UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

# HERRAMIENTAS PARA COMPILADORES: MANEJO VIRTUAL DE MATRICES GRANDES

TESIS QUE PRESENTA

MARIA DE LOS ANGELES LICONA VELO

Para obtener el título de

ACTUARIO

México, D.F. Marzo, 1993


**UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso**

# I N D I C E

Pag.

# I. INTRODUCCION

# I N T R O D U C C I O N

El objetivo principal de este trabajo es resolver a través del manejo de matrices virtuales el problema de saturación de memoria ocasionado por el crecimiento excesivo de tablas en la construcción de un reconocedor no recursivo por predicción.

En el capítulo II, **Marco teórico**, se expone brevemente lo que es teoría de autómatas y lenguajes formales, dando las definiciones y conceptos con los que trataremos a través de este trabajo.

En el capítulo III, **Transformación de gramáticas libres de contexto**, se describe el procedimiento de transformación de gramáticas libres de contexto para que sea posible la construcción automática de un reconocedor no recursivo por predicción. En esta parte también se muestra la forma en la que crecen las gramáticas al ser transformadas y el por qué surge la necesidad de contar con herramientas que permitan la utilización de matrices grandes.

En el capítulo IV, **Memoria virtual**, se enumeran las alternativas que existen para el manejo de tablas, llegando a la conclusión de que la mejor opción es utilizar matrices virtuales. Se define también el concepto de espacio virtual y su mecanismo de funcionamiento (paginación de memoria). En esta parte, también se explica el uso del concepto de espacio virtual para la implementación del módulo de utilerías para el manejo de matrices virtuales.

Dado que la solución al problema de saturación de memoria es usar matrices virtuales, se generó un módulo que cuenta con las funciones y procedimientos necesarios para proveer las herramientas que permiten el manejo de éstas. En el capítulo V, **Instrumentación del módulo**, se especifican las condiciones que requiere el módulo para ser utilizado. Así mismo, se detalla el objetivo, datos de entrada, datos de salida y método utilizado para cada una de las rutinas que integran a dicha biblioteca de procedimientos. Además del módulo, se codificaron dos programas que efectúan la multiplicación lógica de matrices y uno más para el algoritmo de Warshall, cada uno de ellos también documentado en este capítulo.

Por último, en el capitulo VI, **Aplicaciones del módulo**, se muestra utilidad del módulo para el manejo de matrices virtuales dentro de teoría de autómatas. Por otra parte, también se hace notar que este módulo puede usarse en cualquier procedimiento que involucre matrices. Se proporcionan, además, los algoritmos para suma, multiplicación por un escalar y multiplicación de matrices puesto que esta utilería tiene en realidad un campo de aplicación muy amplio.

En el apéndice A se encuentran los programas fuente del módulo de utilerías para el manejo de matrices virtuales. En el apéndice B se incluyen los programas fuente de aplicación que son dos programas que efectúan la multiplicación lógica de matrices y el algoritmo de Warshall.

# II. MARCO TEORICO

**Generalidades**

Este capítulo tiene como objetivo exponer en forma breve las bases de teoría de autómatas y lenguajes formales.

El fundamento matemático de casi todos los procesadores de lenguajes y lenguajes formales, es la teoría de autómatas, es decir, ésta proporciona los modelos de máquinas virtuales de traducción, que se utilizan para construir compiladores.

El *compilador* en sí mismo es un traductor, que tiene la función de pasar a patrones de bits las instrucciones escritas en algún lenguaje específico de programación, modelando éste en algún lenguaje formal.

Un *lenguaje* es un conjunto de enunciados que satisfacen ciertas propiedades o reglas de construcción; los *enunciados* son sucesiones finitas de símbolos que pertenecen a un conjunto, llamado el alfabeto del lenguaje. Un lenguaje es interesante si es infinito, esto es, si tiene un número infinito de enunciados, por lo que uno no puede esperar hacer una lista exhaustiva de todos los enunciados posibles en un lenguaje. De igual manera, uno no puede esperar escribir todos los programas posibles, pues para cualquier programador es factible crear programas antes jamás escritos que funcionan perfectamente en la computadora.

Entonces, el problema central de describir un lenguaje es proveer una especificación finita para una clase esencialmente infinita de objetos. Una *gramática* es un mecanismo que permite hacer esta descripción estructural del lenguaje, a través de la cual podemos hacer una representación precisa de éste.

Las gramáticas son un conjunto finito de símbolos y de *reglas de traducción o producciones* que si se siguen en cualquier orden válido, se podrán construir cadenas que son enunciados correctos para el lenguaje descrito por la gramática.

**Estructura de un compilador.**

El compilador tiene dos funciones principales:

La primera es determinar si un programa es sintácticamente correcto, al mismo tiempo, el compilador intenta construir una derivación (descripción estructural) del programa de acuerdo a la gramática formal que describe al lenguaje; en este caso el compilador actúa como un aceptador.

La segunda función es producir un programa objeto, que es generado en lenguaje de máquina (o en algún lenguaje intermedio como el lenguaje ensamblador) a partir de la descripción estructural del programa fuente. En este caso el compilador actúa como un traductor.

El proceso de compilación es muy complicado, existen cinco conceptos que conforman lo más importante o quizás lo indispensable para organizar un diseño de compilador. Estas subactividades son: proceso léxico, proceso sintáctico, proceso semántico, optimización y generación de código por lo que se le puede considerar como una interconexión de pequeños procesos. Como organizar estos procesos en etapas para subprocesos a un compilador dado depende de los detalles de cada lenguaje.

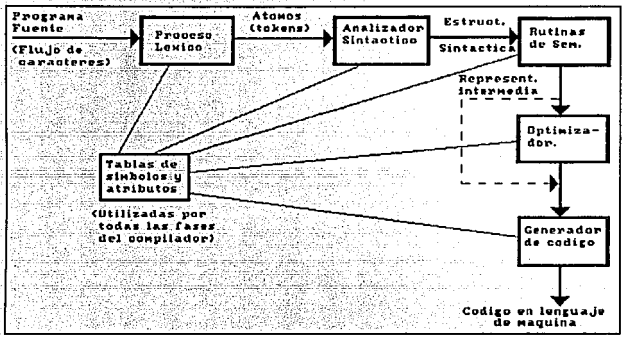
Cada una de estas actividades tiene acceso a un conjunto común de tablas que contienen información global acerca del programa. Una de las tablas es la de símbolos; ésta guarda la información correspondiente a variables o identificadores.

Uno de los problemas centrales en el diseño de un compilador es la selección del método encargado de generar la descripción estructural del programa fuente.

En la mayoría de los compiladores modernos el proceso de compilación es dirigido por la estructura sintáctica del programa fuente y se habla entonces de un compilador dirigido por la sintaxis. El reconocimiento de la estructura sintáctica constituye la mayor parte de la tarea de "análisis".

En la siguiente figura se muestra la organización de un

compilador *dirigido por la sintaxis*.




**Proceso léxico.** Este proceso efectúa las siguientes funciones:

1. Pone al programa en formato compacto y uniforme (tokens).
2. Elimina información innecesaria (como comentarios).
3. Procesa las directivas de control del compilador ("macros").
4. Algunas veces introduce información preliminar en las tablas de símbolos y atributos.
5. Formatea y lista el programa fuente.

**Analizador sintáctico.** Dada una especificación formal de sintaxis (gramática), el analizador lee la cadena de caracteres generada por el proceso léxico y los agrupa en unidades especificadas por las producciones de la gramática que será usada. Los analizadores generalmente son manejados a través de tablas creadas por un generador léxico.

El analizador revisa que la sintaxis sea correcta. Una vez que la estructura sintáctica ha sido aceptada, el analizador sintáctico llama a las rutinas correspondientes a analizar la semántica o bien construye un árbol de sintaxis que represente la estructura del programa, el cual es usado para dirigir el proceso de semántica.

**Rutinas de semántica.** La primer función dentro de este proceso es verificar que la construcción sea legal y con significado correcto; si es así, se efectúa la traducción a lenguaje intermedio o de máquina.

En un compilador dirigido por la sintaxis, las rutinas de semántica usualmente están asociadas con producciones individuales de la gramática o con subárboles del árbol sintáctico.

**Optimizador.** El código de la representación intermedia generado por la rutina de semántica es analizado y transformado en un código funcionalmente equivalente. Esta fase puede ser muy compleja y lenta y con frecuencia involucra a numerosas subrutinas. La mayoría de los compiladores desactivan al optimizador para agilizar la traducción, más aún, algunos no tienen optimizador.

**Generador de código.** La tarea del generador de código es expander el código producido por el analizador sintáctico en una secuencia de instrucciones entendibles por la computadora (en lenguaje de máquina) que realice las operaciones especificadas en el programa fuente.

**Gramáticas formales.**

Como ya mencionamos anteriormente, las gramáticas describen la estructura de un lenguaje. Estas se conforman de tres componentes principales:

1. El alfabeto o conjunto de símbolos con los cuales son construidos los enunciados del lenguaje descrito. A este conjunto se le denomina *símbolos terminales* de la gramática. La generación de un enunciado a través de la aplicación de las reglas de la gramática, debe terminar

en una cadena que contenga solo a estos símbolos.

2. Un conjunto de símbolos que denoten frases, llamados *símbolos no terminales* de la gramática.

3. Una colección de reglas gramaticales o *producciones*.

Para completar la especificación de un lenguaje, es necesario un punto de partida para aplicar las producciones. El símbolo S, denominado símbolo inicial, está reservado para este propósito.

Denotaremos a una gramática formal como $G = (V_N, V_T, P, S)$ donde:

$V_N$ es un conjunto finito de símbolos no terminales,

$V_T$ es un conjunto finito de símbolos terminales,

$V_N$ y $V_T$ son disjuntos: $V_N \cap V_T = \phi$,

P es un conjunto finito de producciones, y

S es el símbolo inicial; $S \in V_N$

**Definición.** Sea G una gramática formal. Una cadena de símbolos en $(V_N \cup V_T)$ es conocida como la forma sentencial. Si $\alpha \rightarrow \beta$ es una producción de G y $w = \varphi\alpha\sigma$ y $w' = \varphi\beta\sigma$ son formas sentenciales, diremos que w' es derivada directamente de w en G, e indicaremos esta relación como $w \Rightarrow w'$. Si $w_1, w_2, \ldots, w_n$ es una secuencia de formas sentenciales tales que $w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n$, diremos que $w_n$ es derivable de $w_1$ e indicaremos esta relación como $w_1 \Rightarrow^* w_n$. La secuencia $w_1, w_2, \ldots, w_n$ se denomina la derivación de $w_n$ a partir de $w_1$ de acuerdo a G.

**Definición.** El lenguaje L(G) generado por la gramática formal G es el conjunto de cadenas terminales derivables de S:

$$L(G) = \{w \in V_T^* \mid S \Rightarrow^* w\}$$

Si $w \in L(G)$, diremos que w es una cadena, enunciado o palabra en el lenguaje generado por G.

**Definición.** La longitud de una cadena **w**, denotada por $|w|$ es el número de símbolos que componen la cadena.

**Definición.** La cadena vacía, denotada por $\lambda$ es la que consta de cero símbolos, es decir $|\lambda| = 0$.

**Tipos de gramáticas.**

Las gramáticas se han clasificado en cuatro grandes grupos, de acuerdo al tipo de restricciones que se aplican a las producciones de la gramática. Esta división se debe a Noam Chomsky.

| Tipo | Formato de las producc.[1] | Características |
|---|---|---|
| 0 | $\varphi A\sigma \rightarrow \varphi w\sigma$ | Reglas de sustitución no restringidas (contraíbles) |
| 1 | $\varphi A\sigma \rightarrow \varphi w\sigma$, $w \neq \lambda$<br>$S \rightarrow \lambda$ | Sensitiva al contexto |
| 2 | $A \rightarrow w$, $w \neq \lambda$<br>$S \rightarrow \lambda$ | Libre del contexto |
| 3 | $A \rightarrow aB$<br>$A \rightarrow a$<br>$S \rightarrow \lambda$<br><br>$A \rightarrow Ba$<br>$A \rightarrow a$<br>$S \rightarrow \lambda$ | Lineal por la derecha<br><br>Lineal por la izquierda |

[1] $A,B \in V_N$, $\varphi,w,\sigma \in (V_N \cup V_T)^*$, $a \in V_T$, $B \in V_N$.

De acuerdo a la tabla que se muestra arriba, la forma de las producciones permitidas son cada vez más restringidas. Es obvio que la gramática de un tipo es también una gramática de cada uno de los tipos listados más arriba en la tabla. Usando la notación $\mathcal{G}_i$ para representar a la clase de gramáticas de tipo i, tenemos que

$$\mathcal{G}_0 \supset \mathcal{G}_1 \supset \mathcal{G}_2 \supset \mathcal{G}_3$$

Como consecuencia, cada gramática debe generar lenguajes que son una subclase de los lenguajes generados por cualquier tipo de gramática listada más arriba en la tabla. Sea $\mathcal{L}_i$ la clase de lenguajes generados por gramáticas de clase $\mathcal{G}_i$, entonces

$$\mathcal{L}_0 \supseteq \mathcal{L}_1 \supseteq \mathcal{L}_2 \supseteq \mathcal{L}_3$$

Es decir, las gramáticas tipo 0 son más generales y por lo tanto más poderosas que las demás, pero no se usan para definir lenguajes de programación, debido a que no existen analizadores eficientes para este tipo de gramáticas y sin un analizador no hay forma de usar la definición de esa gramática para manejar un compilador.

Existen analizadores muy eficientes para muchas clases de gramáticas libres de contexto (tipo 2), por lo que estas gramáticas representan un buen balance entre la generalización y la práctica. Por lo tanto, este trabajo será enfocado a las gramáticas libres de contexto.

## Gramáticas libres de contexto

Una gramática libre de contexto (GLC) está definida por los siguientes cuatro componentes:

1. Un conjunto finito de símbolos terminales $V_T$.
2. Un conjunto finito de símbolos no terminales $V_N$.
3. Un símbolo inicial $S \in V_N$ a partir del cual inician todas las derivaciones.
4. Un conjunto finito de producciones P, de la forma

$A \rightarrow X_1 \ldots X_m$, donde $A \in V_N$, $X_i \in (V_N \cup V_T)$, $1 \leq i \leq m$, $m \geq 0$ y $A \rightarrow \lambda$ es una producción válida.

Estos cuatro componentes son representados como $(V_T, V_N, S, P)$, los cuales son la definición formal de una gramática libre de contexto. El vocabulario V de una gramática libre de contexto es el conjunto de símbolos terminales y no terminales $(V_T \cup V_N)$.

La noción básica de las gramáticas libres de contexto es que principiando con el símbolo inicial S, se deben aplicar las producciones reemplazando los símbolos no terminales hasta que se produzca una cadena que contenga sólo símbolos terminales. El conjunto de cadenas derivables de S comprende al lenguaje libre de contexto de la gramática G, denotado por L(G).

Para hacer más clara la exposición y siguiendo la costumbre establecida, usaremos la siguiente notación:

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

Utilizando esta notación, una producción podría escribirse como $A \rightarrow \alpha$ o $A \rightarrow X_1 \ldots X_m$. Este formato enfatiza que el lado izquierdo de la producción debe ser un símbolo no terminal, mientras que el lado derecho es una cadena de cero o más símbolos del vocabulario, es decir de $V_T \cup V_N$.

De acuerdo a la definición 1, tenemos que si $A \rightarrow \sigma$ es una producción, entonces $\alpha A \beta \Rightarrow \alpha \sigma \beta$, donde $\Rightarrow$ denota un paso de la derivación (usando la producción $A \rightarrow \sigma$). Extenderemos $\Rightarrow$ a $\Rightarrow^+$, que indica que deriva en uno o más pasos, y $\Rightarrow^*$ deriva en cero o más. Si $S \Rightarrow^* \beta$, entonces se dice que $\beta$ es una forma sentencial de GLC. Sea FS(G) el conjunto de formas sentenciales de la gramática G. Similarmente $L(G) = \{ x \in V_T^* \mid S \Rightarrow^+ x \}$. De lo anterior podemos deducir que $L(G) = FS(G) \cap V_T^*$, es decir el lenguaje de G es simplemente el conjunto de aquellas formas sentenciales de G que son cadenas de símbolos terminales.

Una derivación se representa con frecuencia por un *árbol de*

*derivación*. Donde la raíz del árbol es el símbolo inicial S; las hojas son símbolos de la gramática o λ. Los nodos interiores son símbolos no terminales. Cuando la derivación se ha terminado, las hojas del árbol son símbolos terminales o λ.

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

**Análisis sintáctico.**

El analizador sintáctico se encarga de verificar que la cadena de símbolos construída en el proceso léxico pueda ser generada por la gramática que define al lenguaje. Se espera que el reconocedor reporte cualquier error sintáctico en una forma inteligible. Así mismo, debe ser capaz de recuperarse de los errores que ocurren comúnmente, de tal forma que pueda continuar procesando el resto de la cadena de entrada.

Existen distintos tipos de reconocedores sintácticos, que determinan si una cierta cadena pertence o no al lenguaje generado por una gramática libre de contexto. Dos métodos generales para el análisis sintáctico son los que se describen a continuación.

El primero se denomina *arriba-abajo*. Un analizador es considerado arriba-abajo si "descubre" el árbol sintáctico correspondiente de una secuencia de caracteres iniciando en la raíz del árbol (el símbolo inicial), y entonces se expande hacia abajo mediante predicciones. Los analizadores Arriba-abajo son predictivos en naturaleza porque intentan predecir la producción que corresponde.

La segunda se refiere a los analizadores *abajo-arriba*, como su nombre lo indica, éstos generan la estructura del árbol sintáctico iniciando al final, es decir en las hojas del árbol que son símbolos terminales y determinan las producciones usadas para generar las hojas. El analizador continúa hasta llegar a la producción usada para expander el símbolo inicial.

A continuación se muestra un ejemplo del análisis arriba-abajo de una cadena de caracteres:

Sea $G = (V_N, V_T, S, P)$
$V_N = \{A, B\}$
$V_T = \{x, y, z\}$
$S = A$
$P =$

| | |
|---|---|
| $A \rightarrow xB$ (1) | $B \rightarrow y$ (4) |
| $A \rightarrow xC$ (2) | $C \rightarrow xC$ (5) |
| $B \rightarrow xB$ (3) | $C \rightarrow z$ (6) |

Si tratamos de analizar la cadena xxxz podemos proceder como sigue:

Forma Sentencial S = A  Cadena de entrada xxxz

Revisando la forma sentencial A podemos elegir entre la producción (1) y (2). No tenemos ninguna ayuda real revisando el primer símbolo terminal de la cadena de entrada, entonces elijamos la producción (1). Ahora tenemos

Forma Sentencial xB  Cadena de entrada xxxz

Lo cual implica que debemos poder derivar xxz de B. Ahora tenemos que de las producciones de B hay que seleccionar (3), obteniendo

Forma Sentencial xxB  Cadena de entrada xxz

Esto indica que podemos derivar xz de B. Si aplicamos (3) nuevamente obtenemos

Forma Sentencial xxB  Cadena de entrada xz

Ahora, debemos generar z directamente de B, pero no podemos. Esto indica que hicimos una decisión errónea en la selección de alguna de las producciones. Obviamente, aquí estuvimos equivocados al usar la producción (1) y no la (2).

El ejemplo anterior muestra los problemas a los que se llega cuando tenemos producciones alternativas para el mismo símbolo no terminal.

Cuando el analizador se enfrenta con esta clase de dilema, puede adoptar la estrategia de proceder simplemente de

acuerdo a una de las opciones posibles, estando preparado para "retroceder" si el camino elegido no permite seguir adelante. Claramente cualquier acción de retroceso es ineficiente y aun con una gramática simple, casi no hay límite para el número de retrocesos que el analizador debe estar preparado para hacer.

Un tipo de reconocedor arriba-abajo bastante eficiente es el reconocedor no recursivo por predicción.

**Reconocedor no recursivo por predicción.**

El 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 derminada de manera única la producción a aplicar.

Para la construcción de un reconocedor no recursivo por predicción, 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.

No todos los lenguajes libres de contexto pueden ser generados por gramáticas para las cuales es posible la construcción de este tipo de reconocedores.

Para utilizar este tipo de reconocedor, es necesario que la gramática que define al lenguaje sea libre de contexto, esté reducida, no sea recursiva por la izquierda y esté factorizada por la izquierda. Cuando la gramática no cumple con estas características a veces es posible transformarla de tal forma que cumpla con las condiciones requeridas por el reconocedor.

Para que la transformación de la gramática sea posible (y por ende la construcción del reconocedor), la gramática debe ser una gramática LL(1).

**Gramáticas tipo LL(K)**

La terminología LL(k) proviene del hecho de que estamos examinando la cadena de entrada de izquierda a derecha (Left to right) aplicando las producciones en la forma sentencial al símbolo no terminal más a la izquierda (Leftmost derivation) y revisando hacia adelante los primeros k símbolos terminales en la cadena de entrada para ayudarnos a decidir cual producción aplicar en cada paso. En la práctica, el análisis LL(1) es la forma más deseable y común de análisis LL(k).

El tipo más sencillo de gramática LL(1) es una gramática libre de contexto sin producciones vacías, tal que para toda $A \in V_N$, las producciones alternativas de **A** comienzan cada una con un símbolo terminal distinto, es decir todas sus producciones son de la forma:

$A \rightarrow a_1 \alpha_1 | a_2 \alpha_2 | \ldots | a_n \alpha_n$,

donde $a_i \neq a_j$ para $i \neq j$ y $a_i \in V_t$ para $1 \leq i \leq n$.

Existen tipos más generales de gramáticas LL(1), en donde se elimina la restricción anterior para la forma de las producciones, sin embargo no las discutiremos pues no creemos necesario profundizar en este tema.

# III. TRANSFORMACION DE GRAMATICAS LIBRES DE CONTEXTO

## Transformación de gramáticas libres de contexto

En el presente capítulo se describen los métodos de transformación de las gramáticas libres de contexto necesarias para la construcción automática de un analizador sintáctico no recursivo por predicción.

Consideramos que dos gramáticas son *equivalentes* si ambas generan el mismo lenguaje.

Los pasos a seguir para llevar a cabo la transformación de la gramática libre de contexto 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 inútiles.

4. Si la gramática es recursiva por la izquierda:

   4.1 Transformarla a forma normal de Chomsky

   4.2 Transformarla a forma normal de Greibach

5. Factorizar la gramática por la izquierda.

Este es el orden en el que debemos aplicar los algoritmos de transformación, es decir, primero debemos obtener una gramática sin producciones vacías, que será la gramática de entrada para el algoritmo que elimina las producciones unitarias. Posteriormente hay que tomar esta gramática sin producciones vacías ni unitarias para eliminar los símbolos inútiles y obtener así una gramática reducida.

Una vez que se ha obtenido una gramática reducida, habrá que revisar si es recursiva por la izquierda, si lo es hay que transformarla a forma normal de Chomsky y a forma normal de Greibach por último se debe factorizar por la izquierda.

En cada uno de los pasos del proceso debemos obtener gramáticas que sean equivalentes a la anterior.

Cabe aclarar que solo describiremos los métodos, sin construir los algoritmos que generan las transformaciones[1].

**Eliminación de producciones vacías.**

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

Las producciones vacías se utilizan con frecuencia en una gramática libre de contexto como una manera de terminar una recursión.

Podemos eliminar las producciones vacías de una cierta gramática G, siempre y cuando $\lambda \notin L(G)$.

El método para suprimir este tipo de producciones consiste en determinar, para cada símbolo no terminal A, si $A \rightarrow^* \lambda$; si esto ocurre, 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.

Ejemplo

Sea G la siguiente gramática:

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

Aplicando el algoritmo que elimina producciones vacías,

[1] El desarrollo completo de estos algoritmos se encuentra en [Capella Kort, Herramientas para la construcción de compiladores]

obtenemos la gramática G' equivalente a G sin producciones vacías:

| | |
|---|---|
| S → aAbC | B → bSBA |
| S → abC | B → bS |
| A → aB | B → bSA |
| A → a | B → bSB |
| B → A | C → aab |

**Eliminación de producciones unitarias y ciclos.**

**Definición.** Las producciones de la forma $A \rightarrow B$ en las cuales el lado derecho consiste de un solo símbolo no terminal son denominadas producciones unitarias.

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

En primer lugar debemos construir un conjunto cuyos elementos serán símbolos no terminales derivados por algún otro símbolo no terminal, denotaremos a este conjunto como $V_A$, entonces tenemos que

$$V_A = \{ B \mid A \rightarrow^{*} B \text{ y } B \in V_N \}.$$

Posteriormente, habrá que eliminar las producciones unitarias del conjunto de producciones de la gramática y agregar otras que compensen la eliminación, obteniendo de esta manera el conjunto de producciones de la gramática equivalente a G. En notación matemática podemos escribir este proceso como

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

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

$$P = P' \cup P$$

**Eliminación de símbolos inútiles**

**Definición.** Sea $G = (V_N, V_T, P, S)$ cualquier gramática libre de contexto. Una producción $A \rightarrow \alpha$ de G es activa si G permite la derivación $A \rightarrow \alpha \rightarrow^* w$, $w \in V_T^*$. En otro caso $A \rightarrow \alpha$ es una producción inactiva.

Un símbolo no terminal de G es activo si es la parte izquierda de alguna producción activa; en otro caso es un símbolo no terminal inactivo.

**Definición.** Un símbolo A que aparece en alguna forma sentencial derivable del símbolo inicial se llama símbolo alcanzable, es decir, $S \rightarrow^* \varphi A \sigma$, $A \in V_N$. En otro caso se llama símbolo inalcanzable.

**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.

Los símbolos útiles de una gramática libre de contexto G pueden ser identificados aplicando dos procedimientos. En el primero de ellos obtendremos los conjuntos de símbolos y producciones activos, mientras que en el segundo obtendremos al conjunto de símbolos alcanzables extrayéndolos del conjunto de símbolos activos.

Llamemos $P_A$ al conjunto cuyos elementos son las producciones de G tales que existe una derivación de una cadena de símbolos terminales en la cual se aplique la producción.

$$P_A = \{ A \rightarrow \alpha \mid A \rightarrow \alpha \rightarrow^* w, w \in V_T^* \}$$

En términos de $P_A$, tenemos que el conjunto $S_A$ de símbolos no terminales conectados a los terminales está dado por

$$S_A = \{ A \in V_N \mid \exists \text{ una regla } A \rightarrow \alpha \text{ en } P_A \}$$

Obviamente cada producción útil de G debe estar en $P_A$, y

cada símbolo no terminal útil debe estar en el conjunto $S_A$. Sin embargo, el hecho de que una producción esté en $P_A$ o un símbolo esté en $S_A$ no es suficiente para garantizar que la producción o el símbolo no terminal es útil.

El símbolo no terminal A en $S_A$ o la producción $A \rightarrow \alpha$ en $P_A$ es inútil a menos que el símbolo A pueda ser derivado desde S. Tomemos ahora el conjunto $P_S$, que contiene a cada producción $A \rightarrow \alpha$ en $P_A$ sólo si G permite la derivación $S \rightarrow^* \phi A \sigma$. Como estamos usando sólo producciones de $P_A$, G debe permitir la derivación $S \rightarrow^* \phi A \sigma \rightarrow \phi \alpha \sigma \rightarrow^* w$, $w \in V_T^*$.

Tenemos que el conjunto $P_S$ que contiene sólo a las producciones útiles de la gramática es

$$P_S = \{A \rightarrow \alpha \mid S \rightarrow^* \phi A \sigma \rightarrow \phi \alpha \sigma \rightarrow^* w, \quad w \in V_T^* \}$$

Al aplicar cada uno de los métodos que se acaban de describir, (aplicados en el mismo orden) obtendremos una gramática reducida que es equivalente a la gramática original G.

**Diagnóstico de recursividad por la izquierda**

En un reconocedor no recursivo por predicción, no se puede tener recursividad izquierda, puesto que en cada paso del reconocimiento está determinada de manera única la producción a aplicar, de tal forma que si una producción resultara ser recursiva (por la izquierda), ésta volverá a elegirse infinitamente y el proceso de reconocimiento nunca terminará.

Únicamente nos interesa la recursividad izquierda puesto que las producciones se aplican al símbolo no terminal que esté más a la izquierda de la forma sentencial.

**Definición.** Una gramática es recursiva por la izquierda si tiene un símbolo no terminal A tal que haya una derivación $A \rightarrow^+ A\alpha$ para alguna cadena $\alpha$.

Para poder determinar si una gramática es recursiva por la izquierda o no, utilizaremos la relación F (First) que nos va a indicar, dado un símbolo no terminal si el símbolo es recursivo por la izquierda, esto es si en el transcurso de una derivación, vuelve a aparecer el mismo en el extremo izquierdo de la forma sentencial. A continuación damos un algoritmo para determinar esta relación:

Denotaremos al conjunto de símbolos nulos por $V_{Ne}$

$$\text{donde } V_{Ne} = \{ A \in V_N \mid A \Rightarrow^* \lambda \}.$$

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

$X \; F \; Y_1$ pues $Y_1$ aparece en el extremo izquierdo en una derivación de 1 paso

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.

En general, $X \; F \; Y_i$ si $Y_j \in V_{Ne} \; \forall \; j < i$,

donde $X \; F \; Y_i = 1$, si $Y_i \in V_{Ne}$, es decir, la relación F nos indica que símbolos terminales o no terminales ocupan el primer lugar en cada una de las derivaciones de X, para toda $X \in V_N$.

Ahora bien, para determinar si una gramática es recursiva por la izquierda, hay que construir la matriz de relación F de la gramática y con el algoritmo de Warshall, obtener la cerradura transitiva de F, que denotamos por $F^+$ (ver capítulo VI). 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.

Ejemplo

Sea $G=(V_N,V_T,S,P)$ tales que
$V_N = \{S,A,B\}$
$V_T = \{a,b\}$
P =

| | |
|---|---|
| $S \rightarrow AB$ | $B \rightarrow SA$ |
| $A \rightarrow BS$ | $B \rightarrow a$ |
| $A \rightarrow b$ | |

De acuerdo a la gramática anterior, tenemos que $V_{Ne} = \phi$. La matriz de relación R sería:

| | S | A | B | a | b |
|---|---|---|---|---|---|
| S | 0 | 1 | 0 | 0 | 0 |
| A | 0 | 0 | 1 | 0 | 1 |
| B | 1 | 0 | 0 | 1 | 0 |
| a | 0 | 0 | 0 | 0 | 0 |
| b | 0 | 0 | 0 | 0 | 0 |

Aplicando el algoritmo de Warshall a esta matriz, obtenemos la cerradura transitiva, $R^+$:

| | S | A | B | a | b |
|---|---|---|---|---|---|
| S | 1 | 1 | 1 | 1 | 1 |
| A | 1 | 1 | 1 | 1 | 1 |
| B | 1 | 1 | 1 | 1 | 1 |
| a | 0 | 0 | 0 | 0 | 0 |
| b | 0 | 0 | 0 | 0 | 0 |

Observemos que (S,S)=1, (A,A)=1, (B,B)=1 por lo que, esta gramática es recursiva por la izquierda en todos sus símbolos no terminales.

Prueba

$S \rightarrow AB \rightarrow BSB \rightarrow SASB$

$A \rightarrow BS \rightarrow SAS \rightarrow ABAS$

$B \rightarrow SA \rightarrow ABA \rightarrow BSBA$

Como ya mencionamos anteriormente, un reconocedor no

recursivo por predicción no puede manejar gramáticas recursivas por la izquierda, para eliminar dicha recursividad se tiene que transformar a la gramática a forma normal de Chomsky para después pasarla a forma normal de Greibach. Al hacer lo anterior, se elimina la recursividad.

**Forma Normal de Chomsky.**

**Definición.** Una gramática libre de contexto está en forma normal de Chomsky si cada una de sus producciones es de la forma $A \rightarrow BC$ o $A \rightarrow a$, donde $A,B,C \in V_N$ y $a \in V_T$.

Ya que G es una gramática reducida, entonces cada producción de G tiene una de las siguientes formas: $A \rightarrow bB$, $A \rightarrow \alpha$, $A \rightarrow a$.

Podemos construir la gramática equivalente a G expandiendo cada regla $A \rightarrow \alpha$, con $|\alpha|>2$, en un conjunto de producciones cuyo lado derecho contiene solo 2 símbolos.

Para cada regla $A \rightarrow X_1X_2...X_k$, con $k>2$, $X_i \in V_N \cup V_T$, introduciremos k-2 símbolos no terminales, $\alpha_1, \alpha_2, \ldots, \alpha_{k-2}$ y reemplazamos a $A \rightarrow X_1X_2...X_k$ con el conjunto de producciones:

$A \rightarrow X_1\alpha_1$
$\alpha_1 \rightarrow X_2\alpha_2$
$\alpha_2 \rightarrow X_3\alpha_3$
.
.
.
$\alpha_{k-2} \rightarrow X_{k-1}X_k$

Aplicar la producción $A \rightarrow X_1X_2...X_k$ en G corresponde a aplicar estas nuevas producciones en secuencia. Cada producción de la gramática equivalente está ahora en una de las nuevas formas siguientes:

$A \rightarrow BC$
$A \rightarrow a$

$A \rightarrow bC$
$A \rightarrow Bc$
$A \rightarrow bc$

Los tipos de producciones de la columna izquierda están en forma normal de Chomsky, pero los de la columna derecha no lo están porque contienen símbolos terminales en el lado derecho de la producción. Podremos pasarlos a forma normal introduciendo el símbolo no terminal $B_i$ para cada letra terminal x en la gramática, reemplazando cada ocurrencia de x en el lado derecho por $B_i$. La producción $B_i \rightarrow x$ es entonces agregada a la gramática equivalente.

Por ejemplo, una producción de la forma $A \rightarrow bC$ llegaría a ser la producción $A \rightarrow B_1C$, con la regla $B_1 \rightarrow b$ adicionada a la gramática.

Las alteraciones que hemos descrito, producen una nueva gramática equivalente a la original pero en forma normal de Chomsky.

**Forma Normal de Greibach**

Las gramáticas libres de contexto en la forma normal de Greibach empiezan con símbolos terminales en los lados derechos de sus producciones, lo que evita que sean recursivas por la izquierda. Debido a ésto juegan un papel muy importante en el análisis sintáctico.

**Definición.** Una gramática libre de contexto $G=(V_N,V_T,P,S)$ está en forma normal de Greibach si cada una de sus producciones tiene la forma $A \rightarrow a\alpha$, donde $a \in V_T$ y $\alpha \in V_N^*$.

En una gramática en forma normal de Greibach, cada paso después del primero en una derivación por la izquierda tiene la forma $\phi A\sigma \Rightarrow \phi a\alpha\sigma$. Entonces, la derivación de una cadena de símbolos terminales de n letras requiere a lo más de n+1 pasos.

El método para transformar una gramática en forma normal de Chomsky a una en forma normal de Greibach consiste en los siguientes pasos:

1. Le damos un orden al conjunto de símbolos no termi-

nales, $V_n=\{A_1, A_2, \ldots, A_m\}$. No existe ningún criterio específico para ordenar los símbolos no terminales. Sin embargo es conveniente colocar al símbolo inicial a la cabeza de la lista e ir agregando los otros símbolos conforme van apareciendo en las derivaciones, ya que se va a buscar como paso intermedio que si una producción tiene la forma $A_i \rightarrow A_j\alpha$, $j > i$ en el orden que eligimos.

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

3. Hacemos que todas las producciones de los símbolos no terminales $A_1, A_2, \ldots, A_m$ comiencen con un símbolo terminal.

4. Hacemos que las producciones de los símbolos no terminales agregados $B_1, B_2, \ldots, B_m$ comiencen con un símbolo terminal.

La recursividad izquierda no está permitida en una gramática en forma normal de Greibach porque la gramática no acepta derivaciones de la forma $A \rightarrow^* A\alpha$.

**Factorización izquierda.**

Al factorizar por la izquierda una gramática, tenemos que cuando no es claro cuál de dos o más producciones alternativas utilizar para expander a algún símbolo no terminal A, es posible 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.

Esto indica que al tener una gramática factorizada por la izquierda, es posible la construcción automática de un analizador sintáctico no recursivo por predicción.

Para aplicar el algoritmo que factoriza por la izquierda a

la gramática, tomemos en cuenta que ésta debe ser libre de contexto reducida y sin recursividad por la izquierda.

Los pasos a seguir para factorizar por la izquierda son los siguientes:

1. Para cada símbolo no terminal **A**, encontrar todas las producciones que comiencen con el mismo símbolo **X**. Si **X** ≠ **λ**, 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 producciones 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.

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

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

**Crecimiento de gramáticas.**

Como se ha podido observar, al someter a cualquier gramática libre de contexto al proceso de transformación arriba descrito, el número de producciones de la gramática equivalente tiende a aumentar, sobre todo al ser transformada a forma normal de Chomsky y forma normal de Greibach.

Esto ocasiona un problema si tomamos en cuenta que el reconocedor no recursivo por predicción trabaja con tablas

de símbolos terminales y no terminales, y al crecer la gramática crece la tabla, lo que provoca que se pueda llegar a saturar la memoria.

No haremos un cálculo del crecimiento de una gramática que se transforma hasta ser reducida, pues en realidad en este proceso el número de producciones no aumenta tanto. Sin embargo sí es conveniente hacerlo para la transformación a forma normal de Chomsky y Greibach.

**Crecimiento de una gramática que se transforma a Forma Normal de Chomsky.**

Sea **G** una gramática libre de contexto sin producciones vacías ni unitarias, donde

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

sea n igual a la cardinalidad del conjunto **P**, es decir n es el número de producciones de la gramática **G**.

sea $P' = \{ A \rightarrow \alpha \mid A \rightarrow \alpha \in P, |\alpha| > 2, \alpha \in (V_N \cup V_T) \}$

sea n' la cardinalidad de P', es decir, sea n' el número de producciones de G cuyo lado derecho contiene más de dos símbolos.

De acuerdo al proceso de transformación descrito anteriormente, tenemos que, para cada producción $A_i \rightarrow X_1X_2...X_k \in P'$ se deben agregar k-2 símbolos no terminales y reemplazar a $A_i \rightarrow X_1X_2...X_n$ por el conjunto de producciones:

$A_i \rightarrow X_1\alpha_1$
$\alpha_1 \rightarrow X_2\alpha_2$
.
.
.
$\alpha_{k-2} \rightarrow X_{k-1}X_k$

Es decir, en lugar de la producción original con k símbolos no terminales del lado derecho, se agregan k-1 nuevas producciones.

Denotemos como $c_i$ al número de producciones agregadas a la gramática en lugar de la regla $A_i \rightarrow X_1X_2...X_k$. Por lo tanto, el total de producciones adicionadas será $c_1+c_2+...+c_{n'}$.

Hasta ahora, solo hemos logrado que cada una de las producciones de la gramática sean de la forma $A \rightarrow \alpha$, con $|\alpha| \leq 2$. Nos falta sustituir a cada símbolo terminal x en la gramática por un nuevo símbolo no terminal $B_i$, agregando la producción $B_i \rightarrow x$ a la gramática.

$$\text{Sea } P'' = \{ A \rightarrow X_1X_2 \mid X_1 \in V_T^* \text{ o } X_2 \in V_T^* \}$$

Sea n'' igual a la cardinalidad de P'', es decir n'' es igual al número de producciones con al menos un símbolo no terminal del lado derecho. Como por cada símbolo terminal se debe agregar un nuevo no terminal, tenemos que a lo más se tendrán 2n'' nuevas producciones.

De lo anterior deducimos que el nuevo número de producciones de la gramática equivalente está dado por

$$n-n'+c_1+c_2+...+c_{n'}+3n'', \text{ donde}$$

n = Número de producciones de la gramática original

n'= Número de producciones del conjunto P'

$c_i$= Número de producciones agregadas en lugar de la producción $A_i \rightarrow \alpha$ tales que $A_i \rightarrow \alpha \in P'$.

n''= Número de producciones agregadas por cada símbolo no terminal.

Ahora supongamos que P = P', es decir, que todas las producciones de G tienen más de dos símbolos del lado derecho, por lo tanto n = n'.

De acuerdo a la notación que utilizamos, tenemos que $c_i$ es igual al número de nuevas producciones que sustituyen a la

producción $A_i \rightarrow \alpha$, donde $c_i = k-1$ si $|\alpha| = k$. Sea $c = (c_1+c_2+\ldots+c_n)/n$, entonces, el número de nuevas producciones es igual a nc.

Ahora supongamos que todas las nuevas producciones son de la forma $A_i \rightarrow x_1x_2$, donde $x_1,x_2 \in V_T$, es decir que la cardinalidad del conjunto P'' es igual a nc, como $x_1,x_2 \in V_T$ tenemos que tendrán que adicionarse 3nc nuevas producciones.

Si hacemos K = 3c, tenemos que las gramáticas que se transforman a forma normal de Chomsky crecen a razón de Kn, donde n es igual al número de producciones de la gramática original y K es una constante que depende del número de producciones de la forma $A \rightarrow \alpha$, con $|\alpha| > 2$ y del número de símbolos que conforman la producción, es decir de $|\alpha|$. Por último podemos agregar que también depende del número de símbolos terminales de cada producción. De acá, K puede ser relativamente grande, ya que $|\alpha|$ no tiene por que estar dado en función ni del número de símbolos terminales ni del número de símbolos no terminales y que puede haber repeticiones en $\alpha$ de uno o más de ellos.

**Crecimiento de una gramática que se transforma a Forma Normal de Greibach.**

Sea $G = (V_N,V_T,P,S)$ una gramática en forma normal de Chomsky, con $p = |P|$ y $n = |V_N|$, es decir que p es el número de producciones de la gramática y n es el número de símbolos no terminales. Llamaremos G' a la gramática en Forma Normal de Greibach, donde $G' = (V_N',V_T,P',S)$.

De acuerdo al método de transformación descrito anteriormete, en primer lugar hay que reemplazar a todas las producciones con símbolos no terminales "menores" que el símbolo no terminal del lado izquierdo. Es decir hay que reemplazar a las producciones de la forma $A_i \rightarrow A_j\alpha$ con $j < i$ por producciones de la forma $A_i \rightarrow A_k\beta$ con $k \geq i$.

En este primer proceso, el número de producciones crece en función del número de árboles derivables de $A_j$. Ahora bien, sabemos que el número de árboles de cualquier símbolo no terminal es a lo más p. Por lo tanto el número de

producciones aumenta, a lo más, p veces por cada símbolo no terminal $A_j \in V_N$.

En total, si n es $|V_N|$ y p es $|P|$, la gramática crece np veces en cuanto a su número de producciones. Sea $p' = np$.

El siguiente paso es eliminar a las producciones que sean recursivas por la izquierda, sustituyéndolas por otras que sean equivalentes.

Si consideramos el peor caso, es decir, si todas las producciones son recursivas por la izquierda, entonces $V_N$ aumenta en p' símbolos, es decir $|V_N'| = |V_N| + |P'|$, donde P' es el conjunto de producciones después de la primera transformación y $|P'|$ es igual a np. El número de producciones aumenta en forma cuadrática respecto a la gramática original. Entonces, al eliminar la recursividad izquierda, se eleva al cuadrado el número de producciones.

Por lo tanto, después de reemplazar las producciones de la forma $A_i \rightarrow A_j\alpha$ con $j<i$ y de eliminar la recursividad izquierda, tenemos que la gramática original puede crecer hasta $(np)^2$ veces.

Si por ejemplo tuviéramos una gramática en forma normal de Chomsky con veinte símbolos no terminales y cincuenta producciones, es decir con $|V_N| = 20$ y $|P| = 50$, al ser transformada hasta eliminar la recursividad izquierda, tenemos que podría llegar a crecer, por ejemplo, hasta a 1,000,000 de producciones, con alrededor de 400 símbolos no terminales, lo cual implica un gran aumento en la tabla.

Cuando se cambian las producciones de los símbolos no terminales originales y agregados para que comiencen con un símbolo terminal, la gramática crece aún más en cuanto al número de producciones, sin embargo, la tabla ya no crece puesto que el número de símbolos no terminales permanece igual, ya que simplemente se van haciendo sustituciones empezando por el "penúltimo" símbolo no terminal (cuyas producciones comienzan con terminales) hasta que todos los símbolos no terminales con los que inicia cada producción hayan sido reemplazados. Por esta razón no interesa hacer el cálculo del crecimiento de la gramática en este paso.

Ejemplo

sea G una gramática en forma normal de Chomsky, donde:

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

$V_N = \{S, A, B\}$
$V_T = \{a, b\}$
$n = 5$, y
P =

| | |
|---|---|
| $S \rightarrow AB$ | $B \rightarrow SA$ |
| $A \rightarrow BS$ | $B \rightarrow a$ |
| $A \rightarrow b$ | |

Al ordenar el conjunto de símbolos no terminales obtenemos {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, hay que eliminar a la producción $B \rightarrow SA$, sustituyéndola por las producciones derivables de S. Puesto que los árboles derivables de S son:


Obtendremos la siguiente gramática:

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

En donde ya no existen producciones que comiencen con símbolos no terminales "menores" que el símbolo del lado izquierdo de la producción.

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 el resto de las producciones de

B son B → bBA y B → a, agregamos las producciones B → bBAB$_1$ y B → aB$_1$ para B y para B$_1$ adicionamos las producciones B$_1$ → SBA y B$_1$ → SBAB$_1$. Así obtenemos la siguiente gramática:

| | |
|---|---|
| S → AB | B → bBAB$_1$ |
| A → BS | B → aB$_1$ |
| A → b | B$_1$ → SBA |
| B → bBA | B$_1$ → SBAB$_1$ |
| B → a | |

Paso 2 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 → BS que comienza con un no terminal. Las producciones de B son B → bBA, B → a, B → bBAB$_1$ y B → aB$_1$. Ahora agregamos las producciones para A, obteniendo A → bBAA, A → aS, A → bBAB$_1$S y A → aB$_1$S y eliminamos la producción A → BS.

De la misma forma debemos proceder para eliminar la producción del símbolo no terminal B, donde S → AB. De esta manera obtenemos la siguiente gramática:

| | |
|---|---|
| S → bBAAB | A → b |
| S → aSB | A → aB$_1$S |
| S → bBAB$_1$SB | B → bBA |
| S → aB$_1$SB | B → a |
| S → bB | B → bBAB$_1$ |
| A → bBAA | B → aB$_1$ |
| A → aS | B$_1$ → SBA |
| A → bBAB$_1$S | B$_1$ → SBAB$_1$ |

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

Paso 3 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 de B$_1$: B$_1$ → SBA, B$_1$ → SBAB$_1$ y las de S:

$S \rightarrow bBAAB$, $S \rightarrow aSB$, $S \rightarrow bBAB_1SB$, $S \rightarrow aB_1SB$, $S \rightarrow bB$

Es así como obtenemos la gramática equivalente a la original, en forma normal de Greibach:

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

Este ejemplo sirve para ilustrar la manera en que puede llegar a crecer una gramática, ya que iniciamos con una en forma normal de Chomsky con 5 producciones y al transformarla a forma normal de Greibach hemos obtenido una con 24. A su vez esta gramática tiene que ser factorizada por la izquierda lo que implica que el número de producciones aumentará todavía más, junto con el número de símbolos no terminales.

Hemos visto que el problema central en el diseño de un compilador es la selección del método encargado de generar la descripción estructural del lenguaje, es decir, en la elección del analizador sintáctico y por supuesto en la gramática que describe al lenguaje de programación.

Para que la construcción automática de un analizador no recursivo por predicción sea posible, es necesario someter a la gramática libre de contexto a este proceso de transformación, y detectar en su caso, si cumple o no con las características requeridas por dicho reconocedor.

El problema que esto ocasiona es que la gramática puede llegar a crecer mucho y el autómata trabaja en base a tablas, construidas a partir de la gramática, a través de las cuales podrá distinguir si una cadena de entrada es válida o no para el lenguaje.

Como es de suponerse, estas tablas dependen del número de producciones de la gramática, así como del número de elementos del vocabulario $V = V_T \cup V_N$ de ésta, es decir del número de símbolos terminales y no terminales.

Ahora bien, el manejo de estas tablas se vuelve problemático y difícil si éstas son muy grandes.

**Utilización de tablas.**

Los procesos en los que se requiere de tablas se listan a continuación. En primer lugar, el que se encarga de diagnosticar si una gramática es recursiva por la izquierda y el que tiene como tarea construir la tabla de acción del autómata.

En cada uno de ellos, se utiliza el algoritmo de Warshall, para obtener los resultados deseados. Más adelante describiremos a este algoritmo (Capítulo VI).

Para saber si una gramática es recursiva por la izquierda, se requiere construir a la matriz de relación **F** (definida para el algoritmo de diagnóstico de recursividad izquierda) y aplicar el algoritmo de Warshall para obtener la cerradura transitiva de **F**. Esta matriz es de (n x n), donde n es el número de símbolos terminales y no terminales de la gramática.

Para construir la tabla de acción del autómata, es necesario calcular a los conjuntos FIRST y FOLLOW, donde

El conjunto FIRST de una cadena $\alpha$ se define como:

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

Para poder calcular de manera sencilla este conjunto, se debe construir la matriz **F** (definida para el algoritmo de diagnóstico de recursividad izquierda) y utilizar el algoritmo de Warshall para obtener la cerradura transitiva de **F** denotada por $\mathbf{F}^+$.

La construcción del conjunto FIRST de un símbolo terminal es trivial dado que para cada a $\epsilon$ $V_T$ FIRST(a) = {a}

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

Para cada A $\epsilon$ $V_N$

$$FIRST(A) = \{ x \mid x \epsilon V_T \text{ y } A\ F^+\ x \text{ ó } x=\lambda \text{ y } A \epsilon V_{Ne} \}$$

Sea $\alpha \epsilon (V_N \cup V_T)$ y $\alpha=X_1X_2\ldots X_n$, entonces

$FIRST(\alpha) = FIRST(X_1)$

y si $X_1 \epsilon V_{Ne}$, entonces

$FIRST(\alpha) = FIRST(X_1) \cup FIRST(X_2)$

y si $X_1, X_2 \epsilon V_{Ne}$, entonces

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

$FIRST(\alpha) = \bigcup_{i<j} FIRST(X_i)$ si $X_i \epsilon V_{Ne}$ $\forall$ i<j

Si todas las $X_i \epsilon V_{Ne}$ (ver pag. 22), entonces FIRST($\alpha$) también contiene a $\lambda$.

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

FOLLOW(A) = { a | a $\epsilon$ $V_T$ y S $\rightarrow^*$ $\alpha$Aa$\beta$ para cualesquiera $\alpha$ y $\beta$ y donde S es el símbolo inicial}

Otra forma de definir a este conjunto es:

FOLLOW(A) = { a | a $\epsilon$ $V_T$ y S $\rightarrow^*$ $\alpha$A$\beta$ con a $\epsilon$ 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 $F^+$, que denotaremos por $F^*$, y $F^* = I \wedge F^+$, donde I es la matriz identidad.

Ahora definiremos la nueva relación B de la siguiente forma:

Dada una producción

$$X \rightarrow Y_1 Y_2 \ldots Y_n, \quad 1 \leq i \leq n$$

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

$$X \rightarrow Y_1 Y_2 \ldots Y_n, \quad X \in V_N \text{ y } Y_1 \in (V_N \cup V_T)$$

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,

$$\text{FOLLOW}(A) = \{ a \mid a \in V_T \text{ y } A\ (L^* B F^*)\ a \}$$

# IV. MEMORIA VIRTUAL

**Propuestas para el manejo de tablas**

Existen dos formas de trabajar con las tablas en la memoria principal de la computadora, es decir trabajar con la memoria RAM, esto implica un tiempo de respuesta rápido, pero con poca capacidad de almacenamiento, pues el costo por unidad almacenada es muy alto.

La primera sería a través de arreglos. Este método implica un manejo *estático* de la memoria, es decir, aquí se define de antemano el tamaño del arreglo, que para los fines que perseguimos, es obvio que tendría que definirse un arreglo cuyas dimensiones sean lo más grande posible.

En caso de que el tamaño de la gramática y su vocabulario sea muy pequeño, estaríamos desperdiciando recursos, en cuanto a memoria se refiere. Aunque, el problema real de utilizar arreglos es que el espacio disponible es insuficiente y se satura muy rápidamente. Este método nos restringe a trabajar únicamente con gramáticas con pocas producciones.

La segunda alternativa que tenemos es trabajar mediante apuntadores, en donde el manejo de la memoria es *dinámico*, es decir, que se van ocupando o desocupando localidades conforme se vayan necesitando. Claramente este método es mejor y más eficiente que el anterior además de que tiene mayor capacidad, pero para trabajar con tablas de un analizador sintáctico sigue siendo insuficiente porque el espacio disponible en memoria seguirá agotándose, a menos que, como en el caso anterior, se trabaje con gramáticas con pocas producciones.

Una buena alternativa para manejar este tipo de tablas que sabemos que pueden llegar a ser de gran tamaño, sería no trabajar sólo con la memoria principal de la máquina; esto se logra usando otros dispositivos de memoria, como por ejemplo diskettes o disco duro. A los sistemas que utilizan este tipo de memoria se les denomina sistemas de memoria virtual.

Antes que nada describiremos lo que es el concepto de memoria virtual.

**Memoria virtual.**

La memoria virtual ha sido diseñada para resolver los problemas concernientes a asignación dinámica de memoria, ubicación repetitiva de programas en un ambiente de multiprogramación, problemas de traslape (overlay) y errores ocasionados por correr programas o datos que crecen mucho en número.

Para lograr una utilización eficiente de la memoria principal, el sistema debe proveer un mecanismo de asignación de memoria que cumpla con las siguientes características:

1. Hacer disponible a la memoria principal para acomodar otros programas y datos una vez que los programas residentes y sus datos no sean utilizados;

2. Acomodar los programas y datos nuevos en alguna localidad disponible en la memoria principal; y

3. Manejar la asignación dinámica de memoria, puesto que la llegada de nuevos programas en un ambiente de multiprogramación es inpredecible.

El sistema mantiene una lista de espacio disponible. Este sistema está organizado por bloques de distintos tamaños. Al iniciar el sistema su ejecución, esta lista comprende a toda la memoria y resulta ser que los bloques son contiguos. Conforme se va solicitando espacio, el sistema recorre su lista de espacio disponible y mediante algún algoritmo (First Fit, Best Fit, etc.) asigna uno o más bloques. El sistema puede combinar bloques, siempre y cuando éstos estén contiguos en la memoria.

Como los bloques no se asignan secuencialmente, llega un momento en que los bloques disponibles están dispersos y son, en general, pequeños. A esto se le llama "Fragmentación de la memoria". Cuando esto sucede, aún cuando en total hay suficiente espacio para asignar, ninguno de los bloques por sí solo, es capaz de satisfacer la demanda de espacio. Surge entonces la necesidad de desalojar y compactar la memoria.

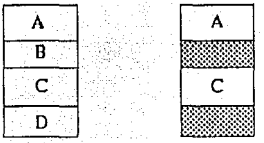
Muchas veces un programa requiere de más espacio del que

está dispuesto a darle el sistema. Entonces se deben encontrar mecanismos para que el programa, en cada momento, ocupe únicamente el espacio relevante o "vivo". Uno de esos mecanismos es el que conocemos como Memoria Virtual.

Para entender la importancia y el concepto de memoria virtual, daremos ejemplos de como se trabajaría en un ambiente de multiproceso con algunos procesos residentes en memoria, en donde se necesita de una utilización más eficiente de ésta.

Ejemplo.

Supongamos que al tiempo $t_0$, la memoria principal está ocupada por los procesos A,B,C y D. Al tiempo $t_1$, los procesos B y D no se encuentran activos, entonces la memoria que ocupan puede ser utilizada por alguien más:



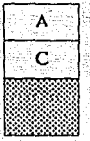

Al tiempo $t_0$ Al tiempo $t_1$

Podemos observar en la figura anterior que la memoria disponible está conformada por dos bloques separados. Supongamos que deseamos colocar al proceso E en la memoria, si el tamaño del programa es mayor que cualquiera de los dos bloques pero menor que el total de la memoria disponible, entonces el sistema debe fragmentar al proceso E en dos partes o consolidar los bloques disponibles en uno antes de colocar al proceso E.

Si se decide fragmentar el proceso, el sistema administrador de memoria debe guardar la dirección de cada fragmento del proceso. Puesto que la fragmentación de la memoria tiende a empeorar conforme transcurre el tiempo, la tarea de almacenar la dirección del proceso fragmentado llegará a

constituir una carga "administrativa" intolerable.

El sistema puede optar por "compactar" los bloques disponibles en memoria, es decir, puede consolidar a los bloques moviendo procesos y sus datos en la memoria. El resultado de esta operación se muestra en la siguiente figura:




Al tiempo $t_2$

Esto permitiría colocar fácilmente al nuevo proceso E. Sin embargo al compactar los bloques se presentan dos problemas:

1. Puesto que la operación de compactar se realiza de la Memoria Principal a la Memoria Principal, la ejecución del proceso actual deberá ser suspendida porque algún otro proceso necesita cargarse.

2. El proceso y los datos que deben ser movidos (el proceso C en el ejemplo) pueden contener direcciones dependientes de información. Puesto que las direcciones de las constantes son guardadas en la memoria como números, no es posible al momento de compactar distinguir direcciones de constantes. Es decir, al compactar no se pueden actualizar las direcciones absolutas con respecto a la nueva localización del proceso.

La falla de "compactar bloques" se debe al hecho de que, en general, una vez que el programa ha sido ubicado estáticamente, no puede ser reubicado.

Para revisar mecanismos de asignación dinámica de memoria más eficientes, debemos proveer una solución al problema de

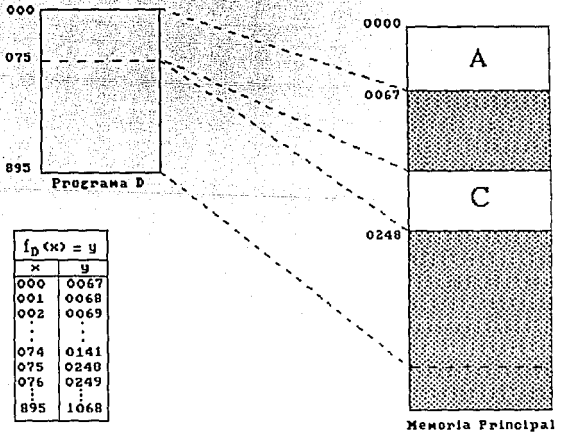
reubicar un programa que ya ha sido colocado una vez.

Una solución a este problema reside en la habilidad del sistema de proveer una función que pueda "recordar" la dirección original del proceso y su nueva ubicación correspondiente. La función puede recordar la única correspondencia entre la dirección original y la nueva. Matemáticamente, diremos que la función es 1 a 1.

Más formalmente, sea **f** una función 1 a 1 cuyo dominio de definición es el conjunto **V** de todas las direcciones del programa, y cuyo rango es el conjunto **M** de todas la direcciones de memoria. En símbolos tenemos

$$f(a_i) = b_i, \quad \text{donde } a_i \in V \text{ y } b_i \in M$$

La correspondencia exacta entre la dirección del programa y la dirección en memoria está provista por la función **f**.




$f_D(x) = y$

| x | y |
|---|---|
| 000 | 0067 |
| 001 | 0068 |
| 002 | 0069 |
| ⋮ | ⋮ |
| 074 | 0141 |
| 075 | 0240 |
| 076 | 0249 |
| ⋮ | ⋮ |
| 895 | 1068 |

De la figura anterior observamos lo siguiente:

1. Puesto que $f_p$ mantiene la correspondencia, la dirección del proceso es la misma y es indiferente a su dirección en memoria. Entonces no es necesaria la reubicación del proceso.

2. Puesto que $f_p$ almacena la dirección de la localidad en memoria, no le importa al proceso si la localidad está separada o continua.

3. La asignación dinámica de memoria para el proceso equivale a una inspección de la memoria disponible y a la construcción de la función $f_p$.

Con la disponibilidad de $f_p$, la reubicación del proceso C y compactar los bloques son innecesarios. Entonces, el sistema debe tener la capacidad de construir y usar la función $f_p$.

Observemos que un proceso ejecutable D, puede ser ubicado dinámica y repetitivamente en cualquier espacio disponible mientras exista una función $f_{pt}$ actualizada después de cada ubicación al tiempo t.

**Dirección virtual, espacio virtual y traducción de direcciones.**

La dirección del programa es utilizada por el sistema para calcular la dirección en memoria basada en la función $f_p$. Una vez que la dirección de la memoria es calculada, se utiliza como referencia actual. Al proceso de calcular la dirección de la memoria se le denomina "traducción de memoria", y la función $f_p$ es generalmente llamada "función de redireccionamiento" o "tabla de redireccionamiento" (si la función está en forma tabular). Para diferenciar a la dirección del programa de la de memoria, a la primera se le nombra *dirección virtual* y a su espacio de direccionamiento *espacio virtual*, mientras que el último es denominado

*dirección física* y su espacio de direccionamiento *memoria física*. Después de que el programa es cargado en la memoria física y antes de su ejecución, la función de traducción es generada para el programa. Para cada referencia al espacio virtual durante la ejecución, la dirección virtual es traducida a la dirección física mediante la función de traducción. La dirección física es entonces usada por el sistema para referencia a memoria física. Debería ser claro que ahora la memoria en la cual la información reside no es la misma en la cual es referenciada por el programa. Cuando un sistema de computación emplea el espacio virtual, es llamado *sistema de memoria virtual*.

Este tipo de sistema utiliza medios de almacenamiento externo de acceso directo (disco) para mantener una imagen completa del espacio de direccionamiento físico del programa, de tal manera que cada vez que se desaloja a un bloque de memoria central, la imagen en disco es actualizada. Esto hace, por supuesto, que la ejecución del programa sea más lenta pero se logra la ejecución de programas que requieren de una cantidad de espacio que no está disponible, ya sea por los mecanismos de asignación de memoria o por el tamaño físico de la memoria.

**Paginación de memoria**

Lo expuesto anteriormente indica que por cada proceso en el sistema debe existir una función de traducción asociada $f_t$ de ubicación al tiempo **t**. Puesto que estos procesos pueden ser dinámica y repetitivamente colocados en memoria, es difícil desarrollar fórmulas para cada función de redireccionamiento $f_t$. Por lo tanto, se utilizan tablas de redireccionamiento y el cambio de las tablas de traducción de $f_{t1}$ a $f_{t2}$ implica únicamente actualizar la tabla $f_{t1}$. Esta es la razón principal por la cual la mayoría de los sistemas de espacio virtual utilizan tablas de redireccionamiento.

Analicemos ahora el caso en el que un proceso ocupa un espacio mayor o igual que la memoria física. Entonces la tabla de redireccionamiento tendrá muchas entradas, y quizás la memoria central se saturará al almacenar la tabla no dejando espacio para el programa, o bien puede suceder que el programa requiera de un espacio mayor que la memoria central misma.

Una solución práctica a este problema es "paginar la memoria", de tal forma que sea posible manejar datos o programas que se encuentran almacenados en un espacio virtual mayor que la memoria física.

En primer lugar, es necesario organizar a la memoria física y al espacio virtual de la siguiente manera:

1. Dividir la memoria física en unidades iguales de palabras o bytes. A cada una de estas unidades les llamaremos bloques.

2. Dividir el espacio virtual en unidades iguales, denominadas páginas, cuyo tamaño es el mismo que el asignado a los bloques.

3. Construir una tabla de redireccionamiento cuyas entradas asocien a las páginas del programa y los bloques en los cuales residen las páginas.

Notemos que el número máximo de entras en la tabla de traducción es igual al número de páginas disponibles en la memoria virtual.

Si hacemos $2^m$ igual al tamaño del espacio virtual, $2^n$ igual al tamaño de la memoria física, y $2^k$ igual al tamaño del bloque o página, todo en números de palabras o bytes, de aquí tenemos que la dirección virtual mide m bits y el número máximo de bits que pueden ser usados para numerar páginas es (m-k). A este (m-k) se le denomina "número de página". Similarmente, (n-k) bits es llamado "número de bloque". De (3) tenemos que, las entradas en la tabla de traducción asocian a los números de página con los de bloque.

El proceso de paginación debe ser capaz de conocer si una página:

1. fue modificada o no;

2. está presente o ausente;

3. es para uso exclusivo del sistema administrador de la memoria o no;

4. es de sólo lectura o lectura/escritura;

5. es media página o página completa; y finalmente también deberá ser capaz de:

6. conocer la dirección de la media página, en caso de que lo sea.

Durante el redireccionamiento de la memoria, la información de status (2), indicará si la página que está siendo referida está ausente en la memoria física. En este caso, la página no está en ningún bloque, resultando un error de página. El error de página se da mediante una interrupción de hardware la cual indica al sistema administrador de memoria que debe iniciar el proceso de paginación para cargar la página deseada en la memoria física, desde el dispositivo de almacenamiento, y para colocarla en un bloque disponible. Mientras la página se carga en memoria, el sistema operativo debe pasar el control a otro programa para reemplazar a la tabla de redireccionamiento actual con la tabla de redireccionamiento del otro programa.

La rutina de paginación del sistema administrador de memoria es responsable de manejar el dispositivo de almacenamiento (disco) y de guardar o cargar páginas. Contrario al proceso de compactar mencionado anteriormente, la paginación no es una operación de memoria principal a memoria principal. Es, en primer lugar, una operación de Entrada/Salida (I/O).

Cuando el espacio virtual es mayor que la memoria física, existen los siguientes requerimientos adicionales:

1. La necesidad de dispositivos de almacenamiento como por ejemplo el disco.

2. La capacidad del sistema administrador de la memoria para cargar páginas y para guardar páginas (nuevas o actualizadas) desde y hacia el dispositivo de almacenamiento. Al proceso de cargar páginas se le llama *operación de entrada de páginas* y al proceso de almacenar se le conoce como *proceso de salida de páginas*. Ambos son referidos como *paginación*.

3. Un director del mapeo del espacio virtual y del mapeo de la memoria física. El primero se utiliza para guardar la dirección de las unidades del dispositivo de

almacenamiento utilizadas para las páginas; el último, en los bloques de la memoria principal para la paginación.

El uso del mapeo de la memoria física habilita al sistema de memoria virtual para soportar varias tablas de traducción (es decir, varios espacios virtuales). Puesto que las páginas de diferentes espacios virtuales pueden estar en los bloques, el mapeo puede mostrar para un bloque dado el espacio del cual proviene la página que le pertenece. Cuando los bloques disponibles son escasos, ciertas páginas deben ser "eliminadas" de la memoria física. En este caso, los espacios virtuales de las páginas eliminadas deben ser identificados. Una vez identificados, sus tablas de traducción correspondientes pueden ser localizadas y las entradas para estas páginas pueden ser entonces actualizadas.

**Uso del concepto de espacio virtual.**

En nuestro caso, requerimos, como ya lo mencionamos, de matrices muy grandes y hemos utilizado el concepto de espacio virtual para poder manejarlas. El dispositivo de almacenamiento de la matriz de (m x n) es el disco (espacio virtual). La memoria física es simplemente una submatriz y la tabla de redireccionamiento tiene una sola entrada en donde únicamente se guardan el renglón i y la columna j a partir de la cual se extrajo la página de la matriz virtual.

A través de este trabajo hemos hablado del manejo de matrices, sin embargo, en la implementación usamos vectores. De tal forma que, en una matriz, el elemento que se encuentra en el renglón i, columna j, equivale al elemento que está en la posición i*m+j del vector que estamos usando.

En los programas hemos definido a los bloques de tamaño (M x N), donde M = N = 32[1] y las páginas, por supuesto, miden lo mismo que los bloques.

---

[1]En caso de que el usuario necesite modificar estos valores puede hacerlo puesto que están definidos a través de la instrucción # define en "matvirt.h"

Para determinar cuál es la nueva dirección i',j' dentro del bloque del elemento i,j, tenemos que: si $i_0$,$j_0$ son los valores almacenados en la tabla de redireccionamiento $i' = i - i_0$ y $j' = j - j_0$.

Para saber si el elemento i,j está contenido en el bloque actual, simplemente hay que revisar que i sea mayor o igual que $i_0$ y menor o igual que $i_0 + M$ y que j sea mayor o igual que $j_0$ y menor o igual que $j_0 + N$. Cuando encontramos que el elemento i,j no se encuentra en el bloque, se cargará en la memoria una nueva página a partir de i,j, asegurándonos de esta manera que contiene al elemento que requerimos.

Todavía nos falta describir como es que se determina el tamaño de la página que se carga en la memoria, puesto que, puede suceder que no siempre se extraiga una página de (32 x 32), sino de dimensiones menores. El mecanismo en realidad es sencillo. Si (m - i) > M entonces el número de renglones de la página será M, en caso contrario los renglones deberán ser igual a (m - i), sucede lo mismo para determinar el número de columnas del bloque: Si (n - j) > N entonces la página contendrá N columnas, de no ser así el número de columnas será (n - j).

En las rutinas que definen el proceso de paginación desarrollado en este trabajo la decisión de a quién sacar es trivial puesto que cuando se requiere de otra página simplemente hay que desalojar a la actual y cargar en memoria a la página solicitada. La página nueva se tomará a partir del elemento i,j que se le pase como parámetro a la función.

De acuerdo a lo explicado anteriormente, si el usuario llegara a requerir trabajar con más de una matriz virtual, será necesario que defina un bloque y una tabla de redireccionamiento para cada una. Por ejemplo, en la aplicación de multiplicación de matrices desarrollada para este trabajo, tenemos tres matrices virtuales, las dos que se van a multiplicar y la resultado, entonces definimos tres bloques en memoria a los que llamamos A, B y R; así mismo también se requiere, para cada matriz de una tabla de redireccionamiento, en el programa les llamamos $A_0$, $B_0$ y $R_0$ (la manera de hacer estas definiciones se explican en el capítulo de instrumentación del módulo).

Generalmente, en computación, cuando se gana espacio, se pierde tiempo de proceso y este caso no es la excepción.

Cuando los datos residen en la memoria principal, la computadora puede accesarlos muy rápidamente. Por esta razón, al utilizar nosotros medios de almacenamiento externo (disco), estamos incrementando el tiempo de proceso. No sólo porque es más tardado accesar al disco, sino también porque estamos paginando la memoria y el procedimiento de cargar y desalojar bloques hace todavía más lento el proceso. Sin embargo, como habíamos mencionado anteriormente, a través de los mecanismos de memoria virtual es posible que se ejecuten programas que requieren de una cantidad de espacio que no está disponible. En el problema que nos ocupa, el tiempo de proceso no es muy relevante puesto que los procedimientos se ejecutan una sola vez.

# V. INSTRUMENTACION DEL MODULO

**Generalidades.**

El módulo de utilerías, consta de un conjunto de funciones y procedimientos que proveen las herramientas necesarias para el manejo de matrices virtuales.

Así mismo, también se crearon dos programas que realizan la multiplicación lógica de matrices virtuales y uno más para el algoritmo de Warshall.

Estos programas han sido creados para ser usados por otros programas, por lo que no manejamos menús ni salidas en pantalla.

En primer lugar, definiremos las condiciones que deben cumplirse para poder utilizar la biblioteca y los programas de aplicación (multiplicación de matrices y algoritmo de Warshall).

**Requerimientos para la utilización del módulo y programas de aplicación.**

1. Las matrices virtuales serán archivos tipo texto. Los datos deberán estar almacenados siguiendo el orden por renglón.

2. Los elementos de las matrices deberán ser números enteros positivos de longitud 5.

3. M, N serán palabras reservadas, pues en el módulo éstas definen la longitud del bloque en memoria física que almacena a la submatriz de la matriz virtual.

4. Para los programas que multiplican matrices y el algoritmo de Warshall, los nombres de los archivos que almacenan a las matrices virtuales y las dimensiones de estas, se definen en el archivo "parametr.h", también aquí se especifica el nombre del archivo resultado. Cuando el usuario genere sus propios programas, podrá utilizar también el archivo "parametr.h", aunque estos

datos puede declararlos directamente en sus programas o bien definir otro archivo header. En caso de que llegara a utilizar dicho archivo o que use alguno de los programas de aplicación, se considerarán como palabras reservadas ren_A, col_A, ren_B, col_B, matrizA, matrizB y matrizR.

5. La instrucción #include "matvirt.h" deberá adicionarse a los programas que utilicen el módulo de matrices virtuales.

6. Cuando se utilice la función que coloca un elemento en la matriz, será necesario que antes de cerrar el archivo, se utilice la función "actualiza" para que se guarde en la matriz virtual la última página que se tiene cargada en memoria, pues si no los últimos cambios no serán registrados.

7. Es necesario que tanto la tabla de redireccionamiento como la variable que almacena las dimensiones de la matriz se definan como variables tipo "pos", este tipo está definido en "matvirt.h". Este tipo de variable, es una estructura compuesta por dos números enteros sin signo, es decir, cualquier variable tipo "pos" consta de una parte que guarda el renglón i y otra que almacena a la columna j.

8. Las estructuras que debe tener construidas el usuario se describen en el siguiente punto, después de definir las estructuras de datos que utiliza el módulo.

**Estructuras de datos.**

Todos los programas, procedimientos y funciones utilizan las estructuras de datos que describimos a continuación:

1. Matriz virtual. Estas matrices son archivos de texto con números enteros positivos. A pesar de que el usuario maneja matrices de dos dimensiones, en realidad están internamente guardadas como vectores.

2. Matriz física o bloque. Es una submatriz de la matriz almacenada en disco. Al igual que la matriz virtual, ésta es un vector. El bloque es una "matriz" de (M x N) donde M = N = 32.

3. Coordenadas de la matriz. Es un par ordenado de números enteros positivos (i,j) que contiene al renglón i y a la columna j de la matriz.

4. Tabla de redireccionamiento. Contiene una sola entrada con las coordenadas que indican a partir de cuál elemento se está tomando la página de la memoria virtual. Por lo tanto, se reduce a una variable de tipo coordenada.

Por lo tanto, las estructuras que debe tener definidas el usuario son:

1. Matriz Virtual: Archivo tipo texto con números enteros positivos de longitud 5, almacenados siguiendo el orden por renglón.

2. Bloque: Esta es una submatriz que se tiene cargada en memoria. Los bloques son vectores con [M x N] columnas. Por supuesto este arreglo deberá contener sólo a números enteros positivos. Debe definirse un bloque por cada matriz virtual que se tenga.

3. Tabla de redireccionamiento. Esta es una variable tipo "pos"; es una estructura que guarda el renglón i y columna j a partir de la cual se extrajo la página. Debe existir una tabla de redireccionamiento para cada matriz virtual.

4. Dimensiones (m,n) de la matriz virtual. Es necesario que se guarde el número de columnas y renglones de cada matriz virtual en una variable tipo "pos".

Las declaraciones dentro de C de las últimas tres estructuras, sería la siguiente:

```
unsigned A[M*N];      /* Define el bloque */
pos A0,               /* Tabla de redireccionamiento */
    An;               /* Dim. (m,n) de la mat. virt. */
```

Donde $A_0$, se debe inicializar haciendo $A_0.i = 0$; $A_0.j = 0$ o utilizando a la rutina inicia (se detalla más adelante); y

$A_n$ debe ser igual a $A_n$.i = m; $A_n$.j = n;

**BIBLIOTECA DE PROCEDIMIENTOS**

**OBJETIVO:** Proveer las herramientas necesarias para el manejo de matrices virtuales. Este módulo podrá ser utilizado por cualquier programa que lo requiera.

A continuación detallamos a cada una de las funciones y procedimientos que incluye la biblioteca. Se han marcado con * las funciones que son de usuario final.

**FUNCION: inicia***

**OBJETIVO:** Abre el archivo de la matriz virtual, inicializa la tabla de redireccionamiento en (0,0) y carga en memoria la primer página o submatriz de la matriz virtual.

**PARAMETROS:**

1. Nombre del archivo de la matriz virtual.
2. Modo de apertura. Los valores que deben pasarse a través de este parámetro son iguales a los utilizados por la función fopen: "r", "w", "a", "r+", "w+", "a+".
3. Apuntador al bloque.
4. Dimensiones de la matriz (m,n).
5. Tabla de redireccionamiento.
6. Variable que indica si se han modificado los valores de algún elemento de la matriz.

**SALIDA:**

Regresa el apuntador al archivo de la matriz Virtual o NULL si hubo error al abrir el archivo.

**METODO:**

1. Abre el archivo de la matriz virtual.
2. Inicializa en ceros el bloque y en (0,0) a la tabla de redireccionamiento.
3. Trae a la memoria principal una submatriz a partir de la posición (0,0).
4. Regresa el apuntador al archivo o NULL si hubo algún error.

**FUNCION: crea***

**OBJETIVO:** Crea una matriz virtual en ceros de (m x n).

**PARAMETROS:**

1. Nombre del archivo que almacenará a la matriz virtual.
2. Dimensiones (m,n) de la matriz.

**SALIDA:**

Apuntador al archivo o NULL si hubo error al crear la matriz.

**METODO:**

1. Crea el archivo que contendrá a la matriz.
2. Llena con ceros la matriz.
3. Cierra el archivo y lo abre en modo de Lectura-Escritura.
4. Regresa el apuntador al archivo o NULL si hubo errores.

**FUNCION: trae_pag***

**OBJETIVO:** Extrae de la matriz virtual una submatriz que contenga al elemento (i,j).

**PARAMETROS:**

1. Apuntador al archivo de la matriz virtual.
2. Apuntador a la submatriz en la memoria física.
3. Las coordenadas i,j que indica a partir de que posición se extraerá la submatriz.
4. Las coordenadas (m,n), donde m es el número de renglones y n es el número de columnas de la matriz.
5. La tabla de redireccionamiento en la cual se guarda la dirección a partir de la que se extrajo la página que está cargada actualmente en memoria.
6. Una bandera que indica si se hicieron cambios en los datos o no.

**SALIDA:**

Si no hubo errores devuelve 0 o -1 en otro caso.

**METODO.**

1. Si hubo algún cambio en la página que se tiene cargada en la memoria actualmente, entonces guardarla en disco.
2. Actualiza la tabla de redireccionamiento con los valores (i,j).
3. Revisa si se tomará la página del tamaño del bloque o más pequeña.
4. Coloca en la memoria principal la submatriz a partir del elemento i,j.
5. Si hubo errores devuelve -1 y 0 en caso contrario.

**FUNCION: extrae***

**OBJETIVO:** Toma el elemento que se encuentra en la posición i,j de la matriz virtual.

**PARAMETROS:**

1. Apuntador al archivo de la matriz virtual.
2. Apuntador a la submatriz en memoria física.
3. Coordenadas i,j
4. La coordenada (m,n), donde m es el número de renglones y n es el número de columnas de la matriz.
5. Tabla de redireccionamiento.
6. Variable que indica si se han modificado los valores de algún elemento de la matriz.
7. Variable que indica si hubo errores o no al extraer el elemento.

**SALIDA:** El elemento i,j

**METODO:**

1. Revisa si la página que contiene al elemento i,j está en memoria, si no está, carga en memoria la página que lo contiene.

2. Si no hubo errores regresa el elemento i,j o cero si existe algún error.

**FUNCION: coloca***

**OBJETIVO:** Modifica el valor del elemento i,j de la matriz virtual.

**PARAMETROS:**

1. Apuntador al archivo de la matriz virtual.
2. Apuntador al vector en memoria física.
3. Valor que tomará el elemento i,j.
4. Posición i,j del elemento que se modificará
5. Dimensiones (m,n) de la matriz.
6. Tabla de redireccionamiento.
7. bandera que indica si hubo cambios.
8. Variable que indica si hubo errores o no al colocar el elemento.

**METODO:**

1. Revisa si la página que contiene al elemento i,j está en memoria, si no está la carga.
2. Modifica el valor del elemento i,j.
3. Prende la bandera que indica que hubo cambios.

**FUNCION: enmemoria**

**OBJETIVO:** Revisa si la página que contiene al elemento i,j está presente en memoria.

**PARAMETROS:**

1. Posición i,j.
2. Tabla de redireccionamiento.

**SALIDA:**

Cierto si i,j está incluido en la página actual y Falso en caso contrario.

**METODO:**

1. Revisa que i sea mayor o igual que el renglón que está guardado en la tabla de redireccionamiento y menor o igual que el renglón de la tabla más M (M es igual al número de renglones del bloque). Efectúa la misma revisión para j, es decir, verifica que j sea mayor o igual que la columna que está guardada en la tabla de redireccionamiento y menor o igual que la columna de la tabla más N, donde N es igual al número de columnas del bloque.

2. Si la coordenada i,j está dentro de la página devuelve CIERTO y FALSO en otro caso.

**FUNCION: actualiza***

**OBJETIVO:** En caso de que se haya hecho una modificación en uno o más elementos de la submatriz que se encuentra en memoria, se actualiza el cambio reescribiéndola en disco.

**PARAMETROS:**

1. Apuntador al archivo de la matriz virtual.
2. Apuntador al bloque.
3. Dimensiones de la matriz.
4. Tabla de redireccionamiento.
5. Bandera para indicar cambios.

**SALIDA:**

-1 si hubo errores o 0 en caso contrario.

**METODO:**

1. Revisa si la página se toma del tamaño del bloque o más

pequeña.

2. Si hubo modificaciones reescribe la submatriz en disco, a partir de la posición i,j de la tabla de redireccionamiento.

3. Si hubo errores devuelve -1 y cero si todo salió bien.

**FUNCION: enceros***

**OBJETIVO:** Inicializa con ceros la matriz que se le pase como parámetro.

**PARAMETROS:**

1. Apuntador al bloque.

**SALIDA:** El bloque con todos sus elementos en ceros.

**METODO:**

1. El algoritmo de inicialización es:

```
Begin
   Para i = 0 hasta M
      Para j = 0 hasta N
         bloque[i,j] = 0
End
```

**FUNCION: copia***

**OBJETIVO:** Copia el contenido de la matriz virtual A de (m x n) hacia la matriz virtual B.

**SALIDA:**

Cero si los archivos se copiaron correctamente o -1 en caso contrario.

**PARAMETROS:**

1. Nombre del archivo fuente.

2. Nombre del archivo destino.

**METODO:**

1. Abre el archivo fuente y destino para lectura y escritura respectivamente.

2. Si no hubo errores en la apertura de los archivos:

```
Begin
   Minetras no sea fin de archivo
      Lee_elemento(A);
      escribe_elemento(B);
End;
```

3. Cierra los archivos fuente y destino.

**PROGRAMA 1: Multiplicación de matrices.**

**OBJETIVO:** Multiplicar lógicamente dos matrices virtuales, utilizando las funciones de extrae y coloca de la biblioteca de procedimientos.

**ENTRADA:**

1. El archivo de texto que contenga los datos de la primera matriz que se va a multiplicar.

2. El archivo de texto, cuyo contenido serán los datos correspondientes a la segunda matriz.

Dichos archivos deberán almacenar sólo ceros y unos o números enteros positivos, donde el número 1 o cualquier entero mayor que cero significa CIERTO y 0 es igual a FALSO.

3. Las dimensiones de cada una de las matrices.

4. El nombre del archivo en el cual se guardará el resultado de la multiplicación de las matrices A y B.

* Los nombres de los archivos y las dimensiones de las matrices son especificadas dentro de parametr.h

**SALIDA:**

Un archivo tipo texto que es el resultado de la multiplicación lógica de las matrices A y B. Sólo contendrá ceros y unos, donde uno significa cierto y cero falso.

**METODO:**

1. Abre los archivos de las matrices A, B y R. Trae a la memoria física la primer submatriz de A y B, tomadas a partir del primer renglón y primera columna.

2. Revisa que el número de columnas de la matriz A sea igual al número de renglones de la matriz B. En caso de que sean diferentes se cancelará la ejecución del programa.

3. Inicializa en ceros el bloque de la matriz resultado.

4. Inicia el proceso de multiplicación utilizando el siguiente algoritmo:

Begin

```
   Para k desde 0 hasta col_B - 1
      Para i desde 0 hasta ren_A - 1
         c = 0;
         Para j des de 0 hasta col_A - 1
            c = c OR (extrae(A,i,j) AND extrae(B,j,k));
         coloca(R,c,i,k);
End;
```

Este proceso es hasta cierto punto ineficiente porque por cada submatriz que se carga en memoria se multiplican sólo un renglón y una columna. Debido a que la multiplicación de matrices es una operación que se usa muy comúnmente, hemos desarrollado otro programa que multiplica matrices en forma más eficiente en el que cada vez que carga una submatriz en memoria, multiplica todos los renglones y columnas de la submatriz. Este utiliza únicamente la función que trae una página de la memoria virtual a la física.

**PROGRAMA 2: Multiplicación eficiente de matrices.**

**OBJETIVO:** Multiplicar lógicamente dos matrices virtuales.

**ENTRADA:**

1. El archivo de texto con los datos de la primera matriz que se va a multiplicar (matriz A).

2. El archivo de texto, cuyo contenido serán los datos de a la segunda matriz (matriz B).

   Dichos archivos deberán almacenar sólo ceros y unos o números enteros positivos, donde el número 1 o cualquier entero mayor que cero significa CIERTO y 0 es igual a FALSO.

3. Las dimensiones de cada una de las matrices.

4. El nombre del archivo en el cual se guardará el resultado de la multiplicación de las matrices A y B.

* Los nombres de los archivos y las dimensiones de las matrices son especificadas dentro de parametr.h

**SALIDA:**

Un archivo tipo texto que es el resultado de la multiplicación lógica de las matrices A y B. Sólo contendrá ceros y unos, donde uno significa cierto y cero falso.

**METODO:**

1. Abre los archivos de las matrices **A**, **B**, y **R**. Inicializa las tablas de redireccionamiento de cada una de las matrices en (0,0) y trae a la memoria física la primer submatriz de **A** y de **B** tomadas a partir del primer renglón y primera columna.

2. Revisa que el número de columnas de **A** y el número de renglones de **B** sean iguales, pues de no ser así se interrumpirá la ejecución del programa.

3. Inicializa en ceros el bloque correspondiente a la matriz resultado y a la matriz temporal, que se usará como auxiliar en el proceso de multiplicación.

4. Da inicio el proceso de multiplicación de matrices utilizando el siguiente algoritmo:

La notación que utilizaremos para describir el algoritmo de multiplicación es la siguiente:

- Para denominar a los bloques o submatrices de cada matriz usamos A, B y R.

- Ao, Bo y Ro designan a las tablas de redireccionamiento de A, B y R respectivamente.

- Dado que la multiplicación de las matrices es lógica,

tenemos que el elemento $r_{ij}$ de la matriz R se define como:

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

4. Da inicio el proceso de multiplicación de matrices utilizando el siguiente algoritmo:

```
Desde x=0 hasta que x=# Columnas de B
Begin
     Desde y=0 hasta que y=# Renglones de  A
     Begin
          Desde z=0 hasta que z=# Columnas de A
          Begin
              Inicializa(Matriz Temporal)
              Ao = (y*M,z*N)
              Bo = (z*N,x*N)
              Ro = (ren de Ao, col de Bo)
              A = trae_pag(MatrizA, a partir de Ao)
              B = trae_pag(MatrizB, a partir de Bo)
              Matriz Temporal = Multiplica(A,B)
              R = R + Matriz Temporal
          End; /* ciclo de z */
          Actualiza(R, a partir de Ro)
          Inicializa(R)
     End;  /* ciclo de y */
End;  /* ciclo de x */
```

5. Finalmente se produce el archivo que guarda el resultado de la multiplicación de las matrices A y B.

**PROGRAMA 3: Algoritmo de Warshall.**

**OBJETIVO:** Calcula la cerradura transitiva de una relación.

**ENTRADA:**

1. Un archivo de tipo texto cuyo contenido será una matriz de relación **A** con n columnas.

Los datos contenidos en este archivo deberán ser sólo ceros y unos o números enteros positivos, donde el número 1 o cualquier entero mayor que cero significa CIERTO y 0 es igual a FALSO.

2. Las dimensiones de cada una de las matrices.

3. El nombre del archivo en el cual se guardará el resultado.

* Los nombres de los archivos y las dimensiones de las matrices son especificadas dentro de parametr.h

**SALIDA:**

Un archivo tipo texto, que contendrá a la matriz de relación R, que es la cerradura transitiva de A.

**METODO:**

```
Begin
     copia(A,R);
     para k desde 1 hasta n
        para i desde 1 hasta n
           para j desde 1 hasta n
              c=extrae(R,i,j)V(extrae(R,i,k)Λextrae(R,k,j));
              coloca(R,c,i,j);
End;
```

# VI. APLICACIONES DEL MODULO

**Aplicaciones del módulo dentro de teoría de autómatas.**

Anteriormente vimos que para la construcción de un reconocedor no recursivo por predicción, es necesario contar con matrices muy grandes. También vimos que una solución a este problema es utilizar memoria virtual.

El módulo de utilerías creado para este trabajo permite manejar matrices muy grandes a través de memoria virtual.

Además se desarrolló un programa que multiplica en forma lógica matrices, utilizando, por supuesto, el módulo de utilerías. Este programa tiene su aplicación en el algoritmo de Warshall, de esta manera se elimina la restricción que obligaba a trabajar con gramáticas libres de contexto con pocas producciones puesto que no importa que tanto crezca la tabla.

A pesar de que únicamente utilizamos la biblioteca de procedimientos para desarrollar rutinas que sólo se aplican al algoritmo de Warshall, también es posible usar el módulo para trabajar con la tabla de acción del autómata. En este caso solamente se usarán las funciones que extraen o colocan un elemento a la matriz.

**Algoritmo de Warshall.**

Como hemos visto en capítulos anteriores, el algoritmo de Warshall juega un papel muy importante en la construcción de un reconocedor no recursivo por predicción.

Se usa, en primer lugar, para determinar si una gramática es recursiva por la izquierda, pues en caso de que lo sea se debe transformar para eliminar la recursividad. Por otra parte, también se utiliza para construir la tabla de acción del autómata.

A continuación presentaremos las definiciones y el algoritmo de Warshall.

**Definición 1.** Sea **R** una relación de **X** a **Y** y **S** una relación de **Y** a **Z**. Entonces la relación **R ° S** se llama la relación de composición 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) pertenece a la relación y 0 en otro caso. Entonces podemos expresar la composición **A ° B** a través de una matriz **C** donde cada elemento de **C** se define como:

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

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

$$1 \wedge 0 = 0 \wedge 1 = 0 \wedge 0 = 0 \quad y \quad 1 \wedge 1 = 1.$$

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

$$1 \vee 1 = 1 \vee 0 = 0 \vee 1 = 1 \quad y \quad 0 \vee 0 = 0.$$

**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 cerradura transitiva 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.

**Método:**

```
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;
```

Para que este algoritmo maneje matrices virtuales, habrá que utilizar las funciones extrae y coloca del módulo de utilerías, entonces el algoritmo se traduce como:

**Método:**

```
begin
  Copia(A,P);
  para k desde 0 hasta n
     para i desde 0 hasta n
        para j desde 0 hasta n
           p_ij=extrae(P,i,j)∨(extrae(P,i,k)∧extrae(P,k,j))
           coloca(P,p_ij)
  actualiza(p)
End;
```

**Otras aplicaciones del módulo.**

Debido a que este trabajo está enfocado hacia herramientas para compiladores, únicamente vimos la aplicación y utilización del módulo dentro de teoría de autómatas. Sin embargo, su aplicación es mucho más amplia, puesto que con estas rutinas es posible efectuar cualquier operación con matrices.

En el campo de las matemáticas, las matrices son de gran utilidad y se aplican dentro de muchas áreas, como son investigación de operaciones, estadística, finanzas, economía y álgebra, por mencionar algunas.

A continuación se describen los algoritmos de las operaciones más importantes con matrices:

**Algoritmo 1: Suma de matrices**

**Entrada:** Matriz A, Matriz B

**Salida:** Matriz C = A + B

**Método:**

```
Begin
   Para i desde 1 hasta n
      Para j desde 1 hasta m
         c_ij = extrae(A,i,j) + extrae(B,i,j)
         coloca(C,c_ij,i,j)
End
```

**Algoritmo 2: Multiplicación de una matriz por un escalar**

**Entrada:** Matriz A, escalar K

**Salida:** Matriz C= k*A, donde K es un escalar.

**Método:**

```
Begin
   Para i desde 1 hasta n
      Para j desde 1 hasta m
         c_ij = k * extrae(A,i,j)
         coloca(C,c_ij,i,j)
End
```

**Algoritmo 3: Multiplicación de matrices**

**Entrada:** Matriz A de (m x n), Matriz B (n x l)

**Salida:** Matriz C = A * B

**Método:**

```
Begin
   Para k desde 1 hasta l
      Para i desde 1 hasta m
         Para j desde  1 hasta n
            c = c + (extrae(A,i,j) * extrae(B,j,k))
         coloca(C,c,i,k)
End
```

# APENDICE A: BIBLIOTECA DE PROCEDIMIENTOS

```
# include <stdio.h>          /* Modulo con funciones de entrada/salida */
# include <math.h>           /* Modulo con funciones matematicas */
# define  M  10              /* Numero de renglones del bloque   */
# define  N  10              /* Numero de columnas  del bloque   */


/* DECLARACION DE TIPOS */

struct pos {                 /* POSición: renglon i, columna j. */
       unsigned i;
       unsigned j;
};

typedef struct pos pos;


/* DECLARACION DE FUNCIONES:  */

FILE * inicia( char *,       /* Nombre del archivo de la matriz virtual   */
               char *,       /* Modo de actualizacion del archivo         */
               unsigned *,   /* Apuntador al bloque                       */
               pos,          /* Dimensiones (m,n) de la matriz virtual    */
               pos*,         /* Tabla de redireccionamiento               */
               unsigned *);  /* Bandera para saber si hubo cambios        */

FILE * crea(char *,     /* Nombre del archivo de la matriz virtual */
            pos);       /* Dimensiones (m,n) de la matriz virtual  */

unsigned extrae( FILE *,     /* Apuntador al arch. de la matriz virtual*/
                 unsigned *, /* Apuntador al bloque                    */
                 unsigned,   /* Renglon I del elemento a extraer       */
                 unsigned,   /* Columna J del elemento a extraer       */
                 pos,        /* Dimensiones (m,n) de la matriz virtual */
                 pos *,      /* Tabla de redireccionamiento            */
                 unsigned*,  /* Bandera para saber si hubo cambios     */
                 int *);     /* Indica si hubo algun error             */

void coloca( FILE *,       /* Apuntador al archivo de la matriz virtual*/
             unsigned *,   /* Apuntador al bloque                      */
             unsigned,     /* Valor a colocar en i,j                   */
             unsigned,     /* Renglon I del elemento a colocar         */
             unsigned,     /* Columna J del elemento a colocar         */
             pos,          /* Dimensiones (m,n) de la matriz virtual   */
             pos *,        /* Tabla de redireccionamiento              */
             unsigned *,   /* Bandera que indica si hubo cambios       */
             int *);       /* Indica si hubo algun error               */

int  trae_pag( FILE *,     /* Apuntador al archivo de la matriz virtual  */
               unsigned *, /* Apuntador al bloque                        */
               unsigned,   /* Ren I a partir del cual se tomara la pagina*/
               unsigned,   /* Col J a partir de la que se tomara la pag. */
               pos,        /* Dimensiones (m,n) de la matriz virtual     */
               pos *,      /* Tabla de redireccionamiento                */
               unsigned *); /* Bandera que indica si hubo cambios        */

unsigned enmemoria(unsigned,   /* Ren I que se verificara si esta en bloque*/
                   unsigned,   /* Col J que se verificara si esta en bloque*/
                   pos);       /* Tabla de redireccionamiento              */

int actualiza( FILE *,     /* Apuntador al archivo de la matriz virtual */
               unsigned *, /* Apuntador al bloque                       */
```

```
                pos,            /* Dimensiones (m,n) de la matriz virtual    */
                pos,            /* Tabla de redireccionamiento               */
                unsigned *);    /* Bandera para indicar si hubo cambios      */

void enceros(unsigned *);       /* Apuntador al bloque */


int       copia(char *,         /* Nombre del archivo fuente */
                char *);        /* Nombre del archivo destino */



/*-------------------------------------------------------------------------*/
/*                          F U N C I O N E S                              */
/*-------------------------------------------------------------------------*/


/* INICIA -> Inicializa en ceros al  bloque  (o submatriz), y  carga en
             la memoria principal la primer pagina de la matriz virtual,
             a partir del renglon 0, columna 0. */

FILE * inicia (char *nombre,char *modo,unsigned *bloque,pos Pn,pos *Po,
               unsigned *ban)
{
    FILE * fp;
    int err = 0;
    fp = fopen(nombre,modo);
    if (fp != NULL){
       enceros(bloque);
       * ban = 0;
       Po->i = 0;
       Po->j = 0;
       err = trae_pag(fp,bloque,0,0,Pn,Po,ban);
    }
    if (err != 0)
       fp = NULL;

    return fp;
} /* Fin de inicia */



/* CREA -> Crea en disco una matriz de (m x n) en ceros */

FILE * crea (char *nombre, pos Pn)
{
    FILE *fp;
    int i,err = 0;

    fp = fopen(nombre,"w");
    if (fp != NULL){
       for (i=0; i < Pn.i*Pn.j; i++)
       {
        fprintf(fp,"%5d",0);
        err = (ferror(fp)!=0)?-1:err;
       }
       if (err == 0)
       {
        fclose(fp);
```

```
        fp = fopen(nombre,"r+");
       }
       else
        fp = NULL;
    }
    return fp;
} /* Fin de crea */



/* TRAE_PAG -> Extrae de la matriz virtual (la que esta en disco) una
              submatriz que contenga al elemento i,j.  */

int  trae_pag(FILE *fp, unsigned *bloque, unsigned i, unsigned j,
              pos Pn, pos *Po, unsigned *ban)
{
    unsigned h,k;
    long  LimRen,       /* Determina el numero de renglones para el bloque */
          LimCol,       /* Determina el numero de columnas  para el bloque */
          desde;        /* Para  posicionar el apuntador del archivo en el
                           elemento deseado. */
    int err,err2;       /* Indican si hubo errores */

    /* Si hubo cambios los escribe en disco */
    err = actualiza(fp,bloque,Pn,(*Po),ban);

    if (err == 0)  /* Si no hubo error al escribir la pag. ant. en disco...*/
    {
       /* Inicializa en  ceros el bloque  para  que en  caso de que  se */
       /* extraiga media pagina, no se quede con los valores anteriores */
       enceros(bloque);

       /* Actualiza la tabla de redireccionamiento */
       Po->i = i;
       Po->j = j;

       /* Determina de que tamaño se tomara la pagina de la matriz virtual */
       LimRen= (Pn.i-i>M)?M:Pn.i-i;
       LimCol= (Pn.j-j>N)?N:Pn.j-j;

       /* Posiciona el apuntador en el elemento i,j a partir del que se
          va a extraer la submatriz */
       desde = (Po->i * Pn.j + Po->j)*5;
       fseek(fp,desde,SEEK_SET);
       err = (ferror(fp)!=0)?-1:err;

       if (err == 0)
       /* Si no hubo error al colocar el apuntador en el lugar deseado */
       {
        /* Carga en memoria la submatriz */
        h = k = 0;
        while (h<LimRen){
           k = 0;
           while (k<LimCol) {
            fscanf(fp,"%5d",bloque+M*h+k);
            err = (ferror(fp)!=0)?-1:err;
            k++;
           }
           h++;
           desde = ((Po->i+h)*Pn.j+ Po->j)* 5;
           fseek(fp,desde,SEEK_SET);
```

```
            err = (ferror(fp)!=0)?-1:err;
          } /* Fin de while h<LimRen */
        } /* if err == 0  (error de posicion del cursor) */
      } /* if err == 0 (error al guardar la pagina anterior) */
      *ban = 0;
      return err;
} /*fin de trae_pag */



/* EXTRAE -> Toma el elemento que se encuentra en la posicion i,j de
            la matriz virtual */

unsigned extrae(FILE * fp, unsigned * bloque, unsigned i, unsigned j,
                pos Pn, pos * Po, unsigned * ban, int *err)
{
    if (!enmemoria(i,j,(*Po)))
       *err = trae_pag(fp,bloque,i,j,Pn,Po,ban);

    if (*err == 0)
       return *(bloque+(M*(i-(Po->i))+(j-(Po->j))));
    else
       return 0;
} /* Fin de extrae */



/* COLOCA -> Modifica el valor  del elemento  I,J  de una matriz virtual
            que ya existe. */

void coloca(FILE *fp, unsigned *bloque, unsigned valor, unsigned i,
            unsigned j, pos Pn, pos *Po, unsigned *ban, int *err)
{
    if (!enmemoria(i,j,(*Po)))
       *err = trae_pag(fp,bloque,i,j,Pn,Po,ban);

    if (*err == 0)
    {
       *(bloque+M*(i-(Po->i))+(j-(Po->j))) = valor;
       *ban = 1;
    }
} /* Fin de coloca */



/* ENMEMORIA -> Revisa que este en memoria la pagina que  contiene al ele-
               mento i,j de la matriz.    */

unsigned enmemoria(unsigned i, unsigned j, pos Po)
{
    return (Po.i<=i && i<Po.i+M) && (Po.j<=j && j<Po.j+N);
}



/* ACTUALIZA -> En caso de que se haya hecho una modificacion en uno de los
               elementos de la submatriz que esta en memoria  actualmente,
               se actualizara la matriz que esta en disco, escribiendo los
               nuevos valores.       */
```

```
int actualiza(FILE *fp, unsigned * bloque, pos Pn,
              pos Po, unsigned *ban)
{
    long desde, LimRen,LimCol;
    int h,k,
      err = 0;   /* Para saber si hubo errores al escribir */

    LimRen= (Pn.i-Po.i>M)?M:Pn.i-Po.i;
    LimCol= (Pn.j-Po.j>N)?N:Pn.j-Po.j;

    if (*ban) {
      desde = (Po.i * Pn.j+ Po.j)*5;
      fseek(fp,desde,SEEK_SET);
      h = k = 0;
      while (h<LimRen){
          k = 0;
          while (k<LimCol) {
            fprintf(fp,"%5d",*(bloque+M*h+k));
            err = (ferror(fp)!=0)?-1:err;
            k++;
          }
          h++;
          desde = ((Po.i+h)* Pn.j + Po.j)* 5;
          fseek(fp,desde,SEEK_SET);
          err = (ferror(fp)!=0)?-1:err;
      } /* Fin del while */
      *ban = 0;
    } /* Fin del if */
    return err;
} /* Fin de actualiza */




/* ENCEROS -> Inicializa con ceros la matriz que se le pase como
              parametro */

void enceros (unsigned *Matriz)
{
    int i,j;

    for (i=0; i<M; i++) {
       for (j=0; j<N; j++){
        *(Matriz+(M*i+j)) = 0;
       }
    }
} /* enceros */




/* COPIA -> Copia el contenido de la matriz fuente en la matriz
           destino */

int copia(char *fuente, char *destino)
{
    unsigned c;
    FILE *fpf;
    FILE *fpd;
    int  err = 0,   /* Indica si hubo error al abrir los archivos */
```

```
    caract=0;  /* Numero de caracteres leidos */

  fpf = fopen(fuente,"r");
  err = (ferror(fpf)!=0)?-1:err;
  fpd = fopen(destino,"w");
  err = (ferror(fpd)!=0)?-1:err;

  if (fpf!=NULL&&fpd!=NULL)
  {
     caract = fscanf(fpf,"%5d",&c);
     while (caract!=EOF)
     {
      fprintf(fpd,"%5d",c);
      err = (ferror(fpd)!=0)?-1:err;
      caract = fscanf(fpf,"%5d",&c);
      err = (ferror(fpf)!=0)?-1:err;
     }
  }
  else
     err = -1;
  fclose(fpf);
  fclose(fpd);
  return err;
} /*fin de copia */
```

# APENDICE B: PROGRAMAS DE APLICACION

```
/* Definicion de parametros */

# define  ren_A    450             /* Numero de renglones de la matriz A  */
# define  col_A    100             /* Numero de columnas  de la matriz A  */

# define  ren_B    100             /* Numero de renglones de la matriz B  */
# define  col_B    750             /* Numero de columnas de la matriz B   */

# define  matrizA "matrizA.dat"    /* Nombre del archivo para la matriz A */
# define  matrizB "matrizB.dat"    /* Nombre del archivo para la matriz B */
# define  matrizR "R.dat"          /* Nombre del archivo resultado        */
```

```
# include <stdio.h>          /* Modulo con funciones de E/S  */
# include <conio.h>          /* Modulo de funciones para pantalla */
# include "matvirt.h"        /* Modulo de matrices virtuales */
# include "parametr.h"       /* Modulo de definicion de parametros */


/* DECLARACION DE FUNCIONES: */

void multiplica(unsigned *,   /* Apuntador a la matriz A          */
                pos,          /* Dimensiones (m,n) de A           */
                pos,          /* Tabla de redireccionamiento de A */
                unsigned *,   /* Apuntador a la matriz B          */
                pos,          /* Dimensiones (m,n) de B           */
                pos,          /* Tabla de redireccionamiento de B */
                unsigned *);  /* Apuntador a Matriz Resultado (R) */

void suma ( unsigned *,       /* Apuntador a la matriz 1          */
            pos,              /* Dimensiones (m,n) de 1           */
            pos,              /* Tabla de redireccionamiento de 1 */
            unsigned *);      /* Apuntador a la matriz 2          */


main()
{
    FILE * matA;      /* Apuntador al archivo de la matriz A       */
    FILE * matB;      /* Apuntador al archivo de la matriz B       */
    FILE * matR;      /* Apuntador al arch. de la matriz resultado (R) */

    unsigned A[M*N],    /* A = Submatriz de Matriz A */
             B[M*N],    /* B = Submatriz de Matriz B */
             R[M*N],    /* R = Submatriz de Matriz R */
             P[M*N];    /* P = Matriz Provisional    */

    pos Ao,Bo,Ro,     /* Posicion I,J a partir de la cual se
                         toma la pagina para las matrices  A,
                         B y R. (Tablas de redireccionamiento */

        An,Bn,Rn;     /* Dimensiones (m,n) de las matrices
                         virtuales: A,B y R */

    unsigned bana,            /* Bandera de matriz A  */
             banb,            /* Bandera de matriz B  */
             banr,            /* Bandera de matriz R  */
             i,j;             /* Variables auxiliares */


    /* Numero de bloques extraibles de las matrices por: */
    int Ranum,     /* Renglones de A                    */
        Canum,     /* Columnas  de A = renglones de B */
        Cbnum,     /* Columnas  de B                    */
        x,y,z,     /* Variables auxiliares */
        err,err1,err2,err3;

    err = err1 = err2 = err3 = 0;


    /* Apertura de matrices virtuales: */
    An.i = ren_A;
    An.j = col_A;
    matA = inicia(matrizA,"r",A,An,&Ao,&bana);
    Bn.i = ren_B;
```

```
            j = 0;
            while ((j < col_A) && (err1==0 && err2==0 && err3==0))
            {
c=c||(extrae(matA,A,i,j,An,&Ao,&bana,&err1)&&extrae(matB,B,j,k,Bn,&Bo,&banb,&e
rr2));
              j++;
            } /* while j */
            banr = 1;
            coloca(matR,R,c,i,k,Rn,&Ro,&banr,&err3);
            i++;
          } /* while i */
          k++;
        } /* for k */

        if (err1==0 && err2==0 && err3==0)
        {
          err = actualiza(matR,R,Rn,Ro,&banr);  /*guarda la ultima pagina*/
          if (err != 0)
          {
            clrscr();
            printf("\n\nERROR de escritura.");
            getchar();
          }
        }
        else
        {
          clrscr();
          printf("\n\nERROR de Lectura/Escritura en disco.\n");
          printf("      Ejecucion cancelada.");
          getchar();
        }

        fclose(matA);
        fclose(matB);
        fclose(matR);
      }
      else
      {
        if (An.j != Bn.i)
        {
          clrscr();
          printf("\n\n\n\nERROR. El No. de columnas  de la matriz A");
          printf("\n       debe ser igual al No. de renglones");
          printf("\n       de la matriz B.");
          getchar();
        }
        else
        {
          clrscr();
          printf("\n\n\n\nERROR. No se puede accesar o crear la");
          printf("\n       matriz virtual");
          getchar();
        }
      }
}
```

```
# include <stdio.h>          /* Modulo con funciones de E/S  */
# include <conio.h>          /* Modulo de funciones de pantalla */
# include "matvirt.h"        /* Modulo de matrices virtuales */
# include "parametr.h"       /* Modulo de definicion de parametros */


main()
{
    FILE * matA;          /* Apuntador al archivo de la matriz A */
    FILE * matB;          /* Apuntador al archivo de la matriz B */
    FILE * matR;          /* Apuntador al arch. de la matriz resultado (R) */

    unsigned A[M*N],      /* A = Submatriz de Matriz A */
             B[M*N],      /* B = Submatriz de Matriz B */
             R[M*N];      /* R = Submatriz de Matriz Resultado */

    struct pos Ao,Bo,Ro,       /* Posicion I,J a partir de la cual se
                                  toma la pagina para las matrices  A,
                                  B y R . */

           An,Bn,Rn;           /* Dimensiones (m,n) de las matrices
                                  virtuales */

    unsigned bana,             /* Bandera de matriz A  */
             banb,             /* Bandera de matriz B  */
             banr,             /* Bandera de matriz R  */
             i,j,k,            /* Variables auxiliares */
             c;                /* c = aij * bjk        */

    int err=0, err1=0,    /* En caso de error estas variables   */
        err2=0, err3=0;   /* tomaran un valor diferente de cero */


    /* Se abren los archivos de las matrices virtuales: */
    /* (Los valores de matrizA, matrizB y matrizR estan */
    /* definidos en "parametr.h").                      */
    An.i = ren_A;
    An.j = col_A;
    matA = inicia(matrizA,"r",A,An,&Ao,&bana);
    Bn.i = ren_B;
    Bn.j = col_B;
    matB = inicia(matrizB,"r",B,Bn,&Bo,&banb);
    Rn.i = An.i;
    Rn.j = Bn.j;
    matR = crea(matrizR,Rn);


    if ((An.j == Bn.i)&&(matA!=NULL&&matB!=NULL&&matR!=NULL))
    {
      Ro.i = Ro.j = 0;
      banr = 0;
      enceros(R);      /* inicializa la matriz resultado */


      /* Inicia proceso de multiplicacion: */
      k = 0;
      while ((k < col_B) && (err1==0 && err2==0 && err3==0))
      {
          i = 0;
          while ((i < ren_A) && (err1==0 && err2==0 && err3==0))
          {
              c = 0;
```

```
Bn.j = col_B;
matB = inicia(matrizB,"r",B,Bn,&Bo,&banb);
Rn.i = An.i;
Rn.j = Bn.j;
matR = crea(matrizR,Rn);


if ((An.j == Bn.i)&&(matA!=NULL&&matB!=NULL&&matR!=NULL))
{
  Ro.i = 0;
  Ro.j = 0;
  banr = 0;
  enceros(R);      /* inicializa la matriz resultado */
  enceros(P);      /* inicializa la matriz provisional */

   /*----------------Inicia el proceso de multiplicacion----------------*/

   /* Calcula cuantos bloques puede extraer de A por  renglon  (Ranum) y
    cuantos por columna (Canum); tambien cuantos bloques puede extraer
    de B por columna (Cbnum):
    Ranum = (# Ren de A) / # Ren del bloque (M)
    Canum = (# Col de A) / # Col del bloque (N)
    Cbnum = (#Col  de B) / # Col del bloque (N)    */ ,

  Ranum = (An.i%M==0)?ceil(An.i/M):ceil(An.i/M)+1;
  Canum = (An.j%N==0)?ceil(An.j/N):ceil(An.j/N)+1;
  Cbnum = (Bn.j%N==0)?ceil(Bn.j/N):ceil(Bn.j/N)+1;

  x = 0;
  while ((x < Cbnum) && (err==0 && err1==0&& err2==0 && err3==0))
  {
     y = 0;
     while ((y < Ranum) && (err==0 && err1==0&& err2==0 && err3==0))
     {
        z = 0;
        while ((z < Canum) && (err==0 && err1==0&& err2==0 && err3==0))
        {
         enceros(P);
         Ao.i = y*M;
         Ao.j = z*N;
         err1 = trae_pag(matA,A,Ao.i,Ao.j,An,&Ao,&bana);
         Bo.i = z*N;
         Bo.j = x*N;
         err2 = trae_pag(matB,B,Bo.i,Bo.j,Bn,&Bo,&banb);
         Ro.i = Ao.i;
         Ro.j = Bo.j;
         multiplica(A,An,Ao,B,Bn,Bo,P);
         suma(R,Rn,Ro,P);
         z++;
        } /* while z */
        banr = 1;     /* Indica que R fue modificada */
        err = actualiza(matR,R,Rn,Ro,&banr);
        enceros(R);
        y++;
     } /* while y */
     x++;
  } /* while x */
  if (err!=0 || err1!=0 || err2!=0 || err3!=0)
  {
     clrscr();
     printf("\n\nERROR de Lectura/Escritura");
     printf("\n      Ejecucion cancelada.");
```

```
      }
      fclose(matA);
      fclose(matB);
      fclose(matR);
       /*-----------------Fin del proceso de multiplicacion----------------*/
    }
    else
    {
      if (An.j != Bn.i)
      {
       clrscr();
       printf("\n\n\n\nERROR. El No. de columnas de la matriz A");
       printf("\n       debe ser igual al No. de renglones");
       printf("\n       de la matriz B.");
      }
      else
      {
       clrscr();
       printf("\n\n\n\nERROR. No se puede accesar o crear la");
       printf("\n       matriz virtual.");
      }
    }
}


/*-------------------------------------------------------------------*/
/*                        F U N C I O N E S                          */
/*-------------------------------------------------------------------*/

/* MULTIPLICA -> Multiplica en forma logica dos matrices con ceros y
                 unos. haciendo la multiplicacion como un AND y la
                 suma como un OR.                                    */

void multiplica (unsigned *MatrizA,pos An,pos Ao,
                 unsigned *MatrizB,pos Bn,pos Bo,
                 unsigned *MatrizR)
{
    int i,j,k,    /* Variables auxiliares */
        lcb,      /* Limite de numero de columnas de la matriz B */
        lca,      /* Limite de numero de columnas de la matriz A */
        lra;      /* Limite de numero de renglones de la matriz A*/

    lcb = (Bn.j-Bo.j>N) ? N : Bn.j-Bo.j;
    lca = (An.j-Ao.j>N) ? N : An.j-Ao.j;
    lra = (An.i-Ao.i>M) ? M : An.i-Ao.i;

    for(k = 0; k < lcb; k++){
       for(i = 0; i < lra; i++){
         for (j = 0; j < lca; j++){

*(MatrizR+M*i+k)=*(MatrizR+M*i+k)||(*(MatrizA+M*i+j)&&*(MatrizB+M*j+k));
        }
       }
    }
} /* multiplica */


/* SUMA -> Suma en forma logica dos matrices, es decir, la matriz
           resultado esta compuesta por elementos del tipo Ri, donde
           Rij = Aij OR Bij.  */
```

```
void suma (unsigned *Matriz1,pos Rn,pos Ro,
           unsigned *Matriz2)
{
    int i,j,        /* Variables auxiliares */
        lc,         /* Limite de numero de columnas  de la matriz R */
        lr;         /* Limite de numero de renglones de la matriz R */

    lc = (Rn.j-Ro.j>N) ? N : Rn.j-Ro.j;
    lr = (Rn.i-Ro.i>M) ? M : Rn.i-Ro.i;


    for( i = 0; i < lr; i++){
       for(j = 0; j < lc; j++){
         *(Matriz1+M*i+j) = *(Matriz1+M*i+j) || *(Matriz2+M*i+j);
       }
    }
} /* suma */
```

```
            getchar();
          }
        }
        else
        {
          clrscr();
          printf("\n\nERROR de Lectura/Escritura en disco.\n");
          printf("      Ejecucion cancelada.");
          getchar();
        }
       }
       else
       {
        clrscr();
        printf("\n\nERROR. Error de Lectura/Escritura.");
        printf("\n       Ejecucion cancelada");
        getchar();
       }
       fclose(matR);
    } /* Fin de if err == 0 */
    else
    {
       clrscr();
       printf("\n\nERROR. Al copiar el archivo");
       printf("\n       Ejecucion cancelada.");
       getchar();
    }
}
```

# B I B L I O G R A F I A

# B I B L I O G R A F I A


- CAPELLA Kort Elke, **Herramientas para la construcción de compiladores** (México D.F. Tesis de Licenciatura UNAM). 1991

- DENNING Peter J., DENNIS Jack B., QUALITZ Joseph E. Machines, Languages and Computation. Prentice Hall, 1978

- HSIAO David K. Systems programming concepts of operating and data base systems. Addison Wesley Publishin Company, 1975.

- KERNIGHAN Brian W., RITCHIE Dennis M. El lenguaje de Programación C. Prentice Hall Hispanoamericana, S. A., 1991.

- LEWIS Philip M., ROSENKRANTZ Daniel J., STERRNS Richard E. Compiler Design Theory. Addison Wesley Publishing Company, 1976.

- TERRY Patrick D. Programming language translation a practial approach. Addison Wesley Publishing Company, 1986.