una cadena

de longitud j que comienza en la posición i. La cadena **w** pertence al lenguaje generado por la gramática si y sólo si **S**, el símbolo inicial, está en T[1,n].

Este algoritmo trabaja en un tiempo proporcional al cubo y requiere una cantidad de espacio proporcional al cuadrado de la longitud de la cadena de entrada.

El algoritmo de Earley, parte de una gramática en forma normal de Greibach (ver formas normales, del capítulo IV), y una cadena de longitud **n**. Al igual que el anterior, trabaja en un tiempo de orden n^3 y ocupa un espacio de orden n^2. Si la gramática no es ambigua, puede reducir el tiempo a un orden de n^2.

Estos métodos universales de reconocimiento, aunque tienen un orden de complejidad menor al del primer algoritmo presentado, siguen siendo ineficientes para ser usados en la producción de compiladores.

Los métodos comúnmente utilizados en compiladores se clasifican en métodos de arriba hacia abajo y métodos de abajo hacia arriba. Como sus nombres lo indican, los reconocedores de arriba hacia abajo construyen los árboles de reconocimiento comenzando por la raiz del árbol y continúan la construcción hacia abajo hasta las hojas. En los reconocedores de abajo hacia arriba la construcción del árbol comienza por las hojas y continúa hacia arriba hasta la raíz.

Reconocimiento de arriba hacia abajo.

El reconocimiento de arriba hacia abajo puede ser visto como la búsqueda de la derivación izquierda de una cierta cadena de entrada. Esto es, la construcción del árbol de derivación para la cadena comenzando por la raiz del árbol y creando los nodos en preorden.

Uno de los métodos más generales de arriba hacia abajo es el llamado reconocimiento descendente recursivo, el cual involucra "dar marcha atrás", esto significa que se hacen lecturas repetidas de los simbolos de la cadena de entrada. Sin embargo, estos reconocedores no son tan frecuentes ya que, aunque requieren un espacio lineal, trabajan en un tiempo exponencial en función a la longitud de la cadena de entrada.

El reconocimiento no recursivo por predicción es otro método de reconocimiento de arriba hacia abajo, que resulta ser más eficiente que el anterior. Este reconocedor hace c_1n operaciones y utiliza c_2n espacio al procesar una cadena de longitud **n**, donde c_1 y c_2 son constantes. El precio que pagamos por esta eficiencia es que no todos los lenguajes libres de contexto pueden ser generados por gramáticas para las cuales es posible la construcción de este

tipo de reconocedores. Sin embargo, la clase restringida de gramáticas libres de contexto para las cuales es posible dicha construcción, es suficiente para la especificación de las reglas sintácticas de los lenguajes de programación.

Este algoritmo de reconocimiento se caracteriza por el hecho de que la cadena de entrada se lee una sola vez de izquierda a derecha y el proceso de reconocimiento es determinístico, esto es, en cada paso del reconocimiento está determinada de manera única la producción a aplicar.

Reconocimiento no recursivo por predicción.

Para la construcción de un reconocedor no recursivo por predicción, en el que no es necesario el "dar marcha atrás", debemos saber, dado el símbolo **a** que estamos leyendo de la cadena de entrada y el símbolo no terminal **A** a ser expandido, cuál de las alternativas de la producción $A \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n$ es la que deriva una cadena que comienza con **a**. Esto significa que se debe poder detectar la alternativa apropiada con sólo ver el primer símbolo de su derivación.

La construcción de este tipo de reconocedores es posible, siempre y cuando la gramática que define al lenguaje satisface ciertas propiedades.

En primer lugar la gramática debe ser libre de contexto, estar reducida, no ser recursiva por la izquierda y estar factorizada por la izquierda. Cuando la gramática no cumple con estas características, es posible someterla a un proceso de transformación, a través del cual obtengamos una gramática equivalente que si satisface las propiedades requeridas. Tanto el significado de estas propiedades, como el proceso de transformación serán discutidos en el capitulo IV.

Finalmente, para que la construcción del reconocedor sea posible, la gramática debe ser una gramática LL(1). Definiremos con precisión a este tipo de gramáticas en el capitulo V.

La construcción de este tipo de reconocedores es posible, como veremos en el capitulo V, si utilizamos para ello un autómata de stack, el cual reconoce a las cadenas por stack vacio.

IV. TRANSFORMACION DE GRAMATICAS LIBRES DE CONTEXTO

Generalidades

Como mencionamos en el capítulo III, para que la construcción automática de un analizador sintáctico no recursivo por predicción del lenguaje que genera cierta gramática libre de contexto sea posible, dicha gramática debe satisfacer ciertas propiedades.

Cuando la gramática no satisface estas propiedades es posible someterla a un proceso de transformación a través del cual obtenemos una gramática equivalente a la original que si las satisface.

Que dos gramáticas sean equivalentes significa que el lenguaje que genera cada una de ellas resulta ser el mismo.

Los pasos de este proceso de transformación son los siguientes:

1. Eliminar de la gramática las producciones vacías.

2. Eliminar de la gramática las producciones unitarias y los ciclos.

3. Eliminar de la gramática los símbolos inactivos.

4. Eliminar de la gramática los símbolos inalcanzables.

5. Si la gramática es recursiva por la izquierda:

    5.1 Transformarla a Forma Normal de Chomsky.
    5.2 Transformarla a Forma Normal de Greibach.

6. Factorizar la gramática por la izquierda.

En cada uno de los pasos del proceso debemos obtener gramáticas que sean equivalentes a la anterior. Como se verá más adelante, el orden en el que se aplican cada una de las transformaciones es importante.

Eliminación de símbolos inactivos.

Definición Si un símbolo no terminal deriva cuando menos una cadena de símbolos terminales, se dice que dicho símbolo es un no terminal <u>activo</u>. Obsérvese que todos los símbolos terminales se consideran activos.

Lema 1 Dada una GLC $G=(V_N,V_T,P,S)$ con $L(G)\neq\phi$, existe una GLC equivalente $G'=(V_N',V_T,P',S)$ tal que para cada $\mathbf{A} \in V_N'$ existe alguna $\mathbf{w} \in V_T^*$ para la cual $A \Rightarrow^* w$.

Demostración Todo símbolo no terminal A con alguna producción $A \rightarrow w$ en P claramente pertenece a V_N'. Si $A \rightarrow X_1X_2...X_n$ es una producción, donde cada X_i es un símbolo terminal o un símbolo no terminal que está en V_N', entonces a través de una derivación que comience con $A \rightarrow X_1X_2...X_n$, podemos derivar una cadena de símbolos terminales y por lo tanto A pertenece a V_N'. El conjunto V_N' puede ser calculado a través de un algoritmo iterativo. P' es el conjunto de todas las producciones cuyos símbolos estén en $V_N' \cup V_T$. Ciertamente $G'=(V_N',V_T,P',S)$ satisface la propiedad de que si $A \in V_N'$, entonces $A \stackrel{*}{\Rightarrow} w$ para alguna w.

Como toda derivación de G' es una derivación de G, entonces $L(G') \subseteq L(G)$. Supongamos que existe alguna w en L(G) que no pertenece a L(G'), entonces cualquier derivación de w en G debe involucrar algún símbolo no terminal en V_N-V_N' o una producción de P-P' (lo que implica que estamos usando algún símbolo de V_N-V_N').

Entonces existe un símbolo no terminal en V_N-V_N' que deriva una cadena de símbolos terminales, lo cual es una contradicción. Por lo tanto L(G)=L(G'). ■

Algoritmo 1 Elimina símbolos no terminales inactivos.

Entrada: Una gramática libre de contexto.

Salida: Una gramática equivalente sin símbolos inactivos.

Método:

```
    begin
1)      ANTERIOR := φ;
2)      NUEVO := {A| A → w para alguna w ∈ V_T*}
3)      while ANTERIOR ≠ NUEVO do
        begin
4)          ANTERIOR := NUEVO;
5)          NUEVO := ANTERIOR ∪ {A| A → α para alguna
                                  α ∈ (V_T ∪ ANTERIOR)*};
        end;
6)      V_N' := NUEVO;
7)      P' := P-{A→β| A ∉ V_N' ó X ∈ β y X ∉ V_N'};
    end;
```

Podemos garantizar que este algoritmo termina en un número finito de pasos, ya que el número de símbolos no terminales N y el número de producciones p son finitos. La construcción del conjunto NUEVO en la línea 2 necesita realizar a lo más p comparaciones, el ciclo de la línea 3 se ejecuta N-1 veces a lo más, la construcción del conjunto NUEVO en la línea 5 necesita a lo más de p pasos y la construcción del conjunto P' en la línea 7 requiere de p comparaciones. Por lo tanto, el algortimo trabaja en un tiempo de O(Np).

En la línea 2 guarda en NUEVO los símbolos no terminales que derivan directamente una cadena de símbolos terminales. En la línea 5 agrega a NUEVO los símbolos no terminales que derivan directamente cadenas formadas por símbolos terminales y símbolos no terminales que ya estaban en NUEVO. Termina cuando ya no hay más símbolos que agregar a NUEVO. En la línea 7 construimos el nuevo conjunto de producciones que resulta de eliminar las producciones que involucran símbolos inactivos.

Observemos que si el símbolo inicial de la gramática resulta ser inactivo, es decir, si al aplicar el algoritmo resulta que el símbolo inicial no pertenece al conjunto V_N', entonces a partir de dicha gramática no podemos derivar ninguna cadena, ya que toda derivación debe partir del símbolo inicial.

Ejemplo

Consideremos la siguiente gramática:

$S \rightarrow aAa$ $\quad B \rightarrow abb$
$A \rightarrow Sb$ $\quad B \rightarrow aC$
$A \rightarrow bBB$ $\quad C \rightarrow aCA$
$A \rightarrow Da$ $\quad D \rightarrow abC$

En esta gramática tenemos que el conjunto inicial NUEVO = {B}. Ahora, como el símbolo no terminal A tiene la producción $A \rightarrow bBB$ entonces NUEVO = {B,A}. Como el símbolo no terminal S tiene la producción $S \rightarrow aAa$, entonces NUEVO = {B,A,S}.

En este momento ya no hay más símbolos que agregar a NUEVO, por lo tanto {B,A,S} es el conjunto de símbolos no terminales activos. C y D son símbolos inactivos, así que eliminamos las producciones que involucran a dichos símbolos, obteniendo la siguiente gramática:

$S \rightarrow aAa$ $\quad A \rightarrow bBB$
$A \rightarrow Sb$ $\quad B \rightarrow abb$

que resulta ser equivalente a la original y en la que todos sus símbolos no terminales son activos.

Eliminación de símbolos inalcanzables.

Definición Un símbolo que aparece en alguna forma sentencial derivable del símbolo inicial se llama símbolo alcanzable, en otro caso se llama símbolo inalcanzable.

Lema 2 Dada una GLC $G=(V_N,V_T,P,S)$ podemos encontrar una GLC equivalente $G'=(V_N',V_T',P',S)$ tal que para cada $X \in V_N' \cup V_T'$ existen α y β en $(V_N' \cup V_T')^*$ para las cuales $S \overset{*}{\rightarrow} \alpha X \beta$.

Demostración El conjunto $V_N' \cup V_T'$ de símbolos que aparecen en alguna de las formas sentenciales de G se construye con el siguiente algoritmo iterativo. Agregar S a V_N'. Si A está en V_N' y $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$, entonces agregar todos los símbolos no terminales de $\alpha_1, \alpha_2, \ldots, \alpha_n$ al conjunto V_N' y todos los símbolos terminales de $\alpha_1, \alpha_2, \ldots, \alpha_n$ al conjunto V_T'. P' es el conjunto de producciones de P que contienen únicamente símbolos de $V_N' \cup V_T'$.

Toda forma sentencial derivable de G' es derivable de G. Toda forma sentencial derivable de G tiene símbolos del conjunto $V_N' \cup V_T'$, por lo tanto también es derivable de G'.
Entonces L(G')=L(G). ■

Algoritmo 2 Elimina símbolos inalcanzables.

Entrada: Una gramática libre de contexto.

Salida: Una gramática equivalente sin símbolos inalcanzables.

Método:
```
    begin
 1)         ANTERIOR := ∅;
 2)         TERM := ∅;
 3)         NOTERM := {S};
 4)         while ANTERIOR ≠ NOTERM do
            begin
 5)               ANTERIOR := NOTERM;
 6)               NOTERM := NOTERM ∪ {X| A → α,  A ∈ ANTERIOR,
                                         X ∈ V_N y X ∈ α};
 7)               TERM := TERM ∪ {x| A → α, A ∈ ANTERIOR,
                                     x ∈ V_T y x ∈ α};
            end;

 8)         V_N':= NOTERM;
 9)         V_T':= TERM;
10)         P' := P-{A→β| A ∉ V_N' ó X ∈ β y X ∉ (V_N' ∪ V_T')};
    end;
```

Podemos garantizar que este algoritmo termina en un número finito de pasos, ya que el número de símbolos no terminales N y el número de producciones p son finitos. El ciclo de la línea 4 se ejecuta N-1 veces a lo más. Las construcciones de los conjuntos NOTERM en la línea 6 y TERM en la línea 7 requieren, cada una, de p pasos a lo más. La construcción del conjunto P' en la línea 10 requiere de p comparaciones. Por lo tanto, el algoritmo trabaja en un tiempo de O(Np).

Ejemplo

Consideremos la siguiente gramática:

$S \to aSb$ $A \to aAc$
$S \to bAB$ $C \to aSbS$
$S \to a$ $C \to aba$
$B \to d$

En esta gramática tenemos que el conjunto inicial NOTERM de símbolos no terminales alcanzables es {S}. Como $S \to aSb$, $S \to a$ y $S \to bAB$ entonces NOTERM = {S,A,B} y el conjunto de símbolos terminales alcanzables TERM = {a,b}. Ahora como A y B son símbolos no terminales alcanzables y tenemos las producciones $B \to d$ y $A \to aAc$ entonces TERM = TERM $\cup$ {d,c} = {a,b,d,c}. Así que el símbolo C es inalcanzable, por lo tanto eliminamos todas las producciones que involucran a dicho símbolo y obtenemos la siguiente gramática:

$S \to aAb$ $A \to aAc$
$S \to bAB$ $B \to d$
$S \to d$

que resulta ser equivalente a la original y en la que todos sus símbolos son alcanzables.

Definición Si un símbolo no terminal es activo y alcanzable decimos que es un símbolo no terminal útil, en otro caso es un símbolo inútil.

La aplicación de los algoritmos 1 y 2 a una cierta gramática nos permite obtener una gramática equivalente sin símbolos inútiles. Observemos que el orden de aplicación de los algoritmos es importante. En primer lugar debemos aplicar el algoritmo que elimina los símbolos inactivos y las producciones asociadas a ellos para después aplicar a la gramática resultante de ello el algoritmo que elimina símbolos inalcanzables y las producciones asociadas a ellos.

La razón por la cual debemos alicar los algoritmos en este orden es que pudiera suceder que algunos símbolos se vuelvan inalcanzables sólo después de que algún símbolo no terminal inactivo y las producciones asociadas a él hayan sido eliminadas.

Ejemplo

Consideremos la siguiente gramática:

$S \to ccc$ $B \to aBa$
$S \to Abccc$ $B \to AC$
$A \to Ab$ $C \to Cb$
$A \to aBa$ $C \to b$

Aplicando el algoritmo que elimina símbolos inactivos obtenemos la siguiente gramática:

$$S \rightarrow CCC$$
$$C \rightarrow Cb$$
$$C \rightarrow b$$

Aplicando ahora el algoritmo que elimina símbolos inalcanzables, obtenemos la gramática:

$$S \rightarrow CCC$$

Obsérvese que el símbolo no terminal C se vuelve inalcanzable después de la eliminación de símbolos inactivos. Si aplicamos los algoritmos en orden inverso, el símbolo C y sus producciones pertenecen a la gramática resultante, que resulta ser una gramática con símbolos inútiles, a saber el símbolo no terminal C.

Teorema 1 Todo lenguaje libre de contexto que no sea vacío es generado por una gramática sin símbolos inútiles.

Demostración Sea L=L(G) un lenguaje libre de contexto no vacío. Sea G_1 el resultado de aplicar la construcción del Lema 1 a G y sea G_2 el resultado de aplicar el Lema 2 a G_1. Supongamos que G_2 tiene un símbolo inútil **X**. Por el Lema 2, sabemos que hay una derivación $S \Rightarrow^* \alpha X\beta$ en G_2. Como todos los símbolos de G_2 son símbolos de G_1, entonces sabemos por el Lema 1 que $S \Rightarrow^* \alpha X\beta \Rightarrow^* w$ en G_1 para alguna cadena de terminales w. Entonces ningún símbolo en la derivación $\alpha X\beta \Rightarrow^* w$ de G_1 es eliminado por el Lema 2. Por lo tanto **X** deriva una cadena de terminales en G_2, y entonces **X** no puede ser símbolo inútil como supusimos. ■

Eliminación de producciones vacías.

Definición Las producciones de la forma $A \rightarrow \lambda$ son las llamadas producciones vacías.

Podemos eliminar las producciones vacías de una cierta gramática G, siempre y cuando $\lambda \notin L(G)$. El método consiste en determinar, para cada símbolo no terminal **A**, si $A \Rightarrow^* \lambda$. Si esto ocurre, dicho símbolo se llama nulo. Debemos reemplazar cada producción $B \rightarrow X_1X_2...X_n$ por todas las producciones que se forman al eliminar los distintos subconjuntos formados por las X_i's nulas, sin incluir la producción $B \rightarrow \lambda$, aún cuando todas las X_i's sean nulas.

Teorema 2 Si $L=L(G)$ para alguna gramática libre de contexto $G=(V_N,V_T,P,S)$, entonces $L-(\lambda)$ es $L(G')$ para alguna gramática libre de contexto G' sin símbolos inútiles ni producciones vacías.

Demostración Podemos determinar los símbolos nulos de G con el siguiente algoritmo iterativo. Para empezar, si $A \to \lambda$ es una producción, entonces **A** es nulo. Ahora si $B \to \alpha$ es una producción y todos los símbolos de α resultan ser nulos, entonces **B** también es nulo. Repetimos este proceso hasta que no podamos encontrar más símbolos nulos.

El conjunto de producciones P' se construye como sigue. Si $A \to X_1X_2 \ldots X_n$ está en P, entonces agregamos a P' todas las producciones $A \to \alpha_1\alpha_2 \ldots \alpha_n$ donde

1) Si X_i no es nulo, entonces $\alpha_i=X_i$
2) Si X_i es nulo, entonces α_i es X_i ó λ
3) No todas las α_i's son λ

Sea $G_1=(V_N,V_T,P',S)$, tenemos ahora que demostrar que para toda **A** en V_N y w en V_T^*, $A \Rightarrow^* w$ en G_1 si y sólo si $w \neq \lambda$ y $A \Rightarrow^* w$ en G.

P.d. Si $w \neq \lambda$ y $A \Rightarrow^* w$ en G entonces $A \Rightarrow^* w$ en G_1

Supongamos que $A \Rightarrow w$ en G en i pasos y $w \neq \lambda$, probaremos por inducción sobre i que $A \Rightarrow^* w$ en G_1.

Sea i=1, $A \to w$ es una producción de P y como $w \neq \lambda$ entonces $A \to w$ es una producción de P'.

PASO INDUCTIVO : Sea i>1 entonces $A \Rightarrow X_1X_2 \ldots X_n$ en G y $X_1X_2 \ldots X_n \Rightarrow w$ en i-1 pasos en G.

Escribimos $w=w_1w_2 \ldots w_n$ tal que para cada j, $X_j \Rightarrow^* w_j$ en G en menos de i pasos.

Si $w_j \neq \lambda$ y X_j es un símbolo no terminal entonces por hipótesis de inducción tenemos que $X_j \Rightarrow^* w_j$ en G_1.

Si $w_j=\lambda$, entonces X_j es nulo. Así, $A \to \beta_1\beta_2 \ldots \beta_n$ es una producción en P', donde $\beta_j=X_j$ si $x_j \neq \lambda$ y $\beta_j=\lambda$ si $w_j=\lambda$.

Como $w \neq \lambda$, no todas las β_j son λ. Por lo tanto tenemos una derivación $A \Rightarrow \beta_1\beta_2 \ldots \beta_n \Rightarrow^* w_1\beta_2 \ldots \beta_n \Rightarrow^* w_1w_2\beta_3 \ldots \beta_n \Rightarrow^* \ldots \Rightarrow^* w_1w_2 \ldots w_n=w$ en G_1.

P.d. Si $A \Rightarrow^* w$ en G_1 entonces $w \neq \lambda$ y $A \Rightarrow^* w$ en G

Supongamos $A \Rightarrow^* w$ en G_1 en i pasos. Ciertamente $w \neq \lambda$, ya que G_1 no tiene producciones vacías. Probaremos por inducción sobre i que $A \Rightarrow^* w$ en G.

Sea $i=1$, entonces $A \rightarrow w$ es una producción de P'. Debe haber una producción $A \rightarrow \alpha$ en P tal que eliminando ciertos símbolos nulos de α obtengamos la cadena w. Entonces hay una derivación $A \Rightarrow \alpha \Rightarrow^* w$ en G donde la derivación $\alpha \Rightarrow^* w$ involucra derivaciones a λ de los símbolos nulos de α que fueron eliminados para obtener w.

PASO INDUCTIVO : Sea $i>1$ entonces $A \Rightarrow X_1X_2...X_n$ en G_1 y $X_1X_2...X_n \Rightarrow w$ en $i-1$ pasos en G_1.

Debe haber alguna $A \rightarrow \beta$ en P tal que $X_1X_2...X_n$ se obtiene eliminando ciertos símbolos nulos de β. Así que $A \Rightarrow^* X_1X_2...X_n$ en G.

Escribimos $w=w_1w_2...w_n$ tal que para cada j, $X_j \Rightarrow^* w_j$ en G_1 en menos de i pasos.

Por la hipótesis de inducción, $X_j \Rightarrow^* w_j$ en G si X_j es un símbolo no terminal. Ciertamente si X_j es un terminal, entonces $w_j=X_j$ y $X_j \Rightarrow^* w_j$ en G es cierto trivialmente. Así que $A \Rightarrow^* w$ en G.

Para completar la demostración observemos que G_1 no tiene producciones vacías. Si aplicamos el Teorema 1 para eliminar los símbolos inútiles (con lo cual no estaremos agregando nuevas producciones) a la gramática G_1, obtenemos una gramática G' que además no tiene símbolos inútiles. Más aún $S \Rightarrow^* w$ en G' si y sólo si $w \neq \lambda$ y $S \Rightarrow^* w$ en G. Esto es, $L(G')=L(G)-\{\lambda\}$. ■

Algoritmo 3 Elimina producciones vacías.

<u>Entrada</u>: Una gramática libre de contexto.

<u>Salida</u>: Una gramática equivalente sin producciones vacías.

<u>Método</u>:

```
/* Calcula el conjunto V_Ne={A ∈ V_N| A ⇒* λ} */
    begin
  1)        ANTERIOR := φ;
  2)        V_Ne={A ∈ V_N| A → λ};
  3)        NUEVO := V_Ne;
  4)        while ANTERIOR ≠ NUEVO do
            begin
  5)              ANTERIOR := NUEVO;
  6)              NUEVO := ANTERIOR ∪ {A ∈ V_N| A → Xα donde
                                              X ∈ ANTERIOR y
                                              α ∈ (V_N ∪ V_T)*};
            end;

  7)        ANTERIOR := φ;
```

```
 8)         while ANTERIOR ≠ V_Ne do
            begin
 9)              ANTERIOR := V_Ne;
10)              V_Ne := ANTERIOR ∪ {A ∈ NUEVO| A → Xα donde
                                                  α ∈ (V_N ∪ V_T)* y
                                                  para toda Y ∈ α
                                                  Y ∈ ANTERIOR};
            end;

/* Elimina las producciones vacias */

 11)        P' := P-{A → λ};

/* Generamos las nuevas producciones */

 12)        NUEVO := φ;
 13)        Repite para cada producción de P' de la forma A → α
                 donde al menos hay una B_i, 1≤i≤n, tal que
                 α = Φ_1 B_1 Φ_2 B_2 Φ_3 ... Φ_n B_n Φ_n+1 , B_1,B_2,...,B_n ∈ V_Ne
                                                y Φ_i ∈ ((V_N-V_Ne) ∪ V_T)*

                 NUEVO := NUEVO ∪ {A → β| β es el resultado
                                    de eliminar los distintos
                                    subconjuntos posibles de
                                    B_i's de α excepto β=λ};
 14)         NUEVO := NUEVO ∪ {S'→S};
 15)         P' := P' ∪ NUEVO;

      end;
```

Podemos garantizar que este algoritmo termina en un número finito de pasos, ya que el número de símbolos no terminales N y el número de producciones p son finitos. La construcción del conjunto V_{Ne} en la línea 2 requiere de p comparaciones a lo más. El ciclo de la línea 4 se ejecuta, a lo más, N-1 veces y la construcción del conjunto NUEVO en la línea 6 necesita de p comparaciones. El ciclo de la línea 8 se ejecuta N-1 veces a lo más y la construcción del conjunto V_{Ne} en la línea 10 requiere de p comparaciones. El ciclo de la línea 13 se ejecuta p veces a lo más, y la construcción del conjunto NUEVO dentro de este ciclo requiere de 2^n-1 pasos, donde n es el número máximo de símbolos nulos en el lado derecho de las producciones. Por lo tanto, el algoritmo trabaja en un tiempo de $O(2^n p)$.

En la línea 2 obtenemos todos los símbolos no terminales que derivan directamente a la cadena vacía. Al salir de la iteración de la línea 4 hemos guardado en NUEVO todos los símbolos no terminales que derivan cadenas que comienzan con λ. Al salir de la iteración de la línea 8 hemos guardado en V_{Ne} los símbolos no terminales de NUEVO que derivan a la cadena vacía, es decir, no tomamos los símbolos que deriven cadenas cuyo primer símbolo es la cadena vacía y los símbolos que le siguen no la deriven.

Ejemplo

Consideremos la siguiente gramática:

$S \rightarrow aAbC$ $\quad$ $B \rightarrow A$
$A \rightarrow \lambda$ $\quad$ $B \rightarrow bSBA$
$A \rightarrow aB$ $\quad$ $C \rightarrow aab$

En esta gramática tenemos la producción vacia $A \rightarrow \lambda$ y como $B \Rightarrow A \Rightarrow \lambda$, entonces $B \Rightarrow^* \lambda$. Por lo tanto $V_{Ne} = \{A,B\}$.

Ahora debemos generar las producciones que compensan la eliminación de la producción $A \rightarrow \lambda$.

A partir de la producción $S \rightarrow aAbC$ generamos la nueva producción $S \rightarrow abC$ ya que $A \in V_{Ne}$.

A partir de la producción $A \rightarrow aB$ generamos la nueva producción $A \rightarrow a$ ya que $B \in V_{Ne}$.

A partir de la producción $B \rightarrow bSBA$ generamos las nuevas producciones $B \rightarrow bS$, $B \rightarrow bSA$ y $B \rightarrow bSB$ ya que $A,B \in V_{Ne}$.

Obteniendo así la nueva gramática:

$S \rightarrow aAbC$ $\quad$ $B \rightarrow bSBA$
$S \rightarrow abC$ $\quad$ $B \rightarrow bS$
$A \rightarrow aB$ $\quad$ $B \rightarrow bSA$
$A \rightarrow a$ $\quad$ $B \rightarrow bSB$
$B \rightarrow A$ $\quad$ $C \rightarrow aab$

Que resulta ser equivalente a la original y que no tiene producciones vacias.

Eliminación de producciones unitarias y ciclos.

Definición Las producciones de la forma $A \rightarrow B$ donde el lado derecho de la producción consiste de un sólo símbolo no terminal se llaman producciones unitarias.

Definición Decimos que una gramática tiene ciclos, si existen derivaciones de la forma $A \Rightarrow^+ A$.

Teorema 3 Todo lenguaje libre de contexto que no tenga a la cadena λ está definido por una gramática sin símbolo inútiles, sin producciones vacias y sin producciones unitarias.

Demostración Sea L un lenguaje libre de contexto que no tiene a la cadena λ y $L=L(G)$ para alguna $G=(V_N,V_T,P,S)$. Asumimos que G no tiene producciones vacias. Se construye un nuevo conjunto de producciones P' de P incluyendo primero a todas las producciones no unitarias de P. Supongamos que $A \Rightarrow^* B$ en G, para A y B en V_N. Agregamos a P' todas las producciones de la forma $A \rightarrow \alpha$ donde $B \rightarrow \alpha$ es una producción no unitaria de P.

Obsérvese que podemos fácilmente checar cuándo $A \Rightarrow^* B$ en G, ya que G no tiene producciones vacías y si

$$A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \ldots \Rightarrow B_m \Rightarrow B \text{ en } G$$

y algún símbolo no terminal aparece dos veces en la secuencia, podemos encontrar una secuencia más corta de producciones unitarias que nos lleve a $A \Rightarrow^* B$. Así que es suficiente con considerar únicamente aquellas secuencias de producciones unitarias que no repitan ninguno de los símbolos no terminales de G.

Tenemos ahora una nueva gramática $G'=(V_N,V_T,P',S)$. Ciertamente, si $A \rightarrow \alpha$ es una producción de P', entonces $A \Rightarrow^* \alpha$ en G. Así que, si hay alguna derivación de w en G', entonces hay una derivación de w en G.

Supongamos que w está en L(G) y consideremos la derivación izquierda de w en G. Sea $S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n = w$ esta derivación.

Si, para $0 \leq i < n$, $\alpha_i \Rightarrow \alpha_{i+1}$ en G por una producción no unitaria, entonces $\alpha_i \Rightarrow \alpha_{i+1}$ en G'.

Supongamos que $\alpha_i \Rightarrow \alpha_{i+1}$ en G por una producción unitaria, pero que $\alpha_{i-1} \Rightarrow \alpha_i$ por una producción no unitaria, o i=0. También supongamos que $\alpha_{i+1} \Rightarrow \alpha_{i+2} \Rightarrow \ldots \Rightarrow \alpha_j$ en G, todas por producciones unitarias, y $\alpha_j \Rightarrow \alpha_{j+1}$ en G por una producción no unitaria. Entonces $\alpha_i, \alpha_{i+1}, \ldots, \alpha_j$ son todas de la misma longitud, y como la derivación es izquierda, el símbolo reemplazado en cada una de éstas debe estar en la misma posición. Pero entonces $\alpha_i \Rightarrow \alpha_{j+1}$ en G por alguna de las producciones de P'-P. Por lo tanto L(G')=L(G).

Para completar la demostración observemos que G' no tiene producciones unitarias ni producciones vacías. Si aplicamos el Teorema 1 para eliminar los símbolos inútiles (con lo cual no estaremos agregando nuevas producciones) a la gramática G', obtenemos una gramática que satisface el teorema. ■

Algoritmo 4 Elimina producciones unitarias y ciclos.

Entrada: Una gramática sin producciones vacías.

Salida: Una gramática equivalente sin producciones vacías, sin producciones unitarias y sin ciclos.

Método:

1) /* Para cada símbolo no terminal A calculamos el conjunto V_A */

$V_A := \{B \mid A \Rightarrow^* B \text{ y } B \in V_N\}$;

2) /* Eliminamos de P las producciones unitarias */

$P := P - \{A \rightarrow B \mid B \in V_N\};$

3) /* Construimos las producciones que compensan la eliminación*/

$P' := \{A \rightarrow \alpha \mid B \rightarrow \alpha \in P \;\; y \;\; B \in V_A\};$

4) /* Construimos el conjunto P' de la gramática resultante */

$P' := P' \cup P;$

Podemos garantizar que este algoritmo termina en un número finito de pasos, ya que el número de símbolos no terminales N y el número de producciones p son finitos. La construcción de los conjuntos V_A se reduce a la multiplicación de matrices dada por el algoritmo de Warshall (ver apéndice I). Por lo tanto el algoritmo trabaja en un tiempo de $O(N^3)$.

Observemos que una gramática sin producciones unitarias y sin producciones vacías es una gramática en la que podemos garantizar que no hay ciclos. Por lo tanto la gramática resultante de aplicar este algoritmo es una gramática sin producciones vacías, sin producciones unitarias y sin ciclos.

Ejemplo

Consideremos la siguiente gramática:

$S \rightarrow abA \qquad B \rightarrow bS$
$A \rightarrow B \qquad B \rightarrow aa$

En esta gramática tenemos la producción unitaria $A \rightarrow B$ que debe ser eliminada. El conjunto V_A resulta ser $\{B\}$. Como las producciones de B son $B \rightarrow bS$ y $B \rightarrow aa$, las nuevas producciones que debemos generar son $A \rightarrow bS$ y $A \rightarrow aa$. Obteniendo así la nueva gramática:

$S \rightarrow abA \qquad B \rightarrow bS$
$A \rightarrow bS \qquad B \rightarrow aa$
$A \rightarrow aa$

Que resulta ser equivalente a la original y que no tiene producciones unitarias.

Con los cuatro algoritmos expuestos hasta ahora podemos, dada una gramática libre de contexto $G=(V_N,V_T,P,S)$, encontrar una gramática equivalente $G'=(V_N',V_T',P',S)$ sin símbolos inútiles, sin producciones vacías y sin producciones unitarias. Cuando una gramática satisface estas propiedades decimos que la gramática ha sido reducida.

El orden en el que debemos aplicar estos algoritmos es importante. Primero aplicamos el algoritmo 3 para obtener una gramática sin producciones vacías, que será la gramática de entrada para el algoritmo 4 que elimina las producciones unitarias. Ahora aplicamos a esta gramática sin producciones vacías ni unitarias el algoritmo 1 que elimina los símbolos inactivos para finalmente aplicar el algoritmo 2 que elimina los símbolos inalcanzables y obtener así una gramática reducida.

Recursividad izquierda

Definición Una gramática es recursiva por la izquierda si tiene un símbolo no terminal A tal que hay una derivación $A \Rightarrow^{+} A\alpha$ para alguna cadena α.

Para poder determinar si una gramática es recursiva por la izquierda o no, utilizaremos la relación **F** que definimos a continuación:

Como en el algoritmo 3, denotaremos al conjunto de símbolos nulos por V_{Ne} donde $V_{Ne} = \{A \in V_N | A \Rightarrow^{*} \lambda\}$.

Dada una producción $X \rightarrow Y_1Y_2Y_3...Y_n$, con $X \in V_N$ y $Y_i \in (V_N \cup V_T)$ tenemos que:

X **F** Y_1 y si $Y_1 \in V_{Ne}$, entonces también

X **F** Y_2 y si $Y_1, Y_2 \in V_{Ne}$, entonces también

X **F** Y_3, etc.

Algoritmo 5 Determina si una gramática es recursiva por la izquierda.

Entrada: Una gramática libre de contexto.

Salida: El diagnóstico de recursividad por la izquierda.

Método:

1) Construir la matriz de relación **F** de la gramática.

2) Con el algoritmo de Warshall, obtener la cerradura transitiva de **F**, que denotamos por F^{+} (ver apéndice I).

3) Si existe algún símbolo no terminal **A**, para el cual (A,A)=1 en F^{+}, entonces la gramática es recursiva por la izquierda. Si para todo símbolo no terminal **A**, (A,A)=0 en F^{+}, entonces la gramática no es recursiva por la izquierda.

Podemos garantizar que este algoritmo termina, ya que el número de símbolos no terminales N y el número de símbolos del lado derecho de las producciones son finitos. El cálculo de la cerradura transitiva de la relación F se reduce a la multiplicación de matrices dada por el algoritmo de Warshall (ver apéndice I), por lo tanto el algoritmo trabaja en un tiempo de $O(N^3)$.

Los métodos de reconocimiento de "arriba hacia abajo" no pueden manejar gramáticas que sean recursivas por la izquierda, así que es necesaria una transformación que elimine dicha recursividad.

Formas Normales

Forma Normal de Chomsky

Definición Decimos que una gramática libre de contexto está en Forma Normal de Chomsky si todas sus producciones son de la forma $A \to BC$ ó $A \to a$, donde A,B y C son símbolos no terminales y a es un símbolo terminal.

Teorema 4 Cualquier lenguaje libre de contexto sin la cadena vacía es generado por una gramática en Forma Normal de Chomsky.

<u>Demostración</u> Sea G una gramática libre de contexto que genera un lenguaje que no contiene a λ. Por el Teorema 3, podemos encontrar una gramática equivalente $G_1=(V_N,V_T,P,S)$, tal que P no contiene ni producciones unitarias ni producciones vacías. Así que, si una producción contiene un único símbolo del lado derecho, este símbolo es terminal, y la producción se encuentra ya en la forma requerida.

Consideremos ahora una producción de P, de la forma $A \to X_1X_2...X_m$, donde $m\geq 2$. Si X_i es un símbolo terminal **a**, introducimos un nuevo símbolo no terminal C_a y una producción $C_a \to a$, que está en la forma requerida. Ahora reemplazamos X_i por C_a. Sea el nuevo conjunto de símbolo no terminales V_N' y el nuevo conjunto de producciones P'. Consideremos la gramática $G_2=(V_N',V_T,P'S)$. Si $\alpha \Rightarrow \beta$ en G_1, entonces $\alpha \Rightarrow^* \beta$ en G_2. Así que $L(G_1) \subseteq L(G_2)$. Ahora probaremos por inducción sobre el número de pasos en una derivación que si $A \Rightarrow^* w$ en G_2 para A en V_N y w en V_T^*, entonces $A \Rightarrow^* w$ en G_1. El resultado es trivial para las derivaciones de un paso. Supongamos que es cierto para derivaciones de más de k pasos. Sea $A \Rightarrow^* w$ en G_2 una derivación de (k+1) pasos. El primer paso debe ser de la forma $A \to B_1B_2..B_m$ $m\geq 2$. Podemos escribir $w=w_1w_2...w_m$, donde $B_i \Rightarrow^* w_i$, $1\leq i\leq m$.

Si B_i es C_{ai} para algún símbolo terminal a_i, entonces w_i debe ser a_i. Por la construcción de P', hay una producción $A \to X_1X_2...X_m$ de P donde $X_i=B_i$ si B_i está en V_N y $X_i=a_i$ si B_i está en $V_N'-V_N$. Para aquellas B_i en V_N, sabemos que la derivación $B_i \Rightarrow^* w_i$ en G_1 tiene no más de k pasos, entonces por la hipótesis de inducción, $X_i \Rightarrow^* w_i$ en G_1. Por lo tanto $A \Rightarrow^* w$ en G_1.

Hasta aquí hemos probado el resultado intermedio de que cualquier lenguaje libre de contexto puede ser generado por una gramática en la cual toda producción es de la forma $A \rightarrow a$ ó $A \rightarrow B_1B_2...B_m$ para $m \geq 2$, donde $A, B_1, B_2, ..., B_m$ son símbolos no terminales, y a es un símbolo terminal.

Consideremos una de estas gramáticas $G_2=(V_N', V_N, P', S)$. Modificamos G_2 agregando algunos nuevos símbolos a V_N' y reemplazando algunas de las producciones de P'. Para cada produción $A \rightarrow B_1B_2...B_m$ de P', donde $m \geq 3$, creamos nuevos símbolos no terminales $D_1, D_2, ..., D_{m-2}$ y reemplazamos $A \rightarrow B_1B_2...B_m$ por el conjunto de producciones

$$\{A \rightarrow B_1D_1,\ D_1 \rightarrow B_2D_2, ...,\ D_{m-3} \rightarrow B_{m-2}D_{m-2},\ D_{m-2} \rightarrow B_{m-1}B_m\}.$$

Sea V_N'' el nuevo conjunto de símbolos no terminales y P'' el nuevo conjunto de producciones. Sea $G_3=(V_N'', V_T, P'', S)$. G_3 está en Forma Normal de Chomsky. Es claro que si $A \Rightarrow^* \beta$ en G_2, entonces $A \Rightarrow^* \beta$ en G_3, así que $L(G_2) \subseteq L(G_3)$. También es cierto que si $A \Rightarrow^* \beta$ en G_3, entonces $A \Rightarrow^* \beta$ en G_2, así que $L(G_3) \subseteq L(G_2)$. Entonces $L(G_3)=L(G_2)$ y por lo tanto $L(G_3)=L(G_1)$. ■

Algoritmo 6 Transforma una gramática a Forma Normal de Chomsky.

Entrada: Una gramática libre de contexto sin producciones vacías ni unitarias.

Salida: Una gramática equivalente en Forma Normal de Chomsky.

Método:

/* Para toda producción $A \rightarrow X_1X_2...X_n$ con $n \geq 2$, si $X_i \in V_T$ */

1. Agregamos al conjunto de símbolos no terminales un nuevo símbolo α_i.
2. Agregamos al conjunto de producciones la nueva producción $\alpha_i \rightarrow X_i$
3. En la producción original $A \rightarrow X_1X_2...X_n$ reemplazamos X_i por α_i.

/* Para toda producción $A \rightarrow X_1X_2...X_n$ con $n \geq 3$ */

1. Agregamos al conjunto de símbolos no terminales los nuevos símbolos $\alpha_1, \alpha_2, ..., \alpha_m$ donde $m=n-2$.
2. Agregamos al conjunto de producciones la nueva producción $A \rightarrow X_1\alpha_1$

3. FOR i=1 to m-1 do
   begin
   agregamos al conjunto de producciones la nueva producción $\alpha_i \rightarrow X_{i+1}\alpha_{i+1}$
   end;

4. Agregamos al conjunto de producciones la nueva producción $\alpha_m \rightarrow X_{n-1}X_n$

5. Eliminamos del conjunto de producciones la producción original $A \rightarrow X_1X_2...X_n$

Podemos garantizar que este algoritmo termina, ya que el número de producciones p y n, el número máximo de símbolos que aparecen del lado derecho de las producciones son finitos. Este algoritmo trabaja en un tiempo de $O(n^2p)$.

Ejemplo

Consideremos la siguiente gramática:

$S \rightarrow bA$	$A \rightarrow a$
$S \rightarrow aB$	$B \rightarrow aBB$
$A \rightarrow bAA$	$B \rightarrow bS$
$A \rightarrow aS$	$B \rightarrow b$

Las producciones con un sólo símbolo del lado derecho están ya en la forma deseada, ya que estamos considerando una gramática sin ciclos. En este caso son las producciones $A \rightarrow a$ y $B \rightarrow b$.

Tomamos las producciones que tienen más de un símbolo del lado derecho. La producción $S \rightarrow bA$ se reemplaza por la producción $S \rightarrow \alpha_1A$ y agregamos $\alpha_1 \rightarrow b$. La producción $S \rightarrow aB$ se reemplaza por la producción $S \rightarrow \alpha_2B$ y agregamos $\alpha_2 \rightarrow a$. La producción $A \rightarrow bAA$ se reemplaza por la producción $A \rightarrow \alpha_3AA$ y agregamos $\alpha_3 \rightarrow b$. La producción $A \rightarrow aS$ se reemplaza por la producción $A \rightarrow \alpha_4S$ y agregamos $\alpha_4 \rightarrow a$. La producción $B \rightarrow aBB$ se reemplaza por la producción $B \rightarrow \alpha_5BB$ y agregamos $\alpha_5 \rightarrow a$. La producción $B \rightarrow bS$ se reemplaza por la producción $B \rightarrow \alpha_6S$ y agregamos $\alpha_6 \rightarrow b$. Obteniendo así la siguiente gramática:

$S \rightarrow \alpha_1A$	$B \rightarrow b$
$S \rightarrow \alpha_2B$	$\alpha_1 \rightarrow b$
$A \rightarrow \alpha_3AA$	$\alpha_2 \rightarrow a$
$A \rightarrow \alpha_4S$	$\alpha_3 \rightarrow b$
$A \rightarrow a$	$\alpha_4 \rightarrow a$
$B \rightarrow \alpha_5BB$	$\alpha_5 \rightarrow a$
$B \rightarrow \alpha_6S$	$\alpha_6 \rightarrow b$

Ahora tomamos las producciones que tienen más de dos símbolos del lado derecho. La producción $A \rightarrow \alpha_3AA$ se reemplaza por la producción $A \rightarrow \alpha_3\alpha_7$ y agregamos $\alpha_7 \rightarrow AA$. La producción $B \rightarrow \alpha_5BB$ se reemplaza por la producción $B \rightarrow \alpha_5\alpha_8$ y agregamos $\alpha_8 \rightarrow BB$. Obteniendo así la siguiente gramática:

$S \rightarrow \alpha_1A$	$\alpha_1 \rightarrow b$
$S \rightarrow \alpha_2B$	$\alpha_2 \rightarrow a$
$A \rightarrow \alpha_3\alpha_7$	$\alpha_3 \rightarrow b$
$A \rightarrow \alpha_4S$	$\alpha_4 \rightarrow a$
$A \rightarrow a$	$\alpha_5 \rightarrow a$
$B \rightarrow \alpha_5\alpha_8$	$\alpha_6 \rightarrow b$
$B \rightarrow \alpha_6S$	$\alpha_7 \rightarrow AA$
$B \rightarrow b$	$\alpha_8 \rightarrow BB$

Que resulta ser equivalente a la original y que está en Forma Normal de Chomsky.

Forma Normal de Greibach

Definición Decimos que una gramática libre de contexto está en Forma Normal de Greibach si todas sus producciones son de la forma $A \rightarrow a\alpha$ donde $a \in V_T$ y $\alpha \in V_N^*$.

Lema 3 Sea $G=(V_N,V_T,P,S)$ una gramática libre de contexto. Sea $A \rightarrow \alpha_1B\alpha_2$ una producción en P y $B \rightarrow \beta_1 | \beta_2 | \ldots | \beta_r$ el conjunto de todas las producciones de B. Sea $G_1=(V_N,V_T,P_1,S)$ la gramática que se obtiene de G eliminando la producción $A \rightarrow \alpha_1B\alpha_2$ de P y agregando las producciones $A \rightarrow \alpha_1\beta_1\alpha_2 | \alpha_1\beta_2\alpha_2 | \ldots | \alpha_1\beta_r\alpha_2$. Entonces $L(G)=L(G_1)$.

<u>Demostración</u> Si la producción $A \rightarrow \alpha_1\beta_i\alpha_2$ es usada en alguna derivación de G_1, entonces $A \Rightarrow \alpha_1B\alpha_2 \Rightarrow \alpha_1\beta_i\alpha_2$ puede ser usada en G. Por lo tanto $L(G_1) \subseteq L(G)$.

Como $A \rightarrow \alpha_1B\alpha_2$ es la única producción de G que no está en G_1, cada vez que ésta es usada en una derivación de G, el símbolo no terminal **B** debe ser reescrito en algún paso más adelante usando una producción de la forma $B \rightarrow \beta_i$. Estos dos pasos pueden ser reemplazados por el único paso $A \Rightarrow \alpha_1\beta_i\alpha_2$ en G_1. ∎

Lema 4 Sea $G=(V_N,V_T,P,S)$ una gramática libre de contexto. Sea $A \rightarrow A\alpha_1 | A\alpha_1 | \ldots | A\alpha_r$ el conjunto de todas las producciones de A en las que el símbolo más a la izquierda del lado derecho de la producción es **A**, es decir, las producciones de **A** que son recursivas por la izquierda. Sean $A \rightarrow \beta_1 | \beta_2 | \ldots | \beta_s$ el resto de las producciones de **A**. Sea $G_1=(V_N \cup \{B\},V_T,P_1,S)$ la gramática libre de

contexto que se forma al agregar el símbolo no terminal **B** a V_N y construir el nuevo conjunto de producciones P_1 como sigue:

1. Eliminar de P todas las producciones de **A**.
2. $P_1 = P$.
3. Agregar a P_1 las producciones:

$$A \rightarrow \beta_1 B \mid \beta_2 B \mid \ldots \mid \beta_s B$$

$$B \rightarrow \alpha_1 B \mid \alpha_2 B \mid \ldots \mid \alpha_r B$$

$$B \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_r$$

Entonces $L(G_1)=L(G)$.

<u>Demostración</u> En una derivación izquierda, una secuencia de producciones de la forma $A \rightarrow A\alpha_i$ debe eventualmente terminar con una producción $A \rightarrow \beta_j$. La secuencia de reemplazos

$$A \rightarrow A\alpha_{i1} \rightarrow A\alpha_{i2}\alpha_{i1} \rightarrow \ldots \rightarrow A\alpha_{ip}\alpha_{ip-1}\ldots\alpha_{i1}$$

$$\rightarrow \beta_j\alpha_{ip}\alpha_{ip-1}\ldots\alpha_{i1}$$

en G, puede ser reemplazada en G_1 por

$$A \rightarrow \beta_j B \rightarrow \beta_j\alpha_{ip}B \rightarrow \beta_j\alpha_{ip}\alpha_{ip-1}B$$

$$\rightarrow \ldots \rightarrow \beta_j\alpha_{ip}\alpha_{ip-1}\ldots\alpha_{i2}B$$

$$\rightarrow \beta_j\alpha_{ip}\alpha_{ip-1}\ldots\alpha_{i1}.$$

El inverso de esta transformación también puede hacerse, por lo tanto $L(G)=L(G_1)$. ■

Teorema 5 Cualquier lenguaje libre de contexto sin la cadena vacía es generado por una gramática en Forma Normal de Greibach.

<u>Demostración</u> Sea $G=(V_N,V_T,P,S)$ una gramática en Forma Normal de Chomsky que genera el lenguaje libre de contexto **L**. Le damos algún orden al conjunto de símbolos no terminales, $V_N=(A_1,A_2,\ldots,A_m)$. El primer paso en la construcción es modificar las producciones de tal forma que si $A_i \rightarrow A_j\alpha$ es una producción, entonces $j>i$. Comenzando con A_1 y prosiguiendo hasta A_m, lo hacemos de la siguiente forma. Asumimos que las producciones han sido modificadas de tal forma que para $1 \le i < k$, $A_i \rightarrow A_j\alpha$ es una producción sólo si $j>i$. Ahora modificamos las producciones de A_k.

Si $A_k \rightarrow A_j\alpha$ es una producción con $j<k$, generamos un nuevo conjunto de producciones sustituyendo el lado derecho de cada producción de A_j de acuerdo con el Lema 3. Repitiendo este proceso k-1 veces cuando más, obtenemos producciones de la forma $A_k \rightarrow A_r\alpha$, $r \geq k$. Las producciones con $r=k$ son reemplazadas de acuerdo con el Lema 4, introduciendo un nuevo símbolo no terminal B_k.

Repitiendo este proceso para cada símbolo no terminal original, obtenemos únicamente producciones en alguna de las siguientes formas:

1) $A_i \rightarrow A_j\alpha$ con $j>i$
2) $A_i \rightarrow a\alpha$ con a en V_T
3) $B_i \rightarrow \alpha$ con α en $(V_N \cup \{B_1, B_2, ..., B_{i-1}\})^*$.

El símbolo más a la izquierda en el lado derecho de cualquier producción de A_m debe ser un terminal, ya que A_m es el símbolo no terminal "mayor". El símbolo más a la izquierda en el lado derecho de cualquier producción de A_{m-1} debe ser A_m ó un símbolo terminal.

Cuando es A_n podemos generar nuevas producciones reemplazando A_m por el lado derecho de las producciones de A_m de acuerdo al Lema 1. En estas nuevas producciones el símbolo más a la izquierda del lado derecho de la producción es un terminal. Aplicamos este proceso a las producciones de los símbolos no terminales $A_{m-2}, ...,$ A_2, A_1 hasta que el lado derecho de cada produción de toda A_i comience con un símbolo terminal.

Finalmente examinamos las producciones de los nuevos símbolos no terminales $B_1, B_2, ..., B_m$. Como empezamos con una gramática en Forma Normal de Chomsky, se puede probar fácilmente por inducción sobre el número de aplicaciones de los Lemas 3 y 4 que el lado derecho de toda producción de A_i para $1 \leq i \leq n$, comienza con un símbolo terminal o con A_jA_k para algunas j y k, por lo tanto ninguna producción de B_i puede comenzar con otra B_j. Así que, en todas las producciones de B_i el símbolo más a la izquierda del lado derecho es un terminal o una A_i. Aplicando una vez más el Lema 3 para cada producción de B_i completamos la construcción. ■

Algoritmo 7 Transforma una gramática a Forma Normal de Greibach.

Entrada: Una gramática libre de contexto en Forma Normal de Chomsky.

Salida: Una gramática equivalente en Forma Normal de Greibach.

Método:

/* Para todo símbolo no terminal hacemos que todas sus producciones comiencen con un terminal o con un no terminal "mayor" que él */

1) Sea $V_N = \{A_1, A_2, \ldots, A_m\}$

2) Para k desde 1 hasta m
begin
3) para j desde 1 hasta k-1
para cada producción de la forma $A_k \to A_j\alpha$
begin
para todas las producciones $A_j \to \beta$
agrega la producción $A_k \to \beta\alpha$
elimina la producción $A_k \to A_j\alpha$
end;

4) para cada producción de la forma $A_k \to A_k\alpha$
begin
agrega las producciones $B_k \to \alpha$ y $B_k \to \alpha B_k$
elimina la producción $A_k \to A_k\alpha$
end;

5) para cada producción $A_k \to \beta$, donde β no empieza A_k
agrega la producción $A_k \to \beta B_k$
end;

/* Hacemos que todas las producciones de los símbolos no terminales $A_1, A_2, \ldots, A_m$ comiencen con un simbolo terminal */

6) Para k desde m-1 hasta 1
para cada producción de la forma $A_k \to A_j\alpha$, $k<j\leq m$
begin
para todas las producciones $A_j \to \beta$
agrega la producción $A_k \to \beta\alpha$
elimina la producción $A_k \to A_j\alpha$
end;

/* Hacemos que las producciones de los símbolo no terminales agregados $B_1, B_2, \ldots, B_m$ comiencen con un simbolo terminal */

7) Para k desde 1 hasta m
para cada producción de la forma $B_k \to A_j\alpha$, $1\leq j\leq m$
begin
para todas las producciones $A_j \to \beta$
agrega la producción $B_k \to \beta\alpha$
elimina la producción $B_k \to A_j\alpha$
end;

Podemos garantizar que este algoritmo termina, ya que el número de producciones p y el número de símbolos no terminales m son finitos. La iteración de la línea 3 se ejecuta $m(m+1)/2$ veces, es decir en un tiempo de $O(m^2)$, así que la iteración de la línea 2 se ejecuta en un tiempo de $O(m^3)$. Las iteraciones de las líneas 6 y 7 se ejecutan $m(m+1)/2$ veces, toamndo un tiempo de $O(m^2)$ cada una. Por lo tanto el algoritmo trabaja en un tiempo de $O(m^3)$.

Observemos que en una gramática en Forma Normal de Greibach, eliminamos la posibilidad de que ésta sea recursiva por la izquierda, ya que todas las producciones de la gramática comienzan con un símbolo terminal.

Ejemplo

Consideremos la siguiente gramática:

$S \rightarrow AB$ $\quad B \rightarrow SA$
$A \rightarrow BS$ $\quad B \rightarrow a$
$A \rightarrow b$

y pensemos en los símbolo no terminales ordenados {S,A,B}.

Paso 1 Tomamos las producciones que comiencen con símbolos no terminales "menores" que el símbolo del lado izquierdo de la producción. En este caso la producción $B \rightarrow SA$. Como $S \rightarrow AB$, agregamos la producción $B \rightarrow ABA$ y eliminamos $B \rightarrow SA$. Volvemos a recorrer las producciones y tomamos otra vez las que comiencen con símbolos no terminales "menores", ahora $B \rightarrow ABA$. Como las producciones de **A** son $A \rightarrow BS$ y $A \rightarrow b$, agregamos las producciones $B \rightarrow BSBA$ y $B \rightarrow bBA$ y eliminamos la producción $B \rightarrow ABA$. De esta forma obtenemos la siguiente gramática:

$S \rightarrow AB$ $\quad B \rightarrow BSBA$
$A \rightarrow BS$ $\quad B \rightarrow bBA$
$A \rightarrow b$ $\quad B \rightarrow a$

Tomamos ahora las producciones que sean recursivas por la izquierda, en este caso $B \rightarrow BSBA$ y agregamos un nuevo símbolo no terminal B_1. Como $B \rightarrow bBA$ y $B \rightarrow a$ son el resto de las producciones de **B**, agregamos las producciones $B \rightarrow bBAB_1$ y $B \rightarrow aB_1$ para **B**; así como $B_1 \rightarrow SBA$ y $B_1 \rightarrow SBAB_1$ para B_1. Así obtenemos la siguiente gramática:

$S \rightarrow AB$ $\quad B \rightarrow bBAB_1$
$A \rightarrow BS$ $\quad B \rightarrow aB_1$
$A \rightarrow b$ $\quad B_1 \rightarrow SBA$
$B \rightarrow bBA$ $\quad B_1 \rightarrow SBAB_1$
$B \rightarrow a$

donde toda producción de los símbolos no terminales originales comienza con un símbolo terminal o con un no terminal que es "mayor" que el del lado izquierdo de la producción.

<u>Paso 2</u> Ahora debemos hacer que todas las producciones de los símbolos no terminales originales comiencen con un símbolo terminal. Como en este momento todas las producciones del "último" símbolo no terminal comienzan con un terminal, vamos recorriendo los símbolos no terminales desde el penúltimo hasta el primero, agregando y eliminando las producciones necesarias.

Para el símbolo no terminal **A**, tomamos la producción $A \rightarrow BS$ que comienza con un no terminal. Como las producciones de **B** son $B \rightarrow bBA$, $B \rightarrow a$, $B \rightarrow bBAB_1$ y $B \rightarrow aB_1$ agregamos las producciones $A \rightarrow bBAA$, $A \rightarrow aS$, $A \rightarrow bBAB_1S$ y $A \rightarrow aB_1S$ y eliminamos la producción $A \rightarrow BS$.

Ahora para el símbolo no terminal **S**, tomamos la producción $S \rightarrow AB$. Como las producciones de **A** son $A \rightarrow bBAA$, $A \rightarrow aS$, $A \rightarrow bBAB_1S$, $A \rightarrow aB_1S$ y $A \rightarrow b$, agregamos las producciones $S \rightarrow bBAAB$, $S \rightarrow aSB$, $S \rightarrow bBAB_1SB$, $S \rightarrow aB_1SB$ y $S \rightarrow bB$ y eliminamos la producción $S \rightarrow AB$. De esta forma obtenemos la siguiente gramática:

$S \rightarrow bBAAB$	$A \rightarrow b$
$S \rightarrow aSB$	$A \rightarrow aB_1S$
$S \rightarrow bBAB_1SB$	$B \rightarrow bBA$
$S \rightarrow aB_1SB$	$B \rightarrow a$
$S \rightarrow bB$	$B \rightarrow bBAB_1$
$A \rightarrow bBAA$	$B \rightarrow aB_1$
$A \rightarrow aS$	$B_1 \rightarrow SBA$
$A \rightarrow bBAB_1S$	$B_1 \rightarrow SBAB_1$

donde todas las producciones de los símbolos no terminales originales comienzan con un símbolo terminal.

<u>Paso 3</u> Ahora revisamos las producciones de los símbolos no terminales agregados, y hacemos que todas ellas comiencen con un símbolo terminal. En este caso tenemos las producciones $B_1 \rightarrow SBA$ y $B_1 \rightarrow SBAB_1$. Como las producciones de **S** son $S \rightarrow bBAAB$, $S \rightarrow aSB$, $S \rightarrow bBAB_1SB$, $S \rightarrow aB_1SB$ y $S \rightarrow bB$, primero agregamos la nuevas producciones $B_1 \rightarrow bBAABBA$, $B_1 \rightarrow aSBBA$, $B_1 \rightarrow bBAB_1SBBA$, $B_1 \rightarrow aB_1SBBA$ y $B_1 \rightarrow bBBA$ y eliminamos la producción $B_1 \rightarrow SBA$. Ahora agregamos las producciones $B_1 \rightarrow bBAABBAB_1$, $B_1 \rightarrow aSBBAB_1$, $B_1 \rightarrow bBAB_1SBBAB_1$, $B_1 \rightarrow aB_1SBBAB_1$ y $B_1 \rightarrow bBBAB_1$ y eliminamos la producción $B_1 \rightarrow SBAB_1$. De esta forma obtenemos la siguiente gramática:

$S \rightarrow bBAAB$	$B \rightarrow bBAB_1$
$S \rightarrow aSB$	$B \rightarrow aB_1$
$S \rightarrow bBAB_1SB$	$B_1 \rightarrow bBAABBA$
$S \rightarrow aB_1SB$	$B_1 \rightarrow aSBBA$
$S \rightarrow bB$	$B_1 \rightarrow bBABB_1SBBA$
$A \rightarrow bBAA$	$B_1 \rightarrow aB_1SBBA$
$A \rightarrow aS$	$B_1 \rightarrow bBBA$
$A \rightarrow bBAB_1S$	$B_1 \rightarrow bBAABBAB_1$
$A \rightarrow aB_1S$	$B_1 \rightarrow aSBBAB_1$
$A \rightarrow b$	$B_1 \rightarrow bBAB_1SBBAB_1$
$B \rightarrow bBA$	$B_1 \rightarrow aB_1SBBAB_1$
$B \rightarrow a$	$B_1 \rightarrow bBBAB_1$

Que resulta ser equivalente a la original y que está en Forma Normal de Greibach.

Los algoritmos 5 y 6 nos permiten, dada una gramática reducida que tenga recursividad por la izquierda, encontrar una gramática equivalente en donde esta recursividad ha sido eliminada. Primero aplicamos el algoritmo 5 con el cual transformamos la gramática a Forma Normal de Chomsky. Esta gramática será la entrada del algoritmo 6 con el cual transformamos la gramática a Forma Normal de Greibach y de esta forma obtenemos una gramática equivalente sin recursividad por la izquierda.

Factorización izquierda.

Factorizar por la izquierda una gramática es una transformación de una gramática que es útil para producir una gramática a partir de la cual sea posible la construcción automática de un analizador sintáctico no recursivo por predicción.

La idea básica es que cuando no es claro cuál de dos o más producciones alternativas utilizar para expander a algún símbolo no terminal **A**, podamos reescribir las producciones de **A** de tal forma que la decisión de cuál de las producciones debemos expander, para que ésta sea la elección correcta, se haga hasta que hayamos visto suficientes símbolos de la cadena de entrada.

Algoritmo 8 Factoriza por la izquierda una gramática.

<u>Entrada</u>: Una gramática libre de contexto reducida y sin recursividad por la izquierda.

<u>Salida</u>: Una gramática equivalente factorizada por la izquierda.

Método:

Para cada símbolo no terminal **A**, encontrar todas las producciones que comiencen con el mismo símbolo **X**. Si **X** $\neq \lambda$, reemplazar todas las producciones:

$$A \rightarrow X\alpha_1 \mid X\alpha_2 \mid \ldots \mid X\alpha_n \mid \beta$$

donde β representa todas las alternativas que no comienzan con **X** por:

$$A \rightarrow XA_1 \mid \beta$$

$$A_1 \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

donde A_1 es un nuevo símbolo no terminal.

Aplicar este proceso repetidamente hasta que no haya dos producciones de un símbolo no terminal que comiencen con el mismo símbolo.

Podemos garantizar que este algoritmo termina, ya que el número de producciones p y q, el número máximo de símbolos del lado derecho de las producciones, son finitos. El algoritmo trabaja en un tiempo de O(pq).

Observemos que al aplicar este algoritmo, pudiera ser que la gramática resultante tenga producciones vacias.

Ejemplo

Consideremos la siguiente gramática:

$S \rightarrow AB$	$A \rightarrow BS$
$A \rightarrow aB$	$B \rightarrow bba$
$A \rightarrow a$	$B \rightarrow ba$

Primero tomamos las producciones de **A** que comienzan con el mismo símbolo, éstas son $A \rightarrow aB$ y $A \rightarrow a$. Agregamos las producciones $A \rightarrow aA_1$, $A_1 \rightarrow B$ y $A_1 \rightarrow \lambda$; y eliminamos las producciones $A \rightarrow aB$ y $A \rightarrow a$.

Ahora tomamos las producciones de B que comienzan con el mismo símbolo, éstas son $B \rightarrow bba$ y $B \rightarrow ba$. Agregamos las producciones $B \rightarrow bB_1$, $B_1 \rightarrow ba$ y $B_1 \rightarrow a$; y eliminamos las producciones $B \rightarrow bba$ y $B \rightarrow ba$.

De esta forma obtenemos la siguiente gramática:

$S \rightarrow AB$	$A_1 \rightarrow B$
$A \rightarrow aA_1$	$A_1 \rightarrow \lambda$
$A \rightarrow BS$	$B_1 \rightarrow ba$
$B \rightarrow bB_1$	$B_1 \rightarrow a$

que resulta ser equivalente a la original y donde no hay dos producciones de un mismo símbolo no terminal que comiencen con el mismo símbolo, es decir, está factorizada por la izquierda.

La aplicación, en el orden adecuado, de los algoritmos expuestos en este capítulo, nos permite dada una gramática libre de contexto cualquiera, obtener una gramática equivalente que satisface las propiedades necesarias para que pueda ser construido automáticamente un analizador sintáctico no recursivo por predicción del lenguaje que genera dicha gramática, siempre y cuando ésta sea LL(1).

V. CONSTRUCCION DE UN RECONOCEDOR NO RECURSIVO POR PREDICCION

Gramáticas LL(1)

Como mencionamos en el capítulo III, la construcción de un reconocedor no recursivo por predicción es posible siempre y cuando las gramáticas sean LL(1). En este capítulo definiremos con precisión este tipo de gramáticas.

Las gramáticas LL(1) han sido utilizadas para definir lenguajes de programación en, aproximadamente, los últimos quince años. El algoritmo de reconocimiento para esta clase de gramáticas resulta ser muy eficiente y su instrumentación resulta ser una función lineal de la longitud de la cadena de entrada. Asi mismo, la detección de errores y las técnicas de recuperación pueden ser fácilmente añadidas al algoritmo de reconocimiento para esta clase de gramáticas.

La primera **L** de LL(1) se refiere a que la lectura de la cadena de entrada a ser reconocida se hace de izquierda a derecha (Left to right scan). La segunda **L** indica que el árbol de reconocimiento de construye con la derivación izquierda de la cadena (Leftmost derivation). Finalmente el 1 significa que en el reconocimiento se lee, de la cadena de entrada, un sólo símbolo a la vez.

Definiremos primero el tipo más sencillo de gramáticas LL(1) que son las gramáticas LL(1) simples.

Definición 1 Una <u>gramática LL(1) simple</u> es una gramática libre de contexto sin producciones vacías, tal que para toda $\mathbf{A} \in V_N$, las producciones alternativas de **A** comienzan cada una con un símbolo terminal distinto, es decir todas sus producciones son de la forma:

$$A \rightarrow a_1\alpha_1 \mid a_2\alpha_2 \mid \ldots \mid a_n\alpha_n \quad \text{donde}$$

$$a_i \neq a_j \quad \text{para} \quad i \neq j \quad y \; a_i \in V_T \quad \text{para } 1 \leq i \leq n.$$

Para poder definir un tipo más general de gramáticas LL(1), en donde eliminemos la restricción anterior para la forma de las producciones, necesitamos definir el conjunto FIRST de una cadena.

Definición 2 Dada una cadena $\alpha \in (V_N \cup V_T)^*$, el conjunto de símbolos terminales que son derivables a la izquierda desde α está dado por:

$$\text{FIRST}(\alpha) = \{w \mid \alpha \rightarrow^* w\ldots \quad y \; w \in V_T\}.$$

Definición 3 Una gramática libre de contexto sin producciones vacías es una <u>gramática LL(1)</u> si para todas las producciones de la forma:

$$A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$$

los conjunto FIRST(α_1), FIRST(α_2),..., FIRST(α_n) son ajenos dos a dos, es decir

$$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi \quad \text{para } i \neq j.$$

Definiremos ahora el tipo más general de gramáticas LL(1), en las cuales la restricción de que no tenga producciones vacías es eliminada. Para ello necesitamos generalizar la definición del conjunto FIRST e introducir una nueva definición, la del conjunto FOLLOW.

Definición 4 El conjunto FIRST de una cadena α se define como: $FIRST(\alpha) = \{w | \alpha \Rightarrow^* w\ldots \ , |w| \leq 1 \ \text{ y } \ w \in V_T^*\}$.

Definición 5 Se define el conjunto FOLLOW(A), para un símbolo no terminal **A**, como el conjunto de símbolos terminales **a** que pueden aparecer inmediatamente a la derecha de **A** en alguna forma sentencial, esto es, el conjunto de símbolos terminales **a** tales que existe una derivación de la forma:

$$S \Rightarrow^* \alpha A a \beta \qquad \text{para cualesquiera } \alpha \text{ y } \beta.$$

Observemos que, en algún paso de la derivación, pudiera ser que hubiera símbolos entre **A** y **a**, en cuyo caso tales símbolos derivarían a la cadena vacía y desaparecerían.

Definición 6 Una gramática libre de contexto G es una <u>gramática LL(1)</u> si y sólo si para cada par de producciones $A \rightarrow \alpha$ y $A \rightarrow \beta$, de algún símbolo no terminal **A** se cumple que:

$$FIRST(\alpha \circ FOLLOW(A)) \cap FIRST(\beta \circ FOLLOW(A)) = \phi$$

Una formulación equivalente a la anterior es:

Para todas las producciones $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$

1. $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi$ para toda $i \neq j$

   y si $\alpha_i \Rightarrow^* \lambda$, entonces

2. $FIRST(\alpha_j) \cap FOLLOW(A) = \phi$ para $j \neq i$

Para cada uno de estos tres tipos de gramáticas LL(1) existe un algoritmo para la construcción del reconocedor sintáctico. En el presente trabajo, nos centraremos en la construcción del reconocedor para las gramáticas LL(1) de la definición 6, por ser éstas las del tipo más general, y por lo tanto nos permite el reconocimiento de un mayor número de lenguajes.

Aspectos Generales

Como mencionamos en el capitulo III, para la construcción de un reconocedor no recursivo por predicción, utilizamos un autómata de stack, el cual reconoce a las cadenas por stack vacio.

Este autómata tiene un stack y un alfabeto de stack formado por la unión del alfabeto de entrada, es decir los símbolos terminales de la gramática, y el conjunto de símbolos no terminales de la gramática. El comportamiento del autómata en cada momento está definido por una tabla de acción que se define para cada pareja:

(símbolo de entrada,símbolo en el tope del stack)

y las posibles acciones del autómata son:

1. Reemplazar el símbolo en el tope del stack por los símbolos del lado derecho de una cierta producción.

2. Sacar el símbolo del tope del stack y leer el siguiente símbolo de entrada.

3. Detectar un error en la cadena de entrada.

La segunda acción únicamente ocurre cuando en el tope del stack hay un símbolo terminal, que casa con el símbolo de entrada. Así, la tabla de acción se define únicamente para símbolos no terminales en el tope del stack.

La tercera acción puede ocurrir en dos casos: el primero de ellos es cuando el símbolo de entrada no casa con el símbolo terminal en el tope del stack, y el segundo cuando la tabla indica que no hay producción del símbolo no terminal en el tope del stack que sea consistente con el símbolo de entrada.

Si la cadena de entrada se termina y el stack está vacio, entonces la cadena es aceptada por el autómata, es decir, la cadena pertenece al lenguaje generado por la gramática.

Construcción de la tabla de acción del autómata

Definimos una función de reconocimiento para la cual, dado el símbolo en el tope del stack y el siguiente símbolo de la cadena de entrada, regresa ya sea la producción a aplicar o una indicación de cómo continuar o terminar. Esta función está dada por:

$$M: (V_N \cup V_T \cup \{\#\}) \times (V_T \cup \{\#\}) \rightarrow \{(\beta, i),\ \text{sacar},\ \text{aceptar},\ \text{error}\}$$

donde # marca el fondo del stack y el final de la cadena de entrada, (β, i) es un par ordenado tal que β es el lado derecho de la producción i.

Si **A** es el símbolo en el tope del stack y **a** es el símbolo leído de la cadena de entrada, entonces **M** se define como:

$$M(A,a) = \begin{cases} \text{sacar} & \text{si } A=a \quad \text{para} \quad a \in V_T \\ \text{aceptar} & \text{si } A=\# \quad y \quad a=\# \\ (\alpha, i) & \text{si } a \in \text{FIRST}(\alpha) \quad y \\ & A \rightarrow \alpha \quad \text{es la i-ésima producción} \\ (\alpha, i) & \text{si } a \in \text{FOLLOW}(A) \quad y \\ & A \rightarrow \alpha \quad \text{es la i-ésima producción y} \\ & \lambda \in \text{FIRST}(\alpha),\ \text{i.e. } A \in V_{N\epsilon} \\ \text{error} & \text{en cualquier otro caso} \end{cases}$$

El corazón del algoritmo de reconocimiento es esta función **M**, la cual representamos en una tabla. Cuando al construir esta tabla tenemos que alguna de sus entradas está multidefinida, entonces la gramática para la cual estamos construyendo la tabla no es LL(1), así que, para que una gramática sea LL(1) debe no ser recursiva por la izquierda y no ser ambigua.

Construcción de los conjuntos FIRST y FOLLOW

Para poder obtener la tabla de acción del autómata para una cierta gramática LL(1), es necesario calcular los conjunto FIRST y FOLLOW, definidos en el capítulo V.

Conjunto FIRST

El conjunto FIRST de una cadena α se define como:

$$\text{FIRST}(\alpha) = \{w \mid \alpha \rightarrow^* w\ldots \quad \text{donde} \quad |w| \leq 1 \quad y \quad w \in V_T^*\}$$

Para poder calcular de manera sencilla este conjunto, redefiniremos la relación FIRST en términos de otra relación, la relación **F** que definimos para el algoritmo 5 del capítulo IV.

Construida la matriz F, utilizamos el algoritmo de Warshall para obtener la cerradura transitiva de **F** donotada por $\mathbf{F}^+$ (ver apéndice I).

Para calcular el conjunto FIRST de una cierta cadena α, debemos primero obtener los conjuntos FIRST de cada uno de los símbolos de α. La construcción del conjunto FIRST de un símbolo terminal es trivial,

$$\text{para cada } a \in V_T \quad \text{FIRST}(a) = \{a\}.$$

El conjunto FIRST de un símbolo no terminal puede ser obtenido directamente de la matriz $\mathbf{F}^+$ de la siguiente forma:

para cada $\mathbf{A} \in V_N$ $\quad \text{FIRST}(A) = \{x \mid x \in V_T \text{ y } A\ \mathbf{F}^+\ x \text{ ó } x = \lambda \text{ y } A \in V_{Ne}\}.$

Sea $\alpha \in (V_N \cup V_T)^*$ y $\alpha = X_1X_2...X_n$, entonces

$\text{FIRST}(\alpha) = \text{FIRST}(X_1)$ y si $X_1 \in V_{Ne}$, entonces

$\text{FIRST}(\alpha) = \text{FIRST}(X_1) \cup \text{FIRST}(X_2)$ y si $X_1, X_2 \in V_{Ne}$, entonces

$\text{FIRST}(\alpha) = \text{FIRST}(X_1) \cup \text{FIRST}(X_2) \cup \text{FIRST}(X_3)$, etc.

Si todas las $X_i \in V_{Ne}$, entonces $\text{FIRST}(\alpha)$ también contiene a λ.

Conjunto Follow

El conjunto FOLLOW de un símbolo no terminal **A** se define como:

$\text{FOLLOW}(A) = \{a \mid a \in V_T \text{ y } S \rightarrow^* \alpha A a \beta$ para cualesquiera α y β y donde S es el símbolo inicial$\}$.

Otra forma de definir a este conjunto es:

$\text{FOLLOW}(A) = \{a \mid a \in V_T \text{ y } S \rightarrow^* \alpha A \beta \text{ con } a \in \text{FIRST}(\beta)\}.$

Igual que para el conjunto FIRST, redefiniremos a la relación FOLLOW en términos de otras tres relaciones, de tal forma que calcular este conjunto resulte más sencillo.

En primer lugar necesitaremos la cerradura transitiva reflexiva de $\mathbf{F}^+$, que denotaremos por $\mathbf{F}^*$. $\mathbf{F}^* = \mathbf{I} \wedge \mathbf{F}^+$, donde **I** es la matriz identidad.

Ahora definimos la nueva relación **B** de la siguiente forma: Dada una producción

$$X \rightarrow Y_1Y_2...Y_n \ , \ 1 \le i \le n$$

sea Y_i **B** Y_{i+1} y si $Y_{i+1} \in V_{Ne}$, entonces también Y_i **B** Y_{i+2}, etc.

Finalmente definimos la relación **L**. Dada una producción

$$X \rightarrow Y_1Y_2...Y_n \ , \ X \in V_N \ \ y \ \ Y_i \in (V_N \cup V_T)$$

sea Y_n **L** X y si $Y_n \in V_{Ne}$, entonces también

Y_{n-1} **L** X y si $Y_n, Y_{n-1} \in V_{Ne}$, entonces también

Y_{n-2} **L** X, etc.

Las cerraduras transitiva y transitiva reflexiva de **L**, denotadas por L^+ y L^*, se calculan de la misma forma que las de **F**.

Tenemos entonces que, para cada símbolo no terminal **A**,

$$\text{FOLLOW}(A) = \{a | a \in V_T \ \ y \ \ A \ (L^*BF^*) \ a\}.$$

Comportamiento del autómata

Definimos la forma general de la configuración del autómata en cada paso del reconocimiento como (**az**,**A**α,P), donde **az** es la subcadena aún no reconocida de la cadena de entrada, **A**α es el contenido del stack con **A** en el tope, y P es el contenido de la cinta de salida, en la que se registran las producciones aplicadas en el reconocimiento.

Utilizamos el valor de M(**A**,**a**), donde **a** es el símbolo que estamos leyendo de la cadena de entrada, para determinar la siguiente configuración del autómata de acuerdo con la relación de producción ($\vdash$) dada por:

$$(az, A\alpha, P) \vdash \begin{cases} (z, \alpha, P) & \text{si } M(A,a) = \text{sacar} \\ \text{Termina exitosamente} & \text{si } M(A,a) = \text{aceptar} \\ (az, \beta\alpha, P_i) & \text{si } M(A,a) = (\beta, i) \\ \text{Termina por error} & \text{si } M(A,a) = \text{error} \end{cases}$$

La configuración inicial del autómata es (**z**#,**S**#,λ), donde **z** es la cadena a ser reconocida y **S** es el símbolo inicial de la gramática.

Algoritmo de reconocimiento

El siguiente algoritmo permite, dada la tabla de acción **M** del autómata, para una gramática LL(1) particular, analizar cadenas y determinar si éstas pertenecen o no al lenguaje generado por la gramática.

Sean :

CAD = cadena de entrada.

p = apuntador al símbolo analizado de la cadena.

PILA = contenido del stack, donde el símbolo más a la izquierda de PILA es el tope del stack. Esta pila representa a la forma sentencial en cada instante de tiempo.

t = símbolo en el tope del stack. Este símbolo es el que está más a la izquierda en la forma sentencial en cada instante de tiempo.

c = símbolo analizado.

PROD = lista de producciones aplicadas.

SI = símbolo inicial.

Algoritmo 9 Analiza cadenas, intentando su reconocimiento.

Entrada: Cadena a ser reconocida.

Salida: Producciones aplicadas en el reconocimiento y el mensaje de si la cadena fue reconocida exitosamente o no.

Método:

```
begin
      /* inicializa */

       CAD := CAD ∘ '#';
         p := 1;
      PILA := SI ∘ '#';
      PROD := ' ';
```

```
      /* reconocimiento */

       mientras el simbolo analizado sea distinto de '≑'
       begin
             t := copy(PILA,1,1);
             c := copy(CAD,p,1);
             case M(t,c) of

                    (β,i) : begin
                                  PILA := β ∘ copy(PILA,2);
                                  PROD := PROD ∘ ' ' ∘ i;
                            end;

                    SACAR : begin
                                  PILA := copy(PILA,2);
                                  p := p+1;
                            end;

                    ACEPTAR : reporta('Reconocimiento exitoso');

                    ERROR : begin
                                 reporta('Falló reconocimiento');
                                 EXIT;
                            end;

             end; /* del case */
       end; /* de mientras */
end.
```

Ejemplo

Consideremos la siguiente gramática LL(1):

1. S → iBba
2. B → AB
3. B → λ
4. A → caCd
5. A → fi
6. C → aC
7. C → λ

Para poder construir la tabla de acción del autómata debemos primero calcular los conjuntos FIRST de los lados derechos de las producciones, y los conjuntos FOLLOW de los símbolos no terminales.

FIRST(iBba) = {i}
FIRST(AB) = {c,f}
FIRST(λ) = {λ}
FIRST(caCd) = {c}
FIRST(fi) = {f}
FIRST(aC) = {a}

FOLLOW(B) = {b}
FOLLOW(A) = {B}
FOLLOW(C) = {d}

La tabla de acción del autómata para ésta gramática es:

Símbolo Tope del stack	Símbolo actual de la cadena de entrada						
	i	b	a	c	d	f	#
S	(iBba,1)						
B		(λ,3)		(AB,2)		(AB,2)	
A				(caCd,4)		(fi,5)	
C			(aC,6)		(λ,7)		
i	sacar						
b		sacar					
a			sacar				
c				sacar			
d					sacar		
f						sacar	
#							aceptar

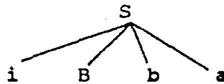
Las entradas en blanco, son entradas de error.

El comportamiento del autómata en el reconocimiento de la cadena 'icadba' es el siguiente:

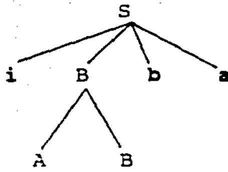
	(icadba# , S# , ' ')
aplica producción 1	(icadba# , iBba# , 1)
sacar	(cadba# , Bba# , 1)
aplica producción 2	(cadba# , ABba# , 1 2)
aplica producción 4	(cadba# , caCdBba# , 1 2 4)
sacar	(adba# , aCdBba# , 1 2 4)
sacar	(dba# , CdBba# , 1 2 4)
aplica producción 7	(dba# , dBba# , 1 2 4 7)
sacar	(ba# , Bba# , 1 2 4 7)
aplica producción 3	(ba# , ba# , 1 2 4 7 3)
sacar	(a# , a# , 1 2 4 7 3)
sacar	(# , # , 1 2 4 7 3)
aceptar	('Reconocimiento exitoso')

Así que las producciones aplicadas en el reconocimiento de la cadena 'icadba' fueron 1,2,4,7,3. Aplicando estas producciones, podemos construir la derivación izquierda del árbol de reconocimiento de la cadena.

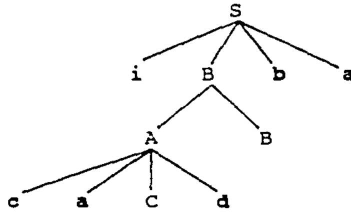
S → iBba



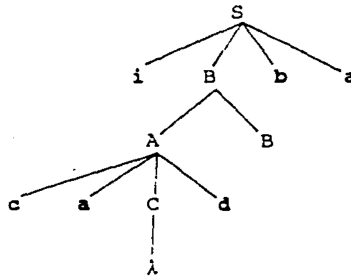
S → iBba → iABba



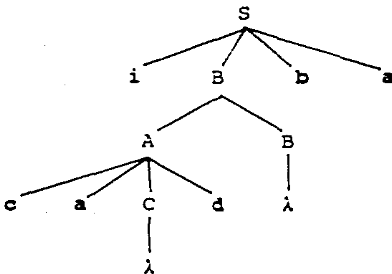
S → iBba → iABba → icaCdBba



S → iBba → iABba → icaCdBba → icadBba



$S \to iBba \to iABba \to icaCdBba \to icadBba \to icadba$



VI. INSTRUMENTACION DEL SISTEMA

Generalidades

El sistema se compone de dos programas. El primero de ellos toma como entrada un archivo con una gramática libre de contexto y hace las transformaciones necesarias para que sea posible la construcción automática del analizador sintáctico del lenguaje que genera dicha gramática. Este primer programa genera archivos que contienen las gramáticas resultantes de cada paso de la transformación y que pueden ser mostrados en pantalla, solicitándolos desde un menú.

El segundo programa toma como entrada la gramática transformada que genera el primero, y trata de construir la tabla de acción del autómata para esta gramática. Si la tabla está multidefinida, el programa reporta que la gramática no es LL(1) y que por lo tanto no es posible la construcción automática del analizador sintáctico no recursivo por predicción. Si la construcción de la tabla es posible, el programa pide la cadena a ser analizada y procede a hacer su reconocimiento. Si la cadena pertenece al lenguaje da el mensaje de que el reconocimiento fue exitoso y cuáles fueron las producciones aplicadas. Si la cadena no pertenece al lenguaje da el mensaje correspondiente, reporta cuál fue la subcadena reconocida y cuáles las producciones aplicadas.

A continuación expondremos, para cada uno de los dos programas, las estructuras de datos y el método utilizados. Los programas no manipulan directamente a los símbolos de la gramática, sino que trabajan con números enteros asociados a ellos.

PROGRAMA 1 : Transformación de gramáticas.

OBJETIVO

Transformar una gramática libre de contexto, para poder construir el analizador sintáctico del lenguaje que genera dicha gramática.

ENTRADA

Un archivo de texto con la lista de símbolos terminales y la gramática propuesta. El símbolo inicial de la gramática.

SALIDA

1. El archivo *.org (el * representa el nombre del archivo de entrada), con la gramática original.

2. El archivo *.vac con la gramática sin producciones vacias.

3. El archivo *.unt con la gramática sin producciones unitarias.

4. El archivo *.utl con la gramática sin símbolos inútiles.

5. Cuando la gramática es recursiva por la izquierda:

   5.1 El archivo *.fnc con la gramática en Forma Normal de Chomsky.

   5.2 El archivo *.fng con la gramática en Forma Normal de Greibach.

6. El archivo *.fiz con la gramática factorizada por la izquierda.

7. Los archivos archnt.dat, archterm.dat y archprod.dat con los datos de las tablas de símbolos no terminales, símbolos terminales y producciones respectivamente, de la gramática transformada en el último paso, es decir, de la gramática que está en *.fiz.

ESTRUCTURAS DE DATOS

La representación y manipulación de los símbolos y producciones de la gramática se hizo a través de las siguientes estructuras de datos.

Símbolos no terminales

La información concerniente a los símbolos no terminales se almacena en una tabla. Dicha tabla es un arreglo de registros, un registro para cada símbolo. Cada registro está formado por los siguientes campos:

simb. Un caracter, que es el símbolo.

cod. El entero que identificará a dicho símbolo. A cada símbolo no terminal se le asigna un entero negativo.

lugini. El entero que apunta al lugar de la tabla de producciones, en el que comienzan las producciones del símbolo.

numprod. Un entero que es el número de producciones que tiene el símbolo.

nulo. Un entero que es 1 si el símbolo es nulo, 0 en otro caso.

activo. Un entero que es 1 si el símbolo es activo, 0 si es inactivo.

alcanza. Un entero que es 1 si el símbolo es alcanzable, 0 si el símbolo es inalcanzable.

aparecio. Un entero que es 1 si el símbolo aparece del lado izquierdo de alguna producción.

sig. Apuntador a una lista ligada de arreglos de enteros, para agregar producciones.

Símbolos terminales

La información relacionada con los símbolos terminales se almacena en una tabla que es un arreglo de registros, un registro para cada símbolo. Cada registro está formado por los siguientes campos:

simb. Un caracter, que es el símbolo.

cod. El entero que identificará a dicho símbolo. A cada símbolo terminal se le asigna un entero positivo.

alcanza. Un entero que es 1 si el símbolo es alcanzable, 0 si el símbolo es inalcanzable.

Producciones

Las producciones son almacenadas en una tabla que es un arreglo de dos dimensiones de enteros. Cada renglón corresponde al lado derecho de una producción y son los códigos, de los símbolos que la forman, los que están en la tabla. La primera columna de la tabla se usa como control para determinar si la producción aún pertenece a la gramática, en cuyo caso tendrá en ese lugar un 1, o si ha sido eliminada y en ese caso tendrá un 0.

Gramáticas

El programa puede desplegar en la pantalla a las gramáticas resultantes en cada paso de la transformación. Para ello, a través de la información contenida en las distintas tablas, escribe en archivos de texto las producciones de la gramática, con sus símbolos originales.

Al final, el programa escribe la información contenida en cada una de las tablas en tres archivos de texto, uno para cada tabla. Estos archivos son los que lee el programa reconocedor para armar sus tablas.

METODO

1. Lee del archivo de entrada la lista de símbolos terminales y construye la tabla respectiva, llenando los campos simb y cod, e inicializa al campo alcanza con 0.

2. Lee el archivo donde está la gramática propuesta y construye las tablas de símbolos no terminales y de producciones, llenando los campos simb, cod, lugini, numprod, apareció e inicializando con 0 los campos nulo, activo y alcanza de la tabla de símbolos no terminales; y poniendo un 1 en la primera columna de cada renglón de la tabla de producciones. Considera a cualquier símbolo que está en la gramática y que no se le haya dado en la lista de terminales como no terminal.

3. Pide el símbolo inicial de la gramática, y checa que éste sea alguno de los símbolos no terminales.

4. Genera el archivo *.org con la gramática original en la presentación adecuada para el despliegue.

5. A través del Algoritmo 3 del capítulo IV,

   5.1 Determina el conjunto de símbolos no terminales nulos y pone en 1 en el campo nulo de dichos símbolos.

   5.2 Elimina las producciones vacías, poniendo un 0 en la primera columna de dichas producciones.

   5.3 Genera las nuevas producciones que compensan esta eliminación, agregándolas a la lista que cuelga de la tabla de símbolos no terminales.

6. Actualiza las tablas de símbolos no terminales y producciones y genera el archivo *.vac con esta nueva gramática.

7. A través del Algoritmo 4 del capítulo IV, elimina las producciones unitarias poniendo un cero en la primera columna de dichas producciones, compensa esta eliminación generando los nuevos símbolos no terminales y las nuevas producciones agregándolas a las listas ligadas.

8. Actualiza las tablas de símbolos no terminales y de producciones y genera el archivo *.unt con esta nueva gramática.

9. A través del Algoritmo 1 del capítulo IV, determina los símbolos activos y pone un 1 en el campo activo de dichos símbolos. Con el Algoritmo 2 del capítulo IV, determina los símbolos terminales y no terminales alcanzables y pone un 1 en el campo alcanza de dichos símbolos. Elimina las producciones que involucran a símbolos inútiles poniendo un 0 en la primera columna de dichas producciones.

10. Actualiza las tablas de símbolos no terminales y producciones y genera el archivo *.utl con esta nueva gramática.

11. A través del Algoritmo 5 del capítulo IV, checa si la gramática es recursiva por la izquierda. En caso de que sí lo sea:

> 11.1 Transforma la gramática a su Forma Norma de Chomsky, a través del Algoritmo 6 del capítulo IV. Elimina las producciones necesarias poniendo un cero en la primera columna, genera los nuevos símbolos no terminales y las nuevas producciones, agregándolas a las listas ligadas.
>
> 11.2 Actualiza las tablas de símbolos no terminales y producciones, y genera el archivo *.fnc con esta nueva gramática.
>
> 11.3 Transforma la gramática a su Forma Normal de Greibach a través del Algoritmo 6 del capítulo IV. Elimina las producciones necesarias poniendo un cero en la primera columna, genera los nuevos símbolos no terminales y las nuevas producciones, agregándolas a las listas ligadas.
>
> 11.4 Actualiza las tablas de símbolos no terminales y producciones, y genera el archivo *.fng con esta nueva gramática.

12. A través del Algoritmo 8 del capítulo IV, factoriza por la izquierda a la gramática, eliminando las producciones necesarias poniendo un cero en la primera columna, genera los nuevos símbolos no terminales y las nuevas producciones, agregándolas a las listas ligadas.

13. Actualiza las tablas de símbolos no terminales y de producciones, y genera el archivo *.fiz con esta nueva gramática.

14. Como pudiera resultar que esta nueva gramática tenga producciones vacías, calcula el conjunto de símbolos no terminales nulos con la primera parte del Algoritmo 3 del capítulo IV, y pone un uno en el campo nulo de dichos símbolos.

15. Genera los archivos ARCHNT.DAT, ARCHTERM.DAT y ARCHPROD.DAT con los datos contenidos en las tablas de símbolos no terminales, símbolos terminales y producciones respectivamente. Estos archivos son los que leerá el programa de reconocimiento.

16. Finalmente despliega el menú, a través del cual seleccionamos la gramática que queremos se despliegue en pantalla.

PROGRAMA 2 : Construcción automática del analizador sintáctico.

OBJETIVO

Construir, si es posible, la tabla de acción del autómata y hacer el reconocimiento de cadenas.

ENTRADA

Los archivos ARCHNT.DAT, ARCHTERM.DAT y ARCHPROD.DAT que genera el Programa 1. Y, si la construcción de la tabla de acción es posible, la cadena a ser reconocida.

SALIDA

1. Si la tabla de acción del autómata está multidefinida, el mensaje que reporta que la gramática no es LL(1) y que por lo tanto no es posible la construcción del analizador sintáctico, así como los símbolos para los cuales la tabla está multidefinida y por cuáles producciones.

2. Si es posible la construcción de la tabla de acción del autómata, la lista de producciones aplicadas en el reconocimiento de la cadena de entrada, así como el mensaje de si la cadena pertenece al lenguaje o no. Si la cadena no pertence al lenguaje, reporta cuál fue la subcadena reconocida.

ESTRUCTURAS DE DATOS

La representación y manipulación de los símbolos y producciones de la gramática se hizo a través de las siguientes estructuras de datos.

Símbolos no terminales

La información concerniente a los símbolos no terminales se almacena en una tabla. Dicha tabla es un arreglo de registros, un registro para cada símbolo. Cada registro está formado por los siguientes campos:

simb. Un caracter, que es el símbolo.

cod. El entero que identificará a dicho símbolo. A cada símbolo no terminal se le asigna un entero negativo.

lugini. El entero que apunta al lugar de la tabla de producciones, en el que comienzan las producciones del símbolo.

numprod. Un entero que es el número de producciones que tiene el símbolo.

nulo. Un entero que es 1 si el símbolo es nulo, 0 en otro caso.

Símbolos terminales

La información relacionada con los símbolos terminales se almacena en una tabla que es un arreglo de registros, un registro para cada símbolo. Cada registro está formado por los siguientes campos:

simb. Un caracter, que es el símbolo.

cod. El entero que identificará a dicho símbolo. A cada símbolo terminal se le asigna un entero positivo.

alcanza. Un entero que es 1 si el símbolo es alcanzable, 0 si el símbolo es inalcanzable.

Producciones

Igual que en el Programa 1.

Conjuntos FIRST

Para representar los conjuntos FIRST de las cadenas que son el lado derecho de las producciones, utilizamos un arreglo de dos dimensiones de enteros. Cada renglón corresponde a una producción y en las columnas metemos los códigos de los símbolos que forman el conjunto FIRST del lado derecho de dicha producción. La primera columna sirve de control para saber si la cadena vacía pertenece al conjunto, en cuyo caso habrá un 1 en ese lugar, si la cadena vacía no pertenece al conjunto habrá un 0.

Conjuntos FOLLOW

Para representar los conjuntos FOLLOW de los símbolos no terminales, utilizamos un arreglo de dos dimensiones de enteros. Cada renglón corresponde a un símbolo no terminal y en las columnas guardamos los códigos de los símbolos que forman el conjunto FOLLOW de dicho no terminal.

Tabla de acción del autómata

La tabla de acción del autómata es un arreglo de dos dimensiones de enteros. Cada renglón corresponde a un símbolo no terminal y cada columna a un símbolo terminal. Las entradas de la matriz son el entero que corresponde al número de producción a aplicar, o un 0 si esa entrada es de error.

Autómata

El stack del autómata se representa a través de un registro con dos campos, uno de los campos es un arreglo de enteros donde irán los códigos de los símbolos que entran al stack, y el otro es un entero que apunta al tope del stack.

La cadena a ser reconocida se guarda en un arreglo de enteros, donde están los códigos de los símbolos que la forman.

La lista de producciones aplicadas es un arreglo de enteros, donde van los números de las producciones aplicadas. Estos números son el renglón en la tabla de producciones que ocupa dicha producción.

METODO

1. Lee los archivos ARCHNT.DAT, ARCHTERM.DAT y ARCHPROD.DAT y construye las tablas de símbolos no terminales, símbolos terminales y producciones.

2. Construye la matriz de relación **F**, definida en el capítulo IV y, utilizando el agoritmo de Warshall (ver Apéndice I), calcula la matriz de relación **F***.

3. Utilizando el método descrito en el capítulo V, construye la tabla FIRST.

4. Construye las matrices de relación **B** y **L**, definidas en el capítulo V, y calcula **F*** y **L***.

5. Construye la tabla de acción del autómata, definida en el capítulo V. Si resulta ser que la tabla está multidefinida, reporta el mensaje correspondiente y termina. Si no, sigue con 6.

6. Solicita la cadena a ser reconocida y traduce los símbolos a sus códigos correspondientes.

7. A través del Algoritmo 9 del capítulo V, hace el análisis de la cadena y reporta el resultado del reconocimiento.

APENDICE I

En el este apéndice se presentan las definiciones y el algoritmo de Warshall, a los que se hace referencia en los capitulos IV, V y VI.

Definición 1 Sea **R** una relación de X a Y y **S** una relación de Y a Z. Entonces la relación $R \circ S$ se llama la <u>relación de composición</u> de R y S donde

$$R \circ S = \{(x,z) \mid x \in X,\ z \in Z \text{ y existe una } y \in Y \text{ tal que } (x,y) \in R \text{ y } (y,z) \in S\}$$

Sean **A** y **B** relaciones representadas por matrices de (n x m) y (m x r) respectivamente; donde la entrada (X,Y) de la matriz es 1 si (X,Y) pertence a la relación y 0 en otro caso. Entonces podemos expresar la composición $A \circ B$ a través de una matriz **C** donde cada elemento de **C** se define como:

$$c_{ij} = \bigvee_{k=1}^{m} a_{ik} \wedge b_{kj} \qquad i=1,2,\ldots,n\ ;\ j=1,2,\ldots,r$$

$a_{ik} \wedge b_{kj}$ indica la conjunción, es decir,

$$1 \wedge 0 = 0 \wedge 1 = 0 \wedge 0 = 0 \quad \text{y} \quad 1 \wedge 1 = 1.$$

$\bigvee_{k=1}^{m}$ indica la disyunción, es decir,

$$1 \vee 1 = 1 \vee 0 = 0 \vee 1 = 1 \quad \text{y} \quad 0 \vee 0 = 0.$$

Definición 2 Sea **X** un conjunto finito y **R** una relación en **X**. Denotamos la composición de una relación consigo misma como:

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

La relación $R^* = R \cup R^2 \cup R^3 \cup \ldots$ en **X** se llama la <u>cerradura transitiva</u> de **R** en **X**.

Algoritmo de Warshall Calcula la cerradura transitiva de una relación.

ENTRADA : Una matriz de relación **A** con **n** columnas.

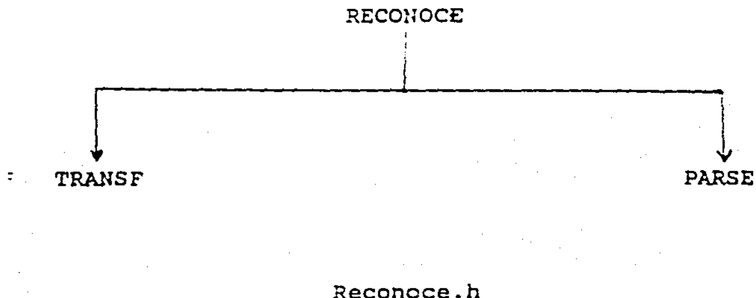
SALIDA : Una matriz de relación **P**, que es la cerradura transitiva de **A**.

METODO :

```
begin
      P := A;
      para k desde 1 hasta n
           para i desde 1 hasta n
                para j desde 1 hasta n
                     p_ij := p_ij ∨ (p_ik ∧ p_kj)
end;
```

APENDICE II

En el este apéndice se presentan los archivos RECONOCE.C, TRANSF.C y PARSE.C que son los programas fuente del sistema, así como el archivo RECONOCE.H que contiene las definiciones utilizadas por los programas TRANSF y PARSE.



Reconoce.h

El programa RECONOCE despliega la presentación y el menú principal del sistema. Dicho menú presenta las opciones:

Transformación de Gramáticas

Análisis Sintáctico.

Si se selecciona **Transformación de Gramáticas**, el programa TRANSF es invocado y ejecutado.

Si se selecciona **Análisis Sintáctico**, se invoca y ejecuta el programa PARSE.

RECONOCE.H es el archivo que contiene las definiciones de los programas TRANSF y PARSE, este archivo se incluye en ambos programas.

El programa TRANSF corresponde al PROGRAMA 1 : Tansformación de gramáticas.

El programa PARSE corresponde al PROGRAMA 2 : Construcción automática del analizador sintáctico.

Las estructuras de datos y el método utilizados en ambos programas están descritos en detalle en el capitulo VI. Instrumentación del Sistema.

```
/* RECONOCE.C

   Programa que despliega el menú principal y desde el cual son
   llamados el programa transf.exe que transforma gramáticas, y
   el programa parse.exe que hace el reconocimiento sintáctico */

# include <stdio.h>
# include <conio.h>
# include <string.h>

# define TRUE 1
# define FALSE 0

main()
{
  int control;
            presenta();
            borra();
            do{
                switch (elige()) {
                                  case 1 : /* invocación a transf */
                                           borra();
                                           system("transf.exe");
                                           control = TRUE;
                                           break;

                                  case 2 : /* invocación a parse */
                                           system("parse.exe");
                                           control = TRUE;
                                           break;

                                   case 3 : control = FALSE;
                                 }
                } while(control);
} /* de main */

int elige() /* elige la opción */
{
   char opcion;

      pantalla();
      do{
          opcion = getch();
       } while (strchr("TtAaSs",opcion) == NULL);
      switch(opcion) {
                       case 'T':
                       case 't': return 1;

                       case 'A':
                       case 'a': return 2;

                       case 'S':
                       case 's': return 3;
                     }
} /* de elige */
```

```
pantalla() /* despliega el menú en pantalla */
{
  int i;

      clrscr();
      gotoxy(20,5);
      printf("M E N U     P R I N C I P A L");
      gotoxy(15,7);
      printf("/");
      for (i=1 ; i<38 ; i++)
          printf("-");
      printf("\");
      gotoxy(15,8);
      printf("|                                      |\n");
      printf("               |   [T] Transformación de Gramáticas. |");
      gotoxy(15,10);
      printf("|                                      |\n");
      printf("               |   [A] Análisis Sintáctico.          |");
      gotoxy(15,12);
      printf("|                                      |\n");
      printf("               |   [S] Salir.                        |");
      gotoxy(15,14);
      printf("|                                      |\n");
      gotoxy(15,15);
      printf("\");
      for (i=1 ; i<38 ; i++)
           printf("-");
      printf("/");
      gotoxy(40,20);
      printf("Elige la opción ");
} /* de pantalla */


borra()
{
   if ( (fopen("archterm.dat","r") ) != NULL)
        system("del archterm.dat");

   if ( (fopen("archnt.dat","r") ) != NULL)
        system("del archnt.dat");

   if ( (fopen("archprod.dat","r") ) != NULL)
        system("del archprod.dat");
}/* de borra */


presenta()
{
  int i;

      clrscr();
      gotoxy(7,1);
      printf("/");
      for (i=1 ; i<72 ; i++)
          printf("-");
```

```
     printf("\");
     gotoxy(7,2);
     puts("|                                                                      |");
     gotoxy(7,3);
     puts("|  H E R R A M I E N T A S   P A R A   L A   C O N S T R U C C I O N  |");
     gotoxy(7,4);
     puts("|                                                                      |");
     gotoxy(7,5);
     puts("|                 D E    C O M P I L A D O R E S.                      |");
     gotoxy(7,6);
     puts("|                                                                      |");
     gotoxy(7,7);
     puts("|                                                                      |");
     gotoxy(7,8);
     puts("|                 TRANSFORMACION   DE   GRAMATICAS.                    |");
     gotoxy(7,9);
     puts("|                                                                      |");
     gotoxy(7,10);
     puts("|                   RECONOCIMIENTO    SINTACTICO.                      |");
     gotoxy(7,11);
     puts("|                                                                      |");
     gotoxy(7,12);
     printf("\");
     for (i=1 ; i<72 ; i++)
          printf("-");
     printf("/");
     gotoxy(20,14);
     printf("/");
     for (i=1 ; i<35 ; i++)
          printf("-");
     printf("\");
     gotoxy(20,15);
     printf("|                                  |\n");
     gotoxy(20,16);
     printf("|  Autor: Elke Capella Kort.       |\n");
     gotoxy(20,17);
     printf("|                                  |\n");
     gotoxy(20,18);
     printf("|          Facultad de Ciencias.   |\n");
     gotoxy(20,19);
     printf("|          U. N. A. M.             |\n");
     gotoxy(20,20);
     printf("|                                  |\n");
     gotoxy(20,21);
     printf("\");
     for (i=1 ; i<35 ; i++)
          printf("-");
     printf("/");
     gotoxy(40,25);
     printf("Oprime cualquier tecla para comenzar... ");
     getch();
} /* de presenta */
```

```
/* Reconoce.h

   Definiciones de los programas transf.c y parse.c */


# define TRUE 1
# define FALSE 0

# define MAXNT 60    /* número máximo de símbolos no terminales */
# define MAXT  20    /* número máximo de símbolos terminales */
# define MAXP  150   /* número máximo de producciones */
# define TOTS 80     /* número máximo de símbolos terminales y no terminales */
# define LPROD 21    /* longitud máxima de las producciones */
# define LCAD 50     /* longitud máxima de la cadena a ser reconocida */

# define TOPE(PILA)       pila.espacio[pila.apunt]
# define PUSH(PILA,t)     pila.espacio[++pila.apunt]=t
# define POP(PILA)        pila.apunt--;

typedef
struct pr    {
                int d[LPROD];
                struct pr *sig;
             } PROD, *PRODS;

typedef
struct note {
                char simb;
                int cod,lugini,numprod,aparecio,nulo,activo,alcanza;
                PRODS sig;
            } NOTERM, *NOTERMS;

typedef
struct tr {
              char simb;
              int cod,alcanza;
            } TERM, *TERMS;


typedef
struct st {
              int espacio[100];
              int apunt;
          }  STACK, *STACKS;
```

```
/* Programa 1 : TRANSF.C
   Transforma una gramática libre de contexto. */

# include <stdio.h>
# include <string.h>
# include <alloc.h>
# include <conio.h>
# include "reconoce.h"

/********** DECLARACIONES GLOBALES *************/

int nnt = 1;    /* lugar disponible en la tabla de no-terminales */
int nt = 1;     /* lugar disponible en la tabla de terminales */
int np = 1;     /* lugar disponible en la tabla de producciones */
int codini;     /* código del símbolo inicial de la gramática */
int nnto;       /* para ir marcando el número de no terminales agregados en cada paso*/
int NTO;        /* número de no terminales originales */
char let ;      /* símbolo de los nuevos símbolos que se generan */
int indice;     /* índice de los nuevos símbolos que se generan */
FILE *ent;      /* para el archivo con la gramática original */

char pref[8];  /* para los archivos que se generan */
char org[12];
char vac[12];
char unt[12];
char utl[12];
char fnc[12];
char fng[12];
char fiz[12];


/************  COMIENZA EL PROGRAMA PRINCIPAL ***************/

main()
{
   NOTERM tabnoterm[MAXNT];          /* tabla de símbolos no-terminales */
   TERM tabterm[MAXT];               /* tabla de símbolos terminales */
   int tabprod[MAXP][LPROD];         /* tabla de producciones */
   int tablaF[TOTS][TOTS];           /* tabla de relacion F */
   int grarec;                       /* booleana que sera verdadera si la gramática
                                                  es recursiva por la izquierda */
   char archivo[30];                 /* archivo con la gramatica de entrada */

/************  COMIENZA LA LLAMADA A SUBRUTINAS *************/

   if (!Arch_ent(archivo))                     /* lee el archivo de entrada */
     exit();

   Presenta();                                 /* despliega presentación */
   Inicializa (tabnoterm,tabprod);             /* inicializa */
   if (!Construye (tabterm,tabnoterm,tabprod,ent) )   /* construye tablas */
        exit();
   Da_símbolos(tabterm,tabnoterm);             /* despliega símbolos de la gramática */
   if (!Símbini (tabnoterm,tabprod))           /* lee el símbolo inicial */
        exit();
```

```
   Pinta_cuadro();
   if (!Procesa (tabnoterm,tabterm,tabprod))  /* procesa gramática */
      exit();

   grarec = Checrec(tablaF,tabnoterm,tabprod); /* checa si la gramática es recursiva */
   if (grarec)
       if (!Fnormales(tabnoterm,tabterm,tabprod))   /* transforma a formas normales */
           exit();

   if (!Factizq(tabnoterm,tabterm,tabprod))         /* factoriza por la izquierda y calcula nulos */
       exit();
   Archtablas(tabnoterm,tabterm,tabprod);     /* genera los archivos que lee el programa parse.c */
   if (Menu (grarec))
     exit();

}  /********** FIN DEL PROGRAMA PRINCIPAL *****************/


/**************DECLARACION DE SUBRUTINAS *******************/

int Arch_ent(char arch[]) /* lee el archivo de entrada */
{
  char opcion;
  int i,sigue;

    clrscr();
    gotoxy(8,4);
    printf("Dame el archivo con la gramática");
    do{
          gotoxy(8,5);
          for (i=1 ; i<30 ; i++)
               printf(" ");
          gotoxy(8,5);
          gets(arch);
          if ( (ent = fopen(arch,"r"))==NULL )
          {
                gotoxy(11,9);
                printf("No puedo leer del archivo\n");
                gotoxy(11,11);
                for (i=1 ; i<30 ; i++)
                    printf(" ");
                gotoxy(11,11);
                printf("%s",arch);
                gotoxy(20,19);
                printf("[L] Lee el archivo otra vez.");
                gotoxy(20,20);
                printf("[R] Regresa al Menú Principal.\n\n");
                gotoxy(42,23);
                printf("Selecciona la opcion");
                sigue = lee_opcion("LlRr",&opcion,'L','l');
          }
          else
          {
               i = 0;
               while ( (arch[i]) != '.') && ( (arch[i]) != '\0') && (i<8) )
```

ESTA TESIS NO DEBE
SALIR DE LA BIBLIOTECA

```
                pref[i] = arch[i++];
             pref[i] = '\0'
             return TRUE;
          }
      } while (sigue);
      if (!sigue)
         return FALSE;
} /* Arch_ent */


Presenta() /* despliega presentación del programa */
{
  int i;
  char opcion;

      clrscr();
      gotoxy(10,5);
      printf("/");
      for (i= 1 ; i<60 ; i++)
          printf("-");
      printf("\\n");
      printf("          |                                                       |\n");
      printf("          | T R A N S F O R M A C I O N   D E   G R A M A T I C A S |\n");
      printf("          |                                                       |\n");
      printf("          \");
      for (i= 1 ; i<60 ; i++)
          printf("-");
      printf("/\n\n\n");
      printf("            [D] Despliega los pasos del proceso de transformación.\n\n");
      printf("            [C] Comienza con el proceso de transformación.\n\n\n");
      gotoxy(42,23);
      printf("            Selecciona la opción  ");

      if (lee_opcion("DdCc",&opcion,'D','d'))
          Explica();
} /* de Presenta */


Explica() /* Despliega explicación del programa */
{
  int i;

     clrscr();
     printf("       /");
     for (i=1 ; i<60 ; i++)
         printf("-");
     printf("\\n");
     printf("       |                                                    |\n");
     printf("       |    PROCESO DE  TRANSFORMACION  DE  GRAMATICAS.     |\n");
     printf("       |                                                    |\n");
     printf("       |    1. Eliminar las producciones vacías.            |\n");
     printf("       |                                                    |\n");
     printf("       |    2. Eliminar las producciones unitarias.         |\n");
     printf("       |                                                    |\n");
     printf("       |    3. Eliminar los símbolos inactivos y los inalcanzables, |\n");
```

```
    printf("        |      i.e. los símbolos inútiles.                          |\n");
    printf("        |                                                           |\n");
    printf("        |   4. Checar si la gramática es recursiva por la izquierda.|\n");
    printf("        |      En caso de que lo sea:                               |\n");
    printf("        |                                                           |\n");
    printf("        |         a) Pasar la gramática a Forma Normal de Chomsky.  |\n");
    printf("        |         b) Pasar la gramática a Forma Normal de Greibach. |\n");
    printf("        |                                                           |\n");
    printf("        |   5. Factorizar por la izquierda las producciones.        |\n");
    printf("        |                                                           |\n");
    printf("        \");
    for (i=1 ; i<60 ; i++)
       printf("-");
    printf("/\n\n\n");
    printf("                   Oprime cualquier tecla para comenzar el proceso...");
    getch();
} /* de Explica */

Inicializa(NOTERM tnt[],int tp[MAXP][LPROD]) /* inicializa estructuras de datos */
{
  int i,j;

   for (i=0 ; i<MAXNT ; i++)
   {
      tnt[i].lugini = 0;
      tnt[i].numprod = 0;
      tnt[i].nulo = 0;
      tnt[i].activo = 0;
      tnt[i].alcanza = 0;
      tnt[i].sig = NULL;
   }
   for (i=0 ; i<MAXP ; i++)
       for (j=1 ; j<LPROD ; j++)
           tp[i][j] = 0;
   for (i=0 ; i<MAXP ; i++)
       tp[i][0] = 1;

   let = 224;
   indice = 1;
} /* de Inicializa */

int Construye(TERM tt[],NOTERM tnt[],int tp[MAXP][LPROD],FILE *e) /* construye las tablas */
{
        if (!consterm(e,tt))
        {
            fclose(e);
            clrscr();
            printf("Espacio insuficiente para los símbolos terminales.\n");
            printf("Oprime cualquier tecla para regresar...");
            getch();
            return FALSE;
        }
        if (!constab (e,tnt,tp,tt))
        {
            fclose(e);
```

```
            clrscr();
            printf("Espacio insuficiente.\n");
            printf("Oprime cualquier tecla para regresar...");
            getch();
            return FALSE;
        }
        fclose(e);
        return TRUE;
} /* de Construye */


Da_símbolos(TERM tt[],NOTERM tnt[]) /* despliega los símbolos de la gramática */
{
    int i;
            clrscr();
            printf("      Lista de símbolos terminales\n");
            printf("      -----------------------------\n        ");
            for (i=1 ; i<nt ; i++)
                printf("%c ",tt[i].simb);
            printf("\n\n      Lista de símbolos no terminales\n");
            printf("      --------------------------------\n        ");
            for (i=1 ; i<nnt ; i++)
                printf("%c ",tnt[i].simb);

} /* de Da_símbolos */


int consterm (FILE *arch,TERM t[]) /* construye la tabla de terminales */
{
  int i,j;
  char c;
  char cadterm[MAXT];

        fgets(cadterm,MAXT+1,arch);
        if (strlen(cadterm) == MAXT) /* trabaja con a lo mas MAXT-2 símbolos terminales */
           return FALSE;
        for (j=0 ; j<strlen(cadterm)-1 ; j++)
        {
             c = cadterm[j];
             t[nt].simb = c;                    /* pone el símbolo en el primer lugar disponible */
             for (i=1 ; t[i].simb != c ; i++);
             if (i == nt)                       /* lo encontré donde lo puse */
             {
                   t[nt].alcanza = 0;
                   t[nt++].cod = i;                 /* el código es el lugar del arreglo */
              }
         } /* del for */
         return TRUE;

} /* de consterm */
```

```
constab (FILE *aux,NOTERM tnt[],int tp[MAXP][LPROD],TERM tt[]) /* construye las tablas de
                                                      noterminales y  de producciones */
{
  int i,j,k,col;
  char c;
  int codigo;
  char cad[LPROD+2];                          /* cadena para leer las producciones */

/* LEE EL LADO IZQ. DE LA PRODUCCION Y ACTUALIZA LA TABLA */
  while ((fgets(cad,LPROD+3,aux)) != NULL)
  {
       if (strlen(cad) == LPROD+2) /* trabaja con producciones de
                                       longitud LPROD-2 a lo mas */
            return FALSE;
       col = 1;
       c = cad[0]; /* ponemos en c el primer caracter de la cadena que es el símbolo
                      del lado izquierdo de la producción */
       tnt[nnt].simb = c; /* ponemos el símbolo en el primer lugar disponible de la tabla */

       for (i=1 ; tnt[i].simb != c ; i++) ; /* lo busco en la tabla */
       if (i == nnt)                        /* lo encontré donde lo puse */
       {
           tnt[nnt].aparecio = 1;
           (tnt[nnt].numprod)++;
           tnt[nnt].lugini = np;
           tnt[nnt++].cod = -(nnt);
           if (nnt == MAXNT) /* solo puede manejar MAXNT-2 no terminales */
              return FALSE;
       }
       else                                 /* si ya estaba en la tabla */
       {
            (tnt[i].numprod)++;
            if (tnt[i].aparecio==0)
            {
                 tnt[i].aparecio = 1;
                 tnt[i].lugini = np;
            }
       } /* del else */

   /* LEE LOS SIMBOLOS DEL LADO DERECHO DE LA PRODUCCION, ACTUALIZA TABLAS
      Y PONE EN CODIGO EL DEL SIMBOLO */

      for (i=2 ; i<(strlen(cad)-1) ; i++) /* para el resto de la cadena, i.e. el lado derecho de la prod. */
      {
          if ((c=cad[i])!='e')    /* si es distinto de la cadena vacía */
          {
               tt[nt].simb = c; /*ponemos el símbolo en el primer disponible de la tabla de terminales */
               for (j=1 ; tt[j].simb != c ; j++); /* lo busco en terminales */
               if (j == nt)                      /* no lo encontré en terminales */
               {
                   tnt[nnt].simb = c;
                   for (k=0 ; tnt[k].simb != c ; k++) ;       /* lo busco en noterminales */
                   if (k == nnt)                             /* es nuevo actualizo tabla */
                   {
                      tnt[nnt].aparecio = 0;
```

```
                    codigo = (tnt[nnt++].cod = -(nnt));
                    if (nnt == MAXNT) /* solo puede manejar MAXNT-2 no terminales */
                        return FALSE;
                 }
                 else                                       /* es un noterminal que ya estaba */
                     codigo = tnt[k].cod;
              }
              else                                        /* es un terminal */
                 codigo = tt[j].cod;

              tp[np][col++] = codigo;

           } /*del if */
           else
              col++;

        } /* del for */
        np++;
        if (np == MAXP) /* solo puede manejar MAXP-2 producciones */
             return FALSE;

  } /* del while */
  return TRUE;
} /* de constab */



int Simbini(NOTERM tnt[],int tp[MAXP][LPROD]) /* Lee el símbolo inicial de la gramática, pone en codini
                                                 su código e incluye la producción cero */
{
  int j;
  int control = TRUE;
  char opcion;
  char c;

  gotoxy(12,11);
  printf("Dame el símbolo inicial  ");
  while (control)
  {
      c = getchar();
      tnt[nnt].simb = c;
      for (j=1 ; tnt[j].simb != c ; j++)  ;
      if (j==nnt) /* si el símbolo no es no-terminal */
      {
         no_simbini(c);
         if (lee_opcion("LlRr",&opcion,'R','r'))
              return FALSE;
         else
         {
             gotoxy(37,11);
             printf(" ");
             gotoxy(37,11);
             c = getchar();
          }
      }
```

```
      else /* si el símbolo es no-terminal */
      {
         control = FALSE;
         codini = tnt[j].cod;
         tp[0][0] = 1;              /* incluye producción cero */
         tp[0][1] = codini;
         return TRUE;
       }
  } /* del while */
}   /* de simbini */


no_simbini(char c) /* manda mensaje de que el símbolo inicial no es no-terminal */
{
  int i;
     gotoxy(11,13);
     printf("/");
     for (i=1 ; i<61 ; i++)
          printf("-");
     printf("\\n");
     printf("           |                                                       |\n");
     printf("           |  El símbolo '%c' no es símbolo no terminal en tu gramática. |\n",c);
     printf("           |                                                       |\n");
     printf("           \");
     for (i=1 ; i<61 ; i++)
         printf("-");
     printf("/");
     gotoxy(20,19);
     printf("[L] Lee el símbolo inicial otra vez.");
     gotoxy(20,20);
     printf("[R] Regresa al Menú Principal.");
     gotoxy(42,23);
     printf("Selecciona la opción ");
} /* de no_simbini */


Pinta_cuadro() /* pinta el cuadro de espera */
{
   char c,i;

        clrscr();
        gotoxy(17,7);
        printf("/");
        for (i=1 ; i<40 ; i++)
             printf("-");
        printf("\\n");
        printf("                 |                                    |\n");
        printf("                 |    Transformación de Gramaticas.   |\n");
        printf("                 |                                    |\n");
        printf("                 |                                    |\n");
        printf("                 |      T R A B A J A N D O ...       |\n");
        printf("                 |                                    |\n");
        printf("                 |                                    |\n");
        printf("                 |            %c",177);
```

```
        for (i=0 ; i<10 ; i++)
             printf("%c",177);
        printf("                    |\n");
        printf("                    |                                      |\n");
        printf("                    \");
        for (i=1 ; i<40 ; i++)
             printf("-");
        printf("/\n");
        gotoxy(29,15);
        getchar();              /* para limpiar el buffer */
        printf("%c",219);
} /* de Pinta_cuadro */

int Procesa(NOTERM tnt[],TERM tt[],int tp[MAXP][LPROD]) /* procesa gramática, eliminando producciones
                                                          vacías, unitarias y símbolos inutiles */
{
   Creaarch(tnt,tt,tp,".org"); /* crea archivo con la gramática original */
   Empty(tnt,tp);                  /* pone 1 en el campo nulo de los no terminales que generan
                                      la cadena vacía */
   if (!Vacia(tnt,tp))                /* elimina las producciones vacías */
   {
     printf("\n\n\n\n\nEspacio insuficiente al eliminar producciones vacias\n\n");
     printf("Oprima cualquier tecla para regresar...");
     getch();
     return FALSE;
   }

   if (!Actualiza(tnt,tt,tp,".vac"))
      return FALSE;
   if (!Unitarias(tnt,tp))              /* elimina las producciones unitarias y agrega las
   {
     printf("\n\n\n\n\nEspacio insuficiente al eliminar producciones unitarias\n\n");
     printf("Oprima cualquier tecla para regresar...");
     getch();
     return FALSE;
   }
                                   que compensan esta eliminación */
   if (!Actualiza(tnt,tt,tp,".unt"))
      return FALSE;

   if (Activo (tnt,tp))
   {
         Alcanza(tnt,tt,tp);      /* determina los símbolos alcanzables */
         if (!Actualiza(tnt,tt,tp,".utl"))
             return FALSE;
         else
             return TRUE;
   } else {
               gotoxy(15,20);
               printf("El símbolo inicial de la gramática es inactivo.\n\n\n");
               printf("                    Oprime cualquier tecla para regresar...");
               getch();
               return FALSE;
           }
} /* de Procesa */
```

```
Crearch(NOTERM tnt[],TERM tt[],int tp[MAXP][LPROD],char *ext) /* crea el archivo con la original */
{
   FILE *sal;
   unsigned i;

   NTO = nnt-1;
   nnto = nnt-1;

   for (i=0 ; i<strlen(pref) ; i++)
       org[i] = pref[i];
   strcat(org,ext);

   sal = fopen(org,"w");
   Nuevo(sal,tnt,tp,tt);
   fclose(sal);

} /* de Crearch */



Empty(NOTERM tnt[],int tp[MAXP][LPROD]) /* encuentra los no terminales que generan la vacía y
                                           pone un 1 en el campo nulo de los simbos nulos */
{
  int tE[MAXNT][MAXNT]; /* matriz de relacion entre no terminales que tienen produccion
                           vacia o producción que comienza con un no terminal */
  int i,j,k;
  int cambio,vacia;  /* booleanas */

  /* inicializo el campo nulo de la tabla de no terminales */
    for (i=0 ; i<MAXNT ; i++)
        tnt[i].nulo = 0;

  /* inicializo tE */
    for (i=0 ; i<nnt+1 ; i++)
       for (j=0 ; j<nnt+1 ; j++)
           tE[i][j] = 0;
    printf("%c",219);


    /* construyo la matriz de relación entre no terminales que tienen producción que comienza
       con no terminal o con vacía. La columna nnt de la matriz tiene un 1 cuando el no terminal
       del renglón tiene producción vacía. Cuando tiene producción que comienza con no terminal,
       tiene 1 en la columna de dicho no terminal */

    for (i=1 ; i<nnt ; i++)                                            /* para cada no terminal */
       for (j=tnt[i].lugini ; j<tnt[i].lugini+tnt[i].numprod ; j++)  /* para cada producción */
       {
         if (tp[j][1] == 0)                  /* si es la producción vacia */
            tE[i][nnt] = 1;
         else
             if (tp[j][1] < 0 )              /* si la prod. comienza con no terminal */
               tE[i][-(tp[j][1])] = 1;
       }
       printf("%c",219);
```

```
/* calculo con Warshall la cerradura para obtener los no terminales que
   producen cadenas que comienzan con la vacía */

   for (k=0 ; k<nnt+1 ; k++)
       for (i=0 ; i<nnt+1 ; i++)
           for (j=0 ; j<nnt+1 ; j++)
               tE[i][j] = tE[i][j] || (tE[i][k] && tE[k][j]);

 /* pongo un uno en el campo nulo de los no terminales que
   producen cadenas que comienzan con la vacía */

   for (i=1 ; i<nnt ; i++)    /* para cada no terminal */
      if (tE[i][nnt] == 1)
         tnt[i].nulo = 1;

 /* pongo en cero el campo nulo de los terminales que producen cadenas que comienzan con la vacía
    pero sigue otra cosa, es decir los elimino del conjunto de nulos */
   printf("%c",219);
   do{
         cambio = FALSE;
         for (i=1 ; i<nnt ; i++)
             if (tnt[i].nulo == 1)            /* para cada no terminal con uno en nulo*/
             {
                vacia = FALSE;
                j = tnt[i].lugini;
                while( (j<tnt[i].lugini+tnt[i].numprod) && ( !vacia) )
                {
                    k=1;
                    while ((tp[j][k]<0) && (tnt[-(tp[j][k])].nulo == 1)) /* mientras la producción
                                                                              sea de noterminales */
                           k++;
                    if (((tp[j][k]<0) && (tnt[-(tp[j][k])].nulo == 0))
                           || (tp[j][k] > 0 ))
                        j++;
                    else
                          if (((tp[j][k]<0) && (tnt[-(tp[j][k])].nulo == 1))
                                || (tp[j][k] == 0))
                              vacia = TRUE;
                } /* del while */
               if (!vacia)
               {
                 tnt[i].nulo = 0;
                 cambio = TRUE;
               }

             } /* del if */
      )   while (cambio);

} /* de Empty */
```

```
int Vacia(NOTERM tnt[],int tp[MAXP][LPROD])  /* elimina las producciones vacias */
{
  int i,j;

  for (i=1 ; i<nnt ; i++) /* para cada no terminal */
        for (j=tnt[i].lugini ; j<tnt[i].lugini+tnt[i].numprod ; j++)
             if (tp[j][0]) /* para cada produccion viva */
             {
               if (tp[j][1] == 0)
                  tp[j][0] = 0;     /* si es la vacia quitala */
               else
                   if (!genera(tnt,tp[j],i,tp)) /* genera las que compensan la eliminación */
                       return FALSE;
             }
   printf("%c",219);
   return TRUE;
} /* de Vacia */


int Actualiza(NOTERM tnt[],TERM tt[],int tp[MAXP][LPROD],char arch[],char *ext)
{
   FILE *sal; /* para los archivos de las nuevas gramaticas */
   unsigned i;

   if (!Acttabla(tnt,tp))              /* actualiza tablas */
       return FALSE;

   for (i=0 ; i<strlen(pref) ; i++)
       arch[i] = pref[i];
   strcat(arch,ext);

   sal = fopen(arch,"w");
   Nuevo(sal,tnt,tp,tt);   /* genera el archivo arch */
   fclose(sal);
   nnto = nnt-1;
   return TRUE;
} /* de Actualiza */


int Unitarias(NOTERM tnt[],int tp[MAXP][LPROD]) /* elimina las producciones unitarias
                                               y  agrega las que lo compensan */
{
  int tE[MAXNT][MAXNT];    /* para la matriz de relación de no terminales con producciones unitarias */
  int tunit[MAXNT][MAXNT]; /* para el registro de unitarias */
  int i,j,k,in;
  int simb;


  /* inicializo tE y tunit */
  for (i=0 ; i<nnt ; i++)
     for (j=0 ; j<nnt ; j++)
         tE[i][j] = 0;
  for (i=0 ; i<nnt ; i++)
     for (j=0 ; j<nnt ; j++)
         tunit[i][j] = 0;
```

```
/* construyo la matriz de relación entre no terminales que tienen
   producciones unitarias,i.e. producciones que constan de un sólo
   símbolo no terminal */

for (i=1 ; i<nnt ; i++)                                          /* para cada no terminal */
   for (j=tnt[i].lugini ; j<tnt[i].lugini+tnt[i].numprod ; j++)  /* para cada producción */
     if ( (tp[j][1] < 0) && (tp[j][2] == 0) )                    /* si es producción unitaria */
        tE[i][-(tp[j][1])] = 1;


/* calculo con Warshall la cerradura para obtener los no terminales que
   producen unitarias*/

   for (k=1 ; k<nnt ; k++)
       for (i=1 ; i<nnt ; i++)
           for (j=1 ; j<nnt+1 ; j++)
               tE[i][j] = tE[i][j] || (tE[i][k] && tE[k][j]);

 /* guardo en la tabla tunit, para cada no terminal, el índice de la tabla de
    no terminales, que corresponde a sus no terminales unitarios */

   for (i=1 ; i<nnt ; i++)      /* para cada no terminal */
   {
       k = 1;
       for (j=1 ; j<nnt ; j++)
           if (tE[i][j] == 1)
               tunit[i][k++] = j;
   }

   /* elimino las producciones unitarias */

   for (i=1 ; i<nnt ; i++) /* para cada no terminal */
        for (j=tnt[i].lugini ; j<tnt[i].lugini+tnt[i].numprod ; j++) /* para cada producción */
            if ( (tp[j][1] < 0) && (tp[j][2] == 0) )
                 tp[j][0] = 0;

   /* agrego las producciones que compensan la eliminación de las unitarias */

   for (i=1 ; i<nnt ; i++) /* para cada no terminal */
   {
     k = 1;
     while ( (simb = tunit[i][k++]) != 0) /* para cada uno de los símbolos en tunit */
     {
       for (j=tnt[simb].lugini ; j<tnt[simb].lugini+tnt[simb].numprod ; j++) /* para cada producción */
          if( (tp[j][0]==1) && (! iguales(tp,i,tnt,tp[j])))
            if (!agrega(tnt,i,tp[j]))
                return FALSE;
     }
   }
   return TRUE;

} /* de Unitarias */
```

```
int Activo(NOTERM tnt[],int tp[MAXP][LPROD]) /* determina los símbolos activos y elimina las
                                                producciones que involucran inactivos */
{
  int i,j,k;
  int cambio,simb,act;

  /* Ponemos uno en el campo activo de los no terminales que tienen producciones
     vacías o producciones con solamente terminales */

   for (i=1 ; i<nnt ; i++)  /* para cada no terminal */
   {
     j = tnt[i].lugini;
     cambio = FALSE;
     while ( (j<tnt[i].lugini+tnt[i].numprod) && (!cambio) ) /* mientras haya prods. y no haya cambio */
     {
       k = 1;
       simb = tp[j][k++];
       if (simb == 0)    /* si es la producción vacía */
       {
         tnt[i].activo = 1;
         cambio = TRUE;
       }
       else
       {
         if (simb > 0)   /* si es un terminal */
         do
            simb = tp[j][k++]; /* recorre mientras sean terminales */
         while (simb > 0);

         if (simb == 0)   /* si es una producción de solamente terminales */
         {
           tnt[i].activo = 1;
           cambio = TRUE;
         }
        }  /* del else */

        j++;
     } /* del while */
   }  /* del for */

  /* Ponemos uno en el campo activo de los no terminales que tienen producciones
     de terminales y no terminales con un uno en el campo activo */

     do
     {
        cambio = FALSE;
        for (i=1 ; i<nnt ; i++)        /* para cada no terminal */
           if (tnt[i].activo == 0)
           {
             j = tnt[i].lugini;
             act = FALSE;
             while ( (j<tnt[i].lugini+tnt[i].numprod) && (!act) ) /* mientras haya producciones
                                                                     y no encuentres activo */
             {
               k = 1;
```

```
            simb = tp[j][k++];
            while ( (simb > 0) || (simb < 0 && tnt[-(simb)].activo))
                  simb = tp[j][k++];
            if (simb == 0)
            {
             act = TRUE;
             cambio = TRUE;
             tnt[i].activo = 1;
            }
             j++;
           } /* del while */

      } /* del if */
    } while (cambio);


  /* Quitamos las producciones (ponemos cero en la primera columna)
     que involucren símbolos inactivos */

     for (i=1 ; i<nnt ; i++)          /* quitamos las producciones que tienen inactivo del lado izquierdo */
          if (tnt[i].activo == 0)
             for (j=tnt[i].lugini ; j<tnt[i].lugini+tnt[i].numprod ; j++)
                 tp[j][0] = 0;

       for (i=0 ; i<np ; i++) /* quitamos las producciones que involucran inactivos del lado derecho */
       {
         k = 1;
         if ( (tp[i][0]) && (tp[i][k] != 0) )  /* si la producción todavía está y no es la vacía */
         {
           simb = tp[i][k++];
           while ( (simb>0) || (simb<0 && tnt[-(simb)].activo) )
                simb = tp[i][k++];

           if (simb < 0)                      /* si te quedas en noterminal inactivo */
             tp[i][0] = 0;

         } /* del if */
       } /* del for */


  if (tnt[-(codini)].activo == 0) /* checa que el símbolo inicial sea activo */
         return FALSE;
     else
         return TRUE;

} /* de activo */
```

```
Alcanza(NOTERM tnt[],TERM tt[],int tp[MAXP][LPROD]) /* determina los símbolos alcanzables y
                                                      elimina las producciones que involucren
                                                      símbolos inalcanzables */
{
  int i,j,k,simb,cambio;


  /* Ponemos uno en el campo alcanza de los no-terminales y terminales alcanzables */

  tnt[-(codini)].alcanza = 1;  /* el símbolo inicial es alcanzable */

  do   /* haz mientras haya cambios, i.e. mientras entren no-terminales a alcanza */
  {
    cambio = FALSE;
    for (i=1 ; i<nnt ; i++)
       if (tnt[i].alcanza && tnt[i].activo) /* para cada no terminal alcanzable y activo */
       {
         for (j=tnt[i].lugini ; j<tnt[i].lugini+tnt[i].numprod ; j++)
             if ( (tp[j][0]) && (tp[j][1] != 0) )   /* para cada producción que este activa y no sea
                                                       la vacía */
             {
               k = 1;
               while ((simb = tp[j][k++]) != 0) /* recorre la producción */
               {
               if (simb>0)  /* si es terminal es alcanzable, pon uno */
                 tt[simb].alcanza = 1;
               else
                     /* si es un no-terminal activo no alcanzable, hazlo alcanzable */
                 if ( (tnt[-(simb)].activo) && (tnt[-(simb)].alcanza == 0) )
                 {
                   tnt[-(simb)].alcanza = 1;
                   cambio = TRUE;
                 }
               } /* del while */
             } /* del if */
       } /* del if */
  } while (cambio);


/* Quitamos las producciones cuyo lado izquierdo es activo pero inalcanzable */

  for (i=1 ; i<nnt ; i++)
      if (tnt[i].activo)   /* para cada no terminal activo */
          if (tnt[i].alcanza == 0)  /* si es un inalcanzable */
              for (j=tnt[i].lugini ; j<tnt[i].lugini+tnt[i].numprod ; j++)
                 tp[j][0] = 0; /* elimina todas sus producciones */

/* Quitamos las producciones que involucren inalcanzables del lado derecho */

   for (i=0 ; i<np ; i++)
   {
     k = 1;
     if ( (tp[i][0]) && (tp[i][k] != 0) )  /* si la producción todavia esta y no es la vacía */
     {
       simb = tp[i][k++];
```

```
      while ( ((simb>0) && (tt[simb].alcanza)) || ((simb<0) && (tnt[-(simb)].alcanza)) )
            simb = tp[i][k++];    /* mientras sea símbolo alcanzable avanza */

      if (simb != 0)  /* si te quedaste en símbolo inalcanzable */
         tp[i][0] = 0; /* elimina la producción */

     } /* del if */
   } /* del for */
}  /* de alcanza */


int Checrec(int tF[TOTS][TOTS],NOTERM tnt[],int tp[MAXP][LPROD]) /* regresa TRUE si la gramática
                                                                    es recursiva por la izquierda */

 /* construye la matriz F y su cerradura, UFX si y solo si existe
    una producción U -> X, UF+X es la cerradura de F
    para checar si la gramática es recursiva por la izq */
{
  int simb;
  int ind;
  int indprod;
  int totsimb;   /* número total de símbolos */
  int i,j,k;

  totsimb = nnt+nt-2;

/* CONSTRUIMOS LA MATRIZ DE RELACION F Y SU CERRADURA */

  for (i=0 ; i<=totsimb ; i++)        /* inicializo la tablaF en ceros */
      for (j=0 ; j<=totsimb ; j++)
          tF[i][j] = 0;

   tF[0][-(codini)] = 1;   /* renglon del símbolo agregado */

   for (i=1 ; i<nnt ; i++)   /* para cada no terminal */

       /* para cada una de sus producciones vivas que no sean la vacía */
      for (k=tnt[i].lugini ; k<tnt[i].lugini+tnt[i].numprod ; k++)
      if ((tp[k][0]) && (tp[k][1] != 0) )
      {
            indprod = 1;
            simb = tp[k][indprod++];
            if (simb<0)
                  ind = -(simb);
            else
                  ind = nnt+simb-1;  /* calcula ind */

            tF[i][ind] = 1;
            while ( (simb<0) && (tnt[-(simb)].nulo) )
            {
                   simb = tp[k][indprod++];
                   if (simb !=0)
                   {
                          if (simb<0)
                               ind = -(simb);
```

```
                        else
                            ind = nnt+simb-1;   /* calcula ind */

                        tF[i][ind] = 1;
                    }
              } /* del while */
         } /* del if */

    /* Construimos la cerradura con Warshall */

    for (k=0 ; k <= totsimb ; k++)
        for (i=0 ; i <= totsimb ; i++)
            for (j=0 ; j <= totsimb ; j++)
                tF[i][j] = tF[i][j] || (tF[i][k] && tF[k][j]);

/* CON LA MATRIZ F CHECAMOS SI LA GRAMATICA ES RECURSIVA POR LA IZQUIERDA */

    for (i=1 ; i<nnt ; i++)   /* si hay algun 1 en la diagonal de F para los
                                 no terminales la gramática es recursiva */
        if (tF[i][i])
              return TRUE;

    return FALSE;
}     /* de Checrec */



int Fnormales(NOTERM tnt[],TERM tt[],int tp[MAXP][LPROD]) /* pasa a formas normales */
{
   if (!Chomsky(tnt,tp))                    /* pasa a forma normal de Chomsky */
   {
     printf("\n\n\n\n\nEspacio insuficiente al pasar a Forma Normal de Chomsky\n\n");
     printf("Oprime cualquier tecla para regresar...");
     getch();
     return FALSE;
   }
   if (!Actualiza(tnt,tt,tp,".fnc"))
      return FALSE;
   if (!Greibach(tnt,tp))                   /* pasa a forma normal de Greibach */
   {
     printf("\n\n\n\n\nEspacio insuficiente al pasar a Forma Normal de Greibach\n\n");
     printf("Oprime cualquier tecla para regresar...");
     getch();
     return FALSE;
   }
   if (!Actualiza(tnt,tt,tp,".fng"))
      return FALSE;
   else
       return TRUE;

} /* de Fnormales */
```

```
int Chomsky(NOTERM tnt[],int tp[MAXP][LPROD]) /* transforma la gramatica a forma normal de Chomsky */
{
  int nvo; /* para ir registrando el lugar en la tabla de los nuevos no terminales a agregar */
  int cont; /* para recorrer las producciones */
  int simb; /* para guardar símbolos leidos de las producciones */
  int nueva[LPROD]; /* para generar las nuevas producciones */
  int ntnvos; /* número de no terminales nuevos a agregar */
  int i,j,k,n;
  int aux  = 0;

  nvo = nnt;
  for (i=1 ; i<nnt ; i++)  /* para cada no terminal */
      for (j=tnt[i].lugini ; j<tnt[i].lugini+tnt[i].numprod ; j++)
         if (tp[j][2] != 0) /* para cada una de sus producciones >= 2 */
         {
           cont = 1;
           while ( (simb = tp[j][cont]) != 0) /* para cada símbolo de la producción */
                 if (simb > 0) /* si es terminal */
                 {
                  nueva[0] = 1;
                  nueva[1] = simb;
                  for (k=2 ; k<LPROD ; k++)
                     nueva[k] = 0;
                  if (!iguales(tp,nvo,tnt,nueva))
                      if (!agrega(tnt,nvo,nueva)) /* agrega la prod. nvo -> simb */
                          return FALSE;
                  tp[j][cont++] = -nvo;
                  nvo++;
                  if (nvo == MAXNT)
                     return FALSE;
                  aux++;
                 } /* del if */
                 else
                     cont++;
         } /* del if */

         /* En este punto todas las producciones > 1 son de puros no terminales */

         for (i=1 ; i<nnt ; i++) /* para cada no terminal */
             for (j=tnt[i].lugini ; j<tnt[i].lugini+tnt[i].numprod ; j++)
                if (tp[j][3] != 0) /* para cada una de sus producciones >= 3 */
                {
                  cont = 1;
                  while (tp[j][cont] != 0) /* calculamos el tamano de la producción */
                        cont++;
                  ntnvos = cont-3; /* número de nuevos no terminales a agregar */
                  if (nvo+ntnvos == MAXNT)
                      return FALSE;
                  aux = aux + ntnvos;
                  tp[j][0] = 0;     /* eliminamos la producción */

                  /* agregamos al no terminal [primer símbolo de la prod] + [primer símbolo nuevo] */
                  nueva[0] = 1;
                  nueva[1] = tp[j][1];
                  nueva[2] = -nvo;
```

```
                for (k=3 ; k<LPROD ; k++)
                    nueva[k] = 0;
                if (!iguales(tp,i,tnt,nueva))
                    if (!agrega(tnt,i,nueva))
                        return FALSE;

                cont = 2;
                k = nvo;
                for (k=nvo ; k<nvo+ntnvos-1 ; k++) /* para cada símbolo nuevo - 1 */
                {
                  /* generamos la prod [el que sigue de la vieja] + [el que sigue de nuevos] */
                  nueva[0] = 1;
                  nueva[1] = tp[j][cont++];
                  nueva[2] = -(k+1);
                  for (n=3 ; n<LPROD ; n++)
                     nueva[n] = 0;
                  if (!iguales(tp,k,tnt,nueva))
                      if (!agrega(tnt,k,nueva))
                          return FALSE;
                } /* del for */

                /* agregamos al último nuevo lo que queda de la producción */
                nueva[0] = 1;
                nueva[1] = tp[j][cont++];
                nueva[2] = tp[j][cont];
                for (n=3 ; n<LPROD ; n++)
                    nueva[n] = 0;
                if (!iguales(tp,k,tnt,nueva))
                    if (!agrega(tnt,k,nueva))
                        return FALSE;
                nvo = nvo + ntnvos;

              } /* del if */

              for (i=nnt ; i<nnt+aux ; i++) /* genera los caracteres de los nuevos símbolos
                                               repitiendolo cada nueve */
                if (indice<10)
                {
                  tnt[i].simb = let;
                  tnt[i].cod = -i;
                  indice++;
                }
                else
                {
                  let++;
                  tnt[i].simb = let;
                  tnt[i].cod = -i;
                  indice = 2;
                }
              nnt = nnt + aux;
              return TRUE;
} /* de Chomsky */
```

```
int Greibach(NOTERM tnt[],int tp[MAXP][LPROD]) /* Pasa la gramática a forma normal de Greibach */
{
  int nvo = nnt;
  int i,j,k;
  PRODS s;

/* HACEMOS QUE TODAS LAS PRODUCCIONES DE LOS NO TERMINALES DE LA FNC
   COMIENCEN CON UN TERMINAL O UN NO TERMINAL MAYOR QUE EL */

   for (i=1 ; i<nnt ; i++) /* para cada no terminal */
   {
     for (j=1 ; j<=i-1 ; j++) /* para cada no terminal anterior a i */
     {

         for (k=tnt[i].lugini ; k<tnt[i].lugini+tnt[i].numprod ; k++) /* para cada una de las prods. de i */

           if ( (tp[k][0]) && (tp[k][1] == -j) ) /* si la producción vive y comienza con el no terminal j */
                 if (!creaGrei(tnt,tp,tp[k],j,i))
                     return FALSE;

          s = tnt[i].sig;
          while (s != NULL)
          {
            if ( (s -> q[0]) && (s -> q[1] == -j) ) /* para cada producción de i en la lista que viva y
                                                       comience con el no terminal j */
              if (!creaGrei(tnt,tp,s -> q,j,i))
                  return FALSE;

            s = s -> sig;
          }
     } /* del for j */
     if (!directa(i,tnt,tp,&nvo))
         return FALSE;
     if (nvo == MAXNT)
         return FALSE;

   } /* del for i */

/* EN ESTE MOMENTO TENEMOS QUE TODAS LAS PRODUCCIONES DEL ULTIMO DE LOS NO TERMINALES DE
   LA FNC COMIENZAN CON UN TERMINAL; ASI QUE VAMOS A RECORRER LOS NO TERMINALES DESDE
   EL PENULTIMO HASTA EL PRIMERO, REEMPLAZANDO EL PRIMER SIMBOLO DEL LADO DERECHO DE LAS
   PRODUCCIONES CUANDO ESTE SEA UN NO TERMINAL, POR LAS PRODUCCIONES DE DICHO NO TERMINAL.
   DE ESTA FORMA TENEMOS QUE TODOS LOS NO TERMINALES DE LA FNC COMIENZAN CON UN TERMINAL */

   for (i=nnt-2 ; i>0 ; i--) /* para cada no terminal desde el penúltimo hasta el primero */
   {
     for (j=tnt[i].lugini ; j<tnt[i].lugini+tnt[i].numprod ; j++)
                                   /* para cada una de sus prods. de la tabla */
         if (!prodGrei(tnt,tp,tp[j],i))
               return FALSE;

     s = tnt[i].sig;
     while (s != NULL)   /* para cada una de sus producciones de la lista */
     {
       if (!prodGrei(tnt,tp,s -> q,i))
```

```
          return FALSE;
       s = s -> sig;
      }
    } /* del for */


/* AHORA ACTUALIZAMOS LAS PRODUCCIONES DE LOS NUEVOS SIMBOLOS QUE FUERON
   AGREGADOS, SI ES QUE HUBO, DE TAL FORMA QUE SUS PRODUCCIONES TAMBIEN
   COMIENCEN TODAS CON UN TERMINAL */


   for (i=nnt ; i<nvo ; i++) /* para cada no terminal de los nuevos */
   {
     for (j=tnt[i].lugini ; j<tnt[i].lugini+tnt[i].numprod ; j++)
                                        /* para cada una de sus prods. de la tabla */
         if (!prodGrei(tnt,tp,tp[j],i))
              return FALSE;
     s = tnt[i].sig;
     while (s != NULL)   /* para cada una de sus producciones de la lista */
     {
       if (!prodGrei(tnt,tp,s -> q,i))
            return FALSE;
       s = s -> sig;
      }
     } /* del for */


/* AHORA GENERAMOS LOS CARACTERES DE LOS NUEVOS SIMBOLOS REPITIENDO EL SIMBOLO CADA NUEVE */


   for (i=nnt ; i<nvo ; i++) /* para cada uno de los no terminales nuevos */

       if (indice<10)
       {
         tnt[i].simb = let;
         tnt[i].cod = -i;
         indice++;
        }
        else
        {
          let++;
          tnt[i].simb = let;
          tnt[i].cod = -i;
          indice = 2;
         }
   nnt = nvo;
   return TRUE;

}  /* de Greibach */
```

```
int iguales(int tp[MAXP][LPROD],int i,NOTERM tnt[],int pr[]) /* regresa true si la producción pr ya
                                                                  es producción del noterminal i */
{
  int k, /* índice para el número de producción */
      p; /* índice para recorrer cada producción */
  PRODS r;
  int encontro;

/* compare con las producciones de la tabla */

  k = tnt[i].lugini;
  encontro = FALSE;
  while( (k < tnt[i].lugini+tnt[i].numprod) && (! encontro) )
  {
    p = 1;
    if (tp[k][0] == 1)
    {
      while ( (tp[k][p] == pr[p]) && (tp[k][p] != 0) )
             p++;
       if ( (tp[k][p] == 0) && (pr[p] == 0) )
           encontro = TRUE;
       else
           k++;
    } /* del if */
    else
        k++;
   } /* del while */

   if (encontro)    /* si ya encontró una igual regresa true */
      return(TRUE);
   else             /* si no busca en los de la lista */
   {

     r = (PRODS)tnt[i].sig;
     while ( (r != NULL) && (! encontro) ) /* mientras haya elementos en la lista
                                              y no encuentres una igual */
     {
      p = 1;

      while ( (r->q[p] == pr[p]) && (r->q[p] != 0) )
            p++;
       if ( (r->q[p] == 0) && (pr[p] == 0) )
           encontro = TRUE;
       r = r->sig;
     } /* del while */

     if (encontro)
        return (TRUE);
     else
        return (FALSE);
   } /* del else */

} /* de iguales */
```

```
int agrega (NOTERM tnt[],int indice,int nueva[LPROD]) /* agrega una producción a la lista de indice */
{
  int j;
  PRODS p;

  if ( (p = (PRODS) malloc(sizeof(PROD))) == NULL)
       return FALSE;

  for (j = 0 ; j < LPROD ; j++)
     p -> q[j] = nueva[j];

   p -> sig = tnt[indice].sig;
   tnt[indice].sig = p;
   return TRUE;
} /* de agrega */


int genera (NOTERM tnt[],int consec[LPROD],int indice,int tp[MAXP][LPROD]) /* Genera las nuevas producciones
                                                                               asociadas a la producción consec
                                                                               para compensar la eliminacion
                                                                               de producciones vacías */
{
  unsigned int producc,  /* para marcar los lugares de los no terminales que generan la vacía */
               numvacia, /* número de símbolos de consec que generan la vacia */
               lugar,    /* indice para consec */
               cualvacia,/* para las combinaciones de que aparezcan o no, en nueva la produccion,
                                                     los símbolos que generan la vacía */
               lugnueva,   /* indice para la nueva producción */
               llevovacia;  /* número de las posibles vacías */

  int tam;         /* número de símbolos de consec */
  int nueva[LPROD];  /* para la nueva producción */
  int i;

  tam = 1;                         /* calculo el tamaño de consec */
  while(consec[tam] != 0) tam++;
  tam = tam--;


  producc = 0;
  numvacia = 0;

/* Contamos el número de símbolos de consec que generan la vacia y marcamos
   en producc los lugares correspondientes a esos símbolos */

  for (lugar = 1; lugar <= tam ; lugar++)
  {
     if ( (consec[lugar] < 0) && (tnt[-(consec[lugar])].nulo) )
     {
       producc = producc | (1 << (tam-lugar + 1)); /* prendemos los bits correspondientes a los
                                                    símbolos de consec que generan la vacía */
       numvacia++; /* queda el número de simbolo que generan la vacía */
     }
   }
```

```
/* El número de producciones nuevas a generar es 2**numvacia.
   En el entero cualvacia el bit i-esimo esta prendido si el símbolo i-esimo,
   de entre los que generan la vacia, debe desaparecer de la nueva producción */

/* Generamos las nuevas producciones */

   for (cualvacia = 1 ; /* desde uno, porque la que no desaparece a ningun simbolo
                         es la original, y a esa la dejamos */

/* Tantas producciones como 2**numvacia-2 si todos los símbolos generan
   a la vacia para no generar la vacia de nuevo, 2**numvacia-1 si no */

          (tam == numvacia) ? cualvacia <= ( (1 << numvacia) - 2)
                            : cualvacia <= ( (1 << numvacia) - 1) ;

        cualvacia++)
   {
     for (i=0 ; i<LPROD ; i++)  /* inicializo la nueva producción */
         nueva[i] = 0;
     nueva[0] = 1;

     lugnueva = 1;
     llevovacia = 0; /* Da el número, respecto a las posibles vacías */

     for (lugar = 1 ; lugar <= tam ; lugar++) /* para cada símbolo de consec */

     /* si ese símbolo esta marcado en producc, se trata de uno que pudiera desaparecer */
         if ( (producc >> (tam-lugar+1) ) & 1)

             /* si el bit i-esimo de cualvacia esta prendido, entonces el
                símbolo i-esimo debe desaparecer */
             if ( (cualvacia >> (numvacia - ++llevovacia)) & 1);

             /* si debe aparecer lo copio a nueva */
             else
                  nueva[lugnueva++] = consec[lugar];

         else /* si no esta marcado en producc debe aparecer y lo copio a nueva */
              nueva[lugnueva++] = consec[lugar];
         if (! iguales(tp,indice,tnt,nueva) )
              if (!agrega(tnt,indice,nueva))
                  return FALSE;
     } /* del for */

     return TRUE;
} /* de genera */
```

```c
int directa(int n,NOTERM tnt[],int tp[MAXP][LPROD],int* aux) /* determina la recursividad izquierda
                                                                directa y genera las producciones
                                                                que compensan esta eliminación */
{
  int i,k,r,in;
  int resto[LPROD];
  int hubo = FALSE;
  PRODS t;

  for (i=tnt[n].lugini ; i<tnt[n].lugini+tnt[n].numprod ; i++)

      if ( (tp[i][0]) && (tp[i][1] == -n) ) /* para cada producción viva de n que sea
                                               recursiva por la izquierda en la tabla */
      {
        hubo = TRUE;
        for (k=1 ; k<LPROD ; k++)
            resto[k] = 0;      /* inicializo el resto */
        resto[0] = 1;

        in = 2;
        r = 1;
        while (tp[i][in] != 0)
              resto[r++] = tp[i][in++]; /* guardo en resto lo que sigue de n en la producción */
        if (!iguales(tp,*aux,tnt,resto))
            if (!agrega(tnt,*aux,resto))           /* agrego la producción resto al nuevo */
                return FALSE;

        resto[r] = -(*aux);
        if (!iguales(tp,*aux,tnt,resto))
            if (!agrega(tnt,*aux,resto)) /* agrego la producción resto+nuevo */
                return FALSE;

        tp[i][0] = 0;           /* quito esa producción */

      } /* del if */

      t = tnt[n].sig;
      while (t != NULL)
      {
         if ( (t -> q[0]) && (t -> q[1] == -n) ) /* para cada producción recursiva de la lista */

         {
           hubo = TRUE;
           for (k=1 ; k<LPROD ; k++)
               resto[k] = 0;     /* inicializo el resto */
           resto[0] = 1;
           in = 2;
           r = 1;
           while (t -> q[in] != 0)
              resto[r++] = t -> q[in++]; /* guardo en resto lo que sigue de n en la producción */
           if (!iguales(tp,*aux,tnt,resto))
               if (!agrega(tnt,*aux,resto))           /* agrego la producción resto al nuevo */
                   return FALSE;

           resto[r] = -(*aux);
```

```
          if (!iguales(tp,*aux,tnt,resto))
              if (!agrega(tnt,*aux,resto)) /* agrego la producción resto+nuevo */
                  return FALSE;

          t -> q[0] = 0;          /* quito esa producción */

        } /* del if */

        t = t -> sig;
     } /* del while */

 if (hubo)
 {
 t = tnt[n].sig;
 for (i=tnt[n].lugini ; i<tnt[n].lugini+tnt[n].numprod ; i++)
     if ( (tp[i][0]) && (tp[i][1] != -n) ) /* para cada producción viva de n
                                              que no sea recursiva */
     {
       for (k= 1 ; k<LPROD ; k++)
           resto[k] = 0;       /* inicializo el resto */
        resto[0] = 1;

       r = 1;
       while (tp[i][r] != 0)
             resto[r] = tp[i][r++]; /* guardala en resto */
       resto[r] = -(*aux);             /* pegale el nuevo */
       if (!iguales(tp,n,tnt,resto))
           if (!agrega(tnt,n,resto))         /* agregala a las producciones de n */
               return FALSE;
     } /* del if */

     while (t != NULL)
     {
       if ( (t -> q[0]) && (t -> q[1] != -n) ) /* para cada producción viva de la tabla no recursiva */
       {
         for (k= 1 ; k<LPROD ; k++)
            resto[k] = 0;       /* inicializo el resto */
         resto[0] = 1;

         r = 1;
         while (t -> q[r] != 0)
              resto[r] = t -> q[r++]; /* guardala en resto */
         resto[r] = -(*aux);             /* pegale el nuevo */
         if (!iguales(tp,n,tnt,resto))
            if (!agrega(tnt,n,resto))         /* agregala a las producciones de n */
               return FALSE;

         } /* del if */

         t = t -> sig;
     } /* del while */
   (*aux)++;
 } /* del if hubo */
 return TRUE;
} /* de directa */
```

```
int creaGrei(NOTERM tnt[],int tp[MAXP][LPROD],int arr[LPROD],int j,int i) /* crea las nuevas producciones de
                                                                              primer paso para pasar a FNG */
{
  int resto[LPROD];
  int nueva[LPROD];
  int r,in,l,n;
  PRODS p;

  for (r=1 ; r<LPROD ; r++)
      resto[r] = 0;
  resto[0] = 1;             /* inicializo resto */

  in = 2;
  r = 1;
  while (arr[in] != 0)   /* guardo en resto la producción arr a partir de su segundo elemento */
        resto[r++] = arr[in++];
  arr[0] = 0;  /* elimino la producción arr */

  for (l = tnt[j].lugini ; l<tnt[j].lugini+tnt[j].numprod ; l++)
      if (tp[l][0]) /* para cada una de las producciones de j vivas en la tabla */
      {
         for (r=1 ; r<LPROD ; r++)
             nueva[r] = 0;
         nueva[0] = 1;             /* inicializo nueva */

         n = 1;
         while (tp[l][n] != 0)
              nueva[n] = tp[l][n++]; /* copio a nueva la producción de j */

         r = 1;
         while (resto[r] != 0)
         {
              nueva[n++] = resto[r++];   /* le pego el resto a nueva */
              if (n == LPROD)
                 return FALSE;
         }
         if ( !iguales(tp,i,tnt,nueva) )
             if (!agrega(tnt,i,nueva))
                 return FALSE;
      } /* del if */


      p = tnt[j].sig;
      while (p != NULL)
      {
        if ( p -> q[0]) /* para cada una de las producciones de j vivas en la lista */
        {
          for (r=1 ; r<LPROD ; r++)
             nueva[r] = 0;
          nueva[0] = 1;             /* inicializo nueva */

          n = 1;
          while (p -> q[n] != 0)
             nueva[n] = p -> q[n++]; /* copio a nueva la producción de j */
          r = 1;
```

```
          while (resto[r] != 0)
          {
              nueva[n++] = resto[r++];    /* le pego el resto a nueva */
              if (n == LPROD)
                  return FALSE;
          }

          if ( !iguales(tp,i,tnt,nueva) )
              if (!agrega(tnt,i,nueva))
                  return FALSE;
        } /* del if */
        p = p -> sig;

      } /* del while */
      return TRUE;
} /* de creaGrei */


int prodGrei(NOTERM tnt[],int tp[MAXP][LPROD],int arr[LPROD],int i) /* crea las nuevas producciones del
                                                                        segundo paso para pasar a FNG */
{
  int resto[LPROD];
  int nueva[LPROD];
  int r,n,k;
  PRODS p;

  if ( (arr[0]) && (arr[1] < 0) )  /* si vive y comienza con un no terminal */
  {
    arr[0] = 0;                 /* la eliminamos */
    for (r=1 ; r<LPROD ; r++)
        resto[r] = 0;
    resto[0] = 1;              /* inicializamos el resto */

    n = 2;
    r = 1;
    while (arr[n] != 0)
         resto[r++] = arr[n++];   /* guardo en resto la producción a partir del segundo símbolo */

    for (k=tnt[-(arr[1])].lugini ; k<tnt[-(arr[1])].lugini+tnt[-(arr[1])].numprod ; k++)
    {
      /* para cada producción en la tabla del no terminal con el que comienza la producción */
      if (tp[k][0]) /* si vive */
      {
        for (n=1 ; n<LPROD ; n++)
            nueva[n] = 0;
        nueva[0] = 1;        /* inicializo nueva */

        n = 1;
        while (tp[k][n] != 0)
             nueva[n] = tp[k][n++]; /* pongo en nueva a la producción */
         r = 1;
         while (resto[r] != 0)
              nueva[n++] = resto[r++];   /* le pego a nueva el resto */
```

```
        if (!iguales(tp,i,tnt,nueva))
           if (!agrega(tnt,i,nueva)) /* le agrego la producción al no terminal i */
                return FALSE;

      } /* del if */
    } /* del for */
    p = tnt[-(arr[1])].sig;
    while (p != NULL)       /* para cada una de las producciones de la lista del no
                               terminal con el que comienza la producción */
    {
      if (p -> q[0]) /* si vive */
      {
        for (n=1 ; n<LPROD ; n++)
            nueva[n] = 0;
        nueva[0] = 1;             /* inicializo nueva */

        n = 1;
        while (p -> q[n] != 0)
              nueva[n] = p -> q[n++];   /* pongo en nueva a la producción */
        r = 1;
        while (resto[r] != 0)
              nueva[n++] = resto[r++];       /* le pego a nueva el resto */
        if (!iguales(tp,i,tnt,nueva))
           if (!agrega(tnt,i,nueva))
               return FALSE;

      } /* del if */

      p = p -> sig;

    } /* del while */

  } /* del if */
  return TRUE;
} /* de prodGrei */


int Factizq(NOTERM tnt[],TERM tt[],int tp[MAXP][LPROD]) /* Factoriza por la izquierda y calcula nulos */
{
   if (!Left(tnt,tp))          /* factoriza por la izquierda */
   {
     printf("\n\n\n\n\nEspacio insuficiente al factorizar por la izquierda\n\n");
     printf("Oprime cualquier tecla para regresar...");
     getch();
     return FALSE;
    }
   if (!Actualiza(tnt,tt,tp,".fiz"))
        return FALSE;
   printf("%c",219);
   Empty(tnt,tp);            /* pone 1 en el campo nulo de los no
                               terminales que generan la cadena vacía */
   printf("%c",219);
   return TRUE;
} /* de Factizq */
```

```
int Left(NOTERM tnt[],int tp[MAXP][LPROD]) /* factoriza por la izquierda */
{
  int i,j,k,n,m;
  int simb;
  int hubo,mas;
  int nueva[LPROD];
  int nvo;
  int prim,ult;
  PRODS p;
  PRODS r;

  nvo = nnt;
  for (i=1 ; i<nnt ; i++)    /* para cada no terminal */
  {
    for (j=tnt[i].lugini ; j<tnt[i].lugini+tnt[i].numprod ; j++) /* para cada una de sus producciones */
    {
      hubo = FALSE;
      if (tp[j][0]) /* si esta viva */
      {
        simb = tp[j][1];  /* tomar el primer símbolo de la producción */

        for (k=j+1 ; k<tnt[i].lugini+tnt[i].numprod ; k++) /* para el resto de las producciones de i */
           if ( (tp[k][0]) && (tp[k][1] == simb) ) /* si vive y comienza con simb */
           {
               tp[k][0] = 2;  /* la marcamos con un dos en el lugar cero */
               hubo = TRUE;
           }

         if (hubo)  /* si hubo producciones marcadas */
         {
           tp[j][0] = 2; /* marcamos la original */
           for (k=tnt[i].lugini ; k<tnt[i].lugini+tnt[i].numprod ; k++)
           if (tp[k][0] == 2) /* para cada una de las producciones marcadas */
           {
              if (tp[k][1] == 0) /* si es la vacia */
              {
                for (n=1 ; n<LPROD ; n++)
                     nueva[n] = 0;
                nueva[0] = 1;
                if (!iguales(tp,nvo,tnt,nueva))
                  if (!agrega(tnt,nvo,nueva)) /* agrega a nuevo la vacía */
                       return FALSE;
              }
              else                          /* si no es la vacía */
              {
                for (n=1 ; n<LPROD ; n++)
                     nueva[n] = 0;
                nueva[0] = 1;
                n = 1;
                for (m=2 ; m<LPROD ; m++)
                     nueva[n++] = tp[k][m];
               if (!iguales(tp,nvo,tnt,nueva))
                if (!agrega(tnt,nvo,nueva))   /* agrega a nvo la prod. a partir del segundo */
                     return FALSE;
               }
```

```
             tp[k][0] = 0;
            } /* del if para cada producción marcada */

            nueva[0] = 1;
            nueva[1] = simb;
            nueva[2] = -nvo;
            for (n=3 ; n<LPROD ; n++)
                nueva[n] = 0;
            if (!iguales(tp,i,tnt,nueva))
               if (!agrega(tnt,i,nueva))
                     return FALSE;
            nvo++;
            if (nvo == MAXNT)
               return FALSE;

        } /* de if(hubo) */
      }
    }
  }

  if (nvo==nnt)
     mas = FALSE;
  else
     mas = TRUE;

  prim = nnt;
  ult = nvo;
  while(mas)
  {
    mas = FALSE;
    for (i=prim ; i<ult ; i++)
    {
      hubo = FALSE;
      p = tnt[i].sig;
      while(p != NULL)
      {
        if (p -> q[0])
        {
          simb = p -> q[1];
          r = p -> sig;
          while (r != NULL)
          {
            if ( (r -> q[0]) && (r -> q[1]==simb) )
            {
              r -> q[0] = 2;
              hubo = TRUE;
            }
            r = r -> sig;
          }
          if (hubo)
          {
            p -> q[0] = 2;
            r = p;
```

```
            while (r != NULL)
            {
              if (r -> q[0] == 2)
              {
                if (r -> q[1] == 0)
                {
                  for (n=1 ; n<LPROD ; n++)
                      nueva[n] = 0;
                  nueva[0] = 1;
                  if (!iguales(tp,nvo,tnt,nueva))
                      if (!agrega(tnt,nvo,nueva))
                          return FALSE;
                }
                else
                {
                  for (n=1 ; n<LPROD ; n++)
                      nueva[n] = 0;
                  nueva[0] = 1;
                  n = 1;
                  for (m=2 ; m<LPROD ; m++)
                      nueva[n++] = r -> q[m];
                  if (!iguales(tp,nvo,tnt,nueva))
                      if (!agrega(tnt,nvo,nueva))
                            return FALSE;
                }
               r -> q[0] = 0;
              }
               r = r -> sig;
            }
            nueva[0] = 1;
            nueva[1] = simb;
            nueva[2] = -nvo;
            for (n=3 ; n<LPROD ; n++)
                nueva[n] = 0;
            if (!iguales(tp,i,tnt,nueva))
               if (!agrega(tnt,i,nueva))
                   return FALSE;
            nvo++;
            if (nvo == MAXNT)
                return FALSE;

            hubo = FALSE;
          } /* del if(hubo) */
        }
        p = p -> sig;
      } /* del while */
    } /* del for */

      if (nvo != ult) /* si se agregaron mas */
      {
        prim = ult;
        ult = nvo;
        mas = TRUE;
      }
  } /* del while(mas) */
```

```
/* AHORA GENERAMOS LOS CARACTERES DE LOS NUEVOS SIMBOLOS REPITIENDO EL SIMBOLO CADA NUEVE */

   for (i=nnt ; i<nvo ; i++) /* para cada uno de los no terminales nuevos */

       if (indice<10)
       {
         tnt[i].simb = let;
         tnt[i].cod = -i;
         indice++;
        }
        else
        {
          let++;
          tnt[i].simb = let;
          tnt[i].cod = -i;
          indice = 2;
         }

nnt = nvo;
printf("%c",219);
return TRUE;
} /* de Left */


int Acttabla(NOTERM tnt[],int tp[MAXP][LPROD]) /* actualiza las tablas */
{
  int tpaux[MAXP][LPROD];     /* matriz temporal de producciones */
  int apaux = 1;              /* apuntador de tpaux */
  int nump;                   /* contador para el número de producciones */
  PRODS p;
  int i,j,k;

  for (i=0 ; i<MAXP ; i++)
      for (j=1 ; j<LPROD ; j++)
          tpaux[i][j] = 0;    /* inicializo tpaux */

   /* Para cada no terminal de los originales */
   for (i=1 ; i<=nnto ; i++)
   {
     nump = 0;

     /* Para cada una de sus prod. vivas en la tabla */
     for (j=tnt[i].lugini ; j<tnt[i].lugini+tnt[i].numprod ; j++)
         if (tp[j][0] == 1)
         {
           nump++;            /* incremento el número de producciones */

           for (k=0 ; k<LPROD ; k++)
                tpaux[apaux][k] = tp[j][k]; /* copio la producción a tpaux */
           apaux++;          /* incremento el apuntador de tpaux */
           if (apaux == MAXP)
           {
             printf("\n\n\n\n\nEspacio insuficiente para producciones\n\n");
             printf("Oprime cualquier tecla para regresar...");
             getch();
```

```
            return FALSE;
          }
        }

      /* Para las producciones de la lista */
      p = tnt[i].sig;
      while (p != NULL) /* mientras haya elementos en la lista */
      {
        nump++;                          /* incremento el número de producciones */
        for (k=0 ; k<LPROD ; k++)
            tpaux[apaux][k] = p -> q[k]; /* copio la producción a tpaux */
        p = p->sig;
        apaux++;                     /* incremento el apuntador de tpaux */
        if (apaux == MAXP)
        {
           printf("\n\n\n\n\nEspacio insuficiente para producciones\n\n");
           printf("Oprime cualquier tecla para regresar...");
           getch();
           return FALSE;
         }
      } /* del while */

      /* Actualizo lugini y numprod y sig de tnt */

      if (i > 1)
        tnt[i].lugini = tnt[i-1].lugini + tnt[i-1].numprod;
      tnt[i].numprod = nump;
      tnt[i].sig = NULL;
    } /* del for */

    for (i=0 ; i<MAXP ; i++)  /* actualizo tabla de producciones */
        for (j=0 ; j<LPROD ; j++)
            tp[i][j] = tpaux[i][j];

     /* Para cada terminal de los nuevos */
     for (i=nnto+1 ; i<nnt ; i++)
     {
       nump = 0;
       /* Para las producciones de la lista */
       p = tnt[i].sig;
       while (p != NULL) /* mientras haya elementos en la lista */
       {
         nump++;                          /* incremento el número de producciones */
         for (k=0 ; k<LPROD ; k++)
             tp[apaux][k] = p -> q[k]; /* copio la producción a tp */
         p = p->sig;
         apaux++;
         if (apaux == MAXP)
         {
              printf("\n\n\n\n\nEspacio insuficiente para producciones\n\n");
              printf("Oprime cualquier tecla para regresar...");
              getch();
              return FALSE;
         }
       } /* del while */
```

```
      /* Actualizo lugini y numprod y sig de tnt */

      tnt[i].lugini = tnt[i-1].lugini + tnt[i-1].numprod;
      tnt[i].numprod = nump;
      tnt[i].sig = NULL;

    } /* del for */

    np = tnt[nnt-1].lugini+tnt[nnt-1].numprod;

    tp[0][0] = 1;                          /* incluye producción cero */
    tp[0][1] = codini;
    return TRUE;
} /* de Actabla */


Nuevo(FILE *aux,NOTERM tnt[],int tp[MAXP][LPROD],TERM tt[])  /* Genera un nuevo archivo con la
                                                                   gramática de las tablas */
{
  int i,j;
  for (i=1 ; i<nnt ; i++) /* para cada no terminal */

    /* Para cada una de sus producciones */
    for (j=tnt[i].lugini ; j<tnt[i].lugini+tnt[i].numprod ; j++) /* para cada producción */
        if (tp[j][0]) /* si esta viva */
           copia(aux,i,tp[j],tnt,tt);

} /* de Nuevo */


copia(FILE *aux,int i,int t[LPROD],NOTERM tnt[],TERM tt[]) /* Este es utilizado por nuevo para
                                                                escribir el archivo */
{
  int k;

  /* escribe el lado izquierdo */

    fprintf(aux,"%c",tnt[i].simb);
    if (i > NTO)
    {
      if ( (i-NTO)%9 == 0)
          fprintf(aux,"%d",9);
      else
         fprintf(aux,"%d",(i-NTO)%9);
     }
     fprintf(aux," ->");

     if (t[1] == 0)  /* si es la vacía */
        fprintf(aux," %c",238);
     k = 1;
     while (t[k] != 0) /* mientras haya elementos en la producción */
     {
      if (t[k] < 0)     /* escribe el lado derecho */
      {
         fprintf(aux," %c",tnt[-(t[k])].simb);
```

```
          if ( -(t[k]) > NTO)
          {
            if ( (-t[k]-NTO)%9 == 0 )
               fprintf(aux,"%d",9);
            else
               fprintf(aux,"%d",(-t[k]-NTO)%9);
          }
       }
       else
            fprintf(aux," %c",tt[t[k]].simb);
        k++;
       } /* del while */
  fprintf(aux,"\n");
} /* de copia */


Archnt(FILE *aux,NOTERM tnt[]) /* Genera el archivo que leera el programa PARSE con
                                  la informacion de la tabla de no terminales */
{
  int i;

  fprintf(aux,"%d\n",codini);
  fprintf(aux,"%d\n",NTO);
  for (i=1 ; i<nnt ; i++)
  {
     fprintf(aux,"%c ",tnt[i].simb);
     fprintf(aux,"%d ",tnt[i].cod);
     fprintf(aux,"%d ",tnt[i].lugini);
     fprintf(aux,"%d ",tnt[i].numprod);
     fprintf(aux,"%d\n",tnt[i].nuto);
  }
} /* de Archnt */



Archterm(FILE *aux,TERM tt[]) /* Genera el archivo que leera PARSE con
                                 la informacion de la tabla de terminales */
{
  int i;

  for (i=1 ; i<nt ; i++)
  {
      fprintf(aux,"%c ",tt[i].simb);
      fprintf(aux,"%d ",tt[i].cod);
      fprintf(aux,"%d\n",tt[i].alcanza);
  }
} /* de Archterm */
```

```
Archprod(FILE *aux,int tp[MAXP][LPROD]) /* Genera el archivo que leera PARSE con
                                           la informacion de la tabla de producciones */
{
  int i,j;

  for (i=0 ; i<np ; i++)
  {
      for (j=0 ; j<LPROD ; j++)
          fprintf(aux,"%d ",tp[i][j]);
      fprintf(aux,"\n");
   }
} /* de Archprod */


Archtablas(NOTERM tnt[],TERM tt[],int tp[MAXP][LPROD]) /* genera los archivos que lee PARSE */
{
   FILE *sal; /* para los archivos de las nuevas gramáticas */

   sal = fopen("archnt.dat","w"); /* genera el archivo de la tabla de no terminales */
   Archnt(sal,tnt);
   fclose(sal);
   sal = fopen("archterm.dat","w"); /* genera el archivo de la tabla de terminales */
   Archterm(sal,tt);
   fclose(sal);
   sal = fopen("archprod.dat","w"); /* genera el archivo de la tabla de producciones */
   Archprod(sal,tp);
   fclose(sal);
   printf("%c",219);
} /* de Archtablas */



int Menu(int g) /* despliega menús */
{
   int acaba;
   char opcion;

   Diagfnorm(g);      /* despliega mensaje de si la gramática fue recursiva o no */
   do{
       cuadro_opciones(g);
       if (g)
          acaba = lee_opcion("OoVvUuAaCcGgIiRr",&opcion,'R','r');
       else
          acaba = lee_opcion("OoVvUuAaIiRr",&opcion,'R','r');

       if (acaba)
          return TRUE;
       else
       {
          clrscr();
          switch (opcion){
               case 'o':
               case 'O': printf("          Gramática original\n\n");
                         escribe(org);
                         break;
```

```
            case 'v':
            case 'V': printf("          Gramática sin producciones vacías\n\n");
                      escribe(vac);
                      break;
            case 'u':
            case 'U': printf("          Gramática sin producciones unitarias\n\n");
                      escribe("unt);
                      break;
            case 'a':
            case 'A': printf("          Gramática sin símbolos inútiles\n\n");
                      escribe(uti);
                      break;
            case 'c':
            case 'C': printf("          Gramática en Forma Normal de Chomsky\n\n");
                      escribe("fnc);
                      break;
            case 'g':
            case 'G': printf("          Gramática en Forma Normal de Greibach\n\n");
                      escribe(fng);
                      break;
            case 'i':
            case 'I': printf("          Gramática factorizada por la izquierda\n\n");
                      escribe(fiz);
                      break;

        }  /* del switch */
      } /* del else */

    }  while(!acaba);
} /* de menu */


Diagfnorm(int g) /* despliega mensaje de si la gramática fue recursiva o no */
{
  int i;

  clrscr();
  gotoxy(10,5);
  printf(" Z");
  for (i=1 ; i<55 ; i++)
     printf("-");
  printf("\\n");
  if (g)
  {
    printf("        |                                                    |\n");
    printf("        | La gramática resultó ser recursiva por la izquierda, |\n");
    printf("        |                                                    |\n");
    printf("        | por lo tanto, fue transformada a:                  |\n");
    printf("        |                                                    |\n");
    printf("        |                1. Forma Normal de Chomsky.         |\n");
    printf("        |                2. Forma Normal de Greibach.        |\n");
    printf("        |                                                    |\n");
    printf("        | y de esta forma la recursividad por la izquierda   |\n");
    printf("        | fue eliminada.                                     |\n");
  }
```

```
 else
 {
   printf("          |                                              |\n");
   printf("          |      La gramática no es recursiva por la izquierda,   |\n");
   printf("          |      por  lo  tanto, no  fue  necesario  hacer las    |\n");
   printf("          |      transformaciones a formas normales de Chomsky    |\n");
   printf("          |      y de Greibach.                                  |\n");
 }
 printf("          |                                              |\n");
 printf("          \");
 for (i=1 ; i<55 ; i++)
     printf("-");
 printf("/\n\n\n");
 printf("                         Oprima cualquier tecla para continuar...");
 getch();
} /* de Diagfnorm */

despliega(FILE *arch) /* despliega en pantalla el archivo arch */
{
   char cad[30];
   char c;
   int cont=0;
   int interr = FALSE;

   while ((fgets(cad,29,arch)) != NULL && (cont < 17) && (interr==FALSE))
   {
         printf("                %s",cad);
         cont++;
         if ( (cont == 16) && (fgets(cad,29,arch) != NULL) )
         {
           printf("\n\n\n              [C] Continúa con el despliegue.\n");
           printf("              [R] Regresa.\n\n");
           printf("                                  Selecciona la opción ");
         do{
             c = getch();
            } while ( (c!='C') && (c != 'c') && (c!='R') && (c != 'r'));
           printf("\n\n");
           if ( (c == 'R') || (c=='r') )
              interr = TRUE;
           else
           {
             cont = 0;
             printf("              %s",cad);
           }
         }
    }
   if (interr == FALSE)
   {
     printf(" \n\n\n\n              [R] Regresa.");
     do{
         c = getch();
        } while ( (c!='R') && (c != 'r') );
   }
   clrscr();
} /* de despliega */
```

```
cuadro_opciones(int g) /* despliega cuadro de opciones */
{
    int i;

    clrscr();
    printf("                              M  E  N  U   \n");
    printf("/");
    for (i=1 ; i<62 ; i++)
        printf("-");
    printf("\\n");
    printf("|                                                            |\n");
    printf("|       [O] Despliega gramática original                     |\n");
    printf("|                                                            |\n");
    printf("|       [V] Despliega gramática sin producciones vacias      |\n");
    printf("|                                                            |\n");
    printf("|       [U] Despliega gramática sin producciones unitarias   |\n");
    printf("|                                                            |\n");
    printf("|       [A] Despliega gramática sin símbolos inútiles        |\n");
    printf("|                                                            |\n");
    if (g)
        {
          printf("|       [C] Despliega gramática en Forma Normal de Chomsky   |\n");
          printf("|                                                            |\n");
          printf("|       [G] Despliega gramatica en Forma Normal de Greibach  |\n");
          printf("|                                                            |\n");
        }
    printf("|       [I] Despliega gramática factorizada por la izquierda |\n");
    printf("|                                                            |\n");
    printf("|       [R] Regresa al Menú Principal                        |\n");
    printf("|                                                            |\n");
    printf("\");
    for (i=1 ; i<62 ; i++)
        printf("-");
    printf("/\n\n");
    printf("  Selecciona la opción  ");
} /* de cuadro_opciones */



int lee_opcion(char *cadena,char* op,char fin,char FIN) /* lee opciones y regresa TRUE
                                                           si la opcion en fin o FIN */
{
  do{
      *op = getch();
     } while (strchr(cadena,*op) == NULL);
  if ( (*op == fin) || (*op == FIN) )
      return TRUE;
  else
      return FALSE;
} /* de lee_opciones */
```

```
escribe(char *archivo) /* despliega en pantalla el contenido de archivo */
{
    FILE *sal;

    sal = fopen(archivo,"r");
    despliega(sal);
    fclose(sal);
} /* de escribe */
```

```
/* Programa 2 : PARSE.C
   Construye el analizador sintactico. */

# include <stdio.h>
# include <string.h>
# include <conio.h>
# include <alloc.h>
# include "reconoce.h"

/********** DECLARACIONES GLOBALES *************/

int nnt;      /* lugar disponible en la tabla de no-terminales */
int nt;       /* lugar disponible en la tabla de terminales */
int np;       /* lugar disponible en la tabla de producciones */
int codini;   /* código del símbolo inicial */
int NTO;      /* número de no terminales originales */


/************  COMIENZA EL PROGRAMA PRINCIPAL ***************/

main()
{
   FILE *archnt, *archt, *archp;       /* para los archivos con las tablas */
   NOTERM  tabnoterm[MAXNT];           /* tabla de símbolos no-terminales */
   TERM tabterm[MAXT];                 /* tabla de símbolos terminales */
   int tabprod[MAXP][LPROD];           /* tabla de producciones */
   int tablaF[TOTS][TOTS];             /* matriz para la relación F */
   int FIRST[MAXP][MAXT];              /* tabla para el conjunto FIRST */
   int FOLLOW[MAXNT][MAXT];            /* tabla para el conjunto FOLLOW */
   int tablaM[MAXNT][MAXT];            /* tabla de acción del autómata */


/*******   CHECA QUE EXISTAN LOS ARCHIVOS DE ENTRADA Y ABRELOS PARA LECTURA  *******/

   if ( (archnt = fopen("archnt.dat","r")) == NULL) /* checa que exista el archivo de no terminales */
   {
     rollo_arch();
     exit();
   }
   if ( (archt = fopen("archterm.dat","r")) == NULL) /* checa que exista el archivo de terminales */
   {
     rollo_arch();
     exit();
   }
   if ( (archp = fopen("archprod.dat","r")) == NULL) /* checa que exista el archivo de producciones */
   {
     rollo_arch();
     exit();
   }
```

```
/********    COMIENZA LA LLAMADA A SUBRUTINAS    *******/

   Presentacion();      /* despliega presentación del programa */
   Cuadro();            /* despliega cuadro de espera */
   Constab(archnt,archt,archp,tabnoterm,tabterm,tabprod); /* construye las tablas de no terminales,
                                                              terminales y producciones */
   fclose(archnt);
   fclose(archt);
   fclose(archp);

   Relaciones(tablaF,FIRST,FOLLOW,tabnoterm,tabprod); /* construye las tablas de relacion F,
                                                          FIRST y FOLLOW */

   if (!constM (tablaM,FIRST,FOLLOW,tabnoterm,tabprod,tabterm)) /* construye la tabla de
                                                                   accion del autómata */
      exit();

   Reconoce (tabnoterm,tabterm,tabprod,tablaM); /* realiza reconocimiento */

} /***************** TERMINA EL PROGRAMA PRINCIPAL *******************/



/************  COMIENZA LA DECLARACION DE SUBRUTINAS ************/

rollo_arch() /* despliega mensaje de cual archivo no encontró */
{
     clrscr();
     gotoxy(10,5);
     printf("No se puede entrar al proceso de análisis sintáctico.);
     gotoxy(10,7);
     printf("No se ha realizado el proceso de transformación o dicho");
     gotoxy(10,9);
     printf("proceso no llego al último paso.");
     gotoxy(20,20);
     printf("                  Oprime cualquier tecla para regresar...");
     getch();
} /* de rollo_arch */


Presentacion() /* despliega presentación del programa */
{
  int i;
  char opcion;

  clrscr();
  gotoxy(10,5);
  printf("/");
  for (i= 1 ; i<60 ; i++)
     printf("-");
  printf("\\n");
  printf("         |                                                    |\n");
  printf("         |          A N A L I S I S   S I N T A C T I C O    |\n");
  printf("         |                                                    |\n");
```

```
  printf("          \");
  for (i= 1 ; i<60 ; i++)
      printf("-");
  printf("/\n\n\n");
  printf("          [D] Despliega los pasos del proceso de análisis sintáctico.\n\n");
  printf("          [C] Comienza con el proceso de análisis sintáctico.\n\n\n");
  gotoxy(42,23);
  printf("          Selecciona la opción  ");
  if ( lee_opcion("DdCc",&opcion,'D','d') )
      Explica();

} /* de Presentacion */



Explica() /* despliega explicación de lo que hace el programa */
{
  int i;

  clrscr();
  printf("        /");
  for (i=1 ; i<60 ; i++)
      printf("-");
  printf("\\n");
  printf("        |                                                          |\n");
  printf("        |       PROCESO  DEL  ANALISIS  SINTACTICO.                |\n");
  printf("        |                                                          |\n");
  printf("        | Construye la tabla de acción del autómata.               |\n");
  printf("        |                                                          |\n");
  printf("        | - Si la tabla de acción del autómata esta multidefinida, |\n");
  printf("        |   nos indica:                                            |\n");
  printf("        |             1. Para cuáles símbolos.                     |\n");
  printf("        |                                                          |\n");
  printf("        |             2. Por cuales producciones.                  |\n");
  printf("        |    y termina.                                            |\n");
  printf("        |                                                          |\n");
  printf("        |                                                          |\n");
  printf("        | - Si la tabla de acción del automata está definida       |\n");
  printf("        |   unívocamente para todo par de símbolos:                |\n");
  printf("        |                                                          |\n");
  printf("        |              1. Solicita la cadena a ser analizada.      |\n");
  printf("        |                                                          |\n");
  printf("        |              2. Acepta o rechaza la cadena.              |\n");
  printf("        |                                                          |\n");
  printf("        \");
  for (i=1 ; i<60 ; i++)
    printf("-");
  printf("/\n\n\n");
  printf("                     Oprime cualquier tecla para comenzar el proceso...");
  getch();
} /* de Explica */
```

```
Cuadro() /* despliega cuadro de espera */
{
  int i;

  clrscr();
  gotoxy(17,7)
  printf("/");
  for (i=1 ; i<40 ; i++)
      printf("-");
  printf("\\n");
  printf("                |                                      |\n");
  printf("                |         Análisis Sintáctico          |\n");
  printf("                |                                      |\n");
  printf("                |                                      |\n");
  printf("                |          T R A B A J A N D O ...     |\n");
  printf("                |                                      |\n");
  printf("                |                                      |\n");
  printf("                |               %c",177);
  for (i=1 ; i<6 ; i++)
       printf("%c",177);
  printf("                 |\n");
  printf("                |                                      |\n");
  printf("                \");
  for (i=1 ; i<40 ; i++)
        printf("-");
  printf("/\n");
  gotoxy(33,15);
} /* de Cuadro */


Constab(FILE *ntt,FILE *t,FILE *p,NOTERM tnt[],TERM tt[],int tp[MAXP][LPROD]) /* construye las tablas */
{
   int i;

   printf("%c",219);
   /* LEE CODINI,NTO Y CONSTRUYE LA TABLA DE NO TERMINALES */
   fscanf(ntt,"%d\n",&codini);
   fscanf(ntt,"%d\n",&NTO);
   nnt = 1;
   while ( (fscanf(ntt,"%c %d %d %d %d\n",&tnt[nnt].simb,
                  &tnt[nnt].cod,&tnt[nnt].lugini,
                  &tnt[nnt].numprod,&tnt[nnt].nulo) != EOF))
        nnt++;


   /* CONSTRUYE LA TABLA DE TERMINALES */
   nt = 1;
   while ( (fscanf(t,"%c %d %d\n",&tt[nt].simb,&tt[nt].cod,&tt[nt].alcanza) != EOF))
          nt++;

   /* CONSTRUYE LA TABLA DE PRODUCCIONES */
   np = 0;
   while ( (fscanf(p,"%d ",&tp[np][0]) != EOF))
   {
        for (i=1 ; i<LPROD ; i++)
```

```
            fscanf(p,"%d ",&tp[np][i]);
         fscanf(p,"\n");
         np++;
    }
    printf("%c",219);
} /* de Constab */


Relaciones(int tF[TOTS][TOTS],int FI[MAXP][MAXNT],int FO[MAXNT][MAXT],
           NOTERM tnt[],int tp[MAXP][LPROD])
                                              /* construye las tablas de relacion */
{
   constF(tF,tnt,tp);        /* construye la tabla de relacion F */
   constFI (FI,tp,tF,tnt); /* construye el conjunto FIRST */
   constFO(FO,tnt,tp,tF);  /* construye el conjunto FOLLOW */
} /* de relaciones */


constF(int tF[TOTS][TOTS],NOTERM  tnt[],int tp[MAXP][LPROD])
/* construye la matriz F y su cerradura, UFX si y solo si existe
   una producción U -> X, UF+X es la cerradura de F */
{
  int simb;       /* códigos de los símbolos en las producciones */
  int ind;        /* para el lugar de cada símbolo en las columnas de F */
  int indprod;    /* para recorrer las producciones */
  int totsimb;    /* número total de símbolos */
  int i,j,k;

  totsimb = nnt+nt-2;
  for (i=0 ; i<=totsimb ; i++)          /* inicializo la tablaF en ceros */
      for (j=0 ; j<=totsimb ; j++)
          tF[i][j] = 0;

   tF[0][-(codini)] = 1;    /* renglón del símbolo agregado */

   for (i=1 ; i<nnt ; i++)    /* para cada no terminal */
       /* para cada una de sus producciones vivas que no sean la vacía */
      for (k=tnt[i].lugini ; k<tnt[i].lugini+tnt[i].numprod ; k++)
      if ((tp[k][0]) && (tp[k][1] != 0) )
      {
           indprod = 1;
           simb = tp[k][indprod++];  /* primer elemento de la producción */
           if (simb<0)               /* calcula ind */
                ind = -(simb);
           else
                ind = nnt+simb-1;
           tF[i][ind] = 1;
           while ( (simb<0) && (tnt[-(simb)].nulo) ) /* mientras los símbolos de la producción sean no
                                                     terminales nulos avanza y pon los unos de F */
           {
                simb = tp[k][indprod++];
                if (simb !=0)
                {
                     if (simb<0)
                          ind = -(simb);
```

```
                else
                    ind = nnt+simb-1;  /* calcula ind */

                tF[i][ind] = 1;
            }
        } /* del while */
    } /* del if */

  /* Construimos la cerradura con Warshall */

  for (k=0 ; k <= totsimb ; k++)
      for (i=0 ; i <= totsimb ; i++)
          for (j=0 ; j <= totsimb ; j++)
              tF[i][j] = tF[i][j] || (tF[i][k] && tF[k][j]);

   printf("%c",219);
}     /* de constF */


int estafi(int tfi[MAXP][MAXT],int sim,int reng) /* regresa cierto si sim está en el renglón
                                                  reng de tfi esta rutina es llamada en constFI() */
{
  int encontro;
  int i;

  encontro = FALSE;
  i = 1;
  while ( (!encontro) && (tfi[reng][i] != 0) ) /* mientras haya símbolos en el renglon
                                                 y no lo hayas encontrado */
  {
       if (sim == tfi[reng][i++])
           encontro = TRUE;          /* si lo encuentra */
  }
  if (encontro)
     return TRUE;
  else
     return FALSE;

} /* de estafi */


constFI (int tfirst[MAXP][MAXT],int tp[MAXP][LPROD],
         int tf[TOTS][TOTS],NOTERM  tnt[])           /* construye la tabla First, una matriz donde
                                                        cada renglon es para una producción */
{
   int i,j,k;
   int prim;   /* primer elemento de la producción */
   int col;    /* indice de las columnas de la matriz */
   int vacia;  /* booleana que es verdadera si FIRST tiene a la vacia */

   for (i=0 ; i < MAXP ; i++)       /* inicializo en ceros first */
       for (j=0 ; j < MAXT ; j++)
           tfirst[i][j] = 0;
```

```
for (i=0 ; i < np ; i++)      /* para cada producción */
if (tp[i][0]) /* si vive */
{
    vacia = FALSE;
    col = 1;
    k = 1;
    prim = tp[i][k++];         /* primer elemento de la producción i */

    if(prim > 0)                  /* si es terminal lo metemos a first */
       tfirst[i][col++] = prim;
    else

      if (prim == 0)          /* si es la vacía */
         tfirst[i][0] = 1;  /* ponemos un uno en la primera columna, la producciones que tengan en
                               first a la vacía tendrán un uno en la columna 0 */

      else  /* si es no terminal */
      {
        for (j=nnt ; j < nnt+nt-1; j++) /* lo buscamos en tablaF */
            if (tf[-(prim)][j])       /* si hay un uno */
               if (!(estafi(tfirst,j-nnt+1,i))) /* y todavía no esta */
                  tfirst[i][col++] = j-nnt+1;  /* lo metemos a first */

        while ( (prim<0) && (tnt[-(prim)].nulo)) /* mientras sea un no terminal
                                                    con uno en nulo */
        {
          prim = tp[i][k++]; /* tomamos el siguiente de la producción */
          if ( (prim > 0) && (!(estafi(tfirst,prim,i))) )
                      /*si es terminal y no esta todavía mételo en first */
             tfirst[i][col++] = prim;
          else
              if (prim == 0)  /* si es la vacía, todos llegan a la vacía */
                 vacia = TRUE;
              else             /* si es no terminal */
                  for (j=nnt ; j < nnt+nt-1 ; j++) /* lo buscamos en tablaF */
                      if ( (tf[-(prim)][j]) && (!(estafi(tfirst,j-nnt+1,i))) )
                                                   /* si hay un uno y no está */
                         tfirst[i][col++] = j-nnt+1; /* lo metemos a first */
        } /* del while */

        if (vacia)
           tfirst[i][0] = 1;  /* si todos van a la vacía pon uno en la primera columna,
                                 la vacía pertenece al conjunto first de esa producción */

      } /* del else */

  } /* del if */
  printf("%c",219);

} /* de constfI */
```

```
constB(int tB[TOTS][TOTS],NOTERM  tnt[],int tp[MAXP][LPROD]) /* construye la matriz de relacion B */
{
  int i,j;
  int totsimb;           /* número total de simbolos */
  int sig,sigaux;        /* para ir avanzando en las producciones */
  int simb1,simb2,aux;  /* para los símbolos de las producciones */
  int ind1,ind2,indaux; /* para las columnas de B de los símbolos de las producciones */

  totsimb = nnt+nt-2;

  for (i=0 ; i<=totsimb ; i++)        /* inicializo la tablaB en ceros */
      for (j=0 ; j<=totsimb ; j++)
          tB[i][j] = 0;
  for (i=0 ; i<np ; i++)
  {
      if ( (tp[i][0]) && (tp[i][1] != 0) ) /* para cada producción viva distinta a la vacía */
      {
        sig = 2;
        simb1 = tp[i][1];      /* primer símbolo de la producción */
        simb2 = tp[i][sig++]; /* segundo símbolo de la producción */
        while (simb2 !=0)      /* mientras haya símbolos en la producción */
        {
          if (simb1<0)
             ind1 = -(simb1);
          else
             ind1 = nnt+simb1-1;  /* calcula ind1 */

          if (simb2<0)
             ind2 = -(simb2);
          else
             ind2 = nnt+simb2-1;  /* calcula ind2 */

          tB[ind1][ind2] = 1;     /* pon el 1 de B */
          aux = simb2;

          while ( (aux<0) && (tnt[-(aux)].nulo) ) /* mientras encuentres no terminales nulos */
          {
            sigaux = sig;
            aux = tp[i][sigaux++];
            if (aux!=0)    /* si no se ha acabado la producción */
            {
              if (aux<0)
                 indaux = -(aux);
              else
                indaux = nnt+aux-1;  /* calcula indaux */
              tB[ind1][indaux] = 1;  /* pon el 1 de B */
            }
          }
          simb1 = simb2;
          simb2 = tp[i][sig++]; /* avanza en la produccion */
        } /* del while */
      } /* del if */
    } /* del for */
  printf("%c",219);
} /* de constB */
```

```
constL(int tL[TOTS][TOTS],NOTERM  tnt[],int tp[MAXP][LPROD]) /* construye la matriz de relacion L */
{
  int i,j,k;
  int totsimb;   /* número total de símbolos */
  int apun;      /* para ir avanzando en las producciones */
  int simb;      /* para los códigos de las producciones */
  int ind;       /* columnas de L que corresponden a los símbolos */


  totsimb = nnt+nt-2;
  for (i=0 ; i<=totsimb ; i++)         /* inicializo la tablaL en ceros */
      for (j=0 ; j<=totsimb ; j++)
          tL[i][j] = 0;

  for (i=1 ; i<nnt ; i++)    /* para cada no terminal */
     for (j=tnt[i].lugini ; j<tnt[i].lugini+tnt[i].numprod ; j++)
        if ((tp[j][0]) && (tp[j][1] != 0)) /* para cada producción viva que no sea la vacía */
        {
           apun = 1;
           while (tp[j][apun] != 0)
                 apun++;
           apun--;              /* apun apunta al último símbolo de la producion */
           simb = tp[j][apun--]; /* simb es el último símbolo de la producción */

           if (simb<0)
              ind = -(simb);
           else
              ind = nnt+simb-1;  /* calcula ind */

           tL[ind][i] = 1;  /* pon el 1 de L */

           while ( (simb<0) && (tnt[-(simb)].nulo) && (apun>1) ) /* mientras encuentres en la produccion
                                                                     símbolos no terminales nulos, avanza de
                                                                     atras hacia adelante */
           {
             apun--;
             simb = tp[j][apun];
             if (simb<0)
                ind = -(simb);
             else
                ind = nnt+simb-1;  /* calcula ind */
             tL[ind][i] = 1;      /* pon el 1 de L */
           }
        } /* del if */


        /* Construimos la cerradura con Warshall */

       for (k=0 ; k <= totsimb ; k++)
           for (i=0 ; i <= totsimb ; i++)
               for (j=0 ; j <= totsimb ; j++)
                  tL[i][j] = tL[i][j] || (tL[i][k] && tL[k][j]);
```

```
        /* Construimos L* ponien 1's en la diagonal de L+ */

        for (i=0 ; i <= totsimb ; i++)
            for (j=0 ; j <= totsimb ; j++)
                if (i == j)
                   tL[i][j] = 1;
    printf("%c",219);
} /* de constL */


int estafo(tfo,sim,reng) /* regresa cierto si sim está en el renglón reng de tfo,
                            esta rutina es llamada en constFO() */

int tfo[MAXNT][MAXT];
int sim,reng;

{
  int encontro;
  int i;

  encontro = FALSE;
  i = 1;
  while ( (!encontro) && (tfo[reng][i] != 0) ) /* mientras no lo hayas encontrado y haya símbolos */
  {
    if (sim == tfo[reng][i++]) /* si lo encuentras */
       encontro = TRUE;
  }
  if (encontro)
     return(TRUE);
  else
     return(FALSE);

} /* de estafo */



constFO(int tfollow[MAXNT][MAXT],NOTERM  tnt[],int tp[MAXP][LPROD],int tf[TOTS][TOTS])

/* Construye la tabla FOLLOW donde cada renglos es para un no terminal */

{
   int totsimb;                         /* número total de símbolos */
   int tablaB[TOTS][TOTS];              /* tabla para la matriz de relacion B */
   int tablaL[TOTS][TOTS];              /* tabla para la matriz de relacion L */
   int apf;                             /* para avanzar en las columnas de follow */
   int i,j,k,l;

   FILE *prueba;

   totsimb = nnt+nt-2;
   constB(tablaB,tnt,tp); /* construye la matriz de relacion B */

   constL(tablaL,tnt,tp); /* construye la matriz de relacion L */
```

```
/* Construimos F* ponien 1's en la diagonal de F+ */

for (i=0 ; i <= totsimb ; i++)
    tf[i][i] = 1;

/* inicializamos Follow */
  for (i=0 ; i<=totsimb ; i++)          /* inicializo la tfollow en ceros*/
     for (j=0 ; j<=totsimb ; j++)
        tfollow[i][j] = 0;


/* construimos la tabla follow */
  for (i=1 ; i<nnt ; i++)  /* para cada no terminal */
  {
   apf = 1;
   for (j=1 ; j<=totsimb ; j++) /* para cada columna ce no terminal de L* */
      if (tablaL[i][j])    /* si i L* j */
         for (k=1 ; k<=totsimb ; k++) /* para cada columna de no terminales de B */
            if (tablaB[j][k]) /* si j B k */
               for (l=nnt ; l<=totsimb ; l++) /* para cada columna de no term. de F* */
                  if (tf[k][l]) /* si k F* l */
                     if ( !(estafo(tfollow,l-nnt+1,i)) )
                        tfollow[i][apf++] = l-nnt+1; /* si no esta en follow metelo */
  } /* del for */

 printf("%c",219);
}   /* de constFO */


int constM(tM,tfi,tfo,tnt,tp,tt) /* construye la tabla ce accion del autómata */

int tM[MAXNT][MAXT];
int tfi[MAXP][MAXT];
int tfo[MAXNT][MAXT];
NOTERM  tnt[];
int tp[MAXP][LPROD];
TERM tt[];

{
 int i,j,k;
 int simb; /* para guardar el símbolo analizado */

 for (i=0 ; i<nnt ; i++)
     for (j=0 ; j<nt ; j++)
         tM[i][j] = 0;          /* inicializo tM en ceros */

 for (i=1 ; i<nnt ; i++) /* para cada no terminal */
    for (j=tnt[i].lugini; j<tnt[i].lugini+tnt[i].numprod ; j++)
         if (tp[j][0]) /* para cada producción viva */
         {
           k=1;
           while ( (simb=tfi[j][k++]) != 0 ) /* para todo elemento de la producción en first */
           {
             if (tM[i][simb] == 0)
                tM[i][simb] = j; /* si la entrada de M esta vacía pon el número de la producción */
```

```
              else /* si la tabla esta multidefinida */
              {
                NO_LL1(tnt,tt,tp,tM,i,j,simb); /* reporta que la gramática no es LL(1) */
                return FALSE;
              }
             }  /* del while */

            k=1;
            while ( (simb=tfo[i][k++]) != 0 ) /* para todo elemento en follow del no terminal */
            {
              if (tfi[j][0]) /* si la vacía esta en first de la producción */
                 if (tM[i][simb] == 0)
                    tM[i][simb] = j; /* si la entrada de M esta vacía pon el número de producción */
              else /* si la tabla esta multidefinida */
              {
                NO_LL1(tnt,tt,tp,tM,i,j,simb); /* reporta que la gramática no es LL(1) */
                return FALSE;
              }

             }  /* del while */

          } /* del if */

  printf("%c",219);
  return TRUE;

} /* de constM */


SI_LL1()  /* reporta que la gramática si fue LL(1) */
{
 int i;
   clrscr();
   gotoxy(10,5);
   printf("/");
   for (i=1 ; i<59 ; i++)
        printf("-");
   printf("\\n");
   printf("          |                                                          |\n");
   printf("          |  La gramática es LL(1). Por lo tanto, sí fue posible     |\n");
   printf("          |                                                          |\n");
   printf("          |  la construcción de la tabla de acción del autómata.     |\n");
   printf("          |                                                          |\n");
   printf("          \");
   for (i=1 ; i<59 ; i++)
       printf("-");
   printf("/\n");
   printf("\n\n\n\n\n                    Oprime cualquier tecla para continuar...");
   getch();
} /* de SI_LL1 */
```

```
int Codifica(TERM tt[],char cad[LCAD],int cc[LCAD]) /* traduce la cadena de entrada a sus código
                                                         y los guarda en cc */
{
  int i; /* para recorrer la cadena de entrada */
  int j;   /* para buscar el símbolo en la tabla de terminales */
  char c;  /* para ir leyendo los símbolos de la cadena de entrada */

  for (i=0 ; i<LCAD ; i++)
      cc[i] = 0;

  i = 0;
  while ( i < (strlen(cad)) ) /* mientras haya símbolos en la cadena de entrada */
  {
    c = cad[i];
    tt[nt].simb = c;
    for (j=0 ; tt[j].simb != c ; j++);
    if ( (j==nt) || (!tt[j].alcanza) )   /* si el símbolo no esta en la tabla de terminales */
    {
      No_terminal(c);
      return FALSE;
    }
    else
        cc[i++] = tt[j].cod; /* si el símbolo es terminal guarda su código en cadcod */
  } /* del while */
  return TRUE;
} /* de Codifica */



No_terminal(char c) /* reporta que la cadena de entrada tiene algun símbolo que no es terminal */
{
  int in;
  char opcion;

  gotoxy(16,10);
  printf("/");
  for (in=1 ; in<30 ; in++)
      printf("-");
  printf("\\n");
  printf("                   |                             |\n");
  printf("                   | El símbolo '%c' de tu cadena |\n",c);
  printf("                   | no es símbolo terminal,     |\n");
  printf("                   | o es inalcanzable.          |\n");
  printf("                   |                             |\n");
  printf("                   \");
  for (in=1 ; in<30 ; in++)
        printf("-");
  printf("/");
} /* de No_terminal */
```

```
int Parse(int tM[MAXNT][MAXT],int tp[MAXP][LPROD],int cc[LCAD],int opt[100]) /* reconocimiento de cadena */
{
  STACK pila;          /* stack del autómata */
  int sigue = FALSE;
  int k,j;
  int i;              /* para avanzar en cadcod */
  int t;              /* para el símbolo que entra al stack */
  int cod;            /* para guardar los códigos de cadcod */
  int codtope;        /* código del símbolo en el tope del stack */


      pila.espacio[0] = 0;          /* inicializa */
      pila.espacio[1] = codini;
      pila.apunt = 1;
      opt[0] = 1;
      for (i=1 ; i<30 ; i++)
            opt[i] = 0;

      i = 0;
      cod = cc[i++];      /* toma el primer código */
      if (cod == 0)       /* si no hay tal */
             return -1;
      else                /* si si hay cadena prosigue */
            sigue = TRUE;

      while(sigue)
      {
        if (TOPE(PILA) == cod) /* si el símbolo de entrada caza con el tope del stack */
        {
          if (cod == 0)   /* si ambos son cero, i.e. fin de cadena y stack vacio */
          {
            sigue = FALSE;
            return -2;
          }
          else              /* si no son ambos cero */
          {
            POP(PILA);          /* saca el tope del stack */
            cod = cc[i++];      /* avanza la cadena de entrada */
          }
        } /* del if */

        else    /* si el símbolo de entrada no caza con el tope del stack */
        {
          if (TOPE(PILA) >= 0) /* si el tope de la pila es un terminal */
          {
             sigue = FALSE;
             return i;
          }
          else   /* si el tope del stack es un no terminal */
          {
           codtope = -(TOPE(PILA));
           if ( (tM[codtope][cod]) != 0 )  /* si la entrada de M no es de error */
           {
             POP(PILA);    /* saca el símbolo del tope del stack */
```

```
          for (j=0 ; (tp[tM[codtope][cod]][j]) != 0 ; j++);
          for (k=j-1 ; k>0 ; k--)
          {
            t = tp[tM[codtope][cod]][k]; /* mete la producción que esta en la entrada */
            PUSH(PILA,t);                /* de M de atras hacia adelante */
          }
          opt[(opt[0])++] = tM[codtope][cod]; /* registra la producción aplicada */
        } /* del if */

        else /* si la entrada de M es de error */
        {
           sigue = FALSE;
           return i;
         }
        } /* del else el tope no es terminal */
      } /* del else no cazan */
     } /* del while */
  } /* de Parse */

desp_prod(n,tnt,tp,tt) /* despliega producción */

int n;
TERM tt[];
NOTERM  tnt[];
int tp[MAXP][LPROD];

{
  int i=1,j=1;
  int mas;
  int sim;
  char car;

  do
  {
     if ( (n>=tnt[i].lugini) && (n<(tnt[i].lugini+tnt[i].numprod)) )
     {
       mas = FALSE;
       car = tnt[i].simb;
      }
      else
      {
        mas = TRUE;
        i++;
       }
   } while ( (i<nnt) && (mas) );

   printf("    %c",car);
   if (i > NTO)
   {
     if( (i-NTO)%9 == 0)
         printf("%d",9);
     else
        printf("%d",(i-NTO)%9);
     printf(" ->");
    }
```

```
    else
       printf("  ->");
   if (tp[n][j] == 0) ,
      printf(" %c",238);
   else
   while ( (sim = tp[n][j++]) != 0)
   {
     if (sim<0)
     {
        car = tnt[-(sim)].simb;
        printf(" %c",car);
        if ((-sim)>NTO)
        {
         if( ((-sim)-NTO)%9 == 0)
            printf("%d",9);
         else
            printf("%d",((-sim)-NTO)%9);
         }
     }
     else
     {
       car = tt[sim].simb;
       printf(" %c",car);
      }

   }  /* del while */

}  /* de desp_prod */



Esc_prods(TERM tt[],NOTERM  tnt[],int tp[MAXP][LPROD],int opt[100],char c[LCAD])

/* despliega las producciones aplicadas */

{
 int i;
 char opcion;

        if (opt[0]>1)
        {
                gotoxy(42,23);
                printf("[D] Despliega producciones aplicadas.");
                gotoxy(42,24);
                printf("[C] Continúa.");
                if (lee_opcion ("DdCc",&opcion,'D','d'))
                {
                  clrscr();
                  printf("                 Producciones Aplicacas en el Reconocimiento.\n");
                  printf("                 ------------------------------------------\n\n");
                  printf("      Cadena : '%s'\n\n",c);
                  printf("      Producciones :\n\n");
                  for (i=1 ; i<opt[0] ; i++)
                  {
                      printf("              ");
```

```
                    desp_prod(opt[i],tnt,tp,tt);
                    printf("\n");
                  }
                }
                else
                {
                  gotoxy(42,23);
                  printf("                                        ");
                }
        }
} /* de Esc_prods */


NO_LL1(tnt,tt,tp,tM,i,j,simb) /* reporta que la gramática no es LL(1) */

NOTERM tnt[];
TERM tt[];
int tp[MAXP][LPROD];
int tM[MAXNT][MAXT];
int i,j;
int simb;

{
  int in; /* para el cuadro */

   printf("%c",219);
   clrscr();
   gotoxy(10,5);
   printf("/");
   for (in=1 ; in<55 ; in++)
       printf("-");
   printf("\\n");
   printf("         |                                                     |\n");
   printf("         |  La gramática no es LL(1).                          |\n");
   printf("         |                                                     |\n");
   printf("         |  No se puede construir el reconocedor.              |\n");
   printf("         |                                                     |\n");
   printf("         |  La tabla de acción del autómata está multidefinida |\n");
   printf("         |  para los símbolos:                                 |\n");
   printf("         |                                                     |\n");
   printf("         |           Símbolo en  el tope  del  stack : %c",tnt[i].simb);
   if (i > NTO)
   {
     if( (i-NTO)%9 == 0)
        printf("%d     |\n",9);
     else
        printf("%d     |\n",(i-NTO)%9);
     }
     else
        printf("      |\n");
        printf("         |          Símbolo de la cadena de entrada : %c      |\n",tt[simb].simb);
        printf("         |                                                    |\n");
        printf("         |           ");
        desp_prod(tM[i][simb],tnt,tp,tt);
        gotoxy(65,17);
```

```
        printf("|\n");
        printf("          |    M [%c",tnt[i].simb);
        if (i > NTO)
        {
          if( (i-NTO)%9 == 0)
             printf("%d,",9);
          else
             printf("%d,",(i-NTO)%9);
         }
         else
              printf(",");
         printf("%c] =",tt[simb].simb);
         gotoxy(65,18);
         printf("|\n");
         printf("          |            ");
         desp_prod(j,tnt,tp,tt);
         gotoxy(65,19);
         printf("|\n");
         printf("          |                                                            |\n");
         gotoxy(10,21);
         printf("\");
         for (in=1 ; in<55 ; in++)
             printf("-");
         printf("/\n");
         printf("\n                    Oprime cualquier tecla para regresar...");
         getch();
} /* de NO_LL1 */



Nohay()  /* manda mensaje de que no hay cadena a reconocer */
{
  int in;

     gotoxy(16,10);
     printf("/");
     for (in=1 ; in<30 ; in++)
         printf("-");
     printf("\\n");
     printf("                |                             |\n");
     printf("                |  No hay cadena a reconocer. |\n");
     printf("                |                             |\n");
     printf("                \");
     for (in=1 ; in<30 ; in++)
         printf("-");
     printf("/\n");
} /* de Nohay */
```

```
Rec_exito()  /* manda mensaje de reconocimiento exitoso */
{
   int in;

   gotoxy(5,10);
   printf("/");
   for (in=1 ; in<66 ; in++)
       printf("-");
   printf("\\n");
   printf("    |                                                                 |\n");
   printf("    |  La cadena si pertenece al lenguaje ");
   printf("generado por la gramática. |\n");
   printf("    |                                                                 |\n");
   printf("    \");
   for (in=1 ; in<66 ; in++)
        printf("-");
   printf("/\n");
} /* de Rec_exito */

Rec_fracaso(int i,char cad[LCAD],int cc[LCAD]) /* reporta que la cadena no pertenece
                                                  al lenguaje */
{
  int j;
  int in;

  gotoxy(5,10);
  printf("/");
  for (in=1 ; in<66 ; in++)
      printf("-");
  printf("\\n");
  printf("    |                                                                 |\n");
  printf("    |  La cadena no pertenece al lenguaje ");
  printf("generado por la gramática. |\n");

  if (i>1)
  {
    if (cc[i-1] != 0)
    {
      printf("    |   La subcadena reconocida fue:                                   |\n");
      printf("    |                                  ");
      for (j=0 ; j<i-1 ; j++)
          printf("%c",cad[j]);
      gotoxy(71,14);
      printf("|\n");
     }
     else
        printf("    |   Hay hojas con símbolos que no forman parte de la cadena.      |\n");
    }
    else
       printf("    |   No hubo subcadena reconocida.                                  |\n");
    printf("    \");
    for (in=1 ; in<66 ; in++)
         printf("-");
    printf("/\n");
} /* de Rec_fracaso */
```

```
int lee_opcion(char *cadena,char* op,char fin,cnar FIN) /* lee opcion y regresa cierto si lee fin o FIN */
{
  do{
      *op = getch();
     } while (strchr(cadena,*op) == NULL);
  if ( (*op == fin) || (*op == FIN) )
       return TRUE;
  else
       return FALSE;
} /* de lee_opciones */


Reconoce(NOTERM tnt[],TERM tt[],int tp[MAXP][LPROD],int tM[MAXNT][MAXT]) /* reconocimiento de la cadena */
{
  char cadent[LCAD];   /* cadena a ser analizada */
  int mensaje;
  int cadcod[LCAD];    /* códigos de la cadena a ser analizada */
  int optape[100];    /* registro de las producciones aplicadas */
  char opcion;
  int i;

   SI_LL1(); /* reporta que la gramática si fue LL(1) */
   do{
              clrscr();
              printf("      Lista de simbolos terminales alcanzables\n");
              printf("     ----------------------------------------\n        ");
              for (i=1 ; i<nt ; i++)
                 if (tt[i].alcanza)
                    printf("%c ",tt[i].simb);
              gotoxy(16,6);
              printf("Dame la cadena a reconocer\n");
              gotoxy(16,8);
              gets(cadent);
              if (Codifica (tt,cadent,cadcod))
              {
                 mensaje = Parse (tM,tp,cadcod,optape);
                 if (mensaje >= 0)
                     Rec_fracaso(mensaje,cadent,cadcod);
                 else
                     if (mensaje == -1)
                          Nohay();
                     else
                          Rec_exito();

                 Esc_prods(tt,tnt,tp,optape,cadent);
               }
               gotoxy(42,23);
               printf("[A] Analiza otra cadena.");
               gotoxy(42,24);
               printf("[R] Regresa al Menú Principal.");

       } while (lee_opcion ("RrAa",&opcion,'A','a'));

} /* de Reconoce */
```

APENDICE III

EN ESTE APENDICE SE PRESENTA LA IMPRESION DE LAS

DISTINTAS PANTALLAS QUE DESPLIEGA EL SISTEMA.

H E R R A M I E N T A S   P A R A   L A   C O N S T R U C C I O N
D E   C O M P I L A D O R E S.

TRANSFORMACION DE GRAMATICAS.
RECONOCIMIENTO SINTACTICO.

Autor: Elke Capella Kort.

Facultad de Ciencias.
U. N. A. M.

Oprime cualquier tecla para comenzar...

M E N U   P R I N C I P A L

[T] Transformación de Gramáticas.

[A] Análisis Sintáctico.

[S] Salir.

Elige la opción

TRANSFORMACION DE GRAMATICAS

[D] Despliega los pasos del proceso de transformación.

[C] Comienza con el proceso de transformación.

Selecciona la opción

PROCESO DE TRANSFORMACION DE GRAMATICAS.

1. Eliminar las producciones vacías.

2. Eliminar las producciones unitarias.

3. Eliminar los símbolos inactivos y los inalcanzables, i.e. los símbolos inútiles.

4. Checar si la gramática es recursiva por la izquierda. En caso de que lo sea:

    a) Pasar la gramática a Forma Normal de Chomsky.
    b) Pasar la gramática a Forma Normal de Greibach.

5. Factorizar por la izquierda las producciones.

Oprime cualquier tecla para comenzar el proceso...

Lista de símbolos terminales

i b c a d f

Lista de símbolos no terminales

A B S C

Dame el símbolo inicial a

El símbolo 'a' no es símbolo no terminal en tu gramática.

[L] Lee el símbolo inicial otra vez.
[R] Regresa al Menú Principal.

Selecciona la opción

Lista de símbolos terminales

a b

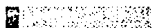
Lista de símbolos no terminales

S A B C

Dame el simbolo inicial S

Transformación de Gramáticas.

T R A B A J A N D O ...



Transformación de Gramáticas.

T R A B A J A N D O ...

Transformación de Gramáticas.

T R A B A J A N D O ...

La gramática no es recursiva por la izquierda, por lo tanto, no fue necesario hacer las transformaciones a formas normales de Chomsky y de Greibach.

Oprima cualquier tecla para continuar...

M E N U

[O] Despliega gramática original

[V] Despliega gramática sin producciones vacías

[U] Despliega gramática sin producciones unitarias

[A] Despliega gramática sin símbolos inútiles

[I] Despliega gramática factorizada por la izquierda

[R] Regresa al Menú Principal

Selecciona la opción

```
Gramática original

   A -> i B b a
   B -> S C
   S -> c a C d
   S -> f i
   C -> a b

     [R] Regresa.
```

```
Gramática sin símbolos inútiles

   S -> c a C d
   S -> f i
   C -> a b

     [R] Regresa.
```

```
Gramática factorizada por la izquierda

   S -> c a C d
   S -> f i
   C -> a b

     [R] Regresa.
```

La gramática resultó ser recursiva por la izquierda,

por lo tanto, fue transformada a:

1. Forma Normal de Chomsky.
2. Forma Normal de Greibach.

y de esta forma la recursividad por la izquierda
fue eliminada.

Oprima cualquier tecla para continuar...

M E N U

[O] Despliega gramática original

[V] Despliega gramática sin producciones vacias

[U] Despliega gramática sin producciones unitarias

[A] Despliega gramática sin símbolos inútiles

[C] Despliega gramática en Forma Normal de Chomsky

[G] Despliega gramática en Forma Normal de Greibach

[I] Despliega gramática factorizada por la izquierda

[R] Regresa al Menú Principal

Selecciona la opción

Gramática sin producciones vacias

```
S -> a A b
S -> a b
A -> b B
B -> a A b C
B -> A C S
B -> S
B -> C S
B -> A S
B -> a b
B -> a b C
B -> a A b
C -> A
C -> C A
C -> C
```

[R] Regresa.

Gramática sin producciones unitarias

```
S -> a A b
S -> a b
A -> b B
B -> a A b C
B -> A C S
B -> C S
B -> A S
B -> a b
B -> a b C
B -> a A b
C -> C A
C -> b B
```

[R] Regresa.

Gramática sin símbolos inútiles

```
S -> a A b
S -> a b
A -> b B
B -> a A b C
B -> A C S
B -> C S
B -> A S
B -> a b
B -> a b C
B -> a A b
C -> C A
C -> b B
```

[R] Regresa.

```
Gramática en Forma Normal de Greibach

    S -> a β6
    S -> a α4
    A -> b B
    B -> b B β9
    B -> a β7
    B -> a Γ1
    B -> a Γ2
    B -> a α9
    B -> b B S
    B -> b B Γ3 S
    C -> b B Γ3
    C -> b B
    α1 -> a
    α2 -> b
    α3 -> a
    α4 -> b


     [C] Continúa con el despliegue.
     [R] Regresa.

                                   Selecciona la opción
```

```
    α5 -> b
    α6 -> a
    α7 -> b
    α8 -> a
    α9 -> b
    β1 -> a
    β2 -> b
    β3 -> a
    β4 -> b
    β5 -> b
    β6 -> b B α2
    β7 -> b B β8
    β8 -> b C
    β9 -> b B S
    β9 -> b B Γ3 S
    Γ1 -> b C
    Γ2 -> b B β4


     [C] Continúa con el despliegue.
     [R] Regresa.

                                   Selecciona la opción
```

Gramática factorizada por la izquierda

$S \rightarrow a\ \Gamma 4$
$A \rightarrow b\ B$
$B \rightarrow a\ \Gamma 6$
$B \rightarrow b\ \Gamma 5$
$C \rightarrow b\ \Gamma 7$
$\alpha 1 \rightarrow a$
$\alpha 2 \rightarrow b$
$\alpha 3 \rightarrow a$
$\alpha 4 \rightarrow b$
$\alpha 5 \rightarrow b$
$\alpha 6 \rightarrow a$
$\alpha 7 \rightarrow b$
$\alpha 8 \rightarrow a$
$\alpha 9 \rightarrow b$
$\beta 1 \rightarrow a$
$\beta 2 \rightarrow b$

[C] Continúa con el despliegue.
[R] Regresa.

Selecciona la opción

A N A L I S I S   S I N T A C T I C O

[D] Despliega los pasos del proceso de análisis sintáctico.

[C] Comienza con el proceso de análisis sintáctico.

Selecciona la opción

PROCESO DEL ANALISIS SINTACTICO.

Construye la tabla de acción del autómata.

- Si la tabla de acción del autómata está multidefinida, nos indica:
  1. Para cuáles símbolos.
  2. Por cuáles producciones.

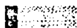
  y termina.

- Si la tabla de acción del autómata está definida unívocamente para todo par de símbolos:
  1. Solicita la cadena a ser analizada.
  2. Acepta o rechaza la cadena.

Oprime cualquier tecla para comenzar el proceso...

Análisis Sintáctico.

T R A B A J A N D O ...



Análisis Sintáctico.

T R A B A J A N D O ...

Análisis Sintáctico.

T R A B A J A N D O ...

La gramatica no es LL(1).

No se puede construir el reconocedor.

La tabla de acción del autómata está multidefinida
para los símbolos:

Símbolo en el tope del stack : Γ4
Símbolo de la cadena de entrada : b

$$M\,[\Gamma 4, b] = \begin{array}{l} \Gamma 4 \rightarrow \alpha 4 \\ \Gamma 4 \rightarrow \beta 6 \end{array}$$

Oprime cualquier tecla para regresar...

La gramática es LL(1). Por lo tanto, sí fue posible la construcción de la tabla de acción del autómata.

Oprime cualquier tecla para continuar...

Lista de símbolos terminales alcanzables

i b c a d f

Dame la cadena a reconocer

ifiab

La cadena no pertenece al lenguaje generado por la gramática.
No hubo subcadena reconocida.

[A] Analiza otra cadena.
[R] Regresa al Menú Principal.

Lista de símbolos terminales alcanzables

i b c a d f

Dame la cadena a reconocer

fiabb

La cadena no pertenece al lenguaje generado por la gramática.
La subcadena reconocida fue:
fi

[D] Despliega producciones aplicadas.
[C] Continúa.

Producciones Aplicadas en el Reconocimiento.

Cadena : 'fiabb'

Producciones :

S -> f'i

[A] Analiza otra cadena.
[R] Regresa al Menú Principal.

Lista de símbolos terminales alcanzables

i b c a d f

Dame la cadena a reconocer

ca

La cadena no pertenece al lenguaje generado por la gramática.
Hay hojas con símbolos que no forman parte de la cadena.

[D] Despliega producciones aplicadas.
[C] Continúa.

Producciones Aplicadas en el Reconocimiento.

Cadena : ´ca´

Producciones :

S -> c a C d

[A] Analiza otra cadena.
[R] Regresa al Menú Principal.

Lista de símbolos terminales alcanzables

i b c a d f

Dame la cadena a reconocer

caabd

La cadena si pertenece al lenguaje generado por la gramática.

[D] Despliega producciones aplicadas.
[C] Continúa.

Producciones Aplicadas en el Reconocimiento.

Cadena : 'caabd'

Producciones :

```
S  -> c a C d
C  -> a b
```

[A] Analiza otra cadena.
[R] Regresa al Menú Principal.

BIBLIOGRAFIA

- AHO Alfred V.,ULLMAN Jeffrey D. The Theory of Parsing Translation, and Compiling. Vol. I: Parsing. Prentice-Hall, 1972.

- AHO Alfred V.,ULLMAN Jeffrey D. The Theory of Parsing Translation, and Compiling. Vol. II: Compiling. Prentice-Hall, 1973.

- AHO Alfred V.,SETHI Ravi, ULLMAN Jeffrey D. Compilers, Principles, Techniques and Tools. Addison-Wesley Publishing Company, 1986.

- GOUGH K. John. Syntax Analysis and Software Tools. Addison-Wesley Publishing Company, 1988.

- HOPCROFT John E., ULLMAN Jeffrey D. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Publishing Company, 1979.

- KERNIGHAN Brian W., RITCHIE Dennis M. El lenguaje de programación C. Prentice-Hall, 1986.

- SCHILDT Herbert. Advanced C. Osborne McGraw-Hill, 1988.

- TREMBLAY Jean Paul, SORENSON Paul G. The Theory and Practice of Compiler Writing. McGraw-Hill, 1987.

- TURBO C. Reference Guide. Borland International, Inc., 1987.