DESARROLLO DE UN TRADUCTOR DE MANTIS A COBOL

UNIVERSIDAD DEL VALLE DE GUATEMALA

Facultad de Ciencias y Humanidades

DESARROLLO DE UN TRADUCTOR DE MANTIS A COBOL

Mynor Iván Muralles Ortiz

Trabajo de investigación presentado para
optar al grado académico de

Ingeniero en Ciencias de Computación

Guatemala

1990

Vo. Bo. :

(f) _____
Licenciado Rodrigo Arias Palomo
Asesor

Tribunal:

(f) _____
Licenciado Rodrigo Arias Palomo

(f) _____
Licenciado Gabriel Basterrechea

(f) _____
Doctor Ricardo Cordón Engel

Fecha de aprobación: 8 de Mayo de 1990.

A mis Padres

CONTENIDO

I. INTRODUCCION

En los últimos años, el desarrollo de herramientas de Cuarta Generación (4GL) para los *mainframes* y minicomputadores IBM ha aumentado. Por ejemplo, se han desarrollado lenguajes como Mantis, el cual facilita, grandemente, la implementación de aplicaciones. Este lenguaje cuenta con un amplio conjunto de instrucciones que lo pone en ventaja ante otros lenguajes tradicionales que operan en dichos computadores, como por ejemplo Cobol, RPG, etc... Sin embargo, Mantis presenta dificultades para ser utilizado en aplicaciones *batch*. Las aplicaciones *batch* son frecuentes en los sistemas de información, ya que, generalmente, es necesario contar con programas que efectúen grandes cantidades de transacciones en una base de datos o en un archivo. La duración de estos procesos puede ser de minutos, horas, o hasta de días en algunos casos. Por eso, es necesario que dichos programas sean eficientes en tiempo. En la mayoría de los casos se utiliza un lenguaje como Cobol para los procesos *batch*, el cual puede ser compilado a lenguaje objeto y ser eficiente.

Programar en Cobol requiere un mayor esfuerzo para el programador. La codificación es bastante larga y complicada, ya que el programador tiene a su cargo

implementar todas las funciones que necesita, definir todos los datos, manejar los *status* de los archivos etc... Esto redunda en un tiempo de implementación largo, con el riesgo de producir un programa complejo (ya que Cobol no tiene muchas facilidades de estructuración) y, posiblemente, con errores. Además, se debe tomar en cuenta que los programas en Cobol requieren largos tiempos de espera (en sistemas multiusuarios) para compilación y pruebas. Cuando se detecta un error, el programador debe modificar el programa fuente y compilarlo de nuevo, ciclo que puede requerir demasiado tiempo.

Debido a esto, se consideró que sería deseable contar con una herramienta que tuviera las facilidades de programación de Mantis y la rapidez de ejecución de Cobol. Entonces se pensó en la posibilidad de desarrollar un traductor que reciba como entrada un programa escrito en Mantis y que produzca como salida un programa equivalente escrito en Cobol.

Lo primero que se hizo fue analizar la factibilidad de la traducción automática. Ya que el intéres principal es su aplicación en los procesos *batch*, se analizó un subconjunto de las instrucciones de Mantis que fuera lo suficientemente completo para dichas aplicaciones. Luego se pensó en la posible traducción de cada instrucción en

Mantis a Cobol, determinando las equivalencias. Se trató de que los esquemas de traducción utilizados fueran lo más eficientes posible para que el programa generado en Cobol sea casi tan rápido como un programa escrito directamente por un programador.

Se optó por traducir a lenguaje Cobol porque éste es un lenguaje más portátil. Lo ideal hubiera sido generar código en *Assembly*, pero no se hizo por lo siguiente:

i) El lenguaje *Assembly* es más difícil de entender y la generación de código hubiera sido demasiado complicada.

ii) El código generado en *Assembly* hubiera sido particular de un sólo modelo de computadores, ya que éste cambia para cada procesador. Por lo tanto, se hubiera perdido la portabilidad.

iii) El lenguaje Cobol es ampliamente utilizado desde hace mucho tiempo y es de esperarse que los compiladores actuales de Cobol hagan uso óptimo de las instrucciones en *Assembly* y generen código eficiente, por lo cual es conveniente generar únicamente instrucciones de Cobol.

El presente trabajo explica los aspectos considerados para la elaboración de un traductor de Mantis a Cobol, al cual se le ha dado el nombre de **FastMan**. Inicialmente se

analizan dos herramientas que se utilizaron para la construcción del traductor: el generador de tablas del *scanner* y el generador de tablas del *parser*. Luego se explica, en términos generales, como se efectúa la traducción de un programa escrito en Mantis a su equivalente en Cobol. Después se muestran algunas comparaciones de la rapidez de programas escritos en Mantis y sus equivalentes generados por el traductor. Debido a que la generación de código en Cobol no es óptima, también se hacen comparaciones con programas escritos en Cobol directamente por un programador, para obtener una idea de la eficiencia de la traducción. Por último, se especifica la gramática de Mantis utilizada para las aplicaciones *batch* y se presentan algunos ejemplos de traducción.

II. VENTAJAS DE PROGRAMAR EN LENGUAJE MANTIS 4.2

Las principales ventajas de desarrollar programas en el lenguaje Mantis 4.2 son:

1.- Mantis es un Lenguaje de Cuarta Generación (4GL) y como tal, facilita el desarrollo de programas. Los programas escritos en Mantis ocultan al usuario la definición de los archivos, y hacen que éstos se manejen en una forma más abstracta. Por ejemplo, un programa en Mantis que hace una operación *INSERT* a un archivo, no considera los códigos sin significado que el manejador de archivos regresa como resultado de hacer la operación; por el contrario, el programa considera los *status* que Mantis maneja, como *GOOD* (si la operación fue exitosa) o *DUPLICATE* (si la llave está repetida). Esto hace que los programas sean más claros en cuanto a su lógica.

2.- Los programas en Mantis pueden ser fácilmente estructurados, ya que este lenguaje permite:

i) Uso de estructuras de control como *WHILE*, *UNTIL*.

ii) Uso de estructuras de decisión como *IF ELSE*, *WHEN*.

iii) Uso de procedimientos con parámetros, lo cual facilita la abstracción de los programas. Los módulos en Mantis

reciben un conjunto de datos de entrada, realizan ciertas operaciones con dichos datos y producen datos de salida.

3.- Mantis facilita el manejo de *strings*, permitiendo efectuar operaciones de suma y resta. Además cuenta con una variedad de funciones para manejo de los mismos (*PAD*, *UNPAD*, *POINT*, *TXT*, *VALUE*). Mantis también permite accesar *substrings*. El manejo de *strings* en Cobol es difícil porque no existen funciones primitivas para hacer dichas operaciones. Cuando se necesitan estas funciones, el programador en Cobol debe implementarlas primero y luego hacer referencia a ellas, lo cual redunda en la complejidad de los programas.

4.- En Mantis existen funciones aritméticas (*ABS*, *EXP*, *INT*, *LOG*, *SGN*, *SQR*, *VAL*) y, además, trigonométricas (*SIN*, *COS*, *TAN*, *ATN*), las cuales pueden ser de utilidad en aplicaciones numéricas. También existen algunas constantes predefinidas (*PI*, *E*, *TRUE*, *FALSE*, *ZERO*).

5.- Mantis es un lenguaje interpretado y esto facilita el desarrollo de los programas. Al tener la posibilidad de probar y corregir errores en los programas "en línea", se evita que los programadores esperen largos tiempos de compilación y prueba.

III. VENTAJAS DE UTILIZAR EL TRADUCTOR DE MANTIS A COBOL

Entre los principales factores que hacen conveniente el uso del traductor como una herramienta de desarrollo están:

1.- El lenguaje Mantis es, principalmente, usado para aplicaciones "en línea", ya que presenta facilidades para el manejo de pantallas para captar y presentar información. Su utilización en aplicaciones *batch* también debería ser factible. Sin embargo, Mantis tiene un problema: detecta posibles *loops* infinitos. Mantis asume que cualquier programa que esté funcionando correctamente debe presentar en pantalla alguna información cada cierto tiempo. Si un programa no hace ésto, Mantis asume que el proceso está en un *loop* infinito y lo cancela, mostrando un mensaje de error *POT* (*POTential program loop condition*). Esto significa que el programa excedió el máximo número de instrucciones sin haber efectuado un *I/O* con la terminal. El problema puede solucionarse parcialmente al modificar desde el programa que se ejecuta, dos parámetros: *SLICE* y *SLOT*. Con el parámetro *SLOT* se puede modificar el número de veces que el programa puede ejecutar el número de instrucciones indicadas por el parámetro *SLICE* antes de que Mantis encuentre un error *POT*. Sin embargo, en la práctica se ha

determinado que aunque estos parámetros se pongan al máximo posible, Mantis es bastante limitado en cuanto al tiempo que puede ejecutar un proceso sin hacer *I/O*, ya que no sobrepasa los 5 minutos. Además, el valor de estos parámetros no debe ponerse muy grande porque se produce una degradación en el tiempo de respuesta del computador.

2.- Se supone que es posible ejecutar un programa en Mantis en una partición *batch* (Mantis *Batch*). Esto no requiere de una terminal para ejecutar el proceso, pero existe la misma limitación mencionada en el punto anterior. El programa en Mantis debe efectuar cada cierto número de instrucciones un *SHOW* seguido de un *WAIT* para simular que se está efectuando *I/O* con la terminal. Ya que el comando *WAIT* ocasiona que el proceso espere intervención del usuario del programa, dentro del *JCL* que ejecuta el proceso hay que incluir tarjetas con la instruccion <*ENTER*> para que el programa siga funcionando. Esto hace que el *JCL* tenga tantos <*ENTER*> como veces se suspenda el programa, lo cual es impredecible en un proceso *batch*. Por esta razón, Mantis *Batch* no ha sido utilizado. Además, no se puede esperar un gran incremento en la velocidad de los programas, ya que aunque sean *batch*, éstos siguen siendo interpretados.

3.- Como ya es sabido, un programa compilado se ejecuta más rápido que un programa interpretado. La diferencia radica en que el programa interpretado tiene que pasar antes de ejecutar en sí cada instrucción, por una fase de chequeo y decodificación de la misma. Cuando ya se sabe que la instrucción está correcta y de qué operación se trata, el intérprete debe investigar los datos a los cuales hace referencia la instrucción, chequear que existan y hacer un acceso a ellos. Por último se ejecuta la instrucción propiamente. Esto es porque el intérprete trabaja por Asociación Dinámica, o sea que busca los datos de las instrucciones en el momento en que se utilizan (tiempo de ejecución). Por el contrario, la compilación de un programa genera código objeto. Dicho código es sumamente rápido y en él están especificados de antemano los datos que se van a utilizar en cada operación, o sea que cuenta con una Asociación Estática. Debido a ésto, se espera que el resultado de traducir un programa en Mantis a Cobol y luego compilar este programa en Cobol, se produzca un programa que sea más rápido. La desventaja del programa compilado es que se necesita una recompilación cada vez que se cambia la definición de un archivo o una vista de datos.

4.- Cobol es un lenguaje ampliamente conocido y mucho más difundido que Mantis. Además, todos los computadores IBM en los que se utiliza Mantis tienen un compilador de Cobol. Esto permite que un programa pueda ser trasladado de un computador a otro, incluso a un ambiente en el que Mantis no está instalado.

5.- El traductor genera como resultado un programa escrito en Cobol, según el *standard* aprobado por ANSI para dicho lenguaje. Esto facilita la portabilidad de los programas en Cobol generados.

IV. DESCRIPCION DEL PROYECTO

El desarrollo del traductor de Mantis a Cobol se dividió, principalmente, en dos partes. La primera parte consistió en implementar dos herramientas de *software* que sirven para la construcción de un compilador, las cuales son:

1.- Generador de tablas del *scanner*.

2.- Generador de tablas del *parser*.

Por medio de estos programas se construyeron las tablas necesarias para reconocer la gramática de FastMan. Estos generadores están escritos en lenguaje Prolog, lo cual resultó una interesante adaptación de los algoritmos. Los programas tienen un tamaño relativamente pequeño y reflejan en forma clara la lógica de los mismos. El hecho de utilizar estas herramientas agrega una gran versatilidad al traductor, ya que por ejemplo, se pueden agregar facilmente las nuevas instrucciones que se incluyan en las siguientes versiones de Mantis.

La segunda parte consistió en el desarrollo del traductor propiamente. En los siguientes capítulos se analiza la construcción y el funcionamiento de cada una de estas partes, mostrando los detalles más relevantes de las mismas.

V. GENERADOR DE TABLAS DEL *SCANNER*

Un *scanner* es la parte de un compilador encargada de reconocer los *tokens* del programa fuente. Cada *token* generalmente es identificado por un número, el cual es utilizado por el compilador para saber qué acción se debe tomar al encontrar dicho *token*. Los *tokens* de un lenguaje pueden ser fácilmente definidos por medio de Expresiones Regulares. A través de esta notación se definen las palabras reservadas del lenguaje, la forma que tendrán los identificadores o variables, la forma que tendrán las constantes (numéricas o *strings* de caracteres) etc... A partir de las expresiones regulares se construye un diagrama de transiciones el cual es no-determinístico (NFA). Por último, se aplica un algoritmo para transformar el NFA en un diagrama de transiciones determinístico (DFA), el cual sirve para identificar los *tokens* del lenguaje.

Cuando los *tokens* de un lenguaje son muchos, la construcción de los diagramas de transición para reconocer dichos *tokens* resulta difícil, ya que pueden resultar muy grandes y se pueden cometer errores en el proceso. Por eso es mejor optar por un proceso automático. La primera parte de este proyecto consistió en desarrollar una herramienta llamada **GenDfa** por medio de la cual, se pueden

reconocer los *tokens* de un lenguaje a partir de la definición de las expresiones regulares de los mismos. Con esta herramienta se obtiene generalidad, ya que funciona para reconocer los *tokens* de cualquier lenguaje. GenDfa se utilizó para generar las tablas del *scanner* de FastMan, brindando gran versatilidad. Al principio, FastMan tenía un conjunto pequeño de instrucciones. Conforme se fue desarrollando el proyecto se fueron agregando otras, para lo cual había necesidad de reconocer un mayor número de *tokens*. El único cambio necesario para ésto fue agregar las definiciones de los *tokens* a la tabla de *tokens* anterior, y generar de nuevo el DFA. Esto se hizo en un tiempo relativamente corto si se toma en cuenta el tiempo que hubiera tomado reconstruir el DFA manualmente.

A. Funcionamiento de GenDfa.

GenDfa es un programa que genera la información necesaria para reconocer los *tokens* de un lenguaje a partir de la definición de los mismos. El programa está escrito en lenguaje Prolog. Se usó Prolog porque se pensó que los datos y el algoritmo del programa se podían acoplar a una representación de grafos, la cual se puede manejar fácilmente en dicho lenguaje. Además, el programa resultaría más fácil y entendible, debido a que el lenguaje Prolog es bastante conciso y oculta los detalles de implementación de las estructuras que maneja.

Para entender fácilmente la operación de GenDfa se analiza un ejemplo sencillo. Suponiendo que se tiene un lenguaje en el cual los *tokens* son los siguientes:

Palabras reservadas:

BEGIN
END
IF
THEN
ELSE

La forma general de un Identificador (Id) es: empieza con una letra y luego siguen letras o dígitos cualquier número de veces. La notación de expresiones regulares para representar esto, es :

Id = letra (letra ¦ dígito) ~

en donde letra y dígito son los conjuntos:

letra = { A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,
U,V,W,X,Y,Z }

dígito = { 0,1,2,3,4,5,6,7,8,9 }

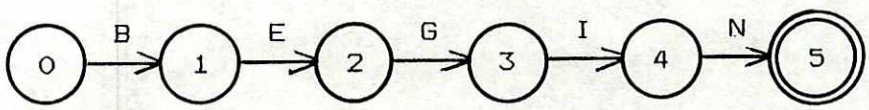
Aquí se ha especificado que el *token* llamado Id está formado por un elemento del conjunto letra seguido por elementos de los conjuntos letra o dígito cero o más veces (el signo ¦ significa ó y el signo ~ significa cero o más veces). Como se puede observar, es necesario poder especificar nombres de conjuntos dentro de las expresiones

regulares, ya que de lo contrario habría que hacer una especificación detallada para cada letra, lo que sería muy difícil.

Dada esta definición de los *tokens* del lenguaje, para usar GenDfa se debe crear un archivo texto (con cualquier editor que no introduzca caracteres extraños en él) que contenga dicha definición. La definición resultaría así:

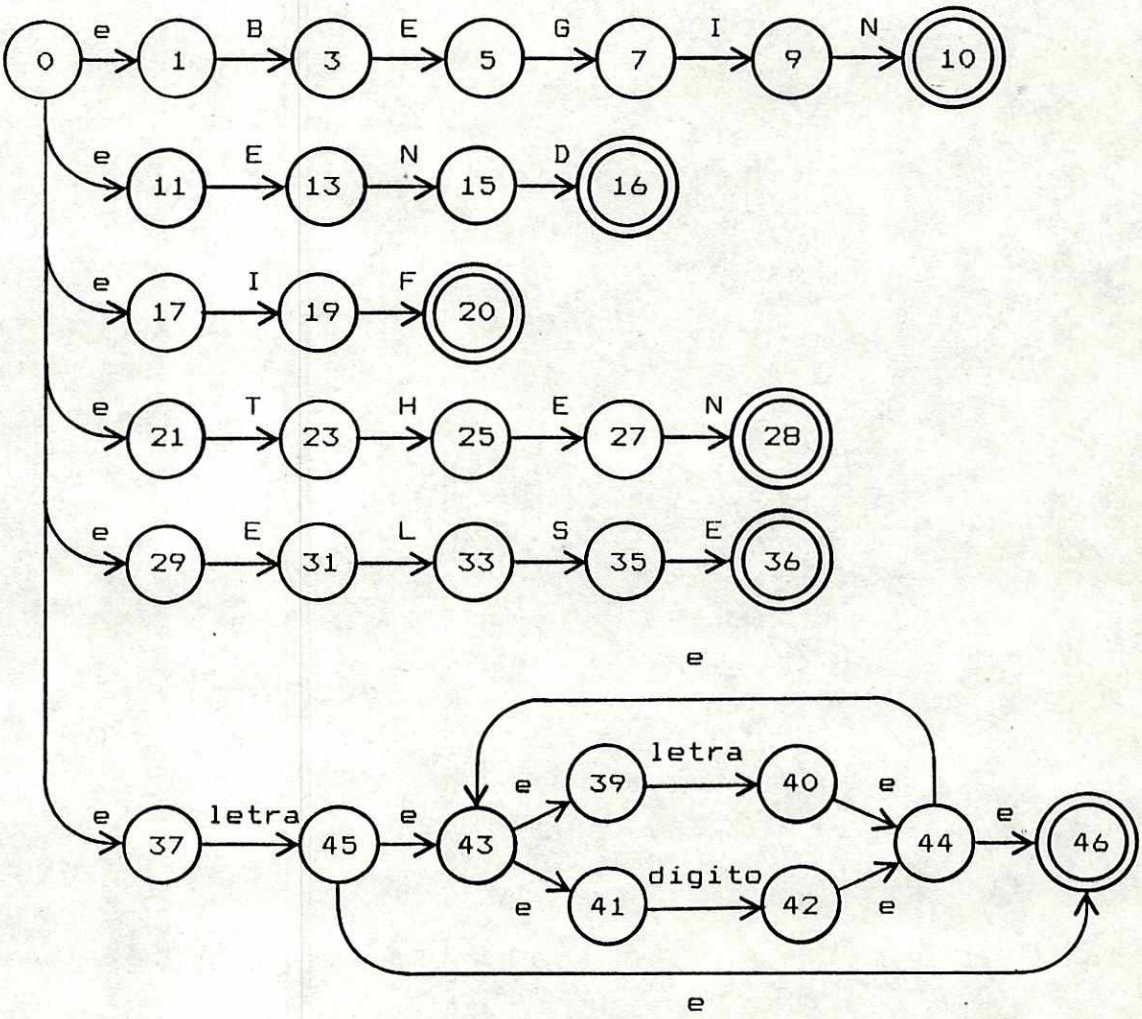
```
Definicion letra=ABCDEFGHIJKLMNOPQRSTUVWXYZ
Definicion digito=0123456789
Token 01 = B E G I N
Token 02 = E N D
Token 03 = I F
Token 04 = T H E N
Token 05 = E L S E
Token 06 = letra [ letra ¦ digito ] ~
```

Luego se ejecuta el programa GenDfa, el cual requiere que se le ingrese el nombre del archivo en donde está contenida la definición de los *tokens*. El primer paso que GenDfa realiza es almacenar las definiciones de los conjuntos especificados. Luego, para cada *token* crea un diagrama de transiciones. Por ejemplo, el diagrama de transiciones del *token* 1 sería:



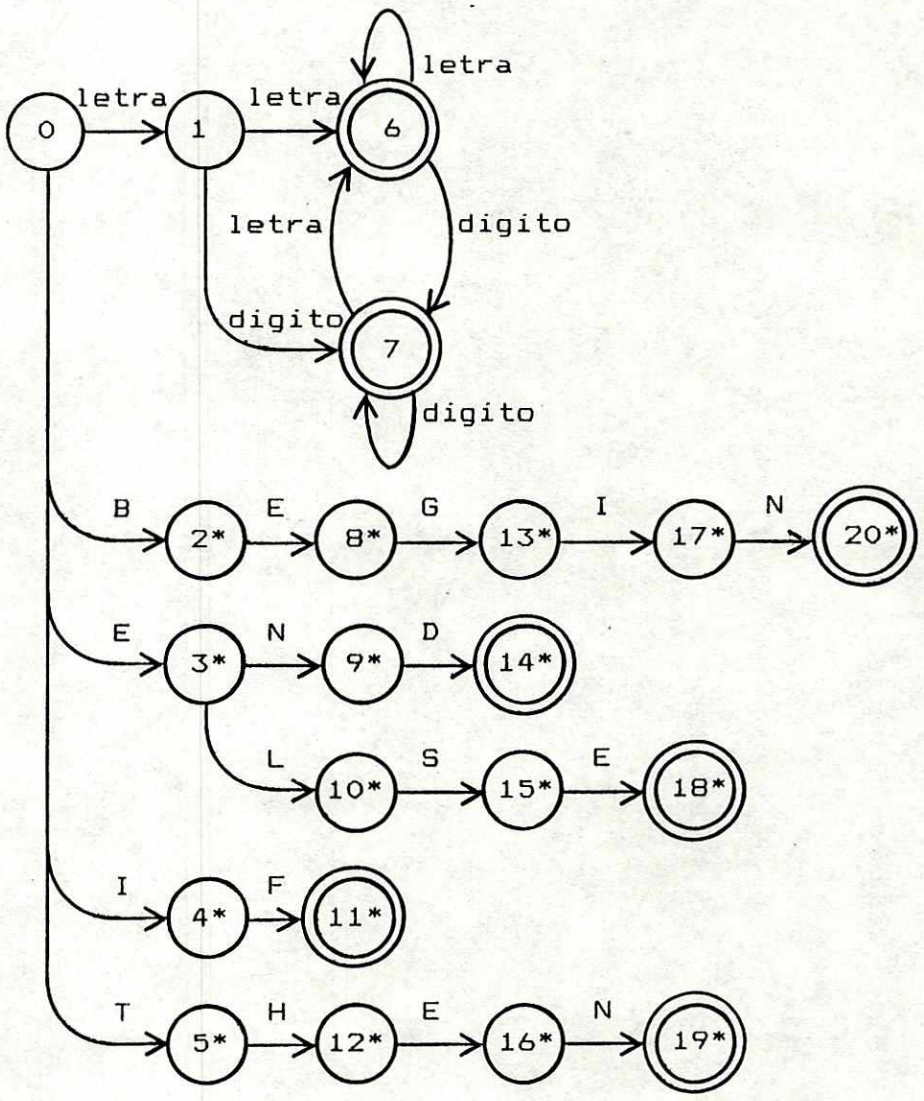
En este caso, el estado 5 es un estado final (lo cual se representa por un circulo doble) en el cual se reconoce

el *token* número 1. El diagrama de transiciones total construido por GenDfa es:



Como se puede apreciar, las transiciones de los estados son ambiguas ya que existen transiciones en vacío (e) y no son mutuamente excluyentes (o sea, se pueden tomar varios caminos sobre el mismo símbolo). Es por esto que dicho diagrama es llamado NFA (*Nondeterministic Finite Automaton*).

El siguiente paso de GenDfa es construir un DFA (*Deterministic Finite Automaton*) a partir del NFA generado. El DFA no tiene transiciones en vacío y es determinístico (o sea, cada estado no tiene varias transiciones a diferentes estados sobre el mismo símbolo). El DFA resultante es:



Todos los estados del DFA marcados con * tienen transiciones a los estados 6 y 7 en los conjuntos letra y

dígito, respectivamente. Además, todos los estados (menos el estado 0 y los indicados explícitamente) son estados finales que reconocen el *token* número 6 (Id). Las transiciones deben ser evaluadas tomando en cuenta, en primer lugar, los símbolos escalares y después los conjuntos, con lo cual se eliminan las ambigüedades.

Utilizando este diagrama de transiciones, es posible reconocer los diferentes *tokens* de un *string* de entrada, o detectar error si se encuentra un *token* que no ha sido definido. El resultado de GenDfa es un DFA representado como un conjunto de tablas, las cuales pueden ser utilizadas por un programa que las interprete y sirva para reconocer los *tokens* de un lenguaje.

A continuación se explica la forma cómo deben especificarse los *tokens* de un lenguaje, cuál es el procedimiento necesario para producir el DFA y el formato de salida usado por GenDfa.

B. Definición de *tokens* del lenguaje.

El archivo de entrada para GenDfa puede contener información de dos tipos:

1.- Especificación de conjuntos.

2.- Especificación de *tokens*.

1. **Especificación de conjuntos.** GenDfa permite definir conjuntos para simplificar la definición de los *tokens*. Para definir un conjunto se utiliza la siguiente notación:

Definicion Nombre_Conjunto = Descripción_Elementos

La palabra **Definicion** sirve para que GenDfa reconozca que se trata de la definición de un conjunto, y puede ser escrita en mayúsculas o minúsculas.

Nombre_Conjunto se refiere al nombre que se da al conjunto que se está definiendo. Este nombre sirve para hacer referencia al conjunto en las especificaciones de *tokens* posteriores.

Descripción_Elementos es la lista de los valores del conjunto. Deben ponerse consecutivamente, ya que GenDfa trata cada carácter como un posible valor.

2. **Especificación de *tokens*.** Los *tokens* se especifican de la siguiente forma:

Token Número_Token = Definición_Token

La palabra **Token** sirve para que GenDfa reconozca que se va a definir un *token*, y puede ser escrita en mayúsculas o minúsculas.

Número_Token se refiere a un número único que se debe asignar a cada *token* del lenguaje. Este número puede servir para establecer relación con un *parser*, el cual toma diferentes acciones con base en los números de *token* que se encuentran en una secuencia.

Definición_Token es la especificación de los símbolos que componen el *token*. La prioridad de los *tokens* está determinada por el orden de aparición de los mismos. Para la definición se ha tomado la notación utilizada para definir expresiones regulares, la cual es la siguiente:

1.- Dos símbolos consecutivos indican que el *token* está compuesto por la concatenación de ambos símbolos.

2.- Dos símbolos X y Y separados por el signo ¦ (o sea X¦Y) indican que el *token* está compuesto por el símbolo X o por el símbolo Y, pero no por ambos.

3.- La operación Cerradura (concatenar con si mismo cero o más veces) del símbolo X se representa como X~.

4.- La operación Cerradura Positiva (concatenar con si mismo una o más veces) del símbolo X se representa como X^.

5.- Para agrupar símbolos se usan los signos [y].

6.- La precedencia de los operadores es la siguiente:

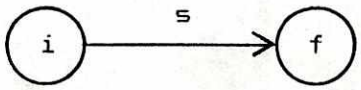
- Cerradura , Cerradura Positiva (~, ^).
- Concatenación.
- Unión (;).

Los símbolos que describen un *token* pueden ser independientes o pueden hacer referencia a un conjunto previamente definido por medio del nombre del conjunto.

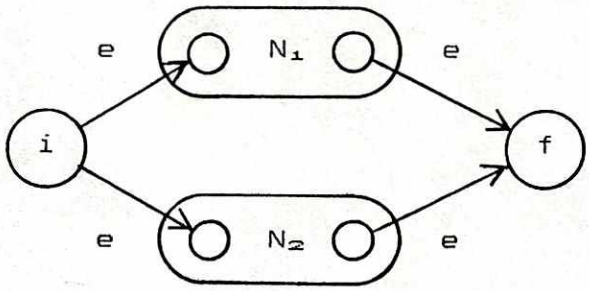
C. Transformación de expresiones regulares a NFA.

Cada vez que se define una expresión regular, GenDfa construye el NFA para dicha expresión. Cada uno de estos NFA individuales es encadenado con el estado inicial del NFA total por medio de una transición en el símbolo vacío (e). Los números de estado del NFA son asignados correlativamente. Para la construcción de los NFA individuales se aplican las siguientes reglas:

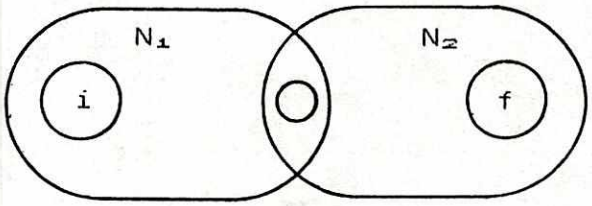
1.- Si existe una transición del estado i al estado f en el símbolo s, se tiene el siguiente NFA:



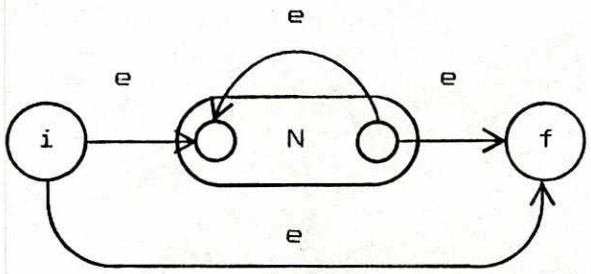
2.- Si se tienen los NFA's N_1 y N_2 para las expresiones regulares R_1 y R_2 respectivamente, para la construcción del NFA de la expresión regular R_1 ¦ R_2 (unión) se tiene:



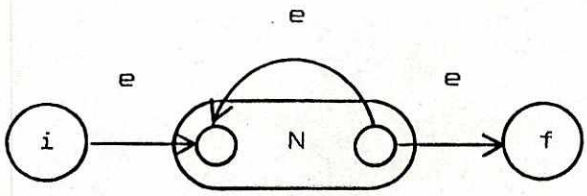
3.- Si se tienen los NFA's N_1 y N_2 para las expresiones regulares R_1 y R_2 respectivamente, para la construcción del NFA de la expresión regular R_1 R_2 (concatenación) se tiene:



4.- Si se tiene el NFA N para la expresión regular R, para la construcción del NFA de la expresión regular R ~ (Cerradura) se tiene:



5.- Si se tiene el NFA N para la expresión regular R, para la construcción del NFA de la expresión regular R ^ (Cerradura Positiva) se tiene:



D. Transformación de NFA a DFA.

El siguiente paso es la construcción del DFA a partir del NFA formado por las expresiones regulares. Para ésto, se construye un nuevo grupo de estados. Cada estado del DFA está compuesto por un conjunto de estados del NFA, a los cuales se hubiera llegado en el NFA después de leer una secuencia de símbolos de entrada. Para un análisis

detallado de los algoritmos utilizados para construir el DFA consultar la referencia de Aho et al (1979 : 92-124). A continuación se explican brevemente dichos algoritmos.

Para generar el DFA se necesita definir una función auxiliar llamada *e-Closure*(S), la cual determina el conjunto de estados a los cuales se puede llegar en el NFA desde estados en S por medio de transiciones en vacío. Para determinar *e-Closure*(S) se siguen las siguientes reglas:

i) Agregar S a *e-Closure*(S).

ii) Si existe un estado t en *e-Closure*(S), y existe una transición en vacío (e) desde t al estado u, entonces el estado u debe agregarse a *e-Closure*(S), si no es que ya existe. Este paso debe repetirse hasta que no se puedan agregar más estados a *e-Closure*(S).

El estado inicial del DFA (D_0) es el conjunto de estados del NFA que pueden ser alcanzados por medio de transiciones en vacío desde el estado inicial del NFA (N_0). Es decir, D_0 es *e-Closure*(N_0). Para generar los siguientes estados del DFA, se mantiene un conjunto de estados del DFA llamados "No-Marcados", el cual toma a D_0 como valor inicial. Luego se realiza el algoritmo V.1.

Después de construir el DFA se deben marcar como estados finales del DFA aquellos conjuntos que contengan por lo menos un estado final del NFA.

Algoritmo V.1. Transformación de NFA a DFA.

```
Mientras ( Exista un estado del DFA  X = { S1,...,Sn }
          que sea No-Marcado ) Hacer:
Inicio
     Marcar el estado X;
     Para cada símbolo de entrada s Hacer:
     Inicio
          Determinar T como el conjunto de estados a
          los cuales existe una transición en s desde
          algún elemento Sj en X;

          Calcular Y = e-Closure(T);

          Si no existe en el DFA un conjunto igual a Y,
          agregar Y al DFA como un conjunto No-Marcado;

          Agregar una transición en el DFA de X a Y en
          el símbolo s, si es que no existe;
     Fin
Fin
```

E. Implementación de GenDfa en Prolog.

La principal estructura para implementar GenDfa en Prolog fue una base de datos de hechos. En dicha base de datos se almacena la información de las transiciones, los estados, los estados finales, la definición de los conjuntos y demás datos. Por ejemplo, una transición del NFA del estado 1 al estado 2 en el símbolo "s", se representa como el siguiente hecho:

```
transición_nfa( 1, 2, "s" )
```

Cuando se necesita saber qué transiciones salen del estado 1 a cualquier otro estado en el símbolo "s", se pregunta a la base de datos por:

transición_nfa(1, X, "s")

El resultado de esta operación es que X tome los posibles valores que satisfagan dicha condición. La recuperación de datos en esta forma es el método de acceso para la información almacenada en la base de datos. Para agregar información a la base de datos se usan los predicados *ASSERTA* y *ASSERTZ*. *ASSERTA* sirve para agregar un hecho al principio de todos lo hechos del mismo tipo, mientras que *ASSERTZ* sirve para agregarlo al final. Para eliminar información de la base de datos se usa el predicado *RETRACT*.

Los principales hechos que GenDfa almacena en la base de datos son:

1.- símbolo(S) : indica que S es un símbolo. Este hecho sirve para generar los estados del DFA al examinar las transiciones existentes en todos los símbolos.

2.- transición_nfa(I,F,S) : establece que en el NFA construido a partir de las expresiones regulares,

existe una transición del estado I al estado F en el símbolo S.

3.- transición_dfa(I,F,S) : indica que en el DFA existe una transición del estado I al estado F en el símbolo S.

4.- final_nfa(E,T) : establece que el estado E del NFA es un estado final y en él se reconoce el *token* identificado como T.

5.- final_dfa(E_D,E_N) : indica que el estado E_D del DFA es un estado final que corresponde al estado final E_N del NFA. E_N es el menor estado final posible contenido en E_D, para que el *scanner* dé prioridad a los *tokens*, según el orden en que han sido definidos.

6.- definición(E,C) : sirve para definir que el estado E del DFA está compuesto por el conjunto C de estados del NFA.

7.- def_auxiliar(N,C) : indica que se ha especificado una definición auxiliar de un conjunto cuyo nombre es N y sus elementos están en C.

Para la creación del DFA se utilizan diferentes predicados que se pueden agrupar en los siguientes módulos:

1.- Conjuntos : que sirve para efectuar operaciones de conjuntos (listas) tales como unión, tamaño, miembro, subconjunto, *append*.

2.- Creación del NFA : el cual consta de predicados para formar los estados y las transiciones a partir de las expresiones regulares, y para crear las definiciones de los conjuntos especificados en el archivo de entrada.

3.- Generación del DFA : que consta de predicados para calcular *e-Closure*, formar los estados del DFA y sus transiciones, y determinar los estados finales.

4.- Generación de Tablas : que sirve para grabar en archivos la información del DFA.

Para ilustrar el funcionamiento de GenDfa en Prolog, se analiza como ejemplo el predicado para calcular *e-Closure*:

```
e_closure( S, Resultado ) :-
    miembro( T, S ),
    transicion_nfa( T, U, 'e' ),
    not( miembro( U, S )),
    append( S, U, Nuevo_S ), !,
    e_closure( Nuevo_S, Resultado ).

e_closure( S, Resultado ) :-
    Resultado = S.
```

El predicado recibe como parámetro de entrada el conjunto de estados S y produce, como salida, el conjunto Resultado. Para cada elemento T de S, se chequea si existe alguna transición del NFA en vacío (e) hacia algún estado U. Utilizando el *backtracking* de Prolog, el predicado miembro(X,Y) produce todos los posibles elementos X del conjunto Y. Las transiciones en vacío que salen del estado T son obtenidas dejando "libre" la variable U y consultando en la base de datos el hecho transicion_nfa(T,U,'e'). Como U está "libre", Prolog le asigna los posibles valores que satisfacen la condición. El próximo paso consiste en verificar si el estado U no pertenece a S y agregarlo, formando un nuevo conjunto Nuevo_S. Se puede observar la presencia del signo !, el cual es llamado *CUT* y sirve para que Prolog ya no realice el *backtracking* de los predicados miembro y transicion_nfa que había dejado pendientes. Luego se realiza una llamada recursiva al predicado con Nuevo_S. La segunda parte del predicado *e-closure* sirve para asignar el Resultado y es necesaria para que éste sea *True* cada vez que sea llamado.

Como se puede observar, los predicados en Prolog, únicamente indican qué operaciones se deben hacer con los datos y no la forma como éstas deben efectuarse. Por eso, Prolog es un lenguaje orientado a declaraciones y no a procedimientos, como sucede con la mayor parte de lenguajes

tradicionales. El desarrollo de GenDfa se hizo en Turbo Prolog versión 1.1.

F. Tablas del *scanner*.

El resultado final de GenDfa son tres tablas, las cuales contienen la siguiente información :

1.- Tabla de conjuntos : contiene los nombres y los elementos de los conjuntos definidos. Esta información es necesaria ya que la tabla de transiciones del DFA hace referencia a los nombres de los conjuntos cuando es necesario.

2.- Tabla de transicíones : contiene las transiciones entre los diferentes estados del DFA. Cada fila indica el estado inicial, estado final, y el símbolo en el cual ocurre la transición.

3.- Tabla de estados finales : indica qué estados del DFA son estados finales y qué *token* es reconocido en cada caso.

G. Ejemplo de GenDfa.

A continuación se muestran los archivos de salida (conjuntos, transiciones y finales) producidos por GenDfa al ejecutarse con la definición de *tokens* del ejemplo anterior.

TABLA DE CONJUNTOS

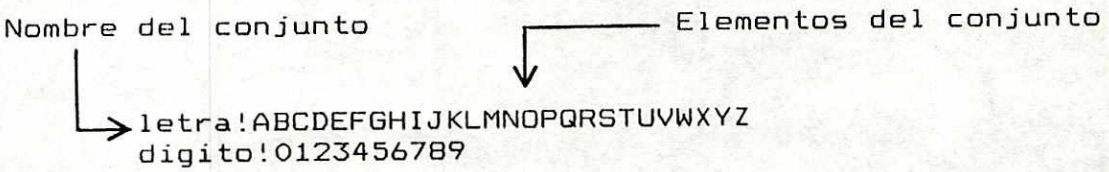
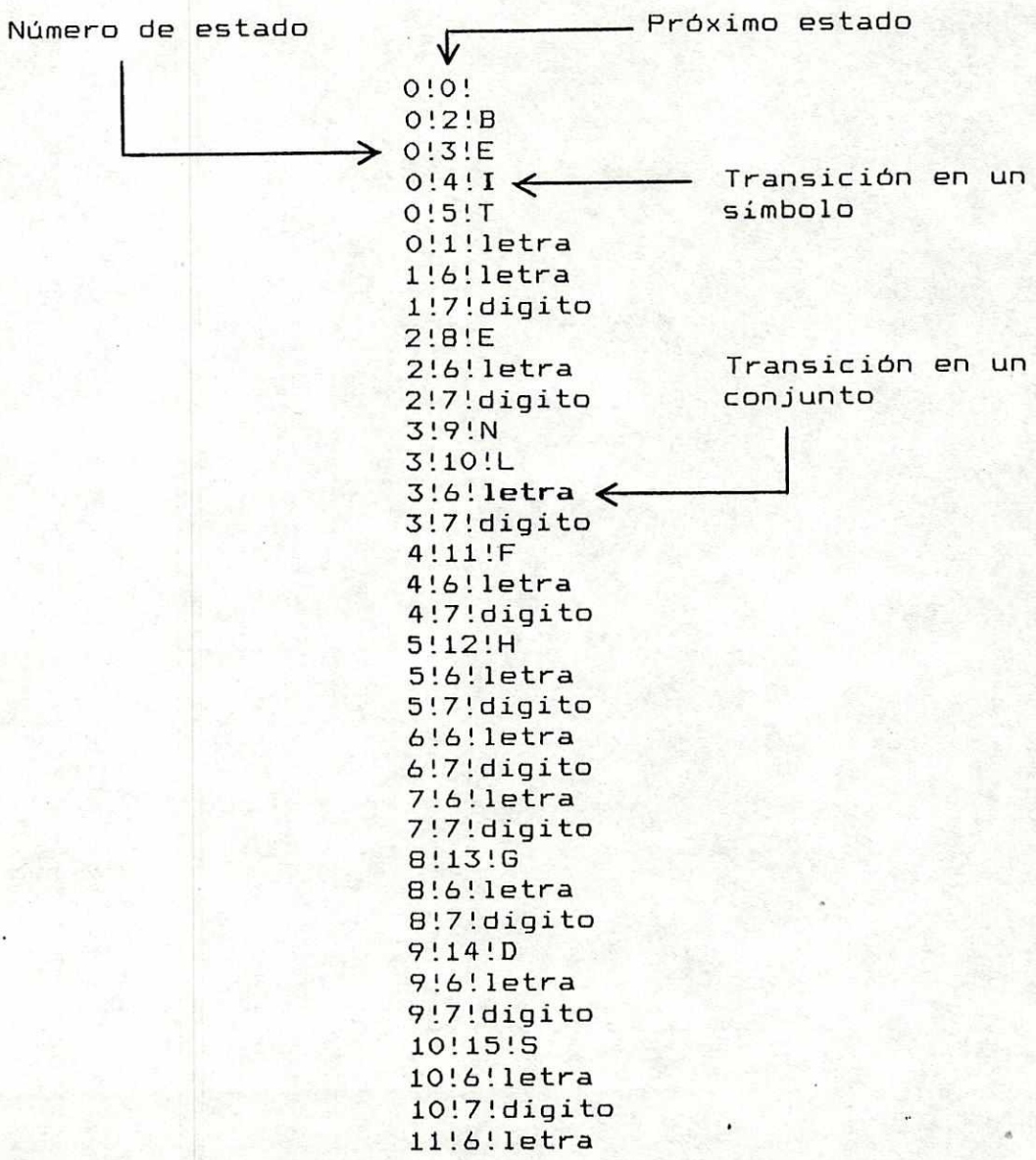


TABLA DE TRANSICIONES



```
11!7!digito
12!16!E
12!6!letra
12!7!digito
13!17!I
13!6!letra
13!7!digito
14!6!letra
14!7!digito
15!18!E
15!6!letra
15!7!digito
16!19!N
16!6!letra
16!7!digito
17!20!N
17!6!letra
17!7!digito
18!6!letra
18!7!digito
19!6!letra
19!7!digito
20!6!letra
20!7!digito
```

TABLA DE ESTADOS FINALES

Estado final del DFA — Número de *token* reconocido

```
1!06
2!06
3!06
4!06
5!06
6!06
7!06
8!06
9!06
10!06
11!03
12!06
13!06
14!02
15!06
16!06
17!06
18!05
19!04
20!01
```

VI. GENERADOR DE TABLAS DEL *PARSER*

Un *parser* es la parte de un compilador encargada de determinar si un programa fuente está correctamente escrito. Por medio del *parser* se genera el árbol de *parse* del programa fuente, el cual establece estrictamente la prioridad y asociatividad de las operaciones que se realizan. Luego el árbol de *parse* es utilizado por el compilador para generar el código objeto.

Muchos de los *parsers* actuales funcionan con base en la información contenida en ciertas tablas constantes: las tablas de *parse*. Este tipo de compiladores consta, principalmente, de dos partes. La primera parte es un conjunto de tablas en las cuales está definida la gramática del lenguaje fuente. La segunda parte consiste en un programa que toma la información de dichas tablas y se encarga de examinar un programa escrito en el lenguaje fuente y determinar si es correcto. Esta clase de *parsers* está siendo ampliamente utilizada debido a que resultan bastante eficientes.

Un tipo especial de *parsers* que utilizan tablas son los llamados LR. Existen diferentes clases de *parser* LR dependiendo de la complejidad de las gramáticas que pueden

reconocer. Aunque los algoritmos para la construcción de las tablas de *parse* están totalmente definidos, son difíciles de implementar manualmente para gramáticas grandes. Por eso, existen programas que sirven para generar, automaticamente, dichas tablas a partir de una gramática.

La segunda parte de este proyecto consistió en desarrollar una herramienta llamada **GenSlr**, la cual sirve para generar las tablas de *parse* de una gramática. Esto facilitó, grandemente, el desarrollo de FastMan, ya que, gradualmente, se fue incrementando la gramática de Mantis para que contara con un mayor conjunto de instrucciones. Utilizando GenSlr, estos incrementos significaron, únicamente, una nueva generación de las tablas de *parse*. Este proceso no involucró cambios de programación, más que los necesarios para generar el código de las nuevas instrucciones.

A. Funcionamiento de GenSlr.

GenSlr es un programa que sirve para generar las tablas de *parse* de un lenguaje a partir de la definición de la gramática del mismo. Las tablas generadas son del tipo SLR (*Simple* LR). Al igual que GenDfa, GenSlr está escrito en lenguaje Prolog por las mismas razones anteriores. La tabla de *parse* es semejante a un grafo, en el cual los

vértices son los estados y las aristas son las transiciones válidas entre los estados según los diferentes *tokens* y producciones. Esta estructura se puede representar fácilmente en Prolog.

A continuación se analiza un ejemplo del funcionamiento de GenSlr. Se supone que se desea generar la tabla de *parse* SLR para una gramática que reconoce expresiones aritméticas con las operaciones suma (+) y multiplicación (*). Utilizando notación BNF, la definición de la gramática es la siguiente:

```
Expresión ::= Expresión + Término ¦ Término
Término   ::= Término * Factor ¦ Factor
Factor    ::= ( Expresión ) ¦ id
```

Esta definición establece la precedencia de la operación multiplicación sobre la operación suma. Se puede observar también que se establecen seis producciones. Hay además ocho símbolos, de los cuales tres son no-terminales (Expresión, Término, Factor) y cinco son terminales (los *tokens* +, *, (,), id). El primer paso para la construcción de la tabla SLR de esta gramática es eliminar la *recursividad inmediata a la izquierda* de la misma. Si se tienen las producciónes A ::= A α ¦ β, las cuales son recursivas a la izquierda, y en donde β no comienza con A, se puede eliminar la recursividad a la izquierda sustituyendo las producciones por:

```
A  ::= β A'
A' ::= α A' | e
```

Aplicando esta transformación a la gramática original y numerando consecutivamente las producciones se tiene:

```
1   Expresión   ::= Término Expresión'
2   Expresión'  ::= + Término Expresión'
3   Expresión'  ::= e
4   Término     ::= Factor Término'
5   Término'    ::= * Factor Término'
6   Término'    ::= e
7   Factor      ::= ( Expresión )
8   Factor      ::= id
```

Para usar GenSlr hay que ingresar esta definición en un archivo texto. Los nombres de las producciones deben ingresarse en minúsculas y los *tokens* en mayúsculas. El texto quedaría así:

```
expresion   ::= termino expresion_p
expresion_p ::= + termino expresion_p | e
termino     ::= factor termino_p
termino_p   ::= * factor termino_p | e
factor      ::= ( expresion ) | ID
```

Al ejecutar el programa GenSlr con este archivo como entrada, se genera la siguiente tabla:

TABLA SLR PARA GRAMATICA DE EXPRESIONES

	A C C I O N							G O T O				
i	ID	+	*	(	)	e	$	E	Ep	T	Tp	F
0	S5			S4				1		2		3
1							A					
2		S7				E8			6			
3			S11			E9					10	
4	S5			S4				12		2		3
5		R8	R8		R8		R8					
6					R1		R1					
7	S5			S4						13		3
8					R3		R3					
9		R6			R6		R6					
10		R4			R4		R4					
11	S5			S4								14
12					S15							
13		S7				E8			16			
14			S11			E9					17	
15		R7	R7		R7		R7					
16					R2		R2					
17		R5			R5		R5					

en donde:

$ = Símbolo para indicar fin de entrada

E = expresión
Ep = expresión_p
T = término
Tp = término_p
F = factor

La tabla consta de dos partes: Acción y *Goto*. La parte de Acción sirve para determinar cuál será el próximo estado en que debe entrar el *parser* cuando en la entrada se presenta determinado *token*. Además, indica si se ha llegado a completar una producción, a aceptar como válida la expresión de entrada, o a una condición inválida. Cada

entrada de Acción en la tabla de parser SLR, tiene el siguiente significado:

- S i significa *Shift* i.
- E i significa *Else* i, que es equivalente a *Shift* i pero sin consumir el *token* de entrada porque es nulo (e).
- R j significa *Reduce* por la producción numerada j.
- A significa aceptar, o sea que la entrada es válida.
- Blanco significa Error.

La parte de *Goto* sirve para determinar el siguiente estado en el que debe entrar el *parser* si se ha reconocido una producción.

B. Construcción de la tabla SLR.

En forma semejante a la construcción de las tablas del *scanner*, para construir las tablas de *parser* se transforma la definición de la gramática en forma de un NFA, a un DFA. Los estados del DFA son conjuntos de *items* y las transiciones entre los estados se realizan por medio de los *tokens* y las producciones de la gramática. La tabla SLR se construye a partir del DFA. La especificación completa de los algoritmos para construir el DFA y la tabla SLR puede encontrarse en Aho et al (1979 : 197-214).

C. Implementación de GenSlr en Prolog.

Al igual que GenDfa, GenSlr hace uso de la base de datos de hechos de Prolog. La gramática se representa como un conjunto de producciones, las cuales indican el nombre de la producción y su especificación en forma de una lista de elementos. A partir de esta información se generan los conjuntos de *items* y las transiciones del DFA. Simultáneamente se generan las entradas de la tabla SLR, las cuales no son almacenadas en la base de datos, ya que son muchas y saturan la memoria principal del computador. Por esa razón son almacenadas directamente en disco.

Los principales hechos que GenSlr almacena en la base de datos de Prolog son:

1.- s(S) : indica que S es un símbolo. Los símbolos son todos los *tokens* (terminales) y los nombres de las producciones de la gramática (no-terminales).

2.- p(S,L,N) : sirve para representar las producciones de la gramática. S es un símbolo que representa el nombre de la producción. L es la definición de la producción representada como una lista de símbolos, los cuales pueden ser terminales o no-terminales. N es un número correlativo que se asigna a la producción y que sirve para identificarla en el *parser*.

3.- definicion(N,L) : se utiliza para almacenar la definición de un conjunto de *items* del DFA. N es un número correlativo asignado a las definiciones. L es una lista de *items*. Los *items* se representan como parejas (P,J), en donde P es el número de producción del *item* y J es la posición del *dot* dentro de la producción.

4.- no_marcado(N) : indica que el conjunto de *items* N es nuevo y no han sido analizadas sus transiciones para generar el DFA.

5.- *first*(S,L) : indica que la lista de símbolos L ha sido calculada como *first*(S), en donde S es un símbolo no-terminal. Ya que *first*(S) es continuamente utilizada durante la creación de la tabla SLR, se optó por calcularla para todos los no-terminales una sola vez al inicio del programa, y luego hacer un acceso a la información almacenada.

6.- *follow*(S,L) : indica que la lista de símbolos L ha sido calculada como *follow*(S), en donde S es un símbolo no-terminal. Al igual que *first*(S), *follow*(S) se usa frecuentemente en la generación de la tabla SLR a partir del DFA. Por eso, al inicio del programa se calcula para todos los no-terminales y

se almacena. Si durante el proceso se necesita, únicamente se hace referencia a la lista ya calculada.

Para la generación de la tabla SLR, GenSlr utiliza diferentes predicados, los cuales se pueden agrupar en los siguientes módulos:

1.- Conjuntos : el cual contiene predicados para efectuar operaciones básicas sobre conjuntos, tales como unión, miembro, subconjunto y *append*.

2.- Lectura de producciones : que consta de predicados para leer las producciones definidas en el archivo fuente de la gramática y formar las listas respectivas.

3.- Cálculo de *First* y *Follow* : que contiene predicados para calcular dichas funciones. Este proceso se realiza inmediatamente después de definir las producciones y antes de generar el DFA.

4.- Cálculo de *Closure* y *Goto* : que consta de predicados para el cálculo de tales funciones. Este cálculo se realiza según lo demanda la generación del DFA.

5.- Generación de la Colección de *items* : que sirve para generar la gramática aumentada del lenguaje y producir los *items* y transiciones del DFA.

6.- Generación de la tabla SLR : que contiene predicados para producir las entradas de la tabla SLR y grabar en los archivos. Además graba los archivos auxiliares.

A continuación se muestra como ejemplo el predicado para calcular la función *Goto*(I,X), la cual se utiliza para determinar las transiciones y construir el DFA de la gramática. El parámetro de entrada I es un conjunto de *items*, y X es un símbolo. *Goto*(I,X) se calcula de la siguiente forma:

i) Calcular G como el conjunto de *items* tal que:
G = { [A ::= α X • β] }
dado que [A ::= α • X β] pertenece a I.

ii) Calcular *Goto*(I,X) como el *Closure* del conjunto G obtenido anteriormente.

Para determinar *Closure*(I) se sigue el siguiente algoritmo:

i) Para cada *item* [A ::= α • B β] en I y cada producción B ::= μ tal que [B ::= • μ] no está

en I, agregar el *item* [B ::= • μ] a I. Repetir este paso hasta que ya no puedan ser agregados más *items* a I.

ii) Regresar I como *Closure*(I).

Para calcular *Goto*(I,X) en Prolog, se utilizan los siguientes predicados:

```
goto( I, X, Gto ) :-
    goto1( I, X, [], G ),
    closure( G, Gto ).


goto1( I, X, Lleva, G ) :-
    miembro_item( item( A, J ), I ),
    siguiente( item( A, J ), S ),
    X = S,
    H = J + 1,
    not( miembro_item( item( A, H ), Lleva )),
    append_item( Lleva, item( A, H ), Nuevo_Lleva ), !,
    goto1( I, X, Nuevo_Lleva, G ).

goto1( I, X, Lleva, G ) :-
    G = Lleva.


closure( I, Resultado ) :-
    miembro_item( item( A, J ), I ),
    siguiente( item( A, J ), P ),
    produccion( P, Expansion_P, B ),
    not( miembro_item( item( B, 0 ), I )),
    append_item( I, item( B, 0 ), Nuevo_I ), !,
    closure( Nuevo_I, Resultado ).

closure( I, Resultado ) :-
    Resultado = I.
```

Como se puede observar, se utiliza un predicado auxiliar para calcular el conjunto de *items* G y otro para

obtener el *Closure* de G. El predicado *goto1* recibe como entradas el conjunto de *items* I, el símbolo X y un acumulador llamado Lleva. El primer paso consiste en determinar un elemento de I, el cual es el *item* (A,J). Luego se determina el símbolo S, el cual está en la posición J+1 de la producción A. Si el símbolo S es igual a X, se debe agregar un nuevo *item* al conjunto Lleva, verificando previamente que no exista ya en él. Luego se efectúa una llamada recursiva al predicado *goto1* con el nuevo conjunto calculado. La forma de calcular *Closure* es semejante. GenSlr está escrito en Turbo Prolog versión 1.1.

D. Tablas del parser.

El resultado final de GenSlr es un conjunto de cuatro archivos, los cuales contienen la siguiente información:

1.- Tabla de acciones : contiene la información de las acciones que se deben realizar (*Shift*, *Reduce*, Aceptación, Error) de acuerdo a los *tokens* que se encuentran en el lenguaje fuente.

2.- Tabla de *Goto* : indica las acciones que se deben tomar cuando se efectúa la reducción de una producción. Las producciones se identifican por el número correlativo que GenSlr les ha asignado.

3.- Tabla de expansiones : contiene la información de las producciones que sirve cuando se ha efectuado una reducción. Principalmente almacena el número de símbolos que cada producción tiene.

4.- Archivo de producciones : sirve para determinar qué número se le ha asignado a cada producción definida en el archivo fuente. Esta información sirve para que el usuario sepa qué producción se ha reconocido en cada reducción, y así pueda generar el código respectivo dentro del *parser*.

Para hacer más eficiente el uso de la tabla de acciones, el archivo resultante se somete a una clasificación (*sort*), la cual ordena la tabla de acuerdo a los números de *token* dentro de cada fila de la misma. Luego, el *parser* que hace uso de esta tabla puede detectar más fácilmente los errores sin necesidad de recorrer toda la fila para buscar un *token* específico. De igual forma, el *parser* puede determinar si la fila tiene un *Else*, y usar esa transición si no existiera otra con el *token* actual.

Debido a que la tabla de acciones resulta ser muy grande (la tabla SLR de la gramática de FastMan tiene aproximadamente 4,500 entradas), resulta conveniente, en función de espacio, hacer una compactación de dicho archivo

eliminando los espacios en blanco. El archivo compactado es, aproximadamente, el 80 % del tamaño del archivo original.

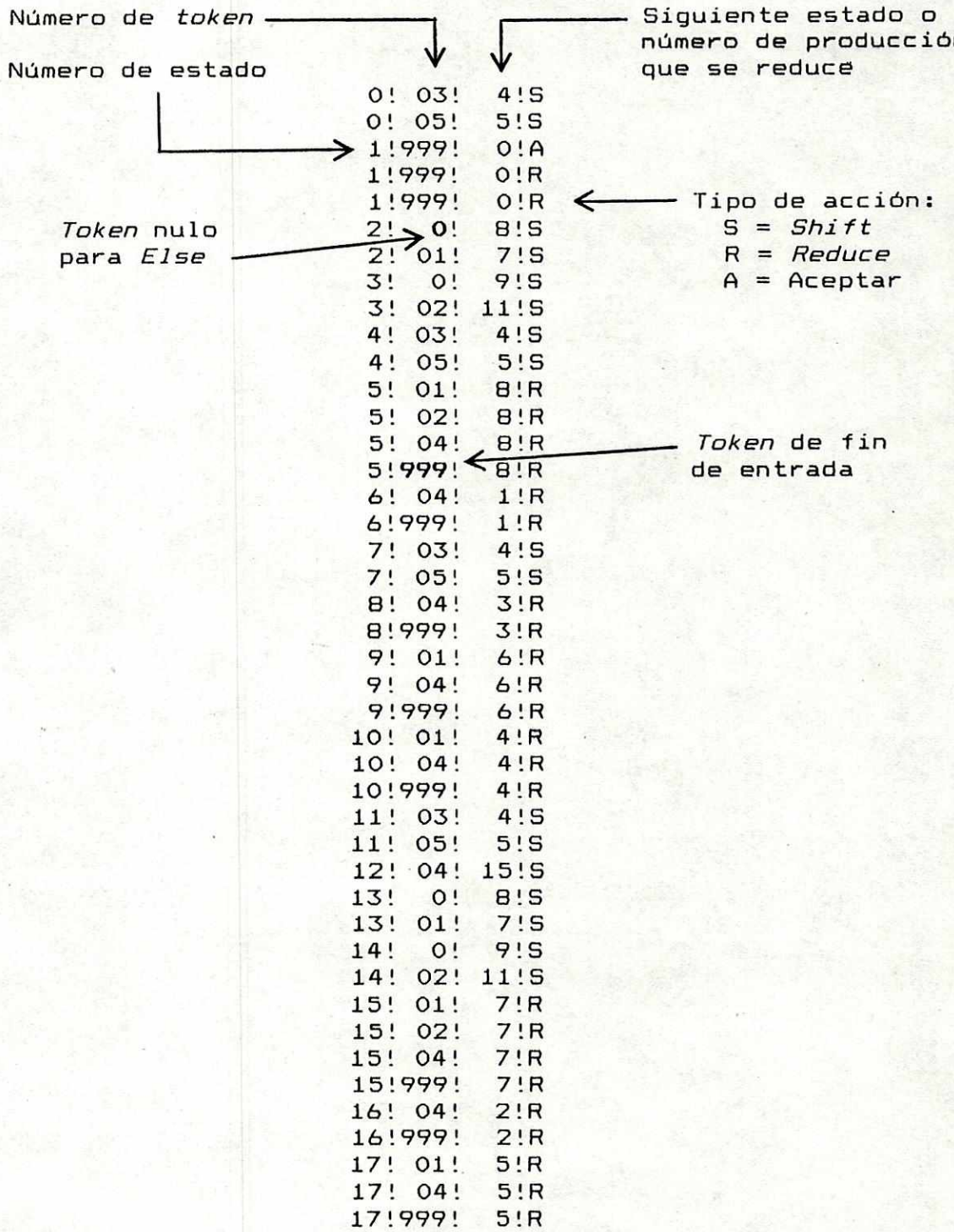
E. Ejemplo de GenSlr.

A continuación se muestra un ejemplo del funcionamiento de GenSlr. Como se podrá notar, al archivo de entrada se ha agregado una lista de *tokens* con sus respectivos números. Esto se hace para poder efectuar el enlace con los *tokens* identificados por las tablas de GenDfa. Dichos números de *token* se usan para producir las transiciones de la tabla de acciones.

Archivo de entrada.

```
Token 01=+
Token 02=*
Token 03=(
Token 04=)
Token 05=ID
expresion ::= termino expresion_p
expresion_p ::= + termino expresion_p ¦ e
termino ::= factor termino_p
termino_p ::= * factor termino_p ¦ e
factor ::= ( expresion ) ¦ ID
```

TABLA DE ACCIONES (NO COMPACTADA).



0	03	4 S
0	05	5 S
1	999	0 A
1	999	0 R
1	999	0 R
2	0	8 S
2	01	7 S
3	0	9 S
3	02	11 S
4	03	4 S
4	05	5 S
5	01	8 R
5	02	8 R
5	04	8 R
5	999	8 R
6	04	1 R
6	999	1 R
7	03	4 S
7	05	5 S
8	04	3 R
8	999	3 R
9	01	6 R
9	04	6 R
9	999	6 R
10	01	4 R
10	04	4 R
10	999	4 R
11	03	4 S
11	05	5 S
12	04	15 S
13	0	8 S
13	01	7 S
14	0	9 S
14	02	11 S
15	01	7 R
15	02	7 R
15	04	7 R
15	999	7 R
16	04	2 R
16	999	2 R
17	01	5 R
17	04	5 R
17	999	5 R

TABLA DE GOTO

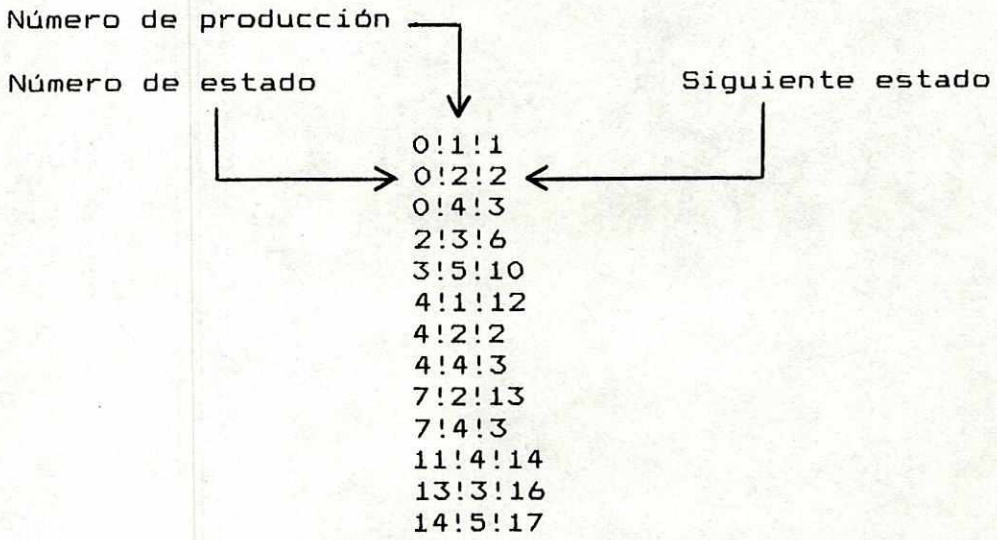


TABLA DE EXPANSIONES

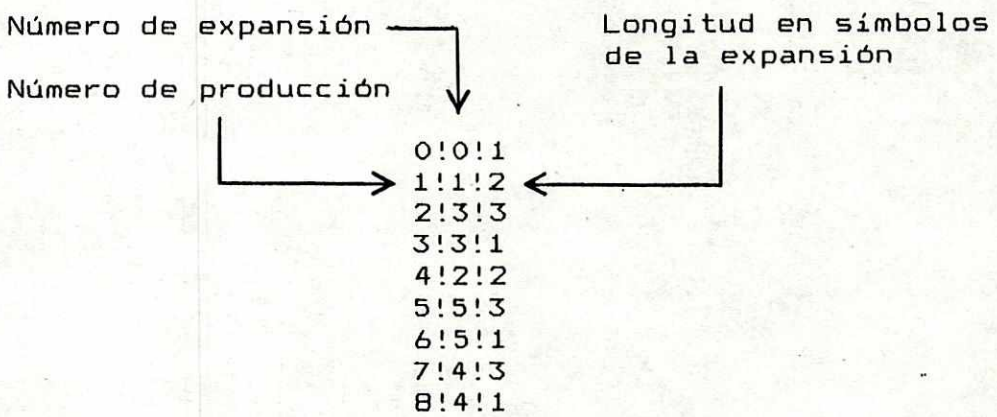
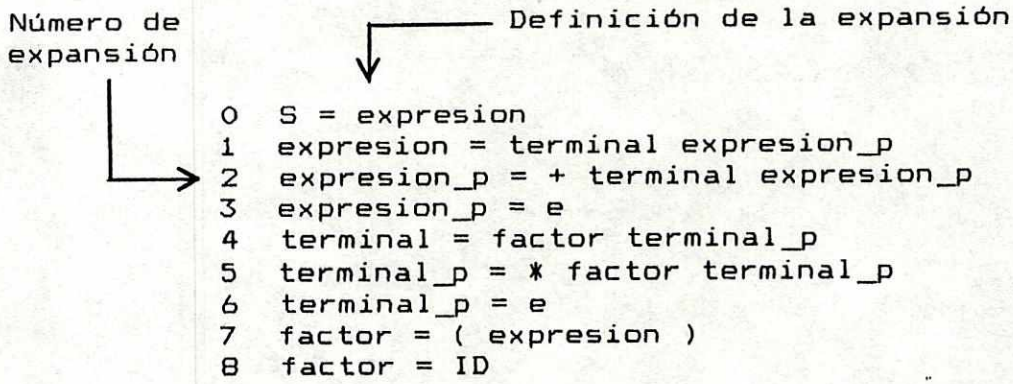


TABLA DE PRODUCCIONES



VII. TRADUCTOR DE MANTIS A COBOL

En este capítulo se describe la implementación del traductor. Se muestran las principales estructuras de datos utilizadas y su funcionamiento. También se detallan los esquemas de traducción de cada instrucción de Mantis. El traductor esta compuesto de las siguientes partes principales:

1.- El *scanner*, que se encarga de reconocer los *tokens* del lenguaje Mantis.

2.- El *parser*, encargado de verificar que el programa fuente esté correcto y de construir el árbol de *parse* para generar el código en Cobol.

3.- El generador de código, cuya función es producir las instrucciones en Cobol respectivas, de acuerdo al árbol de *parse* construido por el *parser*.

A. *Scanner*.

El *scanner* que utiliza el traductor funciona con base en las tablas para reconocimiento de los *tokens* de Mantis generadas por GenDfa. La primera acción que se ejecuta cuando el traductor empieza a funcionar es leer las tablas del *scanner* y colocarlas en memoria principal, para agilizar su acceso. El *scanner* funciona leyendo cada carácter de la

entrada y viajando por los estados del DFA hasta llegar a un estado final o a una condición de error. El DFA (que realmente es un grafo) se representa por medio de un conjunto de estados (vértices), cada uno de los cuales tiene un conjunto de transiciones (aristas) en los diferentes símbolos. Las aristas se almacenan en forma de listas de adyacencia para hacer uso óptimo del espacio. Si las transiciones se efectúan en conjuntos, es necesario que el *scanner* verifique si el carácter que se tiene, actualmente, en la entrada está contenido en dichos conjuntos. Por eso, también se mantiene una tabla de los conjuntos con sus respectivos elementos. El algoritmo utilizado para reconocer los *tokens* puede encontrarse en Aho et al (1979:92). La función final del *scanner* es enviar al *parser* un conjunto de números de *token*, los cuales son utilizados para viajar en el DFA de la gramática de FastMan y reconocer las instrucciones.

B. *Parser*.

La función principal del *parser* es construir el árbol de *parse* del programa fuente en Mantis. Para ello hace uso de las tablas SLR de la gramática de FastMan generadas por GenSlr. El *parser* es del tipo LR y la descripción del algoritmo utilizado se puede encontrar en Aho et al (1979:198). El *parser* funciona viajando a través de los estados del DFA de la gramática. El DFA se forma leyendo

al inicio del programa las tablas SLR y colocándolas en memoria principal. El DFA consta de un conjunto de estados y sus respecivas transiciones con base en los números de *token*. Dichas transiciones se almacenan como listas de adyacencia para optimizar el uso del espacio.

C. Primera y segunda pasadas.

El traductor realiza dos pasadas al programa fuente. En la primera pasada, verifica que el programa esté escrito correctamente de acuerdo a la gramática. Si se detectara algún error de sintaxis, se despliega un mensaje indicándolo. Si no se detectan errores, el traductor prosigue con la segunda pasada. Durante la primera pasada, el traductor genera una tabla con los nombres de las variables y de los parámetros formales de los procedimientos que estan definidos en el programa fuente. Además hay una tabla de los nombres de los procedimientos que se definen. Esto se hace para que durante la segunda pasada se verifique que sólo se hace referencia a procedimientos y variables existentes. Además, durante la primera pasada se definen los archivos que el programa utiliza y sus respectivos campos.

La segunda pasada completa la asignación de tipos a las variables formales, según los parámetros actuales, construye

el árbol de *parse* y efectúa la generación de código de las instrucciones.

D. Generación de código.

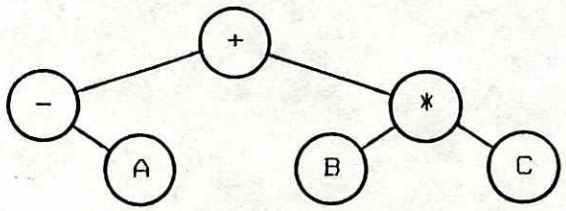
La generación de código se hace conforme se van reduciendo las producciones definidas en la gramática. Cuando se efectúa una operación *Reduce* i en el *parser*, éste invoca a la rutina para generar código con el parámetro i. Esta rutina ejecuta entonces el módulo para generar el código de la producción i. Para poder efectuar esto fue necesario hacer ciertas modificaciones en la forma de escribir la gramática de FastMan, ya que en algunos casos se necesita generar código con sólo parte de la instrucción y no con ella completa.

A continuación se explican los esquemas de traducción utilizados para la generación de código en Cobol a partir de Mantis.

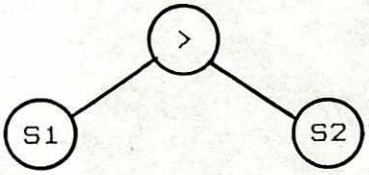
1. **Expresiones aritméticas.** Para el manejo de las expresiones aritméticas se utiliza un procedimiento semejante al utilizado para transformar una expresión en notación *Infix* a notación *Postfix*. Los operandos y los operadores se almacenan en un *stack*, y conforme se van realizando las operaciones, van formando un árbol de la expresión. La sintaxis de Cobol para expresiones

aritméticas es bastante parecida a la de Mantis. Las operaciones permitidas y sus prioridades son las mismas. Por eso, para generar el código en Cobol de la expresión se necesita unicamente realizar un *Inorder* del árbol de *parse*. Se decidió utilizar un árbol ya que cuando hay operadores de comparación o referencias a funciones junto con operadores aritméticos, es necesario eliminar ramas del árbol y sustituirlas por ciertas variables temporales.

Por ejemplo, el árbol generado para la expresión - A + B * C sería de la siguiente forma:



2. Expresiones booleanas. Al igual que las expresiones aritméticas, las expresiones booleanas también generan un árbol de la expresión. Ya que Mantis permite la combinación de ambos tipos de expresiones, hay que efectuar ciertos ajustes en el árbol porque en Cobol no es permitido hacer esta combinación. Estos ajustes consisten en crear ciertas variables temporales para que los tipos de los operandos sean compatibles con el operador. Por ejemplo, el árbol de la expresión A + B > C - D sería el siguiente:

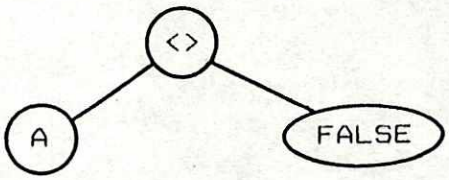


con S1 = A + B
S2 = C - D

Otro aspecto que se toma en cuenta es que en Mantis es posible escribir condiciones como la siguiente:

```
IF A
 SHOW TRUE
ELSE
 SHOW FALSE
END
```

en la cual si A tiene un valor distinto de FALSE (la variable predefinida FALSE tiene el valor 0), la condición booleana es verdadera. Sin embargo, en Cobol no es permitido utilizar este tipo de condiciones. Por eso, para poder generar una condición válida en Cobol, el árbol se construye de la siguiente forma:



3. Expresiones con *strings*. En Mantis se facilita, grandemente, el manejo de los *strings*, ya que se pueden efectuar operaciones de suma y resta. Por otro

lado, Cobol es bastante limitado porque no puede realizar estas operaciones y fue necesario implementarlas. Para efectuar la traducción, cada vez que se hace una operación con *strings* se realiza una llamada a una rutina externa con ciertos parámetros. Esta rutina externa está escrita en Cobol y la llamada se hace por medio de la instrucción *CALL USING*. La rutina recibe ciertos parámetros de entrada, efectúa la operación requerida y produce los resultados. Esta técnica fue la que se utilizó para traducir todas las funciones y operaciones que no existen en Cobol. Durante la compilación de un programa en Cobol, el compilador se encarga de incluir en el programa objeto todos los módulos externos que son referenciados. Por lo tanto, usar esta facilidad genera un código eficiente. Por ejemplo, suponiendo que las variables X, Y y Z son de tipo *TEXT*, la operación en Mantis:

```
Z = X + Y
```

se traduciría a Cobol como:

```
MOVE X TO STR1.
MOVE X-L TO STR1-L.
MOVE Y TO STR2.
MOVE Y-L TO STR2-L.
CALL 'SUMAR' USING STR1, STR1-L,
                   STR2, STR2-L,
                   STR-RES, STR-RES-L.
MOVE STR-RES TO Z.
MOVE STR-RES-L TO Z-L.
```

Como se puede observar, inicialmente se asignan los parámetros actuales a los parámetros formales de la rutina (STR1 y STR2). Se efectua el *CALL* con los parámetros y luego se asigna el resultado (STR-RES). Debido a que los parámetros actuales pueden ser de diferentes tamaños, no pueden incluirse directamente en el *CALL*. Cada variable de tipo *TEXT* tiene asociada en Cobol otra variable numérica (con sufijo -L), la cual sirve para almacenar la longitud actual de la variable. Esto es necesario ya que Cobol no toma en cuenta las longitudes de las variables con *Picture* X. Existen rutinas externas para implementar las operaciones de suma y resta con *strings*.

La comparación de *strings* se tradujo a Cobol en la misma forma como funciona en Mantis. Primero se comparan las longitudes de los *strings* según el operador relacional. Si esta comparación resulta *TRUE*, se comparan los *strings* con el mismo operador, produciendo el resultado. Si la comparación de las longitudes resulta *FALSE*, la comparación total será también *FALSE*.

4. Funciones aritméticas y trigonométricas. Estas funciones se implementaron por medio de llamadas a rutinas externas. Las funciones aritméticas son: *ABS*, *INT*, *RND*, *SGN*, *EXP*, *LOG*. La función *RND* (*Random*), fue implementada como una relación de recurrencia que toma un

valor inicial y produce una sucesión de términos distribuidos en un rango y con igual probabilidad de ocurrencia. Un análisis detallado de dicha función se puede encontrar en Naylor et al (1975). Por medio del comando *SEED* se puede modificar el valor inicial de la sucesión. Las funciones *EXP* y *LOG* se implementaron por medio de aproximaciones con series de Taylor. Para generar estas funciones se utilizó una técnica llamada *bootstrapping*. Las rutinas se escribieron en Mantis y por medio de FastMan se tradujeron a sus equivalentes en Cobol. Luego, estas funciones ya pudieron ser incorporadas en la gramática de FastMan.

Las funciones trigonométricas también se aproximaron por medio de series de Taylor, y son las siguientes: *SIN*, *COS*, *TAN*, *ATN*. Los desarrollos de las series de Taylor se pueden encontrar en Protter et al (1980:589) y Spiegel (1970).

5. Funciones con *strings*. Las funciones con *strings* son: *PAD*, *UNPAD*, *POINT*, *TXT*, y *VALUE*. Además, se tienen las siguientes funciones para manejo de *substrings*:

i) Acceso de *substrings* : la cual sirve para traducir las expresiones de Mantis que hacen acceso parcial de variables tipo *TEXT*. Por ejemplo, X(3,20)

significa el *substring* de X entre las posiciones 3 y 20.

ii) Asignación parcial : que sirve para asignar un *string* a una parte de otro *string*. Por ejemplo, cuando se hace X(5,10)='******', significa que asigne '******' a X pero sólo desde la posición 5 hasta la posición 10, dejando el resto de X intacto.

6. Manejo de arreglos. Dentro de la gramática de FastMan únicamente se consideró el uso de arreglos de una dimensión. Cuando se efectúa la traducción a Cobol de un programa en Mantis que utiliza arreglos, se puede generar opcionalmente un chequeo de rangos sobre los subíndices de los mismos. Esto agrega un pequeño *overhead* a las operaciones con arreglos pero puede servir para detectar situaciones erróneas en los programas. Si durante la ejecución del programa se detecta un acceso a un arreglo fuera de sus límites, el programa finaliza y despliega un mensaje de error indicando en qué variable sucedió el problema.

Si se está seguro de que el programa hace uso correcto del arreglo y no sobrepasa sus fronteras, se puede traducir

el programa sin chequeo de rangos, lo cual hace el código más eficiente.

7. Estructuras de decisión. Las estructuras de decisión de Mantis son *IF* y *WHEN*. Los posibles formatos del *IF* y sus esquemas de traducción a Cobol son:

i) Mantis:

```
IF condición
   sentencias_then
ELSE
   sentencias_else
END
```

Cobol:

```
     IF NOT ( condición )
        GO TO ELSE-IF-N.
     sentencias_then
     GO TO FIN-IF-N.
ELSE-IF-N.
     sentencias_else
FIN-IF-N.
```

ii) Mantis:

```
IF condición
   sentencias_then
END
```

Cobol:

```
     IF NOT ( condición )
        GO TO ELSE-IF-N.
     sentencias_then
ELSE-IF-N.
```

Las etiquetas en Cobol ELSE-IF-N y FIN-IF-N son generadas asignando un número correlativo a cada una, para que no se confundan con las de otras decisiones. Además,

para manejar condiciones anidadas se hace uso de un *stack* de etiquetas. Las instrucciones sentencias_then y sentencias_else pueden ser nulas, en cuyo caso el traductor genera un *warning* para que se tomen las acciones correctivas si fuera necesario.

El formato de la instrucción *WHEN* en Mantis es el siguiente:

```
WHEN condición₁
   sentencias₁
WHEN condición₂
   sentencias₂
      .
      .
      .
WHEN condiciónₙ
   sentenciasₙ
END
```

Tomando en cuenta que el *WHEN* es no-exclusivo, o sea que pueden ejecutarse $sentencias_1$, $sentencias_2$, ..., $sentencias_n$, si las respectivas condiciones se cumplen, la traducción es la siguiente:

```
     IF NOT ( condición1 )
        GO TO ELSE-WHEN-N1.
     sentencias1
ELSE-WHEN-N1.
     IF NOT ( condición2 )
        GO TO ELSE-WHEN-N2.
     sentencias2
ELSE-WHEN-N2.
             .
             .
             .
     IF NOT ( condiciónn )
        GO TO ELSE-WHEN-Nn.
     sentenciasn
ELSE-WHEN-Nn.
```

8. Estructuras de control. Las estructuras de control en Mantis son *WHILE* y *UNTIL*. El formato de *WHILE* es el siguiente:

```
WHILE condición
   sentencias
END
```

y la traducción a Cobol es:

```
LOOP-WHILE-N.
     IF NOT ( condición )
        GO TO FIN-WHILE-N.
     sentencias
     GO TO LOOP-WHILE-N.
FIN-WHILE-N.
```

El formato de *UNTIL* es el siguiente:

```
UNTIL condición
   sentencias
END
```

en donde sentencias se ejecuta al menos una vez. La traducción a Cobol es:

```
     GO TO SENTS-UNTIL-N.
COND-UNTIL-N.
     IF condición
        GO TO FIN-UNTIL-N.
SENTS-UNTIL-N.
     sentencias
     GO TO COND-UNTIL-N.
FIN-UNTIL-N.
```

Si un *WHILE* o un *UNTIL* no tiene instrucciones, el traductor muestra un *warning* indicando que puede producirse un *loop* infinito.

9. Procedimientos y sus parámetros. Los procedimientos de Mantis se traducen a Cobol como subrutinas identificadas con el nombre del procedimiento. Las llamadas a dichos procedimientos se traducen como instrucciones *PERFORM THRU*. Como Mantis es interpretado y crea el espacio para las variables en el momento de su definición, no existe realmente el concepto de variables locales. Por lo tanto, se adoptó la misma política para el programa generado en Cobol. Esto introduce limitaciones, ya que por ejemplo los programas en Mantis no pueden ser recursivos, porque no hay forma de almacenar y recuperar el "ambiente" de los procedimientos. El traductor detecta cuando un procedimiento es directamente

recursivo (se llama a sí mismo), pero no detecta la recursión indirecta.

En Mantis es posible definir parámetros para los procedimientos. Ya que en Cobol esto no es posible, es necesario crear una variable para cada parámetro formal del procedimiento. Para evitar duplicidades en los nombres de las variables, a los parámetros formales se les agrega un prefijo formado por el nombre del procedimiento al cual pertenecen. Estas variables (que sí son locales al procedimiento) tienen prioridad sobre las demás variables del programa dentro del procedimiento respectivo.

Cuando se ejecuta una llamada a un procedimiento se realizan los siguientes pasos:

i) Mover los parámetros actuales a los parámetros formales. Esto se hace porque puede que algún parámetro sea un valor de entrada al procedimiento.

ii) Efectuar la llamada al procedimiento a través de una instrucción *PERFORM THRU*.

iii) Mover los parámetros formales a los parámetros actuales, ya que es posible que el valor de algún

parámetro formal sea un resultado del procedimiento.

Cuando se define un procedimiento que no contiene instrucciones, el traductor produce un *warning* indicándolo.

10. Operaciones con archivos VSAM. El traductor permite traducir operaciones de archivos Secuenciales Indizados VSAM, que se procesen en modo *RANDOM*. Para definir los archivos a utilizar, el programa fuente en Mantis debe incluir uno o más comandos *ACCESS*. Para cada archivo VSAM que se utiliza debe existir un archivo que almacena la descripción del registro de datos. En él se especifican los campos que componen el registro, y se utiliza para definir el registro en la *FILE SECTION* del programa en Cobol. Este archivo debe estar definido en el directorio de archivos de FastMan. El formato para describir un registro es el siguiente:

Nombre, Tipo_Int, Pos, Tipo_Ext, Long, Signo, Decs, Llave

en donde:

Nombre: es el nombre del campo.

Tipo_Int: es el tipo que se dará al campo dentro de Mantis. Puede ser *BIG*, *SMALL* o *TEXT*.

Pos: es la posición (*byte*) en donde el campo comienza dentro del registro.

Tipo_Ext: es el formato como está almacenado el campo dentro del archivo. Puede ser *ZONED* o *PACKED*.

Long: es la longitud del campo (en *bytes*).

Signo: Indica si el campo tiene signo o no (sólo para campos numéricos). Puede ser Y o N.

Decs: Indica el número de posiciones decimales de un campo (sólo para campos numéricos).

Llave: Indica si el campo es una llave del archivo. Es posible tener una llave compuesta de varios campos pero estos deben ser consecutivos dentro del registro.

Por ejemplo, si un programa en Mantis tiene la instrucción:

```
ACCESS ARCHIVO_VSAM( "N_EXTERNO", "PASSWORD" )
```

en el directorio de archivos de FastMan debe existir un archivo llamado N_EXTERNO, el cual contiene una definición como la siguiente:

```
LLAVE_1,BIG,1,PACKED,5,Y,0,KEY
LLAVE_2,SMALL,6,ZONED,2,N,0,KEY
LLAVE_3,SMALL,8,ZONED,2,N,0,KEY
CAMPO_1,BIG,10,PACKED,5,Y,0
CAMPO_2,TEXT,15,TEXT,50,N,0
```

Cuando se procesa la definición de un archivo, todos los campos definidos en el registro están disponibles para ser utilizados por el programa. Si se utilizan varios archivos, puede que existan algunos campos que tengan el mismo nombre y tipo, pero que pertenezcan a diferentes registros. Este tipo de campos deben actualizarse automáticamente en los diferentes archivos cuando alguno sufre modificaciones. Dentro de FastMan estos campos comunes a diferentes archivos se llaman sinónimos. El manejo de los sinónimos se hace definiendo variables temporales que sirven como un *buffer*. Cuando se hace un *GET* a un registro de un archivo que contiene sinónimos, los valores de estos campos se mueven al *buffer*. Cuando se hace un *INSERT* o un *UPDATE* a un registro con sinónimos, los campos del *buffer* se mueven a los sinónimos del registro y después se efectúa la operación al archivo.

Las operaciones de Mantis para manejar los archivos se pueden trasladar directamente a sus equivalentes en Cobol. Para saber si una operación de archivos ha resultado exitosa o no, en Mantis se verifica una variable con el nombre del archivo o se hace por medio de la función *FSI* (*Function*

Status Indicator). Sin embargo, en Cobol este chequeo se hace por medio de la variable *Status-Key*. Por tal razón, inmediatamente después de efectuar una operación de archivos se ejecuta una función externa en Cobol que transforma el *Status-Key* a su equivalente *FSI*. Esta transformación depende de la operación efectuada.

11. Operaciones con vistas de SUPRA. Con FastMan también se pueden traducir programas en Mantis que utilicen vistas de datos de la base de datos SUPRA. Al igual que con los archivos VSAM, cada vista que se utiliza en un programa debe tener su definición de registro. Esto se hace para poder poner los campos del registro en disponibilidad al programa y para el manejo de los sinónimos. Las operaciones sobre vistas en Mantis se traducen a las respectivas instrucciones en RDM Cobol. Esto significa que el programa resultante en Cobol debe pasar previo a su compilación por un pre-compilador, el cual traduzca las operaciones de SUPRA a las respectivas llamadas a las funciones de RDM (*Relational Data Manager*).

Si el programa en Mantis efectúa operaciones con SUPRA, automaticamente se define la vista *TIS-CONTROL*, en la cual están ciertos indicadores y parámetros utilizados por RDM. La definición de las vistas dentro del programa en Cobol se hace por medio de la instrucción *INCLUDE*.

Para poder utilizar los valores de las funciones *FSI* y *VSI* (*Validity Status Indicator*) dentro del programa en Cobol, inmediatamente después de hacer una operacion sobre una vista de SUPRA, se hace una llamada a una rutina externa en Cobol la cual convierte los valores de *TIS-FSI* y *TIS-VSI* a los equivalentes de Mantis. Si se produce un error fatal en alguna operación sobre una vista, y el programa en Cobol ha sido generado con la modalidad de traducción *TRAP OFF*, la ejecución finaliza.

12. **Operaciones de entrada y salida.** Las operaciones de interacción con el usuario permitidas en FastMan son *SHOW*, *OBTAIN* y *WAIT*. Ya que los programas en Cobol son ejecutados en una partición *batch*, la comunicación que se establece con el operador del sistema es por medio de la consola. La instrucción *OBTAIN* genera instrucciones en Cobol para que los datos se ingresen de la consola del sistema. La instrucción *WAIT* suspende la ejecución del proceso y espera que se de una respuesta de continuación también desde la consola.

La instrucción *SHOW* por *default* despliega la información en la "impresora" del sistema. La salida puede

ser redireccionada también a la consola por medio de la instrucción *OUTPUT SCREEN*.

E. Funcionamiento del traductor.

FastMan fue desarrollado en lenguaje Turbo Pascal 5.0 en un computador personal. Por otro lado, el lenguaje Mantis generalmente es utilizado en *mainframes* o minicomputadores IBM. Actualmente se ha tomado la idea de descargar ciertos trabajos específicos de los *mainframes* y procesarlos externamente en computadores más pequeños. Esto se hace porque los *mainframes* no cuentan en la mayoría de los casos con herramientas tan versátiles como las que se encuentran, por ejemplo, en los computadores personales. Además, si dichas herramientas existieran en los *mainframes*, su desarrollo y mantenimiento sería demasiado costoso.

Para integrar el uso de Mantis y de FastMan se propone lo siguiente:

1.- Realizar el programa en Mantis en el *mainframe*. Se deben efectuar todas las pruebas necesarias para garantizar el buen funcionamiento del programa, incluso utilizando datos reales. Si existen errores, se deben corregir hasta que el programa esté totalmente correcto. Ya que este *debugging* se efectua "en

línea", es rápido y se puede obtener un buen tiempo de desarrollo.

2.- Transferir el programa en Mantis del *mainframe* al computador personal. Para esto se puede utilizar una tarjeta de comunicaciones como IRMA. FastMan incluye un programa utilitario encargado de la eliminación de los encabezados del programa en Mantis, eliminación de los números de línea y la conversión de algunos caracteres. Si el programa en Mantis utiliza archivos VSAM o vistas de SUPRA, deben también crearse las definiciones de los registros de dichos archivos en el directorio de FastMan.

3.- Utilizar FastMan para traducir el programa de Mantis a Cobol. Durante la traducción de un programa se pueden producir algunos *WARNINGS*, los cuales deben tomarse en cuenta para corregir el programa fuente si fuera necesario.

4.- Transferir el programa en Cobol del computador personal al *mainframe*. Para esto puede utilizarse también la tarjeta de comunicaciones IRMA.

5.- Compilar el programa en Cobol para generar el programa objeto.

Los pasos números 2, 3 y 4 pueden ser efectuados automaticamente desde el computador personal por medio de un archivo de ejecución *batch* (extensión .BAT). Esto facilita el procedimiento porque el programador no tiene que hacerse cargo de la transmisión de los archivos.

El archivo de entrada para el traductor debe ser un programa en Mantis con las siguientes características:

i) Las instrucciones no deben estar numeradas, ni deben contener los puntos de indentación utilizados por Mantis.

ii) No deben existir líneas en blanco.

iii) Puede utilizarse el signo : para escribir varias instrucciones en la misma línea.

Para traducir un programa en Mantis se ejecuta el programa FASTMAN.EXE indicando el archivo de entrada. Se asume que la extensión del archivo de entrada es .MAN. El programa en Cobol resultado de la traducción se almacena en un archivo con extensión .COB. Para efectuar la traducción, se pueden utilizar dos parámetros opcionales:

1.- Chequeo de rangos: el cual sirve para que dentro del programa en Cobol se efectúen chequeos de rangos de los subíndices para acceso de arreglos. El *default* es R-,

indicando que el chequeo de rangos está inactivo. Para activar el chequeo de rangos se utiliza el parámetro R+.

2.- Chequeo de errores fatales de archivos y vistas: que sirve para que el programa en Cobol suspenda su ejecución automáticamente cuando se detecta un error fatal en un archivo VSAM o una vista de SUPRA. Este parámetro es equivalente a la instrucción *TRAP* utilizada en Mantis. El *default* es T-, indicando que el programa finaliza en errores fatales. Para desactivar el chequeo de errores fatales se debe utilizar el parámetro T+, teniendo cuidado de verificar las condiciones de error desde el programa.

Por ejemplo, para traducir el programa llamado PRUEBA.MAN con la opción de chequeo de rangos se debe ingresar el siguiente comando en DOS:

```
C>FASTMAN PRUEBA R+
```

El resultado será un programa en Cobol llamado PRUEBA.COB.

VIII. RESULTADOS Y CONCLUSIONES

A. Resultados.

Para probar la eficiencia del código generado por FastMan, se compararon las velocidades de ejecución en Mantis y en Cobol de varios programas. Los programas en Mantis repetían un gran número de veces ciertas instrucciones y éstos se tradujeron a sus respectivos programas en Cobol. También, en algunos casos se elaboraron programas directamente en Cobol que realizan la misma función. Esto se hizo para determinar el *overhead* que se introduce al efectuar la traducción automática. Para establecer la comparación, se utilizan las siguientes variables:

t_M: tiempo de ejecución en segundos de un programa en Mantis.

t_C: tiempo de ejecución en segundos de un programa en Cobol escrito directamente por un programador.

t_F: tiempo de ejecución en segundos del programa equivalente en Cobol generado por FastMan.

r_{FM}: razón de eficiencia del programa generado por FastMan comparado con el programa original en Mantis. Se calcula como t_F / t_M y se espera que sea lo más pequeña posible.

r_{CF}: razón de eficiencia del programa escrito directamente en Cobol comparado con el programa generado por FastMan. Se calcula como t_C / t_F y se espera que sea lo más cercano posible a 1.

Los resultados obtenidos se muestran en la siguiente tabla:

Tabla 1

Comparación de tiempos de ejecución

Programa	t_M	t_C	t_F	r_{FM}	r_{CF}
Loop sin instrucciones	120	9	9	0.08	1.00
Expresión aritmética	53	32	32	0.60	1.00
Comparación numérica	27	2	2	0.07	1.00
Expresión booleana	38	3	3	0.08	1.00
Función trigonométrica	25	-	570	22.80	-
Suma de *strings*	40	-	31	0.78	-
Resta de *strings*	45	-	426	9.47	-
DO sin parámetros	8	-	3	0.38	-
DO con 3 parámetros	13	-	4	0.31	-
Lectura archivo VSAM	96	28	40	0.42	0.70
Grabación archivo VSAM	248	25	22	0.09	1.13
Lectura vistas de SUPRA	170	163	168	0.98	0.97
Grabación vistas de SUPRA	92	123	150	1.63	0.82

B. Discusión.

Tomando en cuenta los resultados obtenidos, se puede notar lo siguiente:

1.- Las expresiones aritméticas y booleanas son procesadas más rápido por el programa generado por FastMan. Se puede pensar que en el caso de las expresiones booleanas, el compilador de Cobol realiza alguna optimización de código ya que la diferencia de tiempos es muy grande. No existe diferencia entre el programa de FastMan y el programa escrito directamente en Cobol.

2.- Las funciones aritméticas y trigonométricas no son procesadas tan eficientemente como se quisiera, ya que las aproximaciones requieren muchas iteraciones. Para utilizar el traductor en una aplicación real que involucre estas funciones, se podría hacer lo siguiente:

i) Programar las funciones directamente en lenguaje *Assembly*, lo cual reduciría el tiempo de ejecución.

ii) Usar algoritmos más eficientes para calcular las aproximaciones. Un primer paso sería, por ejemplo, tomar en cuenta el residuo de las series

y detener el cálculo cuando se ha llegado a un determinado número de cifras significativas (lo cual podría depender de cada aplicación).

3.- Las operaciones y funciones con *strings* son también bastante costosas. Los métodos utilizados para implementar dichas operaciones no son los mejores y pueden ser optimizados para uso en aplicaciones reales. Al igual que las funciones aritméticas, las operaciones con *strings* deben codificarse en *Assembly*.

4.- Las operaciones con archivos VSAM son procesadas eficientemente por el programa generado por FastMan. El *overhead* introducido por el traductor es pequeño si se toma en cuenta la facilidad de programación en Mantis.

5.- Las operaciones con vistas de SUPRA no se ven afectadas significativamente por la traducción. Parece que el costo principal de estas operaciones radica en el manejador de la base de datos propiamente y no en el lenguaje que se utiliza. Posiblemente, una forma de incrementar la velocidad de ejecución del programa en Cobol sería traducir las operaciones de SUPRA a llamadas a las rutinas de PDM (*Physical Data Manager*),

en lugar de llamadas a rutinas de RDM como se hace actualmente.

C. Conclusiones.

A partir de los resultados obtenidos se puede concluir lo siguiente:

1.- En general, el programa en Cobol producido por el traductor se ejecuta más rápido que el programa equivalente en Mantis.

2.- El traductor genera un código eficiente comparado con un programa escrito en Cobol directamente por un programador.

3.- Debido a la eficiencia obtenida en la traducción, FastMan resulta ser una herramienta útil que transfiere las ventajas de programación de un 4GL al desarrollo de aplicaciones *batch*.

4.- El traductor de Mantis a Cobol homogeniza el ambiente de programación a un sólo lenguaje. Esto repercute en una mayor facilidad para el desarrollo de las aplicaciones y para el mantenimiento de las mismas.

5.- Por medio del traductor de Mantis a Cobol se puede aumentar el rendimiento de los programadores debido a que:

i) La programación en Mantis es mucho más sencilla y clara. Mantis ayuda a la estructuración de los programas y de esta forma se puede hacer más fácil el desarrollo y mantenimiento de los mismos.

ii) Las pruebas y *debugging* de los programas en Mantis son más rápidos debido a que se hacen "en línea".

iii) Mantis es un lenguaje que puede aprenderse facilmente.

Se espera que FastMan pueda ser utilizado comercialmente como una herramienta de programación. En esta fase se tomarán en cuenta las sugerencias que los usuarios realicen para mejorar el rendimiento del traductor. También se espera realizar las modificaciones a la gramática de FastMan y al traductor para permitir el uso de nuevas instrucciones que surgan en las siguientes versiones de Mantis.

IX. BIBLIOGRAFIA

Aho, A.; J. Ullman. *Principles of Compiler Design*. USA, 1979 Addison-Wesley Pubishing Company. 604 pp.

Clocksin, W.; C. Mellish. *Programming in Prolog*. USA, 1987 Editorial Springer-Verlag. 281 pp.

IBM VS COBOL for DOS/VSE Release 3.0. IBM Corporation, 1981 California USA. 456 pp.

Mantis Programming Reference Manual Release 4.2. Cincom 1986 Systems Inc., Cincinnati USA. 405 pp.

Naylor, T.; J. Balintfy, D. Burdick y K. Chu. 1975 Técnicas de Simulación en Computadoras. México, Editorial Limusa. 390 pp.

Philippakis, A.; L. Kazmier. *Advanced Cobol*. USA, 1982 Editorial McGraw-Hill. 611 pp.

Protter, M; C. Morrey. Cálculo con Geometría Analítica. 1980 3a. edición. México, Fondo Educativo Interamericano. 872 pp.

Relational Application Facilities. Cincom Systems Inc., 1985 Cincinnati USA. 208 pp.

Spiegel, M.; Manual de Fórmulas y Tablas Matemáticas. 1970 México, Editorial McGraw-Hill. 271 pp.

Supra RDM Cobol Programmer's Guide Release 1.0. Cincom 1985 Systems Inc., Cincinnati USA. 116 pp.

APENDICE A

Gramática de FastMan

La gramática aceptada por FastMan es la siguiente:

```
programa ::=
     ENTRY IDENTIFICADOR lista_sentencias EXIT
     declaración_subprogramas

declaración_subprogramas ::=
     declaración_módulo declaración_subprogramas
     | COMENTARIO declaración_subprogramas
     | ϵ

declaración_módulo ::=
     ENTRY definición_módulo lista_sentencias EXIT

definición_módulo ::=
     IDENTIFICADOR parámetros_formales

parámetros_formales ::=
     ( lista_identificadores )
     | ϵ

lista_identificadores ::=
     IDENTIFICADOR
     | lista_identificadores , IDENTIFICADOR

lista_sentencias ::=
     lista_sentencias sentencia
     | ϵ

lista_sentencias_restringidas ::=
     lista_sentencias_restringidas sentencia_restringida
     | ϵ

sentencia ::=
     sentencia_restringida
     | sentencia_when

sentencia_restringida ::=
     COMENTARIO
     | declaración
     | sentencia_elemental
     | sentencia_if_else
     | sentencia_until
     | sentencia_while
```

```
declaración ::=
     tipo lista_declaración

tipo ::=
     SMALL
     ¦ BIG
     ¦ TEXT

lista_declaración ::=
     definición
     ¦ lista_declaración , definición

definición ::=
     IDENTIFICADOR dimensión

dimensión ::=
     ( ENTERO )
     ¦ ϵ

sentencia_elemental ::=
     asignación
     ¦ LET asignación
     ¦ sentencia_subprograma
     ¦ comando

asignación ::=
     variable formato_rounded = expresión

variable ::=
     IDENTIFICADOR subindices

subindices ::=
     ( expresión , expresión )
     ¦ ( expresión )
     ¦ ϵ

formato_rounded ::=
     ROUNDED ( ENTERO )
     ¦ ϵ

expresión ::=
     término_booleano
     ¦ expresión OR término_booleano

término_booleano ::=
     factor_booleano
     ¦ término_booleano AND factor_booleano
```

```
factor_booleano ::=
     expresión_simple
     ¦ expresión_simple operador_relacional expresión_simple

operador_relacional ::=
     =
     ¦ <>
     ¦ >
     ¦ <
     ¦ >=
     ¦ <=

expresión_simple ::=
     término
     ¦ expresión_simple operador_suma término

operador_suma ::=
     +
     ¦ -

término ::=
     factor
     ¦ término operador_multiplicación factor

operador_multiplicación ::=
     *
     ¦ /

factor ::=
     primario
     ¦ primario operador_potenciación factor

operador_potenciación ::=
     **

primario ::=
     elemento
     ¦ signo elemento

signo ::=
     +
     ¦ -

elemento ::=
     ENTERO
     ¦ REAL
     ¦ STRING
     ¦ variable
     ¦ ( expresión )
     ¦ función
```

```
función ::=
      ABS ( expresión )
      ¦ ASI ( IDENTIFICADOR , IDENTIFICADOR )
      ¦ ATN ( expresión )
      ¦ COS ( expresión )
      ¦ DATE
      ¦ E
      ¦ EXP ( expresión )
      ¦ FALSE
      ¦ FSI formato_fsi
      ¦ INT ( expresión )
      ¦ LOG ( expresión )
      ¦ NOT ( expresión )
      ¦ PI
      ¦ POINT ( elemento operador_suma elemento )
      ¦ RND ( expresión )
      ¦ SGN ( expresión )
      ¦ SIN ( expresión )
      ¦ SIZE ( expresión_simple )
      ¦ SQR ( expresión )
      ¦ TAN ( expresión )
      ¦ TIME
      ¦ TRUE
      ¦ TXT ( expresión )
      ¦ VALUE ( expresión )
      ¦ VSI ( IDENTIFICADOR )
      ¦ ZERO

formato_fsi ::=
      ( IDENTIFICADOR , IDENTIFICADOR )
      ¦ ( IDENTIFICADOR )

sentencia_subprograma ::=
      DO IDENTIFICADOR parámetros_actuales

parámetros_actuales ::=
      ( lista_identificadores )
      ¦ ϵ

comando ::=
      ACCESS lista_access
      ¦ COMMIT
      ¦ DELETE IDENTIFICADOR forma_delete
      ¦ GET IDENTIFICADOR formato_get
      ¦ INSERT IDENTIFICADOR forma_insert
      ¦ MARK IDENTIFICADOR AT IDENTIFICADOR
      ¦ OBTAIN lista_identificadores
      ¦ OUTPUT formato_output
      ¦ PAD IDENTIFICADOR formato_pad
      ¦ RELEASE formato_release
      ¦ RESET
```

```
      | SEED
      | SHOW formato_show
      | STOP
      | UNPAD IDENTIFICADOR formato_pad
      | UPDATE IDENTIFICADOR
      | VIEW formato_view
      | WAIT

lista_access ::=
      definición_access
      | lista_access , definición_access

definición_access ::=
      IDENTIFICADOR ( nombre_externo , password )

nombre_externo ::=
      STRING

password ::=
      STRING

forma_delete ::=
      ALL
      | ϵ

formato_get ::=
      AT IDENTIFICADOR
      | llaves forma_get

llaves ::=
      ( lista_expresiones )
      | ϵ

lista_expresiones ::=
      expresión
      | lista_expresiones , expresión

forma_get ::=
      EQUAL
      | FIRST
      | LAST
      | NEXT
      | PRIOR
      | SAME
      | ϵ

forma_insert ::=
      FIRST
      | LAST
      | NEXT
      | PRIOR
```

```
      | ϵ

formato_output ::=
      SCREEN output_printer
      | PRINTER output_screen

output_printer ::=
      PRINTER
      | ϵ

output_screen ::=
      SCREEN
      | ϵ

formato_pad ::=
      expresión forma_pad
      | forma_pad

forma_pad ::=
      AFTER
      | ALL
      | BEFORE
      | ϵ

formato_release ::=
      IDENTIFICADOR
      | ϵ

formato_show ::=
      lista_expresiones
      | ϵ

formato_view ::=
      ON ( usuario )
      | ON ( usuario , password )
      | OFF
      | lista_views

usuario ::=
      STRING

lista_views ::=
      definición_view
      | lista_views , definición_view

definición_view ::=
      IDENTIFICADOR ( nombre_externo )

sentencia_if_else ::=
      IF expresión lista_sentencias formato_else
```

```
formato_else ::=
      END
      | ELSE lista_sentencias END

sentencia_until ::=
      UNTIL expresión lista_sentencias END

sentencia_while ::=
      WHILE expresión lista_sentencias END

sentencia_when ::=
      lista_when END

lista_when ::=
      formato_when
      | lista_when formato_when

formato_when ::=
      WHEN expresión lista_sentencias_restringidas
```

Definiciones Auxiliares.

IDENTIFICADOR = letra (letra ¦ dígito ¦ underline)*

ENTERO = dígito+

REAL = dígito+ punto dígito+

COMENTARIO = barra (letra ¦ dígito ¦ signo ¦ espacio)*

STRING = comilla (letra¦dígito¦signo¦espacio)* comilla

letra = A ¦ B ¦ C ¦ ... ¦ X ¦ Y ¦ Z

dígito = 0 ¦ 1 ¦ 2 ¦ 3 ¦ 4 ¦ 5 ¦ 6 ¦ 7 ¦ 8 ¦ 9

underline = _

punto = .

barra = ¦

espacio = un espacio en blanco.

comilla = "

signo = + ¦ - ¦ * ¦ / ¦ = ¦ > ¦ < ¦ (¦) ¦ : ¦ ; ¦ . ¦
, ¦ ? ¦ # ¦ $ ¦ % ¦ ¦ ¦ _ ¦ " ¦ ' ¦ [¦] ¦ \

APENDICE B

Ejemplos de traducción

A continuación se muestran varios ejemplos de traducción utilizando FastMan. Primero se describen en términos generales los objetivos de los programas y luego se presentan los listados en Mantis y en Cobol de los mismos.

Ejemplo # 1.

Este programa sirve para realizar un *bubblesort* en un arreglo de elementos generados aleatoriamente. La traducción se realizó sin chequeo de rangos (parámetro R-).

```
ENTRY EJEMPLO1
  ¦
  ¦*************************************************************
  ¦* OBJETIVO.....: GENERAR UN ARREGLO DE ELEMENTOS RANDOM     *
  ¦*                Y ORDENARLO UTILIZANDO BUBBLESORT.         *
  ¦* LENGUAJE.....: MANTIS 4.2.                                *
  ¦* FECHA.......: MARZO DE 1990.                             *
  ¦*************************************************************
  ¦
 SMALL I,J,T,N,A(200)
  ¦
 OUTPUT SCREEN
 SHOW "N=?"
 OBTAIN N
 OUTPUT PRINTER
  ¦
 IF N <= 200
  DO GENERAR
  DO SORT
  DO IMPRIMIR
 END
EXIT
```

```
'
'*************************************************************
'* GENERAR ALEATORIAMENTE UN ARREGLO DE N ELEMENTOS          *
'*************************************************************
'
ENTRY GENERAR
 SEED
 I=1
 WHILE I <= N
  A(I)=RND(100)
  I=I+1
 END
EXIT
'
'*************************************************************
'* ORDENAR EL ARREGLO POR BUBBLESORT                         *
'*************************************************************
'
ENTRY SORT
 J=N-1
 WHILE J >= 1
  I=1
  '
  WHILE I <= J
   IF A( I+1 ) < A( I )
    T=A( I )
    A( I )=A( I+1 )
    A( I+1 )=T
   END
   '
   I=I+1
  END
  '
  J=J-1
 END
EXIT
'
'*************************************************************
'* IMPRIMIR LOS ELEMENTOS DEL ARREGLO                        *
'*************************************************************
'
ENTRY IMPRIMIR
 I=1
 WHILE I <= N
  SHOW A( I )
  I=I+1
 END
EXIT
```

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.    EJEMPLO1-P.
 AUTHOR.        FASTMAN.
 DATE-WRITTEN. 20/3/1990; 18:43:24.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. IBM-4381.
 OBJECT-COMPUTER. IBM-4381.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01   A-V                          PIC S9(8)
           OCCURS 200 TIMES.
 01   I-V                          PIC S9(8) VALUE ZERO.
 01   SMALL2                       PIC S9(8) VALUE ZERO.
 01   J-V                          PIC S9(8) VALUE ZERO.
 01   TRUE                         PIC S9(8) VALUE +1.
 01   SMALL1                       PIC S9(8) VALUE ZERO.
 01   SUB1                         PIC S9(8) VALUE ZERO.
 01   PAR-BIG                      PIC S9(8)V9(8) VALUE ZERO.
 01   BIG1                         PIC S9(8)V9(8) VALUE ZERO.
 01   PAR-RNDI1                    PIC S9(8) VALUE ZERO.
 01   SUB-R1                       PIC S9(8) VALUE ZERO.
 01   PAR-RNDI                     PIC S9(8) VALUE +7.
 01   T-V                          PIC S9(8) VALUE ZERO.
 01   OUTPUT-SCREEN                PIC S9(8) VALUE ZERO.
 01   N-V                          PIC S9(8) VALUE ZERO.
 01   OUTPUT-PRINTER               PIC S9(8) VALUE +1.
 01   FALSE                        PIC S9(8) VALUE ZERO.
*
 PROCEDURE DIVISION.
 EJEMPLO1-P.
*
*****************************************************************
** OBJETIVO.....: GENERAR UN ARREGLO DE ELEMENTOS RANDOM       *
**                Y ORDENARLO UTILIZANDO BUBBLESORT.           *
** LENGUAJE.....: MANTIS 4.2.                                  *
** FECHA........: MARZO DE 1990.                               *
*****************************************************************
*
*
      MOVE FALSE TO OUTPUT-PRINTER.
      MOVE TRUE  TO OUTPUT-SCREEN.
      IF ( OUTPUT-PRINTER = TRUE )
         DISPLAY 'N=?'.
      IF ( OUTPUT-SCREEN = TRUE )
         DISPLAY 'N=?' UPON CONSOLE.
      ACCEPT N-V FROM CONSOLE.
      MOVE TRUE  TO OUTPUT-PRINTER.
      MOVE FALSE TO OUTPUT-SCREEN.
```

```
*
      IF NOT  ( N-V NOT > 200 )
         GO TO ELSE-IF-1.
      PERFORM GENERAR-P THRU GENERAR-P-EXIT.
      PERFORM SORT-P THRU SORT-P-EXIT.
      PERFORM IMPRIMIR-P THRU IMPRIMIR-P-EXIT.
 ELSE-IF-1.
*
 FIN-EJEMPLO1-P.
      STOP RUN.
*
*
*****************************************************************
** GENERAR ALEATORIAMENTE UN ARREGLO DE N ELEMENTOS             *
*****************************************************************
*
 GENERAR-P.
      CALL 'MMOSEED'
           USING PAR-RNDI.
      COMPUTE I-V =  1.
 LOOP-WHILE-2.
      IF NOT  ( I-V NOT > N-V )
         GO TO FIN-WHILE-3.
      COMPUTE PAR-BIG =  100.
      CALL 'MMORND'
           USING PAR-RNDI,
                 PAR-RNDI1,
                 PAR-BIG,
                 BIG1.
      COMPUTE SUB1 =  I-V.
      COMPUTE A-V( SUB1 ) =  BIG1.
      COMPUTE I-V =  I-V + 1.
      GO TO LOOP-WHILE-2.
 FIN-WHILE-3.
 GENERAR-P-EXIT.  EXIT.
*
*
*****************************************************************
** ORDENAR EL ARREGLO POR BUBBLESORT                            *
*****************************************************************
*
 SORT-P.
      COMPUTE J-V =  N-V - 1.
 LOOP-WHILE-4.
      IF NOT  ( J-V NOT < 1 )
         GO TO FIN-WHILE-5.
      COMPUTE I-V =  1.
*
 LOOP-WHILE-6.
      IF NOT  ( I-V NOT > J-V )
         GO TO FIN-WHILE-7.
```

```
           COMPUTE SMALL1 =  I-V + 1.
           COMPUTE SMALL2 =  I-V.
           IF NOT  ( A-V( SMALL1 ) < A-V( SMALL2 ) )
              GO TO ELSE-IF-8.
           COMPUTE SMALL1 =  I-V.
           COMPUTE T-V =  A-V( SMALL1 ).
           COMPUTE SMALL1 =  I-V + 1.
           COMPUTE SUB1 =  I-V.
           COMPUTE A-V( SUB1 ) =  A-V( SMALL1 ).
           COMPUTE SUB1 =  I-V + 1.
           COMPUTE A-V( SUB1 ) =  T-V.
       ELSE-IF-8.
      *
           COMPUTE I-V =  I-V + 1.
           GO TO LOOP-WHILE-6.
       FIN-WHILE-7.
      *
           COMPUTE J-V =  J-V - 1.
           GO TO LOOP-WHILE-4.
       FIN-WHILE-5.
       SORT-P-EXIT.  EXIT.
      *
      *
      ****************************************************************
      ** IMPRIMIR LOS ELEMENTOS DEL ARREGLO                          *
      ****************************************************************
      *
       IMPRIMIR-P.
           COMPUTE I-V =  1.
       LOOP-WHILE-9.
           IF NOT  ( I-V NOT > N-V )
              GO TO FIN-WHILE-10.
           COMPUTE SMALL1 =  I-V.
           IF ( OUTPUT-PRINTER = TRUE )
              DISPLAY  A-V( SMALL1 ).
           IF ( OUTPUT-SCREEN = TRUE )
              DISPLAY  A-V( SMALL1 ) UPON CONSOLE.
           COMPUTE I-V =  I-V + 1.
           GO TO LOOP-WHILE-9.
       FIN-WHILE-10.
       IMPRIMIR-P-EXIT.  EXIT.
      *
```

Ejemplo # 2.

El objetivo de este programa es aproximar el valor de una integral utilizando el método de Simpson.

```
ENTRY EJEMPLO2
 !
 !*************************************************************
 !* OBJETIVO.....: APROXIMAR EL VALOR DE UNA INTEGRAL POR     *
 !*                MEDIO DE LA REGLA DE SIMPSON.              *
 !* LENGUAJE.....: MANTIS 4.2.                                *
 !* FECHA........: MARZO DE 1990.                             *
 !*************************************************************
 !
 BIG A,B,H,XI,XIO,XI1,XI2,X,FX,FA,FB
 SMALL I,M
 !
 OUTPUT SCREEN
 SHOW "INGRESE A,B,M:"
 OUTPUT PRINTER
 OBTAIN A,B,M
 !
 H=(B-A)/(2*M)
 DO FUNCION(A,FA)
 DO FUNCION(A,FB)
 XIO=FA+FB
 XI1=ZERO
 XI2=ZERO
 I=1
 !
 WHILE I <= 2*M-1
  X=A+I*H
  DO FUNCION(X,FX)
  !
  IF I/2=INT(I/2)
   XI2 ROUNDED(6)=XI2+FX
  ELSE
   XI1 ROUNDED(6)=XI1+FX
  END
  !
  I=I+1
 END
 !
 XI=H*(XIO + 2*XI2 + 4*XI1 )/3
 SHOW "APROXIMACION=", XI
EXIT
```

```
      ¦
      ¦*****************************************************************
      ¦* FUNCION DE LA CUAL SE CALCULA LA INTEGRAL.                    *
      ¦*****************************************************************
      ¦
      ENTRY FUNCION(Y,FY)
       FY=EXP(Y)
      EXIT



       IDENTIFICATION DIVISION.
       PROGRAM-ID.    EJEMPLO2-P.
       AUTHOR.        FASTMAN.
       DATE-WRITTEN. 22/3/1990; 17:12:19.
      *
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER. IBM-4381.
       OBJECT-COMPUTER. IBM-4381.
      *
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01   FA-V                          PIC S9(8)V9(8) VALUE ZERO.
       01   XI2-V                         PIC S9(8)V9(8) VALUE ZERO.
       01   H-V                           PIC S9(8)V9(8) VALUE ZERO.
       01   A-V                           PIC S9(8)V9(8) VALUE ZERO.
       01   FUNCION-P-Y-V                 PIC S9(8)V9(8) VALUE ZERO.
       01   I-V                           PIC S9(8) VALUE ZERO.
       01   XI0-V                         PIC S9(8)V9(8) VALUE ZERO.
       01   B-V                           PIC S9(8)V9(8) VALUE ZERO.
       01   SMALL2                        PIC S9(8) VALUE ZERO.
       01   FB-V                          PIC S9(8)V9(8) VALUE ZERO.
       01   X-V                           PIC S9(8)V9(8) VALUE ZERO.
       01   TRUE                          PIC S9(8) VALUE +1.
       01   RES-BIG                       PIC S9(8)V9(8) VALUE ZERO.
       01   SMALL1                        PIC S9(8) VALUE ZERO.
       01   FUNCION-P-FY-V-D              PIC S9(8) VALUE ZERO.
       01   PAR-BIG                       PIC S9(8)V9(8) VALUE ZERO.
       01   FUNCION-P-FY-V                PIC S9(8)V9(8) VALUE ZERO.
       01   FX-V                          PIC S9(8)V9(8) VALUE ZERO.
       01   XI1-V                         PIC S9(8)V9(8) VALUE ZERO.
       01   BIG1                          PIC S9(8)V9(8) VALUE ZERO.
       01   PAR-SMALL                     PIC S9(8) VALUE ZERO.
       01   M-V                           PIC S9(8) VALUE ZERO.
       01   OUTPUT-SCREEN                 PIC S9(8) VALUE ZERO.
       01   FUNCION-P-Y-V-D               PIC S9(8) VALUE ZERO.
       01   XI-V                          PIC S9(8)V9(8) VALUE ZERO.
       01   OUTPUT-PRINTER                PIC S9(8) VALUE +1.
       01   FALSE                         PIC S9(8) VALUE ZERO.
      *
```

```
 PROCEDURE DIVISION.
 EJEMPLO2-P.
*
*****************************************************************
** OBJETIVO.....: APROXIMAR EL VALOR DE UNA INTEGRAL POR       *
**                MEDIO DE LA REGLA DE SIMPSON.                *
** LENGUAJE.....: MANTIS 4.2.                                  *
** FECHA........: MARZO DE 1990.                               *
*****************************************************************
*
*
      MOVE FALSE TO OUTPUT-PRINTER.
      MOVE TRUE  TO OUTPUT-SCREEN.
      IF ( OUTPUT-PRINTER = TRUE )
         DISPLAY 'INGRESE A,B,M:'.
      IF ( OUTPUT-SCREEN = TRUE )
         DISPLAY 'INGRESE A,B,M:' UPON CONSOLE.
      MOVE TRUE  TO OUTPUT-PRINTER.
      MOVE FALSE TO OUTPUT-SCREEN.
      ACCEPT A-V FROM CONSOLE.
      ACCEPT B-V FROM CONSOLE.
      ACCEPT M-V FROM CONSOLE.
*
      COMPUTE H-V =  ( B-V - A-V ) / ( 2 * M-V ).
      MOVE A-V TO FUNCION-P-Y-V.
      MOVE FA-V TO FUNCION-P-FY-V.
      PERFORM FUNCION-P THRU FUNCION-P-EXIT.
      MOVE FUNCION-P-Y-V TO A-V.
      MOVE FUNCION-P-FY-V TO FA-V.
      MOVE A-V TO FUNCION-P-Y-V.
      MOVE FB-V TO FUNCION-P-FY-V.
      PERFORM FUNCION-P THRU FUNCION-P-EXIT.
      MOVE FUNCION-P-Y-V TO A-V.
      MOVE FUNCION-P-FY-V TO FB-V.
      COMPUTE XI0-V =  FA-V + FB-V.
      COMPUTE XI1-V =  ZERO.
      COMPUTE XI2-V =  ZERO.
      COMPUTE I-V =  1.
*
 LOOP-WHILE-1.
      COMPUTE SMALL1 =  2 * M-V - 1.
      IF NOT  ( I-V NOT > SMALL1 )
         GO TO FIN-WHILE-2.
      COMPUTE X-V =  A-V + I-V * H-V.
      MOVE X-V TO FUNCION-P-Y-V.
      MOVE FX-V TO FUNCION-P-FY-V.
      PERFORM FUNCION-P THRU FUNCION-P-EXIT.
      MOVE FUNCION-P-Y-V TO X-V.
      MOVE FUNCION-P-FY-V TO FX-V.
*
      COMPUTE PAR-BIG =  I-V / 2.
```

```
      CALL 'MMOINT'
           USING PAR-BIG,
                 SMALL1.
      COMPUTE SMALL2 =  I-V / 2.
      IF NOT  ( SMALL2 = SMALL1 )
         GO TO ELSE-IF-3.
      COMPUTE PAR-BIG =  XI2-V + FX-V.
      MOVE 6 TO PAR-SMALL.
      CALL 'MMOROU'
           USING PAR-BIG,
                 PAR-SMALL,
                 RES-BIG.
      MOVE RES-BIG TO XI2-V.
      GO TO FIN-IF-4.
 ELSE-IF-3.
      COMPUTE PAR-BIG =  XI1-V + FX-V.
      MOVE 6 TO PAR-SMALL.
      CALL 'MMOROU'
           USING PAR-BIG,
                 PAR-SMALL,
                 RES-BIG.
      MOVE RES-BIG TO XI1-V.
 FIN-IF-4.
*
      COMPUTE I-V =  I-V + 1.
      GO TO LOOP-WHILE-1.
 FIN-WHILE-2.
*
      COMPUTE XI-V = H-V * ( XIO-V + 2 * XI2-V + 4 * XI1-V )
                     / 3.
      IF ( OUTPUT-PRINTER = TRUE )
         DISPLAY 'APROXIMACION='
                  XI-V.
      IF ( OUTPUT-SCREEN = TRUE )
         DISPLAY 'APROXIMACION='
                  XI-V UPON CONSOLE.
*
 FIN-EJEMPLO2-P.
      STOP RUN.
*
**************************************************************
** FUNCION DE LA CUAL SE CALCULA LA INTEGRAL.                *
**************************************************************
*
 FUNCION-P.
      COMPUTE PAR-BIG =  FUNCION-P-Y-V.
      CALL 'MMOEXP'
           USING PAR-BIG,
                 BIG1.
      COMPUTE FUNCION-P-FY-V =  BIG1.
 FUNCION-P-EXIT.  EXIT.
```

Ejemplo # 3.

Este programa sirve para leer un archivo VSAM y determinar la cantidad de registros de cada tipo (A, B, C o D) que existen en él.

```
ENTRY EJEMPLO3
 !
 !*************************************************************
 !* OBJETIVO......: LECTURA DE ARCHIVO VSAM                   *
 !* LENGUAJE......: MANTIS 4.2.                               *
 !* FECHA.........: 20 DE MARZO DE 1990                       *
 !*************************************************************
 !
 ACCESS ARCHIVO("N_VSAM","LEER")
 BIG LINEAS, CONTADOR, CONTADOR_A1, CONTADOR_A2
 BIG INMUEBLES_A1, INMUEBLES_A2
 !
 LINEAS=ZERO
 CONTADOR=ZERO
 CONTADOR_A1=ZERO
 CONTADOR_A2=ZERO
 INMUEBLES_A1=ZERO
 INMUEBLES_A2=ZERO
 !
 GET ARCHIVO FIRST
 WHILE ARCHIVO <> "END"
  LINEAS=LINEAS+1
  CONTADOR=CONTADOR+1
  !
  WHEN TIPO_PAN="A"
   CONTADOR_A1=CONTADOR_A1+1
  WHEN TIPO_PAN="B"
   INMUEBLES_A1=INMUEBLES_A1+1
  WHEN TIPO_PAN="C"
   CONTADOR_A2=CONTADOR_A2+1
  WHEN TIPO_PAN="D"
   INMUEBLES_A2=INMUEBLES_A2+1
  END
  !
  GET ARCHIVO NEXT
 END
 !
 SHOW "REGISTROS.....", CONTADOR
 SHOW "AUTOAVALUOS A-1.....", CONTADOR_A1
 SHOW "AUTOAVALUOS A-2.....", CONTADOR_A2
 SHOW "INMUEBLES A-1.......", INMUEBLES_A1
```

```
 SHOW "INMUEBLES A-2.......", INMUEBLES_A2
EXIT


 IDENTIFICATION DIVISION.
 PROGRAM-ID.    EJEMPLO3-P.
 AUTHOR.        FASTMAN.
 DATE-WRITTEN. 20/3/1990; 18:16:14.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. IBM-4381.
 OBJECT-COMPUTER. IBM-4381.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT ARCHIVO-F ASSIGN TO SYS-N-VSAM,
            ORGANIZATION IS INDEXED,
            ACCESS MODE  IS DYNAMIC,
            RECORD KEY   IS ARCHIVO-F-KEY,
            FILE STATUS  IS ARCHIVO-F-SK.
*
 DATA DIVISION.
 FILE SECTION.
 FD  ARCHIVO-F
        LABEL RECORDS ARE OMITTED
        DATA RECORD IS ARCHIVO-F-RECORD.
 01  ARCHIVO-F-RECORD.
     02 ARCHIVO-F-KEY.
        03 AUTOAVALUO-PAN                    PIC 9(7).
        03 ORDEN-PAN                         PIC 9(4).
     02 TIPO-PAN                          PIC X(4).
     02 CORRELATIVO-PAN                   PIC 9(7).
     02 DATOS-PAN                         PIC X(7).
*
 WORKING-STORAGE SECTION.
 01  ARCHIVO-F-FSI              PIC X(16) VALUE SPACES.
 01  ARCHIVO-F-FSI-L            PIC S9(8) VALUE ZERO.
 01  OPERACION                  PIC X(16) VALUE SPACES.
 01  OPERACION-L                PIC S9(8) VALUE ZERO.
 01  CONTADOR-A2-V              PIC S9(8)V9(8) VALUE ZERO.
 01  ARCHIVO-F-SK               PIC X(2) VALUE SPACES.
 01  ARCHIVO-F-SK-L             PIC S9(8) VALUE ZERO.
 01  ARCHIVO-F-MSG-FSI          PIC X(40) VALUE SPACES.
 01  ARCHIVO-F-MSG-FSI-L        PIC S9(8) VALUE ZERO.
 01  ARCHIVO-F-V                PIC X(16) VALUE SPACES.
 01  ARCHIVO-F-V-L              PIC S9(8) VALUE ZERO.
 01  ERROR-FATAL                PIC S9(8) VALUE ZERO.
 01  TRUE                       PIC S9(8) VALUE +1.
 01  SMALL1                     PIC S9(8) VALUE ZERO.
 01  INMUEBLES-A2-V             PIC S9(8)V9(8) VALUE ZERO.
```

```
 01  LINEAS-V                   PIC S9(8)V9(8) VALUE ZERO.
 01  CONTADOR-A1-V              PIC S9(8)V9(8) VALUE ZERO.
 01  DATOS-PAN-L                PIC S9(8) VALUE ZERO.
 01  TIPO-PAN-L                 PIC S9(8) VALUE ZERO.
 01  INMUEBLES-A1-V             PIC S9(8)V9(8) VALUE ZERO.
 01  CONTADOR-V                 PIC S9(8)V9(8) VALUE ZERO.
 01  OUTPUT-SCREEN              PIC S9(8) VALUE ZERO.
 01  OUTPUT-PRINTER             PIC S9(8) VALUE +1.
 01  FALSE                      PIC S9(8) VALUE ZERO.
*
 PROCEDURE DIVISION.
 EJEMPLO3-P.
     OPEN I-O ARCHIVO-F.
*
*****************************************************************
** OBJETIVO......: LECTURA DE ARCHIVO VSAM                      *
** LENGUAJE......: MANTIS 4.2.                                  *
** FECHA.........: 20 DE MARZO DE 1990                          *
*****************************************************************
*
*
     COMPUTE LINEAS-V =  ZERO.
     COMPUTE CONTADOR-V =  ZERO.
     COMPUTE CONTADOR-A1-V =  ZERO.
     COMPUTE CONTADOR-A2-V =  ZERO.
     COMPUTE INMUEBLES-A1-V =  ZERO.
     COMPUTE INMUEBLES-A2-V =  ZERO.
*
     READ ARCHIVO-F RECORD.
     MOVE 'READ' TO OPERACION.
     CALL 'MMOMACC'
          USING OPERACION,
                ARCHIVO-F-SK,
                ARCHIVO-F-V, ARCHIVO-F-V-L,
                ARCHIVO-F-FSI, ARCHIVO-F-FSI-L,
                ARCHIVO-F-MSG-FSI, ARCHIVO-F-MSG-FSI-L,
                ERROR-FATAL.
     IF ERROR-FATAL = TRUE
        PERFORM FATAL-ON-ARCHIVO-F.
     MOVE 1 TO TIPO-PAN-L.
     MOVE 243 TO DATOS-PAN-L.
 LOOP-WHILE-1.
     IF ( ARCHIVO-F-V-L = 3 )
        IF (  ARCHIVO-F-V NOT = 'END' )
             MOVE TRUE  TO SMALL1
        ELSE MOVE FALSE TO SMALL1
     ELSE
        MOVE TRUE  TO SMALL1.
     IF NOT  ( SMALL1 NOT = FALSE )
        GO TO FIN-WHILE-2.
     COMPUTE LINEAS-V =  LINEAS-V + 1.
```

```
      COMPUTE CONTADOR-V =  CONTADOR-V + 1.
*
      IF ( TIPO-PAN-L = 1 )
         IF (  TIPO-PAN OF ARCHIVO-F-RECORD = 'A' )
              MOVE TRUE  TO SMALL1
         ELSE MOVE FALSE TO SMALL1
      ELSE
         MOVE FALSE TO SMALL1.
      IF NOT  ( SMALL1 NOT = FALSE )
         GO TO ELSE-WHEN-3.
      COMPUTE CONTADOR-A1-V =  CONTADOR-A1-V + 1.
 ELSE-WHEN-3.
      IF ( TIPO-PAN-L = 1 )
         IF (  TIPO-PAN OF ARCHIVO-F-RECORD = 'B' )
              MOVE TRUE  TO SMALL1
         ELSE MOVE FALSE TO SMALL1
      ELSE
         MOVE FALSE TO SMALL1.
      IF NOT  ( SMALL1 NOT = FALSE )
         GO TO ELSE-WHEN-4.
      COMPUTE INMUEBLES-A1-V =  INMUEBLES-A1-V + 1.
 ELSE-WHEN-4.
      IF ( TIPO-PAN-L = 1 )
         IF (  TIPO-PAN OF ARCHIVO-F-RECORD = 'C' )
              MOVE TRUE  TO SMALL1
         ELSE MOVE FALSE TO SMALL1
      ELSE
         MOVE FALSE TO SMALL1.
      IF NOT  ( SMALL1 NOT = FALSE )
         GO TO ELSE-WHEN-5.
      COMPUTE CONTADOR-A2-V =  CONTADOR-A2-V + 1.
 ELSE-WHEN-5.
      IF ( TIPO-PAN-L = 1 )
         IF (  TIPO-PAN OF ARCHIVO-F-RECORD = 'D' )
              MOVE TRUE  TO SMALL1
         ELSE MOVE FALSE TO SMALL1
      ELSE
         MOVE FALSE TO SMALL1.
      IF NOT  ( SMALL1 NOT = FALSE )
         GO TO ELSE-WHEN-6.
      COMPUTE INMUEBLES-A2-V =  INMUEBLES-A2-V + 1.
 ELSE-WHEN-6.
*
      READ ARCHIVO-F RECORD.
      MOVE 'READ' TO OPERACION.
      CALL 'MMOMACC'
           USING OPERACION,
                 ARCHIVO-F-SK,
                 ARCHIVO-F-V, ARCHIVO-F-V-L,
                 ARCHIVO-F-FSI, ARCHIVO-F-FSI-L,
                 ARCHIVO-F-MSG-FSI, ARCHIVO-F-MSG-FSI-L,
```

```
                ERROR-FATAL.
     IF ERROR-FATAL = TRUE
        PERFORM FATAL-ON-ARCHIVO-F.
     MOVE 1 TO TIPO-PAN-L.
     MOVE 243 TO DATOS-PAN-L.
     GO TO LOOP-WHILE-1.
 FIN-WHILE-2.
*
     IF ( OUTPUT-PRINTER = TRUE )
        DISPLAY 'REGISTROS..........'
                CONTADOR-V.
     IF ( OUTPUT-SCREEN = TRUE )
        DISPLAY 'REGISTROS..........'
                CONTADOR-V UPON CONSOLE.
     IF ( OUTPUT-PRINTER = TRUE )
        DISPLAY 'AUTOAVALUOS A-1.....'
                CONTADOR-A1-V.
     IF ( OUTPUT-SCREEN = TRUE )
        DISPLAY 'AUTOAVALUOS A-1.....'
                CONTADOR-A1-V UPON CONSOLE.
     IF ( OUTPUT-PRINTER = TRUE )
        DISPLAY 'AUTOAVALUOS A-2.....'
                CONTADOR-A2-V.
     IF ( OUTPUT-SCREEN = TRUE )
        DISPLAY 'AUTOAVALUOS A-2.....'
                CONTADOR-A2-V UPON CONSOLE.
     IF ( OUTPUT-PRINTER = TRUE )
        DISPLAY 'INMUEBLES A-1.......'
                INMUEBLES-A1-V.
     IF ( OUTPUT-SCREEN = TRUE )
        DISPLAY 'INMUEBLES A-1.......'
                INMUEBLES-A1-V UPON CONSOLE.
     IF ( OUTPUT-PRINTER = TRUE )
        DISPLAY 'INMUEBLES A-2.......'
                INMUEBLES-A2-V.
     IF ( OUTPUT-SCREEN = TRUE )
        DISPLAY 'INMUEBLES A-2.......'
                INMUEBLES-A2-V UPON CONSOLE.
*
 FIN-EJEMPLO3-P.
     CLOSE ARCHIVO-F.
     STOP RUN.
*
 FATAL-ON-ARCHIVO-F.
     DISPLAY 'FATAL ERROR'.
     DISPLAY 'ACCESS     =ARCHIVO-F'.
     DISPLAY 'OPERACION =' OPERACION.
     DISPLAY 'STATUS-KEY=' ARCHIVO-F-SK.
     DISPLAY 'FSI        =' ARCHIVO-F-FSI.
     GO TO FIN-EJEMPLO3-P.
*
```

Ejemplo # 4.

El cuarto programa utiliza vistas de una base de datos en SUPRA y sirve para determinar el número de personas cuya suma de inmuebles es mayor que Q. 20,000.00.

```
ENTRY EJEMPLO4
 ¦
 ¦*************************************************************
 ¦* OBJETIVO : CONTAR LAS PERSONAS CON CAPITAL > 20000.00     *
 ¦* LENGUAJE : MANTIS 4.2 / SUPRA.                            *
 ¦* FECHA    : 24 DE FEBRERO DE 1990.                        *
 ¦*************************************************************
 ¦
 VIEW ON("USUARIO","PASSWORD")
 VIEW CUENTAS("RL-NITS")
 VIEW INMUEBLES("DV-NIT-INMUEBLES")
 ¦
 SMALL CONTADOR, C_GRANDES, C_PEQUENIOS
 BIG CAPITAL
 CONTADOR=0
 C_GRANDES=0
 C_PEQUENIOS=0
 ¦
 GET CUENTAS FIRST
 WHILE CUENTAS = "FOUND"
  CONTADOR = CONTADOR+1
  DO CALCULAR( DICABI, CAPITAL )
  IF CAPITAL > 20000
   C_GRANDES = C_GRANDES+1
  ELSE
   C_PEQUENIOS = C_PEQUENIOS+1
  END
  ¦
  GET CUENTAS NEXT
 END
 ¦
 SHOW "CUENTAS LEIDAS.............", CONTADOR
 SHOW "GRANDES CONTRIBUYENTES.....", C_GRANDES
 SHOW "PEQUENIOS CONTRIBUYENTES...", C_PEQUENIOS
 VIEW OFF
EXIT
¦
¦*************************************************************
¦* CALCULAR EL CAPITAL DE UNA CUENTA.                        *
¦*************************************************************
¦
```

```
ENTRY CALCULAR( CUENTA, SUMATORIA )
 SUMATORIA=0
 GET INMUEBLES( CUENTA ) FIRST
 ;
 WHILE INMUEBLES = "FOUND"
  SUMATORIA=SUMATORIA + I_INMUEBLE
  ;
  GET INMUEBLES( CUENTA ) NEXT
 END
EXIT
```

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.    EJEMPLO4-P.
 AUTHOR.        FASTMAN.
 DATE-WRITTEN. 24/2/1990; 19:29:3.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. IBM-4381.
 OBJECT-COMPUTER. IBM-4381.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01   CONST1                      PIC X(16) VALUE
      'USUARIO'.
 01   CONST1-L                    PIC S9(8) VALUE ZERO.
 01   C-PEQUENIOS-V               PIC S9(8) VALUE ZERO.
 01   C-GRANDES-V                 PIC S9(8) VALUE ZERO.
 01   TIS-CONTROL-FSI             PIC X(16) VALUE SPACES.
 01   TIS-CONTROL-FSI-L           PIC S9(8) VALUE ZERO.
 01   OPERACION                   PIC X(16) VALUE SPACES.
 01   OPERACION-L                 PIC S9(8) VALUE ZERO.
 01   DESCRIPCION-DIRECCION-L     PIC S9(8) VALUE ZERO.
 01   C-DEPTO-V                   PIC S9(8) VALUE ZERO.
 01   CALCULAR-P-CUENTA-V         PIC S9(8)V9(8) VALUE ZERO.
 01   INMUEBLES-F-VSI             PIC X(16) VALUE SPACES.
 01   INMUEBLES-F-VSI-L           PIC S9(8) VALUE ZERO.
 01   NOMBRE-PERSONA-L            PIC S9(8) VALUE ZERO.
 01   C-MUNIC-V                   PIC S9(8) VALUE ZERO.
 01   CUENTAS-F-FSI               PIC X(16) VALUE SPACES.
 01   CUENTAS-F-FSI-L             PIC S9(8) VALUE ZERO.
 01   ERROR-FATAL                 PIC S9(8) VALUE ZERO.
 01   TRUE                        PIC S9(8) VALUE +1.
 01   SMALL1                      PIC S9(8) VALUE ZERO.
 01   N-INMUEBLE-L                PIC S9(8) VALUE ZERO.
 01   CUENTAS-F-VSI               PIC X(16) VALUE SPACES.
 01   CUENTAS-F-VSI-L             PIC S9(8) VALUE ZERO.
 01   M-TIS-CONTROL               PIC X(16) VALUE SPACES.
 01   M-TIS-CONTROL-L             PIC S9(8) VALUE ZERO.
```

```
 01   INMUEBLES-F-FSI              PIC X(16) VALUE SPACES.
 01   INMUEBLES-F-FSI-L            PIC S9(8) VALUE ZERO.
 01   CUENTAS-F-MSG-FSI            PIC X(40) VALUE SPACES.
 01   CUENTAS-F-MSG-FSI-L          PIC S9(8) VALUE ZERO.
 01   CUENTAS-F-V                  PIC X(16) VALUE SPACES.
 01   CUENTAS-F-V-L                PIC S9(8) VALUE ZERO.
 01   CALCULAR-P-SUMATORIA-V       PIC S9(8)V9(8) VALUE ZERO.
 01   CALCULAR-P-SUMATORIA-V-D     PIC S9(8) VALUE ZERO.
 01   CONTADOR-V                   PIC S9(8) VALUE ZERO.
 01   INMUEBLES-F-MSG-FSI          PIC X(40) VALUE SPACES.
 01   INMUEBLES-F-MSG-FSI-L        PIC S9(8) VALUE ZERO.
 01   TIS-CONTROL-VSI              PIC X(16) VALUE SPACES.
 01   TIS-CONTROL-VSI-L            PIC S9(8) VALUE ZERO.
 01   OUTPUT-SCREEN                PIC S9(8) VALUE ZERO.
 01   CONST2                       PIC X(16) VALUE
      'PASSWORD'.
 01   CONST2-L                     PIC S9(8) VALUE ZERO.
 01   CALCULAR-P-CUENTA-V-D        PIC S9(8) VALUE ZERO.
 01   CAPITAL-V                    PIC S9(8)V9(8) VALUE ZERO.
 01   INMUEBLES-F-V                PIC X(16) VALUE SPACES.
 01   INMUEBLES-F-V-L              PIC S9(8) VALUE ZERO.
 01   OUTPUT-PRINTER               PIC S9(8) VALUE +1.
 01   FALSE                        PIC S9(8) VALUE ZERO.
*
 01   INCLUDE TIS-CONTROL.
 01   INMUEBLES-F INCLUDE DV-NIT-INMUEBLES.
 01   CUENTAS-F INCLUDE RL-NITS.
*
 PROCEDURE DIVISION.
 EJEMPL04-P.
*
*****************************************************************
** OBJETIVO : CONTAR LAS PERSONAS CON CAPITAL > 20000.00        *
** LENGUAJE : MANTIS 4.2 / SUPRA.                               *
** FECHA    : 24 DE FEBRERO DE 1990.                            *
*****************************************************************
*
      MOVE 'SIGN-ON' TO OPERACION.
      SIGN-ON CONST1 CONST2.
*
      COMPUTE CONTADOR-V =  0.
      COMPUTE C-GRANDES-V =  0.
      COMPUTE C-PEQUENIOS-V =  0.
*
      GET FIRST CUENTAS-F.
      MOVE 'GET' TO OPERACION.
      PERFORM MSG-ON-CUENTAS-F.
      MOVE C-DEPTO OF CUENTAS-F TO C-DEPTO-V.
      MOVE C-MUNIC OF CUENTAS-F TO C-MUNIC-V.
      MOVE 75 TO DESCRIPCION-DIRECCION-L.
      MOVE 60 TO NOMBRE-PERSONA-L.
```

```
LOOP-WHILE-1.
     IF ( CUENTAS-F-V-L = 5 )
        IF (  CUENTAS-F-V = 'FOUND' )
              MOVE TRUE  TO SMALL1
        ELSE MOVE FALSE TO SMALL1
     ELSE
        MOVE FALSE TO SMALL1.
     IF NOT  ( SMALL1 NOT = FALSE )
        GO TO FIN-WHILE-2.
     COMPUTE CONTADOR-V =  CONTADOR-V + 1.
     MOVE DICABI OF CUENTAS-F TO CALCULAR-P-CUENTA-V.
     MOVE CAPITAL-V TO CALCULAR-P-SUMATORIA-V.
     PERFORM CALCULAR-P THRU CALCULAR-P-EXIT.
     MOVE CALCULAR-P-CUENTA-V TO DICABI OF CUENTAS-F.
     MOVE CALCULAR-P-SUMATORIA-V TO CAPITAL-V.
     IF NOT  ( CAPITAL-V > 20000 )
        GO TO ELSE-IF-3.
     COMPUTE C-GRANDES-V =  C-GRANDES-V + 1.
     GO TO FIN-IF-4.
ELSE-IF-3.
     COMPUTE C-PEQUENIOS-V =  C-PEQUENIOS-V + 1.
FIN-IF-4.
*
     GET NEXT CUENTAS-F.
     MOVE 'GET' TO OPERACION.
     PERFORM MSG-ON-CUENTAS-F.
     MOVE C-DEPTO OF CUENTAS-F TO C-DEPTO-V.
     MOVE C-MUNIC OF CUENTAS-F TO C-MUNIC-V.
     MOVE 75 TO DESCRIPCION-DIRECCION-L.
     MOVE 60 TO NOMBRE-PERSONA-L.
     GO TO LOOP-WHILE-1.
FIN-WHILE-2.
*
     IF ( OUTPUT-PRINTER = TRUE )
        DISPLAY 'CUENTAS LEIDAS.............'
                CONTADOR-V.
     IF ( OUTPUT-SCREEN = TRUE )
        DISPLAY 'CUENTAS LEIDAS.............'
                CONTADOR-V UPON CONSOLE.
     IF ( OUTPUT-PRINTER = TRUE )
        DISPLAY 'GRANDES CONTRIBUYENTES.....'
                C-GRANDES-V.
     IF ( OUTPUT-SCREEN = TRUE )
        DISPLAY 'GRANDES CONTRIBUYENTES.....'
                C-GRANDES-V UPON CONSOLE.
     IF ( OUTPUT-PRINTER = TRUE )
        DISPLAY 'PEQUENIOS CONTRIBUYENTES...'
                C-PEQUENIOS-V.
     IF ( OUTPUT-SCREEN = TRUE )
        DISPLAY 'PEQUENIOS CONTRIBUYENTES...'
                C-PEQUENIOS-V UPON CONSOLE.
```

```
      MOVE 'SIGN-OFF' TO OPERACION.
      SIGN-OFF.
*
 FIN-EJEMPL04-P.
      STOP RUN.
*
*
*****************************************************************
** CALCULAR EL CAPITAL DE UNA CUENTA.                           *
*****************************************************************
*
 CALCULAR-P.
      COMPUTE CALCULAR-P-SUMATORIA-V =  0.
      GET FIRST INMUEBLES-F USING
                CALCULAR-P-CUENTA-V.
      MOVE 'GET' TO OPERACION.
      PERFORM MSG-ON-INMUEBLES-F.
      MOVE 16 TO N-INMUEBLE-L.
      MOVE C-DEPTO OF INMUEBLES-F TO C-DEPTO-V.
      MOVE C-MUNIC OF INMUEBLES-F TO C-MUNIC-V.
*
 LOOP-WHILE-5.
      IF ( INMUEBLES-F-V-L = 5 )
         IF (  INMUEBLES-F-V = 'FOUND' )
              MOVE TRUE  TO SMALL1
         ELSE MOVE FALSE TO SMALL1
      ELSE
         MOVE FALSE TO SMALL1.
      IF NOT  ( SMALL1 NOT = FALSE )
         GO TO FIN-WHILE-6.
      COMPUTE CALCULAR-P-SUMATORIA-V = CALCULAR-P-SUMATORIA-V
                      + I-INMUEBLE OF INMUEBLES-F.
*
      GET NEXT INMUEBLES-F USING
               CALCULAR-P-CUENTA-V.
      MOVE 'GET' TO OPERACION.
      PERFORM MSG-ON-INMUEBLES-F.
      MOVE 16 TO N-INMUEBLE-L.
      MOVE C-DEPTO OF INMUEBLES-F TO C-DEPTO-V.
      MOVE C-MUNIC OF INMUEBLES-F TO C-MUNIC-V.
      GO TO LOOP-WHILE-5.
 FIN-WHILE-6.
 CALCULAR-P-EXIT.  EXIT.
*
 ERROR-ON-TIS-CONTROL.
      CALL 'MMOFSI'
           USING OPERACION,
                 TIS-FSI,
                 TIS-CONTROL-FSI, TIS-CONTROL-FSI-L,
                 M-TIS-CONTROL, M-TIS-CONTROL-L,
                 ERROR-FATAL.
```

```
     CALL 'MMOVASI'
          USING TIS-VSI,
                TIS-CONTROL-VSI, TIS-CONTROL-VSI-L.
     PERFORM FATAL-ON-TIS-CONTROL.
*
 FATAL-ON-TIS-CONTROL.
     DISPLAY 'FATAL ERROR'.
     DISPLAY 'VIEW        =TIS-CONTROL'.
     DISPLAY 'OPERACION   =' OPERACION.
     DISPLAY 'TIS-FSI     =' TIS-FSI TIS-CONTROL-FSI.
     DISPLAY 'TIS-VSI     =' TIS-VSI TIS-CONTROL-VSI.
     DISPLAY 'TIS-MESSAGE=' TIS-MESSAGE.
     RESET.
     GO TO FIN-EJEMPLO4-P.
*
 MSG-ON-INMUEBLES-F.
     CALL 'MMOFSI'
          USING OPERACION,
                TIS-FSI,
                INMUEBLES-F-FSI, INMUEBLES-F-FSI-L,
                INMUEBLES-F-V, INMUEBLES-F-V-L,
                ERROR-FATAL.
     CALL 'MMOVASI'
          USING TIS-VSI,
                INMUEBLES-F-VSI, INMUEBLES-F-VSI-L.
     MOVE TIS-MESSAGE TO INMUEBLES-F-MSG-FSI.
     MOVE 40          TO INMUEBLES-F-MSG-FSI-L.
     IF ERROR-FATAL = TRUE
        PERFORM FATAL-ON-INMUEBLES-F.
*
 ERROR-ON-INMUEBLES-F.
     MOVE ZERO TO FALSE.
*
 FATAL-ON-INMUEBLES-F.
     DISPLAY 'FATAL ERROR'.
     DISPLAY 'VIEW        =INMUEBLES-F'.
     DISPLAY 'OPERACION   =' OPERACION.
     DISPLAY 'TIS-FSI     =' TIS-FSI INMUEBLES-F-FSI.
     DISPLAY 'TIS-VSI     =' TIS-VSI INMUEBLES-F-VSI.
     DISPLAY 'TIS-MESSAGE=' TIS-MESSAGE.
     RESET.
     GO TO FIN-EJEMPLO4-P.
*
 MSG-ON-CUENTAS-F.
     CALL 'MMOFSI'
          USING OPERACION,
                TIS-FSI,
                CUENTAS-F-FSI, CUENTAS-F-FSI-L,
                CUENTAS-F-V, CUENTAS-F-V-L,
                ERROR-FATAL.
     CALL 'MMOVASI'
```

```
           USING TIS-VSI,
                 CUENTAS-F-VSI, CUENTAS-F-VSI-L.
       MOVE TIS-MESSAGE TO CUENTAS-F-MSG-FSI.
       MOVE 40          TO CUENTAS-F-MSG-FSI-L.
       IF ERROR-FATAL = TRUE
          PERFORM FATAL-ON-CUENTAS-F.
*
 ERROR-ON-CUENTAS-F.
       MOVE ZERO TO FALSE.
*
 FATAL-ON-CUENTAS-F.
       DISPLAY 'FATAL ERROR'.
       DISPLAY 'VIEW        =CUENTAS-F'.
       DISPLAY 'OPERACION   =' OPERACION.
       DISPLAY 'TIS-FSI     =' TIS-FSI CUENTAS-F-FSI.
       DISPLAY 'TIS-VSI     =' TIS-VSI CUENTAS-F-VSI.
       DISPLAY 'TIS-MESSAGE=' TIS-MESSAGE.
       RESET.
       GO TO FIN-EJEMPLO4-P.
*
```

APENDICE C

Limitaciones del Traductor

Para utilizar el traductor se deben tomar en cuenta las siguientes limitaciones:

1.- No se permite el uso de arreglos de dos dimensiones.

2.- No es permitido seleccionar una lista de campos en la instrucción *VIEW*.

3.- No pueden usarse las opciones *PREFIX* y de manejo de *buffers* en las instrucciones *VIEW* y *ACCESS*.

4.- La instrucción *ACCESS* sólo permite el uso de archivos sequenciales o indizados. No se permiten archivos numerados.

5.- El traductor detecta cuando un programa es directamente recursivo pero no detecta la recursión indirecta. No son permitidos programas que sean recursivos.

6.- La variable utilizada en las instrucciones *PAD* y *UNPAD* no puede hacer referencia a *substrings*.

7.- No es posible enviar vistas o archivos como parámetros a procedimientos.

8.- No se puede utilizar la opción *TAB* en el comando *SHOW*.

9.- Las vistas y archivos dentro del programa en Mantis no se deben cerrar y abrir de nuevo, ya que ésto no se puede hacer en Cobol.

10.- El programa fuente en Mantis no debe contener instrucciones divididas en varias líneas.

11.- No es permitido el uso de números en notación exponencial. El traductor sólo considera 16 dígitos significativos (8 enteros y 8 decimales) para los números tipo *BIG* y 8 dígitos para los datos tipo *SMALL*.

12.- Las instrucciones *ENQUEUE* y *DEQUEUE* de Mantis no son permitidas.

Agradecimientos.

Agradezco a Rodrigo Arias las ideas y sugerencias que aportó para la elaboración del proyecto y la revisión del informe. También, al señor Armando Muralles por la revisión del texto.

Abril de 1990. Iván Muralles.