COLEGIO DE CIENCIAS Y HUMANIDADES

INSTITUTO DE INVESTIGACIONES EN MATEMATICAS APLICADAS Y EN SISTEMAS

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

"FOREST: DISEÑO Y EJECUCION DE UN PREPROCESADOR PARA EXTENDER EL LENGUAJE FORTRAN"

T E S I S

QUE PARA OPTAR AL GRADO DE:

MAESTRO EN CIENCIAS DE LA COMPUTACION

P R E S E N T A :

GUILLERMO LEVINE GUTIERREZ

MEXICO, D. F.

JULIO DE 1979

Este trabajo fue desarrollado en su totalidad en el área de ingeniería electrónica de la Universidad Autónoma Metropolitana, Iztapalapa.

Con agradecimiento para el asesor de esta tesis, Dr. Luis Legarreta; a quien siempre se le ocurre una mejor solución.

Para mis amigos
Ing. Carlos M. Péretz (IBM)
Ing. Edmundo Ponce Adame (U. de G.),
que me dieron la oportunidad.

FOREST : (FORTRAN ESTRUCTURADO)
PREPROCESADOR PARA FORTRAN

1.- INTRODUCCIÓN

Un preprocesador para un lenguaje es un filtro. Tiene una entrada y una salida. Por la entrada ingresan expresiones en un lenguaje fuente y por la salida se obtienen sus correspondientes representaciones en el lenguaje objeto. La diferencia de esto con un compilador es que en el primero no hay realmente tanta diferencia entre los dos lenguajes, más bien sólo se "disfraza" a uno con el ropaje del otro, mientras que en el compilador se hace una traducción más completa. Este hecho habla ya de las diferencias entre ambos procesadores: un preprocesador es más barato (o debe serlo) que un compilador y a la vez se espera sea un poco más sencillo.

Los elementos del lenguaje objeto generados por un preprocesador comparten la forma general de la estructura sintáctica del lenguaje fuente.

Hay, sin embargo, varias diferencias. El lenguaje fuente contiene símbolos adicionales así como construcciones nuevas, basadas en símbolos que no están presentes en el lenguaje objeto. De esta manera, es válido hablar de este proceso como de uno de expansión de macro-expresiones ya que, en general, algunas de estas nuevas construcciones se verán "traducidas", o macro-expandidas, en conjuntos precisamente preestablecidos de elementos del lenguaje objeto.

Recordemos que una macro-expresión es un conjunto bien definido de símbolos que reemplazan a un símbolo particular (llamado "nombre del macro") donde quiera que éste aparezca. No es aquí lugar para discutir sobre macros, por lo que referimos al lector a las fuentes (Refs. 13, 5).

Nuestro preprocesador se encargará de darle a FORTRAN un ropaje que lo haga útil a la enseñanza de la programación moderna (i.e., "estructurada") y por lo tanto, estaremos interesados en extender las estructuras de control de ese lenguaje, que es uno de los más usados en el mundo.

De las estructuras de control DO , GOTO, IF, CONTINUE obtendremos las siguientes: BEGIN-END, IF-(THEN)-ELSE, FOR, DO, WHILE, BREAK, NEXT REPEAT-UNTIL, CASE . Además, lograremos convertir a FOREST en un lenguaje de formato libre. Estamos seguros que estas características harán de FOREST una buena alternativa para aquellos centros de enseñanza de cómputo que no dispongan de ALGOL o PL/I o PASCAL que, en nuestro país y sobre todo en provincia, son la gran mayoría. De cien centros de cómputo tomados al azar, 99 trabajan con FORTRAN y ésta es la gran ventaja. ¿Por qué no utilizar esa especie de lenguaje universal de la programación y hacer de él un arma poderosa para trabajar con la programación moderna? Es sabido que es difícil (si no imposible) enseñar buena programación en FORTRAN. La razón es que el estudiante se pierde entre los árboles, que no le dejan ver el bosque...

Porque la programación no se hace con IF's o GOTO's o CONTINUE's, sino que se hace con BLOQUES y con estructuras bien especificadas, que se demuestra que son suficientes para expresar cualquier algoritmo computacional (Ref. 4). Estos bloques consisten fundamentalmente de reglas de formación de: Iteración y composición.

La "programación estructurada" es el conjunto de reglas, métodos y disciplinas orientadas hacia la utilización de estas reglas de formación (un poco aumentadas) para obtener programas legibles, entendibles y mantenibles.

La ampliación de las reglas mencionadas obedece a una mayor adecuación a los lenguajes de programación actualmente en uso (i.e., aquéllos diseñados para el modelo de Von Neumann; basados en la transición de estados finitos (Ref. 1)). Se reconocen estas reglas como las de: 1) composición (secuenciación) 2) selección y 3) iteración condicional. La literatura sobre estos conceptos es muy amplia (Refs. 17, 7, 12, 21).

FORTRAN provee al programador de solamente un concepto y medio de los tres mencionados; decimos uno y medio porque se permite el IF pero no el ELSE. Sólo se trabaja bien con la secuenciación, que no es más que aprovechar el flujo natural

de todo programa ("de arriba hacia abajo").

FOREST, por el contrario, provee de varios modos de aprovechar estas reglas de formación teóricas. Se espera, de esta manera, que el usuario sea capaz de hacer mejor programación con FOREST que con FORTRAN.

Hay que decir, sin embargo, que las herramientas que le damos al usuario no lo convierten per se en un buen programador, sólo nos comprometemos a que éstas no le estorbarán en el desempeño de su trabajo, como suele suceder tantas veces en FORTRAN. Otra advertencia, las estructuras de control que escogimos no tienen por qué ser las únicas o las mejores, y de hecho hay propuestas para varias mejoras y adiciones (Ref.18). FOREST es una mezcla de RATFOR (Refs.11, 13) con el lenguaje de programación "C" (Ref.14) y con ideas propias.

Ahora pasemos a las desventajas del sistema: la primera y más obvia es que, en su calidad de preprocesador, FOREST requiere de FORTRAN una vez que ha terminado de hacer su trabajo, i.e., una vez que se hizo el pre-proceso. O sea que el lenguaje fuente es FOREST y el objeto es FORTRAN. Ya, y forzosamente, el tiempo de procesamiento (compilación, no ejecución) se dobla: una vez para FOREST y otra para FORTRAN.

Igualmente, y debido a lo mismo, los mensajes de error van a provenir de dos fuentes: del preprocesador y del compilador. Confiamos en que los mensajes del preprocesador serán claros y ayudarán a corregir el programa fuente pero puede suceder que, una vez correcto el programa escrito en FOREST, sea el compilador FORTRAN el que proteste. Se hace necesario entonces dar al usuario una especie de índice cruzado para que pueda hacer referencias al código FORTRAN desde el programa FOREST. Todo esto se discutirá en detalle en el manual de usuario.

Creemos que ya quedará clara la idea general: el usuario aprende técnicas de programación estructurada y las aplica por medio de programas escritos en FOREST. El preprocesador convierte este programa a "código objeto" escrito en FORTRAN, y luego el compilador procesa la imagen del programa original, y se ejecuta.

La traducción entre las estructuras de FOREST y las de FORTRAN se hace en forma mecánica una vez que se ha definido la asociación entre la sintaxis de los dos lenguajes. Es claro que estas manipulaciones sintácticas tienen como finalidad lograr un enriquecimiento semántico en el lenguaje fuente, de tal manera que sea más útil en las aplicaciones de programación estructurada y, en general, más útil que FORTRAN. El preprocesador entonces no hará más que seguir los dictados de la gramática y generar "código" de acuerdo a las "acciones" que la gramática le vaya dictando (Ref. 8).

Ésta es la diferencia fundamental entre nuestro preprocesador y el 'RATFOR' ya mencionado; aquél hace la traducción de manera sui generis, mientras que éste la hace de manera canónica.

Es por esto que el capítulo 2 trata sobre el reconocedor sintáctico sin hacer referencia a algún lenguaje en particular, o a las características especiales de alguna construcción. Primero se hizo el 'parser' (reconocedor sintáctico) y luego se trató de traducirlo a FORTRAN. Porque, por supuesto, FOREST está escrito finalmente en FORTRAN ya que, de otra manera, no tendría utilidad.

Claramente, si deseamos que FOREST sea utilizado en cualquier centro de cómputo que disponga de FORTRAN, entonces habrá que escribir nuestro preprocesador en este lenguaje.

Aún más, FOREST está escrito en un FORTRAN fácilmente transportable; esto es, que no está apegado a ninguna máquina en particular. Por ejemplo, en la codificación no utilizamos las extensiones que algunos compiladores permiten al manejar FORTRAN, ni utilizamos apóstrofos ni nombres de más de seis caracteres, etc., aunque muchos compiladores ya permiten este tipo de cosas. En aquellas partes del código donde sea inevitable utilizar partes no "universales" de FORTRAN, lo hacemos tratando de causar el menor número de molestias a los que decidan transportar FOREST a sus computadoras.

En el capítulo 3 describimos las instrucciones de FOREST como macro-declaraciones y las mostramos ya expandidas en FORTRAN. El siguiente paso será lograr que

la expansión de estas macros se haga en forma automática.

Todos (o por lo menos la gran mayoría de) los puntos obscuros del programa FOREST escrito en FORTRAN (39 subrutinas) se deben a esta conversión forzosa entre el pseudocódigo recursivo en que se diseñó FOREST y el lenguaje FORTRAN en que se codificó.

El capítulo 4 habla de esta triste conversión. Aquí utilizamos algunos conceptos generales de eliminación de recursión, descritos en varias fuentes (Refs. 19, 3). Nos fue muy útil la recomendación que hace Knuth sobre la traducción de programas recursivos a no-recursivos (Ref. 16).

Esta conversión nos da derecho a hablar del preprocesador como un "pseudo-compilador" y nos permite además anunciar que acepta cualquier combinación posible de construcciones (siempre y cuando no violen la sintaxis definida por la gramática) y cualquier nivel de anidamiento que se desee (siempre y cuando se le haga saber tal cosa al compilador FORTRAN con el DIMENSION apropiado). Esto, claro, es un arma de dos filos: un mal programador puede anidar y anidar construcciones y FOREST pacientemente irá deshaciendo los "nudos recursivos" que se vayan formando. Sintácticamente todo irá bien, pero semánticamente puede haber innumerables problemas. Que el programa compile bien pero ejecute mal; que entregue resultados inválidos o ininteligibles; que se comporte de manera errática para diferentes entradas, etc.

Todos estos problemas caen fuera del rango predominantemente sintáctico en que FOREST trabaja aunque, como ya mencionamos, cambios bien pensados en la sintaxis pueden obtener como resultado mejoras semánticas sustanciales.

En este preprocesador no se ha hecho ningún intento de extender las estructuras de datos de FORTRAN. Tal cosa es posible, en grado restringido, a través de un macro-procesador. En la idea original de esta tesis estaba contemplado hacer de FOREST también un macro-procesador, pero preferimos extender aún más las estructuras de control (el CASE no estaba previsto). La razón es que consideramos más útil lo

segundo que lo primero en virtud del uso esperado de FOREST como herramienta didáctica. Es más importante, a nuestro entender, enseñar a programar de forma moderna con estructuras de control ágiles, que tratar que un alumno mal preparado en estos conceptos aprenda los recovecos de los macros.

Una ampliación posterior del preprocesador contemplaría introducir el macroprocesador (escrito en FOREST, claro) junto con las estructuras de datos que se puedan definir por medio de él.

El "código" generado o sea, FORTRAN, estará un poco rebuscado, es cierto: algunos GOTO's alrededor de GOTO's, muchos CONTINUEs seguidos, etc., pero aún así será entendible. Además, no hay que olvidar que es "código objeto"; es decir, se supone que, si todo va más o menos bien, el usuario no tendrá necesidad de examinarlo. La única carga extra para el computador será uno que otro GOTO innecesario, o sea, algunos microsegundos adicionales.

También hay que mencionar que FOREST no se haría acreedor (creemos) a las críticas que para RATFOR tuvo la universidad de Cornell (Ref. 6) en el sentido de que era innecesariamente ineficiente ya que hacía búsquedas secuenciales y "retrocedía" a la menor provocación en el acto de hacer el análisis sintáctico. Nosotros nunca retrocedemos en el sentido de devolver caracteres a la rutina de entrada para releerlos (más que en un caso, y de manera simulada) y hacemos búsquedas binarias, no secuenciales. No creímos necesario hacer 'hash', aunque estuvimos a punto de dejarnos convencer por un novedoso artículo que prometía demasiado, tal como mencionamos en (Ref. 20).

Dicho esto pasaremos entonces a la revisión de la anatomía de nuestro preprocesador para luego dar ejemplos de programas escritos y procesados por él.

2.- DISEÑO DEL PREPROCESADOR ('PARSER')

Partiremos de la gramática de nuestro lenguaje. Es una gramática parecida a la de FORTRAN, pero aumentada. De hecho, esta gramática consiste de dos partes: la gramática usual de FORTRAN, más las añadiduras hechas por nosotros para extender ese lenguaje. De esta manera, si un programa consiste de partes de FOREST entremezcladas con partes de FORTRAN no habrá, más que con una sola excepción, problemas para procesarlo. Esa excepción es la instrucción DO de FORTRAN, que se ve reemplazada por la de FOREST, quedando eliminada la original.

Ésta es la descripción, en BNF, de nuestra gramática:

```
ELEMENTO:      ::= DECLARACIÓN / PROPOSICIÓN
DECLARACIÓN    ::= "declaraciones no ejecutables de FORTRAN".
PROPOSICIÓN    ::= if (-condición-); PROPOSICIÓN
                 / if (-condición-); PROPOSICIÓN else PROPOSICIÓN
                 / while (-condición-); PROPOSICIÓN
                 / for ( -inicio- ';' -condición- ';' -reinicio- ) PROPOSICIÓN
                 / repeat ; PROPOSICIÓN
                 / repeat ; PROPOSICIÓN; until (-condición-)
                 / do -límites- ; PROPOSICIÓN
                 / break
                 / next
                 / case -identificador- ' ; '
                   MASCASO
                 / COMPUESTA
                 / ++ -identificador-
                 / -- -identificador-
                 / -etiqueta- PROPOSICIÓN
                 / FIN
```

```
COMPUESTA       ::= begin MASCOMP end
MASCOMP         ::= PROPOSICION MASCOMP / PROPOSICIÓN
MASCASO         ::= -identificador- ' : ' PROPOSICION ; MASCASO
                  / end
FIN             ::= "cualquier proposición válida de FORTRAN (excepto DO )"
-identificador  ::= "cualquier identificador válido de FORTRAN"
-condición-     ::= "cualquier condición válida de FORTRAN, con los relacionales
                     tipo de PL/I (p. ej.,  = en lugar de .EQ.)"
-inicio-        ::- "cualquier proposición de asignación de FORTRAN"
                  / "cualquier expresión de llamada (que use CALL)"
-reinicio-      ::= -inicio-
-etiqueta-      ::= "cualquier etiqueta válida de FORTRAN, excepto aquéllas que
                     comiencen con 77 y sean de cinco cifras"
```

Aparecen con mayúsculas los elementos no-terminales de la gramática. Aparecen subrayados los elementos terminales que son palabras clave. Aparecen con minúsculas y entre guiones ciertos elementos terminales que merecen mayor explicación. El punto y coma sin apóstrofos es un "brinco de renglón", opcional. Los símbolos entre apóstrofos deberán aparecer textualmente, así como los símbolos adicionales (paréntesis y otros puntos y comas, y signos dobles ++ y --). El símbolo ' / ' representa a la disyunción.

Una vez definida la gramática sólo resta construir un reconocedor para ella, es decir, un 'parser'.

El problema de 'parsing' es un problema muy conocido, sobre todo para gramáticas de tipo "2", como puede ser ésta (Ref. 10).

El reconocedor que nosotros hicimos es dirigido por la sintaxis de la gramática, pero no es de los llamado "dirigidos por tabla". Decidimos construir un 'parser' del llamado de tipo "descenso recursivo" (Refs. 9, 22) por considerarlo

sencillo y canónico[1].

Como su nombre lo indica, éste será un programa recursivo. Quedará para después la conversión de este en uno no-recursivo, apto para ser codificado en FORTRAN.

La idea fundamental del reconocedor de "descenso recursivo" consiste en encontrar una alternativa no-terminal adecuada y seguirla hasta su conclusión. Tal conclusión, sin embargo puede estar lejana, ya que la gramática es recursiva. Es por esto que el reconocedor debe ser también recursivo. Cada vez que un no-terminal apunta a un miembro izquierdo de la gramática el reconocedor hace un "descenso" y vuelve a tratar de agotar la alternativa.

Hay que decir que, como esta gramática es sencilla, el reconocedor no tiene que perder mucho tiempo regresando de alternativas que resultaron no ser las "verdaderas". De hecho, sólo tiene que regresarse en dos ocasiones; cuando no sabe si un if o un repeat van a estar seguidos de sus (opcionales) else o until, respectivamente.

Antes mencionamos (ver supra) que el reconocedor de RATFOR está construido de tal manera que constantemente tiene que regresar caracteres leídos al archivo de entrada. Ese reconocedor no es de tipo "descenso recursivo", sino que más bien es una mezcla de esto con conceptos ad-hoc y, por lo tanto, no lo consideramos como canónico.

La construcción del reconocedor canónico consiste en hacer un programa para "seguir" los múltiples caminos abiertos por la gramática ya descrita.

Supondremos que tenemos una rutina (llamada -sigue-) que nos devuelve el siguiente componente lexicográfico del archivo de entrada. Esto presupone, necesariamente, una rutina de más bajo nivel que haga el papel de reconocedor léxico[2], pero por lo pronto no nos preocuparán estos detalles.

---

† Decimos "canónico" porque no es 'ad hoc', sino que sigue la representación preformulada por la gramática recursiva.

Siguiendo el método de "refinamiento progresivo" de Wirth (Ref.21) proponemos el siguiente pseudo-código recursivo para la rutina maestra.

```
procedure DCL recursive;

    if -sigue- = 'IF' then
     begin
         "acción" IF;
         call DCL ;
         if -sigue- = 'ELSE' then
                                  begin
                                      "acción" ELSE;
                                      call DCL;
                                  end
    end
    else if -sigue- = 'WHILE' then
                                    begin
                                        "acción" WHILE ;
                                        call DCL;
                                    end

    else if -sigue- = 'FOR' then
                                   begin
                                       "acción" FOR;
                                       call DCL;
                                   end
```

[2] Recordemos que un compilador está dividido en analizador lexicográfico, analizador sintáctico y analizador semántico. Estamos, pues, en la primera parte.

```
else if -sigue- = 'REPEAT' then
                                    begin
                                        "acción" REPEAT;
                                        call DCL;
                                        if -sigue- = 'UNTIL' then "acción"
                                        UNTIL;
                                    end

else if -sigue- = 'DO' then
                                begin
                                    "acción" DO;
                                    call DCL;
                                end

else if -sigue- = -ETIQ- then
                                    begin
                                        "acción" -ETIQ-;
                                        call DCL;
                                    end

else if -sigue- = 'BREAK' then "acción " BREAK ;
else if -sigue- = 'NEXT' then "acción" NEXT ;
else if -sigue- = 'BEGIN' then
                                    begin
                                        while -sigue- ≠ 'END'
                                            begin
                                                "regresa el componente";[3]
                                                call DCL;
                                            end
                                    end
```

[3] Se necesita "regresar" el componente al renglón de entrada porque fue leído de más.

```
else if -sigue- = 'CASE' then
                                    begin
                                        "acción CASE;
                                        call DCL;
                                    end

else if -sigue- = -FTN- then
                                    begin
                                        "cópialo a la salida";
                                    end

else if -sigue- = ++id / --id then begin "expansión"; call DCL; end

end DCL .
```

Y el siguiente programa principal

```
repeat
    call DCL
until -sigue- = EOF
```

Este programa recursivo hace efectivamente el 'parsing' sobre la gramática antes mencionada además de llevar a cabo el análisis semántico del lenguaje y la generación de código. Está escrito en pseudo-código y es bastante liberal con respecto a las restricciones que surgirán a tiempo de codificación. Pero esta es precisamente la gran ventaja de los métodos de refinamiento progresivo: permiten al programador liberarse de la tiranía del código y de los detalles para concentrarse en el algoritmo a seguir.

El programa se ve muy sencillo, pero no es tan sencillo de seguir. Para prueba sugerimos al lector que trate de "correrlo" sobre un conjunto de instrucciones

tomadas de la gramática. La razón de esto es, por supuesto, la recursividad.

No hay mejor cosa que un algoritmo recursivo para tratar con estructuras inherentemente recursivas (tales como una gramática), pero a la vez no hay algoritmo tan engañoso a primera vista como uno recursivo, precisamente por su aparente sencillez...

Tal como habíamos mencionado antes, esta gramática tiene añadidas (en el reconocedor correspondiente) "acciones": instrucciones para generar código que ayude a la posterior "compilación". Por lo pronto quedan indefinidas, pues lo que nos interesa es recorrer (reconocer) los caminos de la gramática.

Recorrer tales caminos significa, para cada posible palabra clave (i.e., terminal), comparar el árbol de derivación, de manera 'top-down', es decir, comenzando por los no-terminales.

En el programa anterior seguimos las convenciones usuales del pseudo-código, i.e., subrayar las instrucciones y poner en mayúsculas las cadenas terminales buscadas y los nombres de los procedimientos. Igualmente "EOF" significa "fin de archivo de entrada".

Como todavía no hemos hablado de las instrucciones particulares de FOREST, el lector que desee hacer el 'parse' estará haciendo un juego formal no-interpretado, es decir, estará siguiendo reglas sintácticas sin preocuparse de los aspectos semánticos.

Nos referimos a algo así:

Aplicar el algoritmo recursivo a lo siguiente:

IF (-condición-) -FTN- ELSE BEGIN WHILE (-condición-) -FTN- END .

Esto daría como resultado el "seguir" los árboles de derivación de las expresiones anteriores, sin preocuparnos por su significado. Y esto es perfectamente posible de hacer con el procedimiento DCL :

PROPOSICIÓN

'IF' (-condición-) PROPOSICIÓN 'ELSE' PROPOSICIÓN

FTN COMPUESTA

'BEGIN' MASCOMP 'END'

PROPOSICIÓN

'WHILE' (-condición-) PROPOSICIÓN

FTN

El árbol nos muestra los niveles de recursión en los que se mete el reconocedor (los niveles de"descenso recursivo").

Tal como mencionamos antes, es posible hacer casi cualquier número o combinación de anidamientos de las estructuras. El 'parser' simplemente hará tantos "descensos" como sea necesario.

Hay que hacer notar que si MASCOMP es tipo -FTN-, entonces no se creará un nuevo nivel recursivo, sino que simplemente se terminará el nivel actual. De esta manera, bastarán unos pocos niveles para reconocer la sintaxis de casi cualquier programa. Es raro encontrar más de, digamos, 10 niveles recursivos anidados.

3.- DISEÑO DEL PREPROCESADOR - GENERADOR DE "CÓDIGO"

El "código" generado por nuestro preprocesador no será otra cosa que FORTRAN "puro". Recordemos que el objetivo es hacer una herramienta para extender FORTRAN, para convertirlo en un lenguaje moderno, estructurado.

En este capítulo desarrollaremos las "acciones" que están inmersas en la gramática que recorre el 'parser', y que están destinadas a generar las macro-expansiones de las estructuras de FOREST.

Desde un punto de vista de bloques, esta parte del preprocesador debería ser sencilla, ya que las funciones que desempeñará son directas y no envuelven ni recursión ni ningún otro proceso sofisticado. Veremos más adelante, sin embargo, que esta es precisamente la parte más rebuscada del código, porque hay que recordar que FOREST deberá estar escrito en FORTRAN.

Vamos a desarrollar los bloques principales, que serán más o menos diez: uno por cada estructura de control. Antes de esto, sin embargo se hace necesario describir la estructura de las instrucciones de FOREST.

Las instrucciones propias de FOREST son:

BEGIN-END BREAK CASE DO FOR IF-ELSE NEXT REPEAT-UNTIL y WHILE.

Junto con sus correspondientes en español que son:

COMIENZA-TERMINA FUERA CASO EJECUTA PARA SI-OTRO SIGUE REPITE-HASTA MIENTRAS.

FOREST es un lenguaje de formato libre, es decir, las proposiciones no tienen que comenzar en determinada columna (p. ej. en la 7) ni se restringe a una sola proposición por renglón: puede haber varias, separadas por un punto y coma. De la misma manera, las etiquetas (si es que se requieren) pueden ir en cualquier columna del renglón. En general no está permitido seguir una proposición en otro renglón; pero si un renglón termina o comienza en coma, entonces automáticamente se generará una línea de continuación para FORTRAN. Si comienza con ' / ' también se hará.

Esto es útil para declaraciones INTEGER o DATA o COMMON largas. Sólo en el caso de las condiciones (dentro de un IF, por ejemplo) se permite brincar de renglón en ciertos puntos (luego de una coma, etc.) ya que estos serán los únicos casos donde pueda suceder tal cosa. También se puede continuar un FOR y un WHILE en otro renglón, aunque es dudable que se requiera.

El RANGO de las proposiciones se extiende tan sólo a una proposición. Si se quiere que abarque más de una proposición, habrá que usar los paréntesis BEGIN-END.

Todas las instrucciones de FORTRAN (excepto DO) pueden ser usadas en el preprocesador. El caso del DO de FORTRAN ya está previsto por la correspondiente proposición DO de FOREST. Esto significa que el DO de FORTRAN será siempre reemplazado por el de FOREST.

Vamos entonces a explicar las demás instrucciones, comenzando por el DO.

PROPOSICIÓN DO: El EJECUTA/DO de FOREST es idéntico a su homólogo de FORTRAN, con la diferencia de que aquí no se requiere la etiqueta de control.

Así, pues, las instrucciones

```
      DO  100  I = 1,25
          A(I) = A(I) + 1
100   CONTINUE
```

Se expresan mejor por:

```
      DO  I = 1,25
          A(I) = A(I) + 1
```

Esto es todo en ese caso, ya que el DO sólo controla una proposición. Pero si controla más entonces se hace necesario usar los "paréntesis".

Por ejemplo:

```
      DO  100  I = 1,25
          A(I) = A(I) + 1
          C(I) = C(I) * D(I)
100   CONTINUE
```

Se expresa como:

```
      DO  I = 1,25
      BEGIN
          A(I) = A(I) + 1
          C(I) = C(I) * D(I)
      END
```

Por lo demás, se siguen las reglas semánticas de FORTRAN.

PROPOSICIÓN BREAK

La proposición BREAK/FUERA sirve para, dentro de un DO o un WHILE o un FOR o un REPEAT o CASE, "salirse" del rango de control. Toma el lugar de un GOTO -etiqueta-, donde la -etiqueta- sea la siguiente instrucción luego del DO o WHILE, etc.

Por ejemplo:

```
      DO  100  I = 1,30
          A(I) = A(I) + 1
          IF( C(I) .EQ. D(I) ) GO TO 120
          D(I) = N1
100   CONTINUE
120   DELTA = GAMA
```

Se expresa como:

```
        DO  I  = 1,30
      BEGIN  A(I) = A(I) + 1
             IF( C(I) = D(I) ) BREAK
             D(I) = N1
        END
        DELTA = GAMA
```

PROPOSICIÓN NEXT:

NEXT/SIGUE sirve para mandar el control al comienzo del DO otra vez. Esto es, hace que una interación tipo DO o WHILE o REPEAT se vuelva a efectuar de manera forzosa.

Por ejemplo:

```
        DO 100 I = 1,35
           A(I) = A(I) + 1
           IF((C(I) .AND. ALFA) .EQ. BETA ) GO TO 100
           GAMA = DELTA
100        CONTINUE
```

Se expresa como:

```
        DO  I = 1,35
        BEGIN
           A(I) = A(I) + 1
           IF( (C(I) & ALFA) = BETA ) NEXT
           GAMA = DELTA
        END
```

PROPOSICIÓN IF:

La proposición IF/SI, que ya hemos utilizado en los ejemplos, es muy parecida a la de FORTRAN. Las diferencias son que

```
IF( A(I)  .EQ. BETA  .AND.  C(J)  .EQ.  ALFA )  CALL ALGO
```

Se expresa como

```
IF( A(I)  = BETA & C(J)  =  ALFA )  CALL ALGO
```

De la misma manera, si se requiere más de una proposición luego del IF, se puede encerrar el grupo en paréntesis BEGIN-END:

Por ejemplo:

```
        IF( A(I)  .NE. BETA ) GO TO 125
        ALFA = 1
        BETA = 2
        ETA = 3
125     CONTINUE
```

Se expresará así:

```
        IF( A(I) = BETA )
        BEGIN
            ALFA = 1
            BETA = 2
            ETA = 3
        END
```

PROPOSICIÓN IF-ELSE:

FOREST permite la utilización de una cláusula alterna en el IF. Esto es, si la condición no se cumple se ejecutará esta cláusula alterna y el control "saldrá" del rango del IF-ELSE (SI - OTRO).

Por ejemplo:

```
        IF( A(K)  .EQ.  123.5 ) GO TO 125
        ALFA = BETA
        DELTA = ETA
        GO TO 126
125     CONTINUE
        C(J) = D(M)
126     CONTINUE
```

Podría mejor expresarse como:

```
        IF( A(K) = 123.5 ) C(J) = D(M)
        ELSE BEGIN
             ALFA = BETA
             DELTA = ETA
        END
```

Nuestro preprocesador sigue la costumbre de asignar cada ELSE al IF libre más anterior a él. Esto es, es perfectamente válido usar la construcción IF-ELSE con la segunda parte omitida. Los posibles problemas de la lógica de un tal programa no son, por supuesto, responsabilidad de FOREST. En el manual de usuario se darán ejemplos de esto, reconocido por cualquier programador de ALGOL o PL/I.

PROPOSICIÓN WHILE:

El WHILE/MIENTRAS de FOREST es la representación de la estructura fundamental mencionada en (Ref. 17).

Mientras la (-condición-) sea verdadera, se ejecutarán las instrucciones encerradas en los paréntesis BEGIN-END que vienen luego.

Por ejemplo:

```
120     IF(  ALFA.NE.BETA ) GO TO 125
        CALL UNO
        CALL DOS
        GO TO 120
125     CONTINUE
```

Se verá expresada por:

```
        WHILE ( ALFA = BETA)
        BEGIN
           CALL UNO
           CALL DOS
        END
```

Es necesario notar que si la condición nunca se cumple, entonces el grupo de proposiciones que sigue nunca se ejecutará.

PROPOSICIÓN REPEAT-UNTIL:

Esta estructura difiere de la anterior en que siempre ejecuta la proposición que le sigue al menos una vez, sin importar el valor de la condición. En este sentido, se comporta igual que el DO de FORTRAN.

Por ejemplo:

```
125     CONTINUE
        ALFA = BETA
        DELTA = ETA
        IF(  C(I) .NE. TRES ) GO TO 125
```

Se podrá expresar como:

```
        REPEAT
        BEGIN
           ALFA = BETA
           DELTA = ETA
        END
        UNTIL ( C(I) = TRES )
```

Si eliminamos la cláusula UNTIL estaremos en el caso de una iteración "infinita". Es perfectamente válido usar tal cosa, unida al uso de la instrucción BREAK ya descrita.

Por ejemplo:

```
REPEAT
BEGIN
   ALFA = BETA
   DELTA = 3
   IF(  C(I) = 31.33 ) BREAK
END
```

PROPOSICIÓN FOR:

Esta proposición, FOR/PARA, no tiene paralelo en FORTRAN. Está tomada del lenguaje "C", y no es más que un DO generalizado.

Por ejemplo, en:

```
      ALFA = BETA
125   CONTINUE
      IF(  C(I) .NE. D(J))GO TO 135
      CALL UNO
      CALL DOS
      C(I) = C(I) + 1
      GO TO 125
135   CONTINUE
```

Podemos distinguir varias partes:

una sección ALFA=BETA de "inicialización";

una prueba condicional;

una sección CALL UNO de ejecución condicional y,

CALL DOS

una sección C(I) = C(I) + 1 de "reinicialización", que servirá para decidir -de acuerdo a la prueba condicional- si se ejecutan los dos CALL otra vez o no.

Todo esto se ve más claramente expresado por:

```
FOR(  ALFA = BETA ;  C(I)  = D(J) ; C(I) = C(I) + 1 )
  BEGIN
      CALL UNO
      CALL DOS
  END
```

Esto es, se ejecuta primero una proposición de "inicialización"; luego se pregunta por la condición; si es verdadera entonces se ejecutan las proposiciones que siguen al FOR y se regresa a preguntar, habiendo antes ejecutado el código de "reinicialización". Pero si la condición no fue verdadera entonces el control pasa por

encima a la proposición siguiente (o grupo BEGIN-END) sin ejecutarlas.

Decimos que es un DO generalizado porque puede haber cualquier tipo de instrucción en la reinicialización, y no solamente la suma implícita que hace el DO de FORTRAN sobre la variable de control.

Esto es, el DO ilícito DO 100 I = 25,0,-1 que a veces es muy útil se transforma en FOR (I=25 ; I = 0; I = I - 1).

PROPOSICIÓN CASE:

Cuando uno enseña programación estructurada surge siempre el problema de manejar los IF-THEN-ELSE. Como (en PL/I, por ejemplo) es posible anidar varias de estas instrucciones entonces surge el problema, ya mencionado, de los IF sin ELSE.

Se aconseja entonces manejar estas estructuras en la forma IF....THEN.... ELSE IF....THEN....ELSE IF....THEN...., etc.

El CASE/CASO no es más que la representación de esto.

Por ejemplo:

```
100     IF( A .NE. 1. 0) GO TO 110
        CALL UNO
        CALL DOS
        GO TO 200
110     IF( A .NE. BETA(I))GO TO 120
        CALL TRES
        CALL CUATRO
        GO TO 200
120     IF( A .NE. DELTA ) GO TO 200
        CALL CINCO
        CALL SEIS
200     CONTINUE
```

se ve expresado como:

```
        CASE A ;
           1.0:    BEGIN  CALL UNO ; CALL DOS; END
        BETA (I):  BEGIN  CALL TRES; CALL CUATRO;  END
        DELTA:     BEGIN  CALL CINCO; CALL SEIS; END
        END
```

FOREST además reconoce otro tipo de expresiones, que se usan mucho. En lugar de decir A = A + 1 diremos ++A ; y en lugar de C(I) = C(I) - 1

diremos --C(I) . Sirven sólo para el caso de sumar/restar la unidad.

Así pues, nuestro FOR anterior se convierte en:

```
FOR ( ALFA = BETA ; C(I) = D(J) ; ++C(I)  ) etc.
```

Teniendo ahora más o menos claras las estructuras que vamos a manejar podemos entonces proponer el mecanismo de traducción.

Éste consiste, fundamentalmente, de las "acciones" del pseudo-código (v. supra, pág. 11).

Como regla general, cada "acción" se codificó como una subrutina separada, con algunas excepciones; ya que, por ejemplo, el WHILE, FOR, REPEAT-UNTIL, IF, etc. hacen uso de rutinas compartidas para traducir las condicionales.

La idea fundamental es la siguiente: para cada nuevo "renglón" (renglón físico o continuación luego de un punto y coma) se analiza la palabra que lo encabeza. Si es una palabra reservada de FOREST, entonces el 'parser' llama a las rutinas generadoras de código que corresponden a tal palabra. Si no es así, entonces se llama a una subrutina que copia (a partir de la columna 7) lo que se encuentre, suponiendo que se trata de FORTRAN "puro".

Las etiquetas de FORTRAN se reconocen también como una especie de palabras "clave" (i.e., sólo números, y no más de 5) y se copian a la salida , pero a la columna 1. Si la etiqueta acompaña a una instrucción "pura" entonces se copia todo igual a la salida; p. ej., 125 A = B * D se copia igual, pero alineando a las columnas 1 y 7.

Si lo que se lee es, por ejemplo, 125 WHILE(C = D) ....

entonces se genera

```
125   CONTINUE
77715 IF( .NOT. ( C = D ) ....
```

esto es, se respetan las etiquetas que acompañan a instrucciones de FOREST, acompañándolas de un CONTINUE. La razón de esto es que probablemente la instrucción de FOREST genere internamente otra etiqueta, pudiendo presentarse el peligro de

generar 125 77715 IF(.NOT(..... que, para FORTRAN, no tendrá ningún sentido.

Ya en este momento surgen varias observaciones:

La necesidad de invertir la condición de todas las proposiciones que tengan, en la gramática, la parte -condición- y,

la necesidad de generar internamente etiquetas de control desconocidas (e inaccesibles) para el programador.

Si el lector observa el ejemplo de la pág. 19 que se refiere a la proposición IF sencilla verá que la condición A(I) .NE. BETA aparece invertida en la traducción a FOREST. Esto se debe a que así podemos seguir leyendo y traduciendo renglón por renglón, sin necesidad de acordarnos de instrucciones ya leídas pero aún no "expandidas".

Esto se hará más claro posteriormente, cuando demos los ejemplos completos de las "acciones".

La otra observación se deduce de leer todos los ejemplos que hemos venido dando, pero al revés: primero leer el código FOREST y luego leer la traducción a FORTRAN. Porque en FOREST no utilizamos etiquetas y en FORTRAN sí, se hace necesario generar internamente las que se vayan requiriendo.

Una última observación: como permitimos el uso de punto y coma para separar "renglones" surge la necesidad imperiosa de controlar su uso dentro de la notación Hollerit de FORTRAN. Si escribimos en FOREST, ALFA = 7HALGO;ES por ejemplo, el preprocesador entenderá que la frase ES pertenece a otro renglón, cuando lo que se quiere es algo muy diferente (hacer la variable ALFA igual a una constante Hollerit de siete caracteres). Éste es un problema grave, y tiene dos soluciones: prohibir el uso de punto y coma dentro de la notación Hollerit o bien, darle la vuelta.

FOREST le da la vuelta al problema exigiendo usar la notación 'CUALQUIER COSA' en lugar de 14HCUALQUIER COSA . Evitamos así el problema del punto y

coma y le ahorramos al programador la molestia de tener que contar caracteres. Por supuesto, FOREST tendrá que contar los caracteres internamente; pero esa es precisamente una de sus funciones.

Así pues, si en FOREST leemos ALFA = 'ALGODON' entonces se invocará una "acción" que convertirá esto en ALFA = 7HALGODON. Se sigue la convención de representar un apóstrofo por dos pegados.

Se traduce a notación Hollerit para asegurar máxima portabilidad al pseudo-compilador: i.e., no todos los FORTRAN aceptan la notación de apóstrofos.

Las "acciones" son las siguientes, manejadas aquí como macro-expresiones, que se "expandirán" cuando el 'parser' encuentra una palabra clave al comienzo de un "renglón" de FOREST.

if (-condición-) PROPOSICIÓN1 else PROPOSICIÓN2

se convierte en:

```
        IF(.NOT. (-condición-) ) GO TO  -e_i-
        PROPOSICIÓN1
        GO TO  -e_i+1-
-e_i-   CONTINUE
        PROPOSICIÓN2
-e_i+1- CONTINUE
```

$-e_i-$ es una etiqueta que debe ser generada internamente por el preprocesador. Se escogieron etiquetas internas de cinco cifras, a partir de 77700. Constituye un error el tratar de utilizar una tal etiqueta en un programa FOREST.

while (-condición-) PROPOSICIÓN

se convierte en:

```
-e_i-   CONTINUE
        IF( .NOT. (-condición-) ) GO TO -e_i+1-
        PROPOSICIÓN
        GO TO -e_i-
-e_i+1- CONTINUE
```

for (-inicializa- ; -condición- ; -reinicializa-) PROPOSICION

se convierte en:

```
        -inicializa-
-e_i-   CONTINUE
        IF( .NOT. (-condición-) ) GO TO -e_i+1-
        PROPOSICIÓN
        -reinicializa-
        GO TO -e_i-
-e_i+1- CONTINUE
```

repeat PROPOSICIÓN until (-condición-)

se convierte en:

```
-e_i-   CONTINUE
        PROPOSICIÓN
        IF( .NOT. (-condición-) ) GO TO -e_i-
-e_i+1- CONTINUE
```

En este caso, la etiqueta $-e_{i+1}-$ sirve para un posible BREAK en PROPOSICIÓN.

```
do -límites- PROPOSICIÓN
```

se convierte en:

```
        DO -e_i- -límites-
        PROPOSICIÓN
-e_i-   CONTINUE
-e_i+1- CONTINUE
```

Otra vez, la etiqueta adicional sirve para un posible BREAK.

```
case -identificador1- ;

    -id2-: PROPOSICIÓN2
    -id3-: PROPOSICIÓN3
      .
      .
      .
    -idn-: PROPOSICIÓNn
end
```

se convierte en:

```
        IF (-identificador1-  .NE.  -id2- ) GO TO  -e_i-
        PROPOSICIÓN2
        GO TO  -e_m-
-e_i-   CONTINUE
        IF (-identificador1-  .NE.  -id3- ) GO TO  -e_i+j-
            PROPOSICIÓN3
        GO TO  -e_m-
          .
          .
          .
          .
          .
          .
-e_i+j- CONTINUE
        IF (-identificador1-  .NE.  -idn- ) GO TO  -e_k-
        PROPOSICIÓNn
        GO TO  -e_m-
-e_k-   CONTINUE
-e_m-   CONTINUE
```

En este caso hay que acordarse de varias etiquetas, no sólo de una, y del -identificador1-, ya que se permite que la PROPOSICIÓN sea a su vez un CASE.

Esto mismo sucede en el FOR, donde se permite que la PROPOSICIÓN sea a su vez otro FOR, lo que obliga a guardar el código -reinicializa-.

De la misma manera, el BREAK y el NEXT tienen que "recordar" a dónde deben hacer el GOTO, para lo cual usan una pila de etiquetas.

No se hace ningún intento por eliminar varios CONTINUEs que estén seguidos, como en el ejemplo anterior, por considerarlo demasiado complejo, ya que habría que alterar también las correspondientes etiquetas en los GOTOs.

FOREST reconoce los comentarios (y los ignora) si êstos van precedidos del símbolo " ! ".

Por ejemplo en

```
WHILE(  ALFA = BETA )  ! esta es una iteración condicional.
   CALL DOS
```

Puede haber varios comentarios en un renglón, o intercalados entre las proposición, si terminan en punto y coma.

Por ejemplo:

```
    DO  ! este es el DO de FOREST; I = 1,25  !los comentarios se ignoran
```

es equivalente a `DO  I = 1,25`

Obsérvese que el uso del punto y coma en medio de un comentario está prohibido, porque entonces no se sabría donde termina.

4.- DISEÑO DEL PROGRAMA FORTRAN

Como dijimos anteriormente, nuestro 'parser' simula recursividad a través del uso de una pila donde se guardan las direcciones a las que habrá de regresarse cuando se terminó de procesar una DECLARACION.

Las dos rutinas que se encargan de esto se llaman PUSH y POP y manejan una pila común.

El programa principal se llama PARSER y no es más que la traducción directa del pseudo-código expuesto en la pág. 16. Cada que encuentra una "acción" llama a la subrutina que la genera y luego vuelve al ciclo principal, habiendo antes guardado en la pila la dirección (etiqueta) donde estaba. El direccionamiento a las "acciones" se hace a través de un GOTO computado, dirigido por una rutina que reconoce las palabras claves y sus posiciones.

El regreso recursivo se simula por otro GOTO computado, dirigido éste por la etiqueta al tope de la pila.

FOREST funciona así:

1.- Se lee el siguiente componente sintáctico ('token') del archivo de entrada, por medio de la rutina OTRO. Esta rutina, a su vez, llama a COMP para que lo obtenga. COMP llama a TENCAR para ir recogiendo, carácter por carácter, los componentes sintácticos. En el momento en que COMP ha obtenido un 'token' completo (i.e., cuando TENCAR le entrega un blanco o un símbolo que no es letra ni número)entonces OTRO decide qué tipo de 'token' es: alfanumérico, etiqueta, o símbolo suelto (p. ej. punto y coma).

Por medio de llamadas a subrutinas explicadas más adelante, OTRO averigua si es palabra clave y direcciona al GOTO computado al comienzo de la rutina que

generará la "acción" correspondiente en PARSER.

La rutina CLAVE decide si una palabra es clave o no. Lo hace por medio de una búsqueda binaria (Ref. 15) en una tabla prealmacenada de palabras clave. Suma los caracteres de la palabra en forma numérica y obtiene un factor numérico que luego tratará de encoentrar en la tabla. Si lo encuentra, entonces devuelve la dirección de la "acción"; si no, entonces supone que se trata de una declaración de FORTRAN "puro", por lo cual simplemente se copiará al archivo de salida.

2.- Una vez generado el código FORTRAN correspondiente a la "acción" se procede a terminar ese nivel recursivo. Se hace un PUSH en la pila (guardando la dirección actual) y se procede a continuar leyendo el siguiente componente sintáctico mayor. Esto es, se pide leer la siguiente PROPOSICION.

Es claro que las "acciones" significarán múltiples llamadas a las rutinas que devuelven componentes sintácticos, pero no serán componentes "mayores", es decir, inicio de proposición en el sentido de que no son PROPOSICIONes sino tan sólo -condiciones- o -límites- o -inicializa- etc. Conviene en este punto repasar la gramática, expuesta en la pág. 18.

3.- Cuando, finalmente y después de los "nudos" recursivos, se llega a un end del pseudocódigo entonces es hora de volver a la última de las invocaciones recursivas. Esto se hace por medio de un POP en la pila, que direccionará al 'parser' a donde se había quedado por última vez.

4.- Cuando se lee un END que no tiene su correspondiente BEGIN simulado en la pila de recursividad entonces se ha llegado al final. En este momento termina la generación de código y el preprocesador ha terminado de trabajar.

Esto significa que sólo puede haber un END que no sea de FOREST: el END de fin de programa que todo programa FORTRAN (y por ende, FOREST) debe llevar.

Pero suele suceder que se compilan varias subrutinas juntas donde no hay separadores ni tarjetas de control entre ellas. Por ejemplo:

```
SUBROUTINE  UNO
  .
  .
  .
  .
RETURN
END
SUBROUTINE  DOS
  .
  .
  .
  .
RETURN
END
```

etc. En estos casos FOREST protestaría al encontrar el segundo END. Para evitar esto utilizaremos un truco ingenioso que aprendimos de FLECS (Ref. 2) y que consiste sencillamente en escribir.

```
E   ND
```

en lugar de

```
END
```

Los espacios en blanco engañarán a FOREST, que de esta manera no protestará por un END sin su correspondiente BEGIN, pues no lo reconocerá y lo copiará tal cual a la salida. El compilador FORTRAN, por otro lado, reconocerá E ND como el legítimo terminador de programa.

No es lugar aquí para entrar en más detalles de cómo se hace el 'parsing' sobre una gramática. Referimos al lector a los textos sobre compilación (Refs. 8, 9, 22).

Se ha incluido un conjunto de mensajes de error referentes a FOREST. Hay que hacer mención, sin embargo, que una vez que un programa FOREST ha sido traducido a FORTRAN, de ninguna manera se asegura que va a trabajar sin problemas. Le toca

ahora al compilador FORTRAN de la instalación en particular, compilar lo que el preprocesador generó. Es perfectamente posible que haya errores que FORTRAN reclame y que FOREST no. Si escribimos RETUN en lugar de RETURN, por ejemplo, FOREST supondrá que se trata de una palabra de FORTRAN, pero el compilador marcará un error.

FOREST sólo verifica lo que a él concierne; esto es, que la sintaxis esté correcta; pero la sintaxis de las extensiones de la gramática de FOREST, que es mucho más restringida que la de FORTRAN.

En general, se ha tratado de que el preprocesador se "recupere" de la mayor parte de los errores que encuentre. Si un FOR está mal construido, por ejemplo, ignorará el renglón, pero tratará de generar código para el siguiente. El preprocesador sólo es "derrotado" por el error **FIN DE ARCHIVO PREMATURO**: cuando se ha llegado al fin del archivo de entrada sin haber terminado de vaciar la pila de recursión.

Se han tratado de evitar las "cadenas de errores" que surgen cuando un error previo hace que proposiciones correctas aparezcan como inválidas. Esto se logra manteniendo el rango de todas las proposiciones de FOREST, de tal manera que la estructura sintáctica se mantenga lo más robusta posible.

Para ayudar al programador en su trabajo, FOREST genera números de línea secuenciales del programa fuente y una especie de referencia cruzada a los renglones de código FORTRAN generados internamente. De esta manera, si el compilador FORTRAN señala error en la línea 138, se podrá buscar en el listado fuente de FOREST y ver a cuál línea fuente corresponde (por regla general corresponderá a una línea menor numéricamente en FOREST, ya que una instrucción en FOREST equivale a varias de FORTRAN.

Esta será una referencia aproximada, que se refiere a un grupo reducido de renglones de FORTRAN generados y no a uno en particular. Aún así, es de gran utilidad. Siempre hay que buscar en la referencia cruzada que sigue al renglón FOREST en particular.

Más adelante exhibimos listados de corridas de FOREST que demuestran todo esto.

Mostramos ahora la estructura jerárquica de las 39 subrutinas que componen FOREST:

```
FOREST
        PARSER
        CARGA
                OTRO    EOFPRE
                        COMP
                        NUMER
                        CLAVE
                                TENCAR
                                TIPO
                                REGRES

                INICIA
                PUSH
                POP
                GENET
                QUITET
                GENSAL
                CODIF
                        SACAIF
                        COND
                                SACREL
                                SACA

                SACONT
                SAGOTO
                CODFR1
                        SACTOK

                CODFR2
                        SIGREN

                CODDO
                        CARDEC
                                CONV

                        COPIA
                                HOLLER
                                EXPR

                CODCAS
                        LIMPIA

                MUEVE
                ERROR
                        FIN
```

Y este es un directorio de las subrutinas, en orden alfabético:

CARDEC (LBL) : Convierte las etiquetas numéricas de las pilas en etiquetas de caracteres numéricos para el archivo de salida.

CARGA : Es un BLOCK DATA usado para "inicializar" arreglos de variables.

CLAVE (BI, KWORD) : Averigua si un 'token' alfanumérico es palabra clave o no. Si lo es entonces devuelve la dirección de la "acción" que debe ser generada en la variable KWORD.

CODCAS (NCASO, TOKEN, BI): Rutina que simula la "acción" del CASE y genera "código". NCASO controla el manejo de las "etiquetas" del CASE.

CODDO (LBL) : Rutina que simula la "acción" del DO y genera "código".

CODFR1 (L1, L2, CASO) : Rutina que procesa la parte -inicialización- y -condición- del FOR. Guarda además en un diccionario las partes -reinicialización- del FOR para su posterior expansión. L1 y L2 son etiquetas que imprimirá. CASO maneja las posibles variedades de FOR (sin consición, sin -inicio-, etc.).

CODFR2 : Saca la parte -reinicialización- del diccionario donde fue guardada por CODFR1. Esto se hace cuando se ha regresado recursivamente de los FOR anidados.

CODIF (LBL) : Rutina que simula la "acción" del IF y genera código GO TO -LBL-.

COMP (BI) : Devuelve el siguiente componente sintáctico simple del archivo de entrada (puede ser alfanumérico, numérico o un símbolo suelto). Ignora blancos y comentarios. El componente tiene BI-1 caracteres. El último carácter es el "fin de componente".

COND (BAL, CASO) : Procesa las partes -condición- de la gramática. Va generando las expresiones condicionales de FORTRAN, y lleva la cuenta de los paréntesis y los fines de renglón. BAL controla el balanceo de los paréntesis. CASO avisa situaciones especiales (p. ej., condición nula).

CONV (NUM, ARR) : Convierte un número a su representación de caracteres; es llamado por CARDEC. Se necesita para llenar el 'buffer' de salida. Deja la representación en el arreglo ARR(5).

COPIA (BI) : Copia a la salida aquellos componentes sintácticos que se estima son de FORTRAN "puro". Cuida la columna 7. Logra la impresión del renglón cuando detecta punto y coma. BI controla las continuaciones automáticas a nuevo renglón.

EOFPRE : Rutina que maneja el caso de fin de archivo prematuro.

ERROR (NUM) : Rutina de mensajes de error de FOREST. Imprime el NUM-ésimo mensaje de error.

EXPR (TEMP) : Rutina que procesa las expresiones del tipo ++var y --var. TEMP avisa si hubo error.

FIN : Rutina que hace el cálculo de las líneas leídas y los errores generados al terminar una corrida.

FOREST : Programa principal. Llama a PARSER y termina al acabar éste. Aquí pueden incluirse una serie de comandos cercanos al sistema operativo particular, que manejen los archivos donde FOREST lee/escribe en disco, etc. Por lo pronto la rutina FOREST pregunta al usuario (interactivo) si quiere leer de disco o escribir en disco. Esto variará de acuerdo a si el preprocesador se usa en forma 'batch' o interactivamente.

El sistema lee de un archivo direccionado por la variable IDISC. Si IDISC vale 5 entonces leerá del teclado de la pantalla. Si se quiere leer el programa fuente de otra unidad lógica, habrá que cambiar el valor de IDISC.

El sistema escribe la copia del programa fuente (con los diagnósticos de error) en la unidad lógica IMP.

En la rutina INICIA se colocó la variable UL, que controla la unidad lógica donde se escribirá el "código objeto" FORTRAN generado. Si se desea, se puede traer también a la rutina FOREST. Se hizo originalmente así por requerimientos del microcomputador Z2D donde se comenzó a desarrollar el preprocesador.

GENET (N) : Genera N etiquetas secuenciales (internas) a partir del número 77700 -modificable- y lleva el control de la pila formada por ellas.

GENSAL (N) : Selecciona algunas de las etiquetas y las mete en dos pilas de control para los casos BREAK y NEXT con diversos grados de anidamiento.

HOLLER (I) : Convierte una cadena entre apóstrofos a su representación en Hollerit, para lo cual cuenta los caracteres (serán I).

INICIA : Hace las diversas "inicializaciones" de las variables de control de FOREST. Se llama una vez por proceso.

LIMPIA : Limpia el renglón de salida para evitar la generación de código espurio cuando se encontró un error a medias de una "acción".

MUEVE (N) : Mueve a la columna N de FORTRAN el renglón de salida antes de llenarlo e imprimirlo.

NUMER (BI) : Averigua si un componente que mide BI-1 caracteres es estrictamente numérico o no.

OTRO (TOKEN, BI, CASO) : Regresa el siguiente 'token' a la rutina PARSER. Averigua su longitud y genera el "fin de componente"; la longitud + el "fin de componente" se llama BI. CASO es para detectar fin de archivo prematuro.

PARSER : Rutina principal. Simula estrictamente el pseudo-código recursivo que hace el 'parsing'.

POP (R) : "Levanta" la pila de direcciones. Se llama luego de un end del pseudocódigo recursivo. Devuelve el control a donde se había quedado antes de la última invocación recursiva; o sea, la dirección 'R'.

PUSH (R) : Lo contrario de POP. Guarda la dirección actual 'R' antes de hacer una llamada recursiva a PARSER. Las direcciones son etiquetas en FORTRAN.

QUITET (N, CASO) : Quita N etiquetas de la pila (generadas por GENET) cuando ya no se requieren. Lleva el control del tope de pila. CASO sirve para quitar etiquetas de las pilas BREAK y NEXT.

REGRES : "Regresa" el último carácter leído, cuando ya se ha identificado un 'token'. Trabaja simbólicamente sobre un apuntador al renglón de entrada.

SACA (CAR) : "Saca" un carácter al 'buffer' único de salida. También maneja un apuntador. Cuando el renglón se llena entonces ordena su impresión.

SACAIF : "Saca" " IF(.NOT. (" al 'buffer' de salida. Llamada por CODIF.

SACONT (LBL) : "Saca" "-etiqueta- CONTINUE " al 'buffer' de salida.

SACREL (NUM) : "Saca" los equivalentes FORTRAN de las condicionales de FOREST (i. e., ".EQ." en lugar de "="). Procesa el relacional NUM-ésimo.

SACTOK (BI) : "Saca" un 'token' al 'buffer' de impresión. Es llamado cuando se quiere copiar un componente solo de FORTRAN "puro" que mide BI caracteres-1.

SAGOTO (LBL) : "Saca" " GO TO -etiqueta- " al 'buffer' de impresión.

SIGREN : Llamada por SACA, imprime el siguiente renglón de código FORTRAN generado. Controla líneas de continuación y maneja el 'buffer' de impresión.

TENCAR (TEMP) : Rutina de bajo nivel que pide leer un renglón cada que el 'buffer' de entrada está vacío. Hace lo contrario que SACA y SIGREN. Controla fin de archivo. Devuelve el siguiente carácter cada que le es solicitado en la variable TEMP.

TIPO (C) : Determina el tipo de un carácter leído: si es numérico o alfabético.

Mencionaremos ahora aquellas Partes de FOREST que son dependientes de detalles de alguna máquina en particular.

Se ha tratado de hacer el preprocesador lo más transportable posible, es por esto que está escrito en FORTRAN, sin usar apóstrofos ni otros detalles no demasiado comunes.

Hay, sin embargo, una parte "débil" en este sentido. La búsqueda binaria, como dijimos, toma los valores numéricos de cada carácter ASCII y los va sumando, afectados de un multiplicador, para obtener un identificador numérico (no necesariamente único) que luego tratará de encontrar en una tabla, por medio de una búsqueda binaria. No todas las máquinas tienen la misma representación interna de los caracteres ASCII, porque esto depende de cómo distribuyan los siete bits ASCII en la palabra de máquina. En el microprocesador Z80 donde se comenzó este trabajo, como la palabra es de 8 bits, no hay ambigüedad alguna. Pero en otras máquinas (en la Hewlett-Packard 3000 donde se terminó de probar los programas) hay varias maneras de representar los 7 bits.

Lo que cambiará de máquina a máquina entonces, será la manera de representar (en el BLOCK DATA) la tabla de palabras claves en forma de caracteres. Se había escogido la forma 1HL para representar la letra "L", por ejemplo, pero hay algunos compiladores FORTRAN que colocan el valor numérico de la letra en la parte izquierda (no en la derecha) de la palabra, resultando así una incompatibilidad con los valores previamente establecidos en la tabla de valores numéricos de las palabras clave.

Esto obligó a representar este arreglo (en el BLOCK DATA) de una manera tal que justifique a la derecha y no a la izquierda, en el caso particular de la HP-3000 usando %'L' en lugar de 1HL.

Si se va a llevar FOREST a otra máquina, simplemente hay que asegurar que los caracteres que se manejen internamente y que se lean del archivo de entrada, se justifiquen a la derecha de la palabra de máquina; hay que cambiar de la forma %'L' a la forma 'L' o 1HL en todos los DATA de FOREST. En la HP-3000 también, se

usó el formato 1R1, en lugar de 1A1 por las razones antes mencionadas.

Diremos que, con esta simple modificación, FOREST ha "corrido" bien en dos computadoras hasta la fecha: una micro Z2D (basada en el Z80) y una Hewlett-Packard 3000 (de 16 bits).

Finalmente, y refiriéndonos al mismo problema, a la hora de hacer la suma numérica de los caracteres ASCII, hay algunos compiladores que le suman a cada valor numérico interno ASCII una cierta constante (en el caso del Z2D es el número 8192) por razones desconocidas para mí. La rutina CLAVE simplemente resta esta constante a los valores antes de hacer las sumas.

Es muy fácil averiguar el valor de esta constante (que muchas veces será cero): se usa un programita como el siguiente:

```
100   CONTINUE
      READ(5, 110)  LETRA
110   FORMAT(1A1)
      WRITE(6, 120)  LETRA, LETRA,LETRA
120   FORMAT('   LETRA LEIDA = ' ,1A1, '  VALOR NUMERICO = ' ,16,
     $        '   VALOR OCTAL INTERNO =  ' , 04 )
      GO TO 100
```

Si el compilador no trabaja con formatos octales, entonces bastará con los dos primeros.

Con sólo leer la letra "A" (VALOR NUMERICO = 61) podremos determinar el valor de esta constante, si obtenemos un número mayor que 61.

De la misma forma, se podrá variar el formato 1A1 por algún otro similar (p.ej. 1R1).

5.- CONCLUSIONES

Creemos, aunque sin prueba alguna más que la lectura de las especificaciones de otros preprocesadores (Refs. 11, **6**, 2), que FOREST es un preprocesador versátil y transportable, aunque tal vez no exento de errores o particularidades de comportamiento.

Se trató de aplicar la técnica de "programación de arriba-abajo" en el desarrollo de este trabajo y casi todas las rutinas están razonablemente estructuradas, con la excepción de una o dos que tienen partes obscuras. (COND es una de ellas: sólo la falta de tiempo evitó que se rehiciera).

No creemos sea tan difícil hacerle extensiones a FOREST; los listados completos de las rutinas aparecen al final de este trabajo y son moderadamente entendibles, ya que no claros como el agua. El único problema sería encontrar representaciones internas de nuevas palabras clave que fueran diferentes de las ya utilizadas en el DATA DIC, para que la búsqueda binaria no fracase.

Posibles extensiones podrían contemplar la adición de un macro-procesador integrado en forma de palabras clave nuevas. Por ejemplo, MACRO podría ser una palabra clave para definir macros, etc.

Tal vez más importante aún sería la adición de rutinas de monitoreo de frecuencias de ejecución ('profiler') o de 'debugging' o 'trace'. Esto podría constituir un proyecto interesante que daría a FOREST más potencia como herramienta pedagógica. Podrían ser integradas en el PARSER como "acciones" adicionales de la gramática.

Se han respetado hasta dos símbolos (definibles por el usuario) como símbolos especiales del compilador (de tarjetas de control por ejemplo) que se copiarán a la salida a partir de la columna uno, para permitir comandos de compilador. Si uno de estos símbolos es el 'slash' '/', entonces un renglón que comience con este símbolo NO se supondrá como continuación del anterior.

FOREST genera siempre el primer renglón de "código" FORTRAN en la forma de un comentario (una 'C' en la columna 1). Si llegara a estorbar a algún comando de compilador, que debiera ser el primer renglón físicamente del "código" generado, entonces habría que eliminar este renglón de comentario. Ver la rutina INICIA para solucionar esto.

El objetivo principal fue y sigue siendo crear una herramienta adecuada para enseñar programación "estructurada".

He utilizado estas técnicas (e inclusive partes de FOREST mismo) en los cursos de programación a mi cargo con buenos resultados: mucho mejores que si se utiliza sólo FORTRAN.

Es necesario volver a decir que la motivación de este trabajo fue el hecho de que FORTRAN sea lo más cercano a un lenguaje universal de la programación "científica", y que es enseñado en casi todas las carreras de ingeniería en el país. Considero que, aún al costo de doble tiempo de compilación, vale la pena intentar que los alumnos y programadores obtengan mejores resultados en la práctica diaria de la programación y no se desesperen por las faltas que FORTRAN comete en contra de la estructura de los programas.

Si FOREST sirve para hacer más claro el estilo de programar, y por lo tanto para hacer notar que la programación puede ser interesante y aún bella, entonces estaremos satisfechos.

Aunque no hay que olvidar que el futuro (tal vez lejano) de la programación apunta por otros caminos:

"Mi propia opinión acerca del efecto de FORTRAN en los lenguajes posteriores y el impacto colectivo de tales lenguajes en la programación no constituye, generalmente, una opinión popular. Este punto de vista es el tema de un artículo más largo (Ref. 1).

Ahora considero a los lenguajes convencionales (p. ej., los FORTRANs, los ALGOLs y sus sucesores y derivados) como elaboraciones crecientemente complejas del estilo de programación dictado por la máquina de Von Neumann. Estos lenguajes Von Neumann

crean obstáculos intelectuales enormes y complejos en el pensar acerca de los programas y en el crear las formas combinatorias de alto nivel que se requieren en una metodología de la programación realmente poderosa. Los lenguajes de Von Neumann constantemente tienen nuestras narices pegadas en el suelo del cálculo de direcciones y de las computaciones separadas de palabras sueltas, mientras que deberíamos estar concentrados en la forma y el contenido del resultado global que estamos tratando de producir.

Hemos llegado a suponer el DO, FOR, WHILE y demás, como herramientas poderosas, mientras que sólo son débiles paliativos que son necesarios para lograr que se sostenga el estilo primitivo de la programación de Von Neumann.

Al hacer la división entre el mundo de las expresiones y el de las instrucciones, los lenguajes de Von Neumann evitan el uso efectivo de formas de combinación de más alto nivel.

La programación estructurada puede ser vista como un modesto esfuerzo para introducir una pequeña cantidad de orden en el caótico mundo de las instrucciones. (....) " [1]

[1] John Backus, "The History of FORTRAN I, II and III", en ACM SIGPLAN Notices, Vol. 13, No. 8, Aug. 1978.

6.- BIBLIOGRAFIA COMENTADA

1.- J. Backus, "Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs", CACM, Vol. 21, No. 8, Aug. 1978.

El nombre de John Backus es conocido para todo aquél aún medianamente asociado con la computación. En 1977 la ACM (Association for Computing Machinery) le otorgó el premio anual "Alan M. Turing", y éste es el discurso técnico en honor al premio. Artículo interesantísimo que habla sobre la filosofía de las máquinas de Von Neumann y los lenguajes asociados con ellas, sus ventajas y sus fallas (entre ellas el "cuello de botella de Von Neumann"). Backus propone un álgebra ("aplicativa") que libere a la computación de tales sufrimientos Neumannianos. Hay que recordar que Backus es uno de los principales responsables del surgimiento de los lenguajes "tradicionales" (FORTRAN, ALGOL, PL/I, etc.) y ahora, 20 años más tarde propone nuevos caminos.

2.- T. Beyer, FLECS, Computing Center, Univ. of Oregon, USA, 1975.

Este es un preprocesador escrito para las máquinas de la serie PDP-11. Partes de él están codificadas en ensamblador, por lo cual se ve grandemente reducida su transportabilidad. Existe también una versión para la IBM 370.
Este preprocesador no es de formato libre, pero tiene una característica que lo hace atractivo, sobre todo para los programadores acostumbrados a COBOL: permite la definición de procedimientos que pueden ser invocados por su nombre en cualquier punto del programa. Es algo muy parecido al PERFORM de COBOL.

5.- R. S. Bird, "Notes on Recursion Elimination", CACM, Vol. 20, No. 6, Jun. 1977.

Artículo mucho más general que el anterior, en el sentido de que propone tres métodos generales para eliminar recursión basado en la descripción matemática del programa recursivo original. En el caso general se hace uso de pilas para eliminar la recursividad. Este artículo está basado en una sugerencia de Knuth, ver ref. 16.

4.- C. Böhm & G. Jacopini, "Flow Diagrams, Turing Machines and Languages with only two Formation Rules", Communications of the ACM (CACM), Vol. 9, No. 5, May. 1966.

Artículo teórico que muestra cómo normalizar diagramas de flujo para convertirlos en (descomponerlos) gráficas dotadas de dos reglas de formación. Lo que se logra con esto es demostrar que se puede obtener una máquina de Turing generada por composición e iteración que sea funcionalmente equivalente a cualquier máquina de Turing de dos vías.
Este artículo es muchas veces citado como base de la programación "estructurada", pero formalmente no tiene mucho que ver con ella. Es un artículo teórico presentado en un coloquio internacional sobre lingüística y autómatas (Jerusalén, 1964).

5.- P. J. Brown, Macro Processors and Techniques for Portable Software, Wiley, N.Y., 1974,

Buen libro sobre técnicas generales de macros.
Discute varios modelos teóricos y prácticos de macro-procesadores. El autor (Brown) ha tenido mucha experiencia y es de los pioneros en este campo. La segunda parte del libro trata sobre cómo lograr "software' transportable por medio de macro-definiciones.

6.- D. Comer, "MOUSE4: an Improved Implementationof the RATFOR Preprocessor", Software-Practice and Experience, Vol. 8, 35-40, 1978.

Después de hacer comentarios muy positivos sobre RATFOR (muy adecuado para la programación, muy transportable, etc.) el autor hace una lista de sus desventajas: hace búsquedas lineales, que son ineficientes; obliga a reinsertar caracteres al archivo de entrada para releerlos si el proceso así lo requiere, etc. Finalmente, explica cómo MOUSE4 resuelve estos problemas.

7.- O. Dahl, E. Dijkstra & C. Hoare, Structured Programming, Academic Press, N.Y., 1972.

Libro compuesto de tres secciones, a nosotros nos interesa la primera, escrita por Dijkstra, y que lleva el nombre de "Notes on Structured Programming". Aquí el autor menciona los problemas relacionados a nuestra "incapacidad para hacer mucho" y para entender y probar la correctitud de los programas. Propone varios esquemas para diagramas de flujo y da un ejemplo completo del sistema para generar programas correctos. Artículo un poco oscuro (tal vez debido a "mis fricciones con el idioma inglés") y hasta un poco rebuscado.

8.- J. M. Foster, Automatic Syntactic Analysis, Macdonald, London, 1970.

Librito sencillo sobre análisis sintáctico. Foster propone el concepto de "acciones" como una manera de agilizar el 'parsing'. Tales "acciones" se introducen en la especificación formal de la gramática y se entrena al 'parser' para que las reconozca y llame las rutinas apropiadas en los momentos apropiados de la posterior compilación.

9.- D. Gries, Compiler Construction for Digital Computers, Wiley, N.Y., 1971.

Buen libro sobre "todo acerca de compiladores". Comienza con un repaso sobre gramáticas formales (preferimos el de (Ref, 10)) y pasa a construir diversos tipos de reconocedores. Tiene interesantes referencias históricas. Propone diversos programas para ejemplificar la construcción de las diferentes partes del compilador.

10.- C. Hopcroft & J. Ullman, Formal Languages and their Relation to Automata, Addison-Wesley, N.Y., 1969.

Excelente libro sobre gramáticas formales.
Tratado todo a un muy buen nivel, explica la teoría de las gramáticas y las va relacionando con los autómatas, generadores y reconocedores, aunque no ataca los problemas de 'parsing'.
Hace un recuento completo de la jerarquización de Chomsky sobre las gramáticas, y los autómatas asociados con ellas, desde el autómata finito hasta la máquina universal de Turing.

11.- B. Kernighan, "RATFOR -A Preprocesor for a Rational Fortran", Software-Practice and Experience, Vol. 5, 395-406, 1975.

Artículo introductorio al preprocesador RATFOR, a nivel de usuario y sin mayor explicación de su estructura. RATFOR está mucho mejor explicado en un libro posterior, ver referencia 13.
Éste es el preprocesador en el que está basado FOREST.

12.- B. Kernighan & P. Plauger, The Elements of Programming Style, McGraw-Hill, N. Y., 1974.

Librito fácil de leer y seguir. Manual de estilo que trata de seguir a un homólogo literario. Consiste en una serie apabullante de ejemplos tomados de libros de texto de programación elemental que muestran cómo NO hay que enseñar a programar, junto con un conjunto de"reglas" de estilo. Buen libro de introducción, pero alejado de formalismos y teoría.

13.- B. Kernighan & P. Plauger, Software Tools, Addison-Wesley, N. Y., 1976.

Este es el libro que trae el diseño del preprocesador RATFOR, y por lo tanto, es la referencia principal. Hay que notar que es un libro difícil, ya que muestra, con ejemplos reales, un conjunto de "herramientas" de software que van desde las más sencillas (un programa que cuenta caracteres) hasta las más sofisticadas (RATFOR).
RATFOR, sin embargo, como dijimos antes, no sigue una técnica canónica, y eso lo vuelve difícil de entender en su funcionamiento. FOREST difiere de él en muchos aspectos, sobre todo constructivos, ya que no aparentes.
El libro contiene más de 150 subrutinas diversas, que los autores llaman "herramientas".

14.- B. Kernighan & D. Ritchie, The C Programming Language, Prentice-Hall, New Jersey, 1978.

Libro interesante sobre un lenguaje interesante. El lenguaje "C" fue producido para el sistema operativo UNIX de las máquinas PDP11 y actualmente se está extendiendo su uso debido a su facilidad de manejo. No es un lenguaje tan "formal" como PASCAL por ejemplo, pero tiene muchas buenas ideas en su diseño. FOREST se apropió de varios conceptos de aquí, algunos de los cuales ya estaban descritos en RATFOR.

15.- Donald Knuth, The Art of Computer Programming, Addison-Wesley, Mass. Vol. III, 1972.

No podíamos dejar de mencionar, aunque sea sólo con el pretexto de una simple búsqueda binaria, estos libros. Son de sobra conocidos, así que no hay mucho más que decir.

Sólo mencionaremos que Knuth confesó, ocho años después, en 1976, que su idea de la enciclopedia estaba teniendo serios problemas debido a su cercana dependencia con MIX, que es un lenguaje ensamblador, no-recursivo y sin estructuras de datos. Aún así, estos libros siguen siendo fundamentales.

16.- D. E. Knuth, "Structured Programming with GOTO Statements" Computing Surveys, Vol. 6, No. 4, Dec. 1974.

Artículo largo y complejo ("me costó cientos de horas hombre de trabajo") que trata de eliminar algunos prejuicios sobre la finalidad de la "programación estructurada". De manera sagaz y burlona Knuth hace una buena exposición sobre la programación moderna y se refiere a su inconclusa "Enciclopedia" y hace reflexiones graves sobre ella al hablar del porqué está basada en un lenguaje ensamblador como MIX, mismo que corre peligro de sentirse acomplejado por PASCAL por ejemplo. Lo que a nosotros nos interesa en lo que se refiere a la recursión lo menciona casi de pasada en un punto del artículo: la conveniencia -o necesidad- de prescindir de la recursión en algunos casos y el porqué.

17.- C. McGowan & J. Kelly, Top-Down Structured Programming Techniques, Petrocelli-Charter, N.Y., 1975.

Este libro enseña programación estructurada, y está dividido en tres secciones: primero habla sobre la programación estructurada, claro y con ejemplos; luego dedica un capítulo a un esfuerzo moderno de programación "Top-Down" (la base de

datos del periódico "New York Times") y termina con un ejemplo de un preprocesador para procesos concurrentes en PL/I. Buen libro, claro y amplio.

18.- W. Peterson, T. Kasami & N. Tokura, "On the Capabilities of While, Repeat and Exit Statements", CACM, Vol. 16, No. 8, Aug. 1973.

Artículo que busca explorar los límites de las reglas de composición mencionadas y propone probables extensiones. Hace uso de la teoría de gráficas para demostrar que un programa estará bien formado si se compone de ciertas estructuras de control (If, Repeat y "Exit de varios niveles").

19.- J. S. Rohl, "Converting a Class of recursive Procedures into Non-Recursive Ones", Software-Practice and Experience, Vol. 7, 231-238, 1977.
Artículo práctico de cómo convertir un conjunto restringido de programas recursivos en correspondientes que no lo sean. No hace uso de pilas ('stacks') en esta clase de problemas, sino de algunas características de sus flujos de control. Se refiere a problemas de matemáticas combinatorias.

20.- R. Sprugnoli, "Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets", CACM, Vol. 20, No. 11, Nov. 1977.

Decimos que el artículo prometía demasiado porque las funciones de 'hash' perfectas que proponen están limitadas a conjuntos de sólo 12 elementos como máximo. Para conjuntos mayores es necesario hacer ciertas modificaciones que vuelven al método -ya por sí complicado- en demasiado complicado si se lo compara al, más humilde, método de búsqueda binaria, que fue el que finalmente usamos.

21.- N. Wirth, "Program Development by Stepwise Refinement", CACM, Vol. 14, No. 4, Apr. 1971.

Artículo que hace énfasis en la diferencia que hay y debe haber entre "programar" y "codificar". Pide la descomposición de las tareas en sub-bloques más sencillos y la conversión de los datos en estructuras apropiadas de datos. Trae como ejemplo el problema clásico (desde tiempos de Gauss) de las "ocho reinas". Artículo claro y útil.

22.- N. Wirth, Algorithms + Data Structures = Programs, Prentice-Hall, N. J., 1976.

Buen libro del creador de PASCAL. Ya el título del libro nos habla de las finalidades del autor: enseñar a programar en lenguajes de alto nivel, para lo cual dedica buena parte del libro a convencernos de las bondades de sus sofisticadas estructuras de datos (para un ejemplo de lo sofisticadas que pueden llegar a ser véase Operating Systems, de Brinch Hansen). Finalmente incluye un ejemplo completo de un reconocedor recursivo.

7.- ANEXO: MANUAL DE USUARIO DE FOREST

Antes de pasar a la descripción de FOREST diremos que, en la forma en que está terminado, simplemente deja en un archivo en disco el "código" FORTRAN generado, y en otro archivo el listado fuente, con los posibles mensajes de error y advertencia y las referencias numéricas cruzadas.

Se requiere, entonces, de una pequeña interfase entre FOREST y el sistema operativo de la máquina en particular en la cual se vaya a implantar. Por lo pronto FOREST no puede leer (a tiempo de ejecución) datos, porque requiere que la última tarjeta (físicamente hablando) sea la de END. Se pueden usar varios END de FORTRAN si se requieren (al compilar varios programas juntos, por ejemplo) escribiendo E ND así, como un espacio en blanco intermedio. De esta manera FOREST no reconocerá esto como un END (pues el blanco lo impide) y no dará mensaje de error.

Pero esto es fácilmente modificable para lograr que, desde el punto de vista del usuario, FOREST "compile" y ejecute programas en un solo paso aparente. Se requiere, insistimos, una sencilla interfase; pero necesariamente de carácter dependiente de la máquina particular de que se disponga.

Una vez advertidos de eso, pasamos a la descripción.

FORtran ESTructurado (FOREST) es un lenguaje de programación que se obtuvo a través de "doblar" FORTRAN hacia aplicaciones de programación estructurada por medio de macro-definiciones.

Esto es, las instrucciones de FOREST son, en realidad, instrucciones de FORTRAN "disfrazadas". El papel del preprocesador es entonces el convertir tales instrucciones a FORTRAN "puro" otra vez.

El usuario maneja instrucciones nuevas (son catorce) que conservan una cierta semejanza con FORTRAN, y que pueden "convivir" en un programa dado con instrucciones

de FORTRAN normales (excepto DO). Por medio de estas nuevas estructuras es posible manejar conceptos de programación que se ven grandemente obscurecidos para aquél que maneja FORTRAN.

El resultado esperable es una mayor eficiencia en la producción de programas, debido a la mayor facilidad de leer, entender y escribir código FOREST.

Una vez que el preprocesador terminó de procesar el programa fuente escrito en FOREST, le toca al compilador FORTRAN compilar y ejecutar el programa "objeto" generado. Es decir, FOREST toma el programa fuente y produce un programa en FORTRAN, equivalente funcional al original. Hasta aquí llega el papel del preprocesador; lo demás es función del compilador normal.

Descripción de las estructuras de FOREST

Este preprocesador maneja doce instrucciones y dos notaciones que difieren de las de FORTRAN. Las doce instrucciones, que a continuación describiremos, pueden ser nombradas indistintamente en Inglés o en Español.

Supondremos que el lector está familiarizado con los conceptos de programación "estructurada", aunque sea sólo a nivel de conocer los conceptos elementales de "bloque" y "módulo".

Si no fuera el caso, recomendamos leer primero algún manual de programación de ALGOL o PL/I o PASCAL [1,2]

A continuación reproducimos la gramática de FOREST.

1 D. McCracken, Programación ALGOL, Limusa, México, D.F. 1963.

2 R. Sprowls, Introducción a la Programación en lenguaje PL/I, Harper & Row, España, 1971.

Esta es la descripción, en BNF, de nuestra gramática:

```
ELEMENTO        ::=  DECLARACION / PROPOSICION
DECLARACION     ::=  "declaraciones no ejecutables de FORTRAN"
PROPOSICION     ::=  if (-condición-) ; PROPOSICION
                   / if (-condición-); PROPOSICION; else PROPOSICION
                   / while (-condición-); PROPOSICION
                   / for ( -inicio- ';' -condición- ';' -reinicio- ) PROPOSICION
                   / repeat ; PROPOSICION
                   / repeat ; PROPOSICION; until (-condición-)
                   / do -límites- ; PROPOSICION
                   / break
                   / next
                   / case -identificador- ' ; '
                     MASCASO
                   / COMPUESTA
                   / ++ -identificador-
                   / -- -identificador-
                   / -etiqueta- PROPOSICION
                   / FTN
COMPUESTA       ::=  begin MASCOMP end
MASCOMP         ::=  PROPOSICION MASCOMP / PROPOSICION
MASCASO         ::=  -identificador- ' : ' PROPOSICION ; MASCASO
                   / end
FTN             ::=  "cualquier proposición válida de FORTRAN (excepto DO )"
-identificador- ::=  "cualquier identificador válido de FORTRAN"
-condición-     ::=  "cualquier condición válida de FORTRAN, con los relacionales
                      tipo de PL/I (p. ej.,  = en lugar de .EQ.)"
```

-inicio- ::= "cualquier proposición de asignación de FORTRAN"
/ "cualquier expresión de llamada (que use CALL)"

-reinicio- ::= -inicio-

-etiqueta- ::= "cualquier etiqueta válida de FORTRAN, excepto aquéllas que comiencen con 77 y sean de cinco cifras"

Aparecen con mayúsculas los elementos no-terminales de la gramática.
Aparecen subrayados los elementos terminales que son palabras clave.
Aparecen con minúsculas y entre guiones ciertos elementos terminales que merecen mayor explicación.
Los símbolos entre apóstrofos deberán aparecer textualmente, así como los símbolos adicionales (paréntesis y otros puntos y comas, y signos dobles ++ y --). El símbolo / representa a la disyunción.
El punto y coma sin apóstrofos es un "brinco de renglón", opcional.

El mecanismo general es que cada instrucción de FOREST tiene un rango de una sola proposición, la que le sigue en secuencia. Si se desea que el rango de una proposición de FOREST alcance a más de una proposición, entonces hay que utilizar una especie de paréntesis, cuya función es la de agrupar varias proposiciones y hacer que se comporten como si fueran una sola.

El formato es libre; es decir, las proposiciones pueden comenzar en cualquier columna del renglón, y es posible incluír varias en un renglón, separándolas por medio de un punto y coma aunque esto tiene a dificultar la lectura.

Si un renglón termina en coma o comienza en coma o 'slash' ' / ' entonces se supondrá que es continuación del anterior.

Una -condición- (de un IF, UNTIL, WHILE o FOR) se puede continuar en el siguiente renglón si se corta en algún símbolo relacional o en coma.

Por ejemplo

```
IF  ( ALFA >
                BETA )   CALL DOS
```

es lo mismo que IF (ALFA > BETA) CALL DOS

Los símbolos dobles (<= , >= , <>) no se pueden "cortar" a la mitad.

Los comentarios de FOREST comienzan con el símbolo de admiración '!' y terminan con punto y coma. Pueden aparecer en cualquier lugar donde aparezca un blanco, pero no se pueden extender a otro renglón.

Si el comentario es lo último que hay en el renglón (o lo único), no necesita terminar con punto y coma.

Por ejemplo

```
A = B ; !comentario ; C = D  ! otro comentario.
```

Los "paréntesis" antes mencionados se llaman BEGIN -el que "abre"- (o bien, COMIENZA), y END -el que "cierra"- (también llamado TERMINA).

PROPOSICIÓN DO (EJECUTA):

Es fundamentalmente igual a la de FORTRAN. Sólo que aquí se ha eliminado la etiqueta numérica que sirve para delimitar el alcance, para sustituírla por el mecanismo general antes mencionado. Es decir, si el DO sólo controla a una proposición, entonces no se hace necesario más que escribir

```
DO  -límites-
      PROPOSICIÓN      o bien,      DO  -límites- ; PROPOSICIÓN
```

donde -límites- son los límites normales de un DO de FORTRAN (I = 1,40 por ejemplo), y-PROPOSICIÓN- es cualquier proposición de FOREST o de FORTRAN.

Pero si el alcance es mayor que una proposición, entonces el mecanismo general nos dice que hay que encerrar el grupo de instrucciones entre "paréntesis".

```
DO   -límites-
 BEGIN
 -PROPOSICIÓN-
 -PROPOSICIÓN-
     .
     .
     .
 -PROPOSICIÓN-
END
```

Proposición IF (SI):

Esta proposición es parecida a la de FORTRAN, aplicando el mecanismo fundamental antes descrito para extender su alcance.

Si la -condición- es verdadera a tiempo de ejecución, entonces el flujo de control pasa a la siguiente proposición (o grupo entre BEGIN-END). En otro caso el control pasa a la primera proposición fuera del alcance del IF.

-condición- es cualquier condición de FORTRAN, donde se han hecho las siguientes sustituciones:

=	por	.EQ.	
<> o bien ¬=	por	.NE.	
|	por	.OR.	(se usa '?' en el listado)
&	por	.AND.	
¬	por	.NOT.	(se usa '^' en el listado)
<=	por	.LE.	
>=	por	.GE.	
<	por	.LT.	
>	por	.GT.	

Por ejemplo: IF(ALFA(I) = BETA(J,K)) M = M + 1

que es prácticamente igual en funcionamiento que en FORTRAN, con la diferencia de que se sustituyó el .EQ. por el = en la expresión entre paréntesis (la -condición-).

En

```
IF(  TEMP & CAR   )
BEGIN
  CALL UNO
  CALL DOS
  L = J / (S-P)
END
WRITE(5,112)
READ(5,222)
```

las tres proposiciones encerradas entre el BEGIN-END se ejecutarán sólo si la -condición- TEMP & CAR es verdadera. Luego se ejecutará el WRITE(5,112) y luego el READ(5,222).

Si se desea lograr que solamente una de las dos posibilidades sea ejecutada, dependiendo del valor de verdad de la -condición-, entonces hay que usar la combinación IF-ELSE.

Proposición ELSE (OTRO):

Para el ejemplo anterior, si escribimos

```
IF(  TEMP & CAR  )
 BEGIN
    CALL UNO;  CALL DOS;  L = J / (S-P)
 END
ELSE  WRITE(5,112)
READ(5,222)
```

entonces lograremos que, si la condición es verdadera, se ejecute el grupo de tres proposiciones. Si, por el contrario, la condición haya sido falsa, entonces se ejecutará el WRITE(5,112).

EN AMBOS CASOS la siguiente proposición que se ejecutará será el READ(5,222). Esto es, o se ejecuta la proposición "verdadera" o se ejecuta la "falsa", pero nunca ambas.

No todo IF tiene por fuerza que ir asociado a un ELSE, e inclusive puede haber ELSEs "vacíos": ELSE; es un ejemplo.

FOREST sigue la costumbre de asociar cada ELSE al IF anterior libre más cercano.

Proposición WHILE:

Esta proposición es muy poderosa, y sirve admirablemente bien para "estructurar" programas, al igual que el FOR.

Si la -condición- es verdadera, entonces se ejecuta la PROPOSICIÓN y se vuelve a preguntar por la condición. Esto es, la PROPOSICIÓN se ejecutará mientras la -condición- se cumpla. Si la -condición- nunca es verdadera, entonces la PROPOSICIÓN nunca se ejecutará, y el control seguirá, pasando de largo el bloque WHILE.

Por ejemplo, en:

```
WHILE (ALFA >= 3.81)
 BEGIN
    CALL UNO
    CALL DOS
    CALL TRES
 END
WRITE(5,222)
```

Si ALFA es mayor o igual a 3.81 entonces se ejecutará el grupo de instrucciones entre el BEGIN y el END, y se volverá a preguntar por la condición. En el momento en que ALFA sea menor que 3.81, entonces se ejecutará el WRITE(5,222).

Proposición FOR:

Esta proposición es un WHILE generalizado, y un DO generalizado. Consiste de tres partes, además de la PROPOSICIÓN: primero se ejecuta incondicionalmente la instrucción -inicializa-; luego se pregunta (como en el MIENTRAS) por la -condición-;

si fue verdadera entonces se ejecuta la PROPOSICION y luego la instrucción -reinicia liza-, para volver a preguntar por la -condición-.

Pero si la -condición- fue falsa, entonces el control pasa de largo a todo el FOR.

Por ejemplo:

```
FOR(  J = M ;  J<>1 ;  J = J - 1  )  PRINT, ALFA(J)
```

Este renglón logra que se impriman los valores de

ALFA(M), ALFA(M-1), ALFA(M-2), ..., ALFA(2), ALFA(1), en ese orden; cosa imposible de lograr directamente con un DO de FORTRAN.

Esta es una proposición muy poderosa y versátil: casi desde el principio surgen aplicaciones para ella en cualquier programa escrito en FOREST.

Existe un FOR "nulo", aquél donde la DECLARACION se reemplaza por un punto y coma. Util para"recorrer" arreglos esperando hallar una cierta condición en alguno de sus elementos, pero sin hacer otra cosa con ellos. Por ejemplo:

```
FOR (I=1; ALFA(I)  =  BLANCO & I = 80 ; I = I+1);
```

recorre el arreglo ALFA(I) y se detiene en la posición I-ésima, que contiene un no-blanco. O bien se detiene al hacerse I=80, como caso límite.

También es válido eliminar cualquiera de los tres elementos dentro del paréntesis del FOR, por medio de un punto y coma.

Por ejemplo:

```
FOR (I=0; I=80 ; )  ALGO
```

no tiene la parte -reiniciliza- y por lo tanto equivale a un REPEAT "infinito", ya que I nunca se hará igual a 80.

Proposición REPEAT:

Esta proposición es una abreviatura del último ejemplo del FOR.

La PROPOSICION se ejecutará "infinitas" veces. Claro que, como la PROPOSICION puede ser compuesta (entre BEGIN-END), se podrá incluir un BREAK para terminarla en algún punto.

Por ejemplo:

```
REPEAT
 BEGIN
  CALL UNO
  CALL DOS
  IF( C = 999 ) BREAK
 END
```

reperirá las llamadas a UNO y DOS tantas veces como sea necesario hasta que el valor de la variable C alcance el de 999.
Esto se puede decir directamente usando el complemento HASTA/UNTIL.

Proposición UNTIL:

El complemento UNTIL sirve para terminar un REPEAT, por ejemplo:

```
REPEAT
 CALL UNO
UNTIL( BETA = ZETA )
```

Ejecuta múltiples llamadas a UNO y no termina sino hasta que BETA iguala a ZETA.

Proposición DO:

Es el DO de FORTRAN, pero sin la necesidad de tener que pensar en números apropiados para las etiquetas de control. Si la PROPOSICION es una sola, no hará falta

más que escribirla en el siguiente renglón al DO. Si se trata de varias, se agruparán estas con BEGIN-END. Por ejemplo:

```
DO   I = 1,25
     DO  J = 1,25
         ALFA(I,J) = 0
```

inicializa el arreglo ALFA de 25 X 25 a ceros.

Y

```
DO   I = 1,33
 BEGIN
     ALFA(I) = 0
     BETA(I) = 0
 END
```

hará lo mismo con los arreglos ALFA y BETA de tamaño 33.

Proposición BREAK:

La proposición FUERA logra sacar al flujo de control fuera del rango más cercano (porque puede haber varios rangos anidados) de un WHILE, FOR, REPEAT, DO o CASE. Vimos un ejemplo en la instrucción REPEAT.

Otro ejemplo:

```
DO   I = 1,35
     DO J = 1,45
       BEGIN
          CALL UNO
          CALL DOS
          IF(  ALFA = BETA  )  BREAK
       END
   CALL TRES
```

Si ALFA = BETA, entonces el control se sale del DO más interno y pasa a CALL TRES, que está controlada por el primer DO.

Proposición NEXT:

La proposición SIGUE fuerza al control a iniciar la siguiente iteración de un WHILE, FOR, REPEAT o DO.

Por ejemplo:

```
REPEAT
 BEGIN
    CALL UNO
    IF(  ALFA = 1  )  NEXT
    IF(  ALFA = 2  )  BREAK
    CALL DOS
 END
UNTIL (  ALFA = 10  )
```

Si ALFA = 1 entonces NO se ejecuta el llamado a DOS, sino que se vuelve a iniciar el REPITE llamando a UNO otra vez.

Si ALFA = 2 entonces se termina todo el caso REPITE.

Esta proposición compuesta terminará finalmente cuando ALFA = 10, si antes no fue ALFA = 2.

Igual que con el BREAK, en NEXT devuelve el control a la iteración correspondiente más cercana (es decir, la que corresponda al nivel actual de anidamiento).

Proposición CASE:

La proposición CASO es una abreviatura muy cómoda de los anidamientos IF...ELSE IF....ELSE IF....

Por ejemplo:

```
CASE  VAR2 ;
  ALFA:   ALFA = ALFA + 1
  BETA:   BETA = BETA + 1
  ZETA:   ZETA = ZETA + 1
 END
PRINT, M
```

Dependiendo del valor actual de la variable VAR2, si es igual al de algunas de las listadas (ALFA, BETA o ZETA) entonces hará la proposición (o grupo BEGIN-END) correspondiente a ese caso y luego proseguirá con la siguiente proposición después del CASE. Hará PRINT, M en este ejemplo.

Esto es, ejecutará UNA y sólo una de las múltiples proposiciones listadas (si es que la variable de control (VAR2) es igual a alguna de las variables que sirven como "etiquetas" (las que llevan dos puntos)) y luego ignorará todo el CASE.

Se permite que una o más proposiciones (no "etiquetas") del CASE sean "nulas", es decir, que sean un simple punto y coma. Esto tiene el efecto de ignorar el CASE en algún punto determinado de las comparaciones.

Por ejemplo:

```
CASE   VAR2;
 5:    ;
 6:    PRINT, "FUE SEIS"
 8:    PRINT, "FUE OCHO"
END
```

También se permite anidar varios CASEs:

```
CASO   VAR1 ;
 ALFA:   CALL UNO
 BETA:   CALL DOS
 ZETA: CASE     VAR2;
         2:  CALL CUATRO
         3:  CALL SEIS
       END
 GAMMA:   CALL TRES
END
```

Si se cumple que VAR2 = 3 entonces se llama a CUATRO y se termina. Pero hay que notar que esto sólo sería posible si ANTES se cumplió que VAR1 fue igual a ZETA. Esto es, un CASE depende del otro.

También se puede usar una "etiqueta vacía" (un simple ':'), que tendrá el efecto de "sumidero"; es decir, por ahí "se irá" el flujo de control si todas las anteriores "etiquetas" fallaron.

En

```
CASE    VAR1;

   A:   CALL UNO
'BB':   CALL DOS
    :   CALL TRES

END
```

Si VAR1 no es igual (en valor) a A ni a la cadena ZHBB, entonces se ejecutará CALL TRES. Sólo puede haber una de estas "etiquetas vacías", y debe ir inmediatamente antes del END.

Expresión ++ALGO y --ALGO:

Estas no son más que abreviaturas de los casos tan comunes ALGO = ALGO + 1 y ALGO = ALGO - 1, respectivamente.

Donde "ALGO" es cualquier identificador válido de FORTRAN. Así pues, ++BETA(I,J,K) es lo mismo que BETA(I,J,K) = BETA(I,J,K) + 1.

A continuación pondremos un ejemplo completo. Será el programa que encuentra las posibles maneras de colocar ocho reinas de ajedrez en un tablero, de tal manera que no se ataquen. Este es un problema que el gran matemático Gauss (1777-1855) comenzó a investigar.

Habrá varios listados, donde se podrán observar las referencias cruzadas entre FOREST y FORTRAN.

```
1    :  (F2   ):    ! ESTE PROGRAMA ENCUENTRA LAS -NUM- (10 EN ESTE CASO)
2    :  (F2   ):    ! PRIMERAS SOLUCIONES AL PROBLEMA DE COLOCAR OCHO
3    :  (F2   ):    ! REINAS EN UN TABLERO DE AJEDREZ DE TAL MANERA QUE
4    :  (F2   ):    ! NO SE ATAQUEN.
5    :  (F2   ):    ! PROGRAMA ESCRITO EN  FOREST  .
6    :  (F2   ):    !   N. WIRTH.  PROGRAM DEVELOPMENT
7    :  (F2   ):    !              BY STEPWISE REFINEMENT
8    :  (F2   ):    !              CACM, APR. 1971.
9    :  (F2   ):      LOGICAL A,B,C,SEG
10   :  (F2   ):      COMMON A(8),B(15),C(15),M(8),SEG,I,J,MAS
11   :  (F4   ):      DATA NUM  / 10 / !SI SE QUIEREN OBTENER TODAS LAS
12   :  (F5   ):                       !SOLUCIONES, HACER  NUM = 0 .
13   :  (F6   ):      MAS = 0
14   :  (F6   ):      EJECUTA K = 1,8
15   :  (F7   ):           A(K) = .TRUE.
16   :  (F8   ):      EJECUTA K = 1,15
17   :  (F11  ):      COMIENZA
18   :  (F12  ):           B(K) = .TRUE.
19   :  (F12  ):           C(K) = .TRUE.
20   :  (F13  ):      TERMINA
21   :  (F16  ):      J = 1; !VARIAS PROPOSICIONES POR RENGLON;  I = 0
22   :  (F17  ):      REPITE
23   :  (F19  ):      COMIENZA
24   :  (F19  ):           REPITE
25   :  (F20  ):           COMIENZA
26   :  (F20  ):                ++I  !ESTO ES UNA ABREVIATURA DE  I = I + 1
27   :  (F20  ):                CALL PRUEBA
28   :  (F21  ):           TERMINA
29   :  (F22  ):           HASTA( SEG ? ( I=8 ) )  !EL SIMBOLO  ?  EQUIVALE A .
30   :  (F24  ):           SI( SEG )                                        .OR.
31   :  (F25  ):           COMIENZA
32   :  (F25  ):                CALL PON
33   :  (F25  ):                M(J) = I
34   :  (F26  ):                ++J
35   :  (F27  ):                I = 0
36   :  (F28  ):                SI( J > 8 )
37   :  (F30  ):                COMIENZA
38   :  (F30  ):                     CALL SALE
39   :  (F30  ):                     CALL REGRES
40   :  (F31  ):                TERMINA
41   :  (F32  ):           TERMINA
42   :  (F33  ):           OTRO   CALL REGRES  !'OTRO' ES 'ELSE'.
43   :  (F35  ):      TERMINA
44   :  (F37  ):      HASTA( ( J<1 ) ? ( MAS=NUM ) )
45   :  (F39  ):      STOP
46   :  (F39  ):      E ND   !NOTESE QUE ESTO NO ES EL  END  DE FOREST.
47   :  (F40  ):    !
48   :  (F41  ):      SUBROUTINE REGRES
49   :  (F41  ):      LOGICAL A,B,C,SEG
50   :  (F42  ):      COMMON A(8),B(15),C(15),M(8),SEG,I,J,MAS
51   :  (F44  ):      --J     !ABREVIATURA DE  J = J - 1 .
52   :  (F45  ):      I = M(J)
53   :  (F46  ):      SI( J >= 1 )
54   :  (F48  ):      COMIENZA
55   :  (F48  ):           CALL QUITA
56   :  (F48  ):           SI( I = 8 )
```

```
(F50   ):            COMIENZA
(F50   ):                --J
(F50   ):                I = M(J)
(F51   ):                SI( J >= 1 )   CALL QUITA
(F53   ):            TERMINÁ
(F55   ):        TERMINA
(F56   ):        RETURN
(F57   ):        E ND
(F58   ):    !
(F59   ):        SUBROUTINE PRUEBA
(F59   ):        LOGICAL A,B,C,SEG
(F60   ):        COMMON A(8),B(15),C(15),M(8),SEG,I,J,MAS
(F62   ):        SEG = ( A(I) & B(I+J-1) & C(I-J+8) )
(F63   ):        RETURN
(F64   ):        E ND
(F65   ):    !
(F66   ):        SUBROUTINE PON
(F66   ):        LOGICAL A,B,C,SEG
(F67   ):        COMMON A(8),B(15),C(15),M(8),SEG,I,J,MAS
(F69   ):        A(I) = .FALSE.
(F70   ):        B(I+J-1) = .FALSE.
(F71   ):        C(I-J+8) = .FALSE.
(F72   ):        RETURN
(F73   ):        E ND
(F74   ):    !
(F75   ):        SUBROUTINE QUITA
(F75   ):        LOGICAL A,B,C,SEG
(F76   ):        COMMON A(8),B(15),C(15),M(8),SEG,I,J,MAS
(F78   ):        A(I) = .TRUE.
(F79   ):        B(I+J-1) = .TRUE.
(F80   ):        C(I-J+8) = .TRUE.
(F81   ):        RETURN
(F82   ):        E ND
(F83   ):    !
(F84   ):        SUBROUTINE SALE
(F84   ):        INTEGER T(8,8)
(F85   ):        LOGICAL A,B,C,SEG
(F86   ):        COMMON A(8),B(15),C(15),M(8),SEG,I,J,MAS
(F88   ):        DATA T / 64*0 /
(F89   ):        ++MAS
(F90   ):      ! IMPRIME SEIS SOLUCIONES POR HOJA.
(F91   ):        IF( MOD(MAS,7) = 0 ) WRITE(6,120) ; 120 FORMAT(1H1)
(F95   ):        EJECUTA L = 1,8
(F95   ):            T( M(L),L ) = 1
(F96   ):        WRITE(6,100) MAS, (( ( T(K,L), L=1,8), K=1,8 ) )
(F100  ):        100 FORMAT(' SOL. NUM. ',I4, 8(/,30X, 8(I1,1X) ) )
(F102  ):        EJECUTA L = 1,8
(F102  ):            T( M(L),L ) = 0
(F103  ):        RETURN
(F106  ):        END
##NUMERO DE ADVERTENCIAS =      0
##NUMERO DE ERRORES =      0
##NUMERO DE LINEAS (TARJETAS) LEIDAS =    106
##FIN.##
```

```
C                              "FOREST"  VERSION 1.2          CHUIK.
      LOGICAL A , B , C , SEG
      COMMON A ( 8 ) , B ( 15 ) , C ( 15 ) , M ( 8 ) , SEG , I , J , MA
     $S
      DATA NUM / 10 /
      MAS = 0
      DO 77700 K = 1 , 8
      A ( K ) = . TRUE .
77700 CONTINUE
77701 CONTINUE
      DO 77702 K = 1 , 15
      B ( K ) = . TRUE .
      C ( K ) = . TRUE .
77702 CONTINUE
77703 CONTINUE
      J = 1
      I = 0
77704 CONTINUE
77706 CONTINUE
      I = I + 1
      CALL PRUEBA
      IF(.NOT.(SEG .OR. (I .EQ. 8)))GOTO 77706
77707 CONTINUE
      IF(.NOT.(SEG))GOTO 77708
      CALL PON
      M ( J ) = I
      J = J + 1
      I = 0
      IF(.NOT.(J .GT. 8))GOTO 77710
      CALL SALE
      CALL REGRES
77710 CONTINUE
      GOTO 77709
77708 CONTINUE
      CALL REGRES
77709 CONTINUE
      IF(.NOT.((J .LT. 1) .OR. (MAS .EQ. NUM)))GOTO 77704
77705 CONTINUE
      STOP
      E ND
      SUBROUTINE REGRES
      LOGICAL A , B , C , SEG
      COMMON A ( 8 ) , B ( 15 ) , C ( 15 ) , M ( 8 ) , SEG , I , J , MA
     $S
      J = J - 1
      I = M ( J )
      IF(.NOT.(J .GE. 1))GOTO 77712
      CALL QUITA
      IF(.NOT.(I .EQ. 8))GOTO 77714
      J = J - 1
      I = M ( J )
      IF(.NOT.(J .GE. 1))GOTO 77716
      CALL QUITA
77716 CONTINUE
77714 CONTINUE
77712 CONTINUE
```

PAGE 2 HEWLETT-PACKARD 32201A.7.04 EDIT/3000 FRI, JUL 6, 1979, 2:38 PM (C) HEW

```
57          RETURN
58          E ND
59          SUBROUTINE PRUEBA
60          LOGICAL A , B , C , SEG
61          COMMON A ( 8 ) , B ( 15 ) , C ( 15 ) , M ( 8 ) , SEG , I , J , MA
62         $S
63          SEG = ( A ( I )  .AND. B ( I + J - 1 )  .AND. C ( I - J + 8 ) )
64          RETURN
65          E ND
66          SUBROUTINE PUN
67          LOGICAL A , B , C , SEG
68          COMMON A ( 8 ) , B ( 15 ) , C ( 15 ) , M ( 8 ) , SEG , I , J , MA
69         $S
70          A ( I ) = . FALSE .
71          B ( I + J - 1 ) = . FALSE .
72          C ( I - J + 8 ) = . FALSE .
73          RETURN
74          E ND
75          SUBROUTINE QUITA
76          LOGICAL A , B , C , SEG
77          COMMON A ( 8 ) , B ( 15 ) , C ( 15 ) , M ( 8 ) , SEG , I , J , MA
78         $S
79          A ( I ) = . TRUE .
80          B ( I + J - 1 ) = . TRUE .
81          C ( I - J + 8 ) = . TRUE .
82          RETURN
83          E ND
84          SUBROUTINE SALE
85          INTEGER T ( 8 , 8 )
86          LOGICAL A , B , C , SEG
87          COMMON A ( 8 ) , B ( 15 ) , C ( 15 ) , M ( 8 ) , SEG , I , J , MA
88         $S
89          DATA T / 64 * 0 /
90          MAS = MAS + 1
91          IF(.NOT.(MOD(MAS,7) .EQ. 0))GOTO 77718
92          WRITE ( 6 , 120 )
93    77718 CONTINUE
94    120   FORMAT ( 1H1 )
95          DO 77720 L = 1 , 8
96          T ( M ( L ) , L ) = 1
97    77720 CONTINUE
98    77721 CONTINUE
99          WRITE ( 6 , 100 ) MAS , ( ( ( T ( K , L ) , L = 1 , 8 ) , K = 1 ,
100        $ 8 ) )
101   100   FORMAT ( 12H  SOL. NUM. , I4 , 8 ( / , 30X , 8 ( I1 , 1X ) ) )
102         DO 77722 L = 1 , 8
103         T ( M ( L ) , L ) = 0
104   77722 CONTINUE
105   77723 CONTINUE
106         RETURN
107         END
```

SOL. NUM. 1

1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0

SOL. NUM. 2

1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 0

SOL. NUM. 3

1 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0

SOL. NUM. 4

1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0

SOL. NUM. 5

0 0 0 0 0 1 0 0
1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0

SOL. NUM. 6

0 0 0 1 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0

SOL. NUM. 7

0 0 0 0 1 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0

SOL. NUM. 8

0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0

SOL. NUM. 9

0 0 0 0 1 0 0 0
1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0

SOL. NUM. 10

0 0 0 0 0 0 1 0
1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0

A continuación pondremos dos ejemplos más.

El primero es un listado de la manera en que FOREST genera "código" FORTRAN . Se ponen varios ejemplos que no se veían en el programa de "las ocho reinas".

El segundo ejemplo es de la capacidad que tiene FOREST de detectar errores en el listado fuente. Es necesario repetir que los únicos errores que se detectarán serán aquéllos de FOREST. El preprocesador no tiene capacidad de detectar errores de FORTRAN "puro". Si el usuario escribe SUBRAUTINE en lugar de SUBROUTINE, por ejemplo, FOREST no se quejará y generará "código", pero entonces el compilador FORTRAN detectará el error cuando se procese el programa "objeto" generado.

Si FOREST reporta que se encontró un error o más, entonces también manda el mensaje ***EL CODIGO NO SIRVE*** , dando a entender que el programa "objeto" generado puede tener errores (etiquetas inexistentes, pero referenciadas, etc.) que harán que fracase la compilación. Si éste fuera el caso, FOREST no tendría responsabilidad, puesto que claramente se hizo notar tal situación.

```
HP32201A.7.04 EDIT/3000 FRI, JUL  6, 1979,  3:00 PM (C) HEWL

(F2    ):      !ESTE ES UN EJEMPLO DE COMO  FOREST  GENERA
(F2    ):      !"CODIGO" FORTRAN.
(F2    ):      !SERA UNA SERIE DE PROPOSICIONES (QUE NO TIENEN
(F2    ):      !SENTIDO SEMANTICO) SEPARADAS POR RENGLONES QUE
(F2    ):      !CONTIENEN EL SIMBOLO ESPECIAL '$', PARA MAYOR
(F2    ):      !COMODIDAD VISUAL.
(F2    ):      !SE INTENTA DEMOSTRAR LAS "MACRO-EXPRESIONES" DE
(F2    ):      !FOREST YA "EXPANDIDAS".
(F2    ):      REPEAT
(F3    ):         BEGIN
(F3    ):            PROP. A1
(F3    ):   $
(F4    ):            IF( COND1 ) BREAK
(F7    ):            ELSE
(F8    ):            BEGIN
(F8    ):               WHILE( COND2 ) DO LIMITE1
(F10   ):                              DO LIMITE2
(F11   ):                                 PROP. A2
(F12   ):   $
(F19   ):                  PROP. A3
(F20   ):   $
(F21   ):            END
(F23   ):   $
(F23   ):            FOR( ++I4 ; COND4 ; --R4 )
(F27   ):                 PROP. A4
(F27   ):   $
(F31   ):         END
(F32   ):   $
(F34   ):         PROP. A5
(F35   ):   $
(F36   ):         PROP. A6
(F37   ):   $
(F38   ):         CASE VAR1;
(F39   ):           EXP11:    PROP. 11
(F40   ):           EXP12:    PROP. 12
(F44   ):           EXP13:    CASE  VAR2;
(F48   ):                       EXP21:  PROP. 21
(F49   ):                       EXP22:  PROP. 22
(F53   ):                     END
(F59   ):           EXP14:    PROP. 14
(F60   ):           EXP15:    ;  !NOTESE QUE LA PROP. ES NULA .
(F64   ):                 :   PROP. 16  !NOTESE QUE LA "ETIQUETA" ES VACIA
(F66   ):         END
(F70   ):   $
(F70   ):   END
##NUMERO DE ADVERTENCIAS =       0
##NUMERO DE ERRORES =       0
##NUMERO DE LINEAS (TARJETAS) LEIDAS =      45
##FIN.##
```

```
C                              "FOREST"  VERSION 1.2            CHUIK.
77700 CONTINUE
      PROP . A1
$
      IF(.NOT.(COND1))GOTO 77702
      GOTO 77701
77702 CONTINUE
77704 CONTINUE
      IF(.NOT.(COND2))GOTO 77705
      DO 77706 LIMITE1
      DO 77708 LIMITE2
      PROP . A2
77708 CONTINUE
77709 CONTINUE
77706 CONTINUE
77707 CONTINUE
      GOTO 77704
77705 CONTINUE
$
      PROP . A3
$
77703 CONTINUE
$
      I4 = I4 + 1
77710 CONTINUE
      IF(.NOT.(COND4))GOTO 77711
      PROP . A4
      R4 = R4 - 1
      GOTO 77710
77711 CONTINUE
$
      GOTO 77700
77701 CONTINUE
$
      PROP . A5
$
      PROP . A6
$
      IF(VAR1.NE.EXP11)GOTO 77713
      PROP . 11
      GOTO 77712
77713 CONTINUE
      IF(VAR1.NE.EXP12)GOTO 77714
      PROP . 12
      GOTO 77712
77714 CONTINUE
      IF(VAR1.NE.EXP13)GOTO 77715
      IF(VAR2.NE.EXP21)GOTO 77717
      PROP . 21
      GOTO 77716
77717 CONTINUE
      IF(VAR2.NE.EXP22)GOTO 77718
      PROP . 22
      GOTO 77716
77718 CONTINUE
77716 CONTINUE
```

```
57          GOTO 77712
58    77715 CONTINUE
59          IF(VAR1.NE.EXP14)GOTO 77719
60          PROP = 14
61          GOTO 77712
62    77719 CONTINUE
63          IF(VAR1.NE.EXP15)GOTO 77720
64          GOTO 77712
65    77720 CONTINUE
66          PROP = 16
67          GOTO 77712
68    77721 CONTINUE
69    77712 CONTINUE
70    $
71          END
```

```
PAGE 1      HEWLETT-PACKARD 32201A.7.04 EDIT/3000 FRI, JUL  6, 1979,  4:19 PM (C) HEW

1    :   (F2    ):      !ESTE ES UN EJEMPLO DE ALGUNOS DE LOS ERRORES
2    :   (F2    ):      !QUE  FOREST  PUEDE DETECTAR.
3    :   (F2    ):      !EL PROGRAMA QUE SIGUE (SIN SENTIDO SEMANTICO)
4    :   (F2    ):      !TIENE VARIOS ERRORES DELIBERADOS.
5    :   (F2    ):      INTEGER ALGO(10)
6    :   (F2    ):      DATA ALGO
7    :   (F3    ):      !FOREST AUTOMATICAMENTE GENERA LINEAS DE CONTINUACION...
8    :   (F4    ):            /1,2,3,4,5,6,7,8,9,0/
9    :   (F4    ):      PROP. 1
10   :   (F5    ):      PROP. 2
11   :   (F6    ):      FOR( **I; C > 0 ; ++A ) !AQUI HAY UN ERROR (EL SIMBOLO '*'
                                ^
**ERROR: 'FOR' INCORRECTO, SE IGNORA TODO EL RENGLON**
12   :   (F7    ):      PROP. 3
13   :   (F7    ):      WHILE  C >= 0 )  !AQUI FALTO UN PARENTESIS.
                                ^
**ERROR: MALA SINTAXIS EN LA CONSTRUCCION, SE IGNORA TODO EL RENGLON**
14   :   (F9    ):      BREAK  !ESTE 'BREAK' ES VALIDO; DEPENDE DEL 'WHILE' ANTERI
15   :   (F12   ):             !NO IMPORTA QUE HAYA TENIDO ERROR: SU RANGO SE
16   :   (F12   ):             !MANTIENE.  ESTO EVITA "CADENAS DE ERRORES"
17   :   (F12   ):             !CAUSADAS POR UNO PREVIO.
18   :   (F12   ):      PROP. 3
19   :   (F12   ):      BREAK
                         ^
**ERROR: 'BREAK' ILICITO**
20   :   (F13   ):      ?   !ESTE ES UN SIMBOLO "RARO".
                         ^
**ERROR: SIMBOLO IRRECONOCIBLE**
21   :   (F13   ):      ALGO = 'LASQUINCELETRAS
22   :   (F13   ):      PROP. 4
**ERROR: HACE FALTA UN APOSTROFO**
23   :   (F13   ):      DO +3   !'DO' MAL CONSTRUIDO.
                              ^
**ERROR: MALA SINTAXIS EN EL 'DO', SE IGNORA EL RENGLON**
24   :   (F14   ):      PROP. 5
25   :   (F14   ):      NEXT  ALGO  !AQUI HAY DOS ERRORES...
                         ^
**ERROR: 'NEXT' ILICITO**
                                  ^
**CUIDADO: SE IGNORA EL RESTO DEL RENGLON**
26   :   (F17   ):      END;  PROP. 6    !MAL PUESTO (NO ESTA APAREADO A 'BEGIN' O
                         ^                                                  CASE.
**ERROR: USO ILICITO DE PALABRA CLAVE**
                         ^
**CUIDADO: SE IGNORA EL RESTO DEL RENGLON**
27   :   (F17   ):      77700  CONTINUE
                         ^
**ERROR: POSIBLE CONFLICTO DE ETIQUETAS**
28   :   (F17   ):        PROP. 7
29   :   (F18   ):      123456  CONTINUE
                              ^
**ERROR: ETIQUETA CON MAS DE CINCO CARACTERES**
30   :   (F19   ):      CASE VARI:
31   :   (F20   ):        EXP11:  *   !SIMBOLO "RARO".
                                 ^
**ERROR: SIMBOLO IRRECONOCIBLE**
```

```
HP32201A.7.04 EDIT/3000 FRI, JUL  6, 1979,  4:19 PM (C) HEW
PAGE 2

32    :    (F21    ):          EXP12:  PROP. 12
33    :    (F21    ):                  PROP. 13   !"ETIQUETA" VACIA (VALIDA), PERO MAL
34    :    (F24    ):                             !PORQUE NO ESTA ANTES DEL 'END' .
35    :    (F27    ):          EXP13:  PROP. 13
**CUIDADO: LA "ETIQUETA VACIA" DEL 'CASE'  ESTA MAL POSICIONADA, O HAY MAS DE UNA*
36    :    (F28    ):          (((((:  ;          !"ETIQUETA" INVALIDA.
**CUIDADO: LA "ETIQUETA VACIA" DEL 'CASE'  ESTA MAL POSICIONADA, O HAY MAS DE UNA*
                               ^

**ERROR: MALA SINTAXIS EN EL 'CASE', SE IGNORA COMPLETO**
37    :    (F31    ):          EXP15:  PROP. 15
38    :    (F31    ):        END
39    :    (F32    ):      PROP. 5
40    :    (F32    ):      IF( C >> D ) ALGO    !RELACIONAL MAL FORMADO CON ELEMENTOS
                                  ^                                          VALIDOS.
**CUIDADO: COMBINACION IRRECONOCIBLE DE SIMBOLOS, SE TOMARA EL PRIMERO**
41    :    (F34    ):      PROP. 6
42    :    (F37    ):      PROP. 7
43    :    (F37    ):      END
##NUMERO DE ADVERTENCIAS =      5
##NUMERO DE ERRORES =     12
***EL CODIGO NO SIRVE***
##NUMERO DE LINEAS (TARJETAS) LEIDAS =     43
##FIN.##
```

8.- ANEXO: LISTADO COMPLETO DE FOREST

A continuación sigue el listado completo (compilado ya) del preprocesador escrito en FORTRAN.

Consiste de un BLOCK DATA, un programa principal (FOREST), y 37 subrutinas. La estructura de esto está explicada a partir de la página 35 (ver supra).

En total se trata de 2083 líneas de FORTRAN, incluyendo comentarios.

```
$CONTROL USLINIT
$CONTROL FILE=5-50
      BLOCK DATA CARGA
C     ESTE 'BLOCK DATA' SIRVE PARA "INICIALIZAR" LAS
C     VARIABLES QUE SE USAN EN FORMA COMUN.
      INTEGER APDIC,CLDIC,DIC,ULTIMO,TEXTO
      COMMON /DIC1/ DIC(113),CLDIC(24),APDIC(24)
      COMMON /LEX/ ULTIMO,TEXTO(95),IULT,LIMIZ,LIMDR
      COMMON /NUMR/ NUMRS(10)
C  APDIC CONTIENE LOS APUNTADORES A LAS PALABRAS CLAVE EN DIC .
      DATA APDIC /  1,  3,  5,  7, 10, 13, 17, 21,
     $             25, 29, 33, 37, 42, 47, 52, 57,
     $             62, 67, 72, 78, 84, 91, 98,106 /
C  CLDIC CONTIENE LAS CLAVES NUMERICAS DE LAS PALABRAS CLAVE.
      DATA CLDIC / 283, 302, 305, 643, 717,1140,1160,1195,
     $            1230,1294,1313,1787,1821,1829,1841,1855,
     $            1873,1934,2681,2715,3600,3639,4824,4935 /
      DATA DIC
     $ / %'I',%'F',%'S',%'I',%'D',%'O',%'E',%'N',%'O',%'F',%'O',%'R',
     $   %'P',%'A',%'R',%'A',%'C',%'A',%'S',%'E',%'E',%'L',%'S',%'E',
     $   %'C',%'A',%'S',%'O',%'O',%'T',%'R',%'O',%'N',%'E',%'X',%'T',
     $   %'B',%'R',%'E',%'A',%'K',%'W',%'H',%'I',%'L',%'E',
     $   %'F',%'U',%'E',%'N',%'A',%'B',%'E',%'G',%'I',%'N',
     $   %'H',%'A',%'S',%'T',%'A',%'S',%'I',%'G',%'U',%'E',
     $   %'U',%'N',%'T',%'I',%'L',%'N',%'E',%'P',%'E',%'A',%'T',
     $   %'R',%'E',%'P',%'I',%'T',%'E',%'I',%'E',%'R',%'M',%'I',
     $   %'N',%'A',
     $   %'E',%'J',%'E',%'C',%'U',%'T',%'A',%'C',%'O',%'M',%'I',
     $   %'E',%'N',%'Z',%'A',
     $   %'M',%'I',%'E',%'N',%'T',%'R',%'A',%'S'   /
C
C         COMO EL RENGLON DE COPIA ES DE 95 POSICIONES,
C         PORQUE DEDICA 23 CARACTERES A LOS NUMEROS DE
C         SECUENCIA, LOS 72 CARACTERES QUE SE LEEN VAN
C         DE LA COLUMNA 24 A LA 95 .
      DATA LIMIZ,LIMDR  /24, 95/
      DATA NUMRS / %'0',%'1',%'2',%'3',%'4',
     $             %'5',%'6',%'7',%'8',%'9'       /
C                ESTE ES EL FORMATO DEL RENGLON DONDE SE
C                COPIA EL LISTADO FUENTE, ADICIONADO DE SUS
C                NUMEROS SECUENCIALES Y DE REFERENCIA .
      DATA TEXTO  /  6*%' ', %':', 3*%' ', %'(', %'F',
     $               5*%' ', %')', %':', 4*%' '       /
      END


PROGRAM UNIT CARGA COMPILED
```

```
C*********************************************************
      PROGRAM FOREST
C
C     ***********************************************************
C  *                                                             *
C  *  PROGRAMA PRINCIPAL DE 'FOREST' (FORTRAN ESTRUCTURADO)      *
C  *                                                             *
C     ***********************************************************
C
      INTEGER NO,SI,TEXOBJ,UL
      COMMON /DISCO/ IDISC
      COMMON /SALE/ TEXOBJ(80),JR,UL,IMP,NGOTO,NFTN
      DATA NO,SI  / %'N', %'S'  /
      UL = 6
  100 CONTINUE
      WRITE(6,650)
  650 FORMAT(/,'?QUIERE ESCRIBIR LA COPIA EN DISCO (FTN10)?')
      READ(5,550) IX
      IMP = 6
      IF( IX .EQ. SI ) IMP = 10
C     ---- LEER DE DISCO O DE PANTALLA ----
      WRITE(6,600)
  600 FORMAT(/,'?VA A LEER DE DISCO (FTN09)?')
      READ(5,550) IX
      IDISC = 5
      IF( IX .EQ. SI ) IDISC = 9
C
      CALL PARSER
      WRITE(6,500)
  500 FORMAT(///,40H  ?QUIERE CORRER OTRO PROGRAMA? ( S/N ) )
      READ(5,550) IX
  550 FORMAT(1A1)
      IF( IX .NE. NO ) GO TO 100
      WRITE(6,570)
  570 FORMAT(///,17H      ADIOS PUES. )
      STOP
      END


PROGRAM UNIT FOREST COMPILED
```

```
C*******************************************************
      SUBROUTINE PARSER
C RUTINA PRINCIPAL DE  FOREST  .  HACE 'PARSING' SOBRE LA GRAMATICA
C DE  FOREST .  EL RECORRIDO SOBRE LA GRAMATICA ES DE TIPO DE "DES-
C CENSO RECURSIVO".   EL PROGRAMA SIMULA LA RECURSION POR MEDIO DE
C UNA PILA DONDE SE GUARDAN LAS DIRECCIONES DE REGRESO.
C A LA GRAMATICA SE LE HAN INCORPORADO "ACCIONES" QUE GENERAN CODI-
C GO UNA VEZ QUE SE HIZO EL RECONOCIMIENTO SINTACTICO.
C SE COMENTARAN LAS RUTINAS QUE SE USEN POR PRIMERA VEZ TAN SOLO.
C ESTE PROGRAMA LLAMA A: CODCAS, CODDO, CODFR1, CODFR2, CODIF,
C                        COPIA,EOFPRE,EXPR,ERROR,FIN,GENET,GENSAL,
C                        INICIA, MUEVE,OTRO, POP,
C                        PUSH,QUITET,SACA,SACONT,SACTOK,SAGOTO.
C ---TESIS,   GUILLERMO LEVINE.     ABRIL 1979.---
C
      INTEGER ADV,APDIC,BI,BJ,BK,BUF
      INTEGER CAS,CASAP,CASO,CASTP,CASVAR,CLDIC,DIC
      INTEGER ERRORS,ETIQ,ETIQBK,ETIQNX,INADA,JR
      INTEGER NCASOF(15),NFOK,NLIN,NUMRS,NV,NX,P
      INTEGER PILA,PETIQ,R,REINIC,STACK,TEMP,TEMP1,TEMP2
      INTEGER SESP1,SESP2,TAB,TEXOBJ,TEXTO,TOKEN,UL,ULTIMO
      INTEGER WCOMA,WDATA,WPESOS
C
      COMMON /CASS/ CASVAR(100),CASAP(15,2),CAS,CASTP,NUTHER
      COMMON /CONT/ NLIN,ERRORS,ADV
      COMMON /DIC1/ DIC(113),CLDIC(24),APDIC(24)
      COMMON /DISCO/ IOISC
      COMMON /ETIQT/ PILA(50),ETIQ,PETIQ,ETIQBK(15),
     $               BK,ETIQNX(15),NX
      COMMON /FOR/ REINIC(150),NV
      COMMON /ITEXT/ BUF(80)
      COMMON /LEX/ ULTIMO,TEXTO(95),IULT,LIMIZ,LIMDR
      COMMON /NUMR/ NUMRS(10)
      COMMON /SALE/ TEXOBJ(80),JR,UL,IMP,NGOTO,NFTN
      COMMON /TORRE/ STACK(50),P
C
C         ESTOS SON SIMBOLOS ESPECIALES QUE PUEDEN
C         REQUERIRSE EN ALGUN COMPILADOR.
      DATA SESP1  / %'$' /
C     DATA SESP2  /  /
      DATA LE,LN,LD  /%'E',%'N',%'D'/
      DATA IBEGIN,ICOM,IELSE,IEND,IEOF,IFTN,IHASTA
     $     /  15,   23,   8,    4,  26,  30,   16     /
      DATA INLIN,IOTRO,IPCOMA,ISIMB,ITERM,IUNT
     $     /  25,   10,   29,   28,  21,   18    /
      DATA WCOMA,WDATA,WPESOS  / %',', %'/', %'$' /
C          "TAB"  DEPENDE DE LA TERMINAL...
C      DATA TAB / %'\'  /
C                    INICIALIZACIONES VARIAS....
      NFOR = 0
      IT = 0
      CALL INICIA
C                    CARGA LA TABLA DE PALABRAS CLAVE DE FOREST.
C ------------------------------CICLO PRINCIPAL-----------------.
C                   (TODO EL PROGRAMA NO ES MAS QUE UNA
C                   ITERACION DE TIPO "CASO".)
   50 CONTINUE
```

```
      PARSER

C                     "OTRO" OBTIENE UN NUEVO COMPONENTE SINTACTICO
      CALL OTRO(TOKEN,BI,1)
   60 CONTINUE
      GO TO (  300, 300, 800,1900, 600, 600, 900,2500, 900,
     $        2500,1200,1100, 500,1100, 200,2500,1200,2500,
     $         700, 700,1900, 800, 200, 500,  50,2600,1300,
     $        2300,  50,2000  ), TOKEN
C       GO TO 2300
C     ---------------CASO 'BEGIN'-------------------------
  200 CONTINUE
      CALL OTRO(TOKEN,BI,1)
      IF( TOKEN .EQ. INLIN .OR. TOKEN .EQ. IPCOMA ) GO TO 200
      IF( TOKEN .EQ. IEND .OR. TOKEN .EQ. ITERM ) GO TO 2100
C                     SE GUARDAN LAS DIRECCIONES DE REGRESO .
      CALL PUSH(2)
      GO TO 60
C     ---------------CASO 'IF'----------------------------
  300 CONTINUE
C                     SE GENERAN 2 ETIQUETAS NUEVAS .
      CALL GENET(2)
      TEMP1 = PILA( ETIQ-1 )
C                     GENERA CODIGO PARA 'IF' .
      CALL CODIF( TEMP1 )
      CALL PUSH(3)
      GO TO 50
C                     REGRESO DE LA RECURSION 'IF' .
  350 CONTINUE
C                   LA VARIABLE IULT CONTROLA EL CASO ESPECIAL
C                   DE LEER UN RENGLON LLENO HASTA LA ULTIMA
C                   COLUMNA.  ES UN CASO LIMITE.
      IULT = 1
  351 CALL OTRO(TOKEN,BI,1)
      IF( TOKEN .EQ. INLIN .OR. TOKEN .EQ. IPCOMA ) GO TO 351
      IF( TOKEN .EQ. IELSE .OR. TOKEN .EQ. IOTRO ) GO TO 400
      TEMP1 = PILA( ETIQ-1 )
C                     GENERA  "  -ETIQUETA-    CONTINUE   " .
      CALL SACONT( TEMP1 )
C                     BORRA LAS ETIQUETAS GENERADAS ANTES .
      CALL QUITET(2,1)
C                     QUITAR LOS 'ELSE' DE LA PILA...
  355 CALL POP(R)
      IF( R .NE. 4 ) GO TO 360
      TEMP2 = PILA( ETIQ )
      CALL SACONT( TEMP2 )
      CALL QUITET(2,1)
      GO TO 355
  360 CALL PUSH(R)
C         ES NECESARIO "REGRESAR" CARACTERES YA LEIDOS
C         AL RENGLON DE ENTRADA.
      IF( IT .EQ. 0 ) ULTIMO = ULTIMO - BI + 1
      IT = 0
      GO TO 2100
C     ---------------CASO 'ELSE'--------------------------
  400 CONTINUE
      IULT = 0
      TEMP1 = PILA( ETIQ-1 )
      TEMP2 = PILA( ETIQ )
```

```
C                       GENERA  "      GO TO  -ETIQUETA-  " .
C            CUIDAMOS DE NO IMPRIMIR 2 O MAS GOTO'S SEGUIDOS.
      IF( NGOTO .NE. 0 ) GO TO 410
      NGOTO = 1
      CALL MUEVE(6)
      CALL SAGOTO( TEMP2 )
  410 CONTINUE
      CALL SACONT( TEMP1 )
C               CON  IT  EVITAREMOS QUE SE VUELVA A LEER EL  ELSE .
      IT = 1
      CALL PUSH(4)
      CALL OTRO(TOKEN,BI,1)
      IF( TOKEN .EQ. IPCOMA ) GO TO 355
      IT = 0
      GO TO 60
C                       REGRESO DE LA RECURSION 'ELSE' ...
  450 CONTINUE
      TEMP2 = PILA( ETIQ )
      CALL SACONT( TEMP2 )
      CALL QUITET(2,1)
      GO TO 2100
C     ---------------CASO 'WHILE'-------------------------
  500 CONTINUE
      CALL GENET(2)
      TEMP1 = PILA( ETIQ-1 )
      TEMP2 = PILA( ETIQ )
C                     GUARDA ETIQUETAS PARA 'BREAK' Y 'NEXT'
      CALL GENSAL(1)
      CALL GENSAL(2)
      CALL SACONT( TEMP1 )
      CALL CODIF( TEMP2 )
      CALL PUSH(5)
      GO TO 50
C                       REGRESO DE LA RECURSION 'WHILE' ...
  550 CONTINUE
C                       AVANZA A LA POSICION 7 DEL RENGLON .
      TEMP1 = PILA( ETIQ-1 )
      TEMP2 = PILA( ETIQ )
C                   NO IMPRIMIR DOS  GOTO  SEGUIDOS.
      IF( NGOTO .NE. 0 ) GO TO 553
      CALL MUEVE(6)
      CALL SAGOTO( TEMP1 )
      NGOTO = 1
  553 CONTINUE
      CALL SACONT( TEMP2 )
      CALL QUITET(2,2)
      GO TO 2100
C     ---------------CASO 'FOR'----------------------------
C                       SE COMPLICA UN POCO CON VARIABLES ESPECIALES
C                       PARA MANEJAR LOS DIVERSOS CASOS POSIBLES...
  600 CONTINUE
      CALL GENET(2)
      CALL GENSAL(1)
      CALL GENSAL(2)
      NFOR = NFOR + 1
      IF( NFOR .LE. 15 ) GO TO 610
      CALL ERROR(2)
```

```
      PARSER

          ULTIMO = LIMDR
          CALL QUITET(2,2)
          GO TO 50
      C
        610 CONTINUE
          TEMP1 = PILA( ETIQ-1 )
          TEMP2 = PILA( ETIQ )
      C                    GENERA CODIGO 'FOR' .
          CALL CODFR1( TEMP1,TEMP2,CASO )
          NCASOF(NFOR) = CASO
          CALL PUSH(6)
          CALL LIMPIA
        611 CALL OTRO(TOKEN,BI,1)
          IF( TOKEN .EQ. INLIN ) GO TO 611
          IF( TOKEN .NE. IPCOMA ) GO TO 60
          GO TO 2100
      C                    REGRESO DE LA RECURSION 'FOR' ...
      C                    CASO = 0 : 'FOR' INCORRECTO.
      C                    CASO = 2 : NO HUBO -CONDICION-.
      C                    CASO = 3 : NO HUBO -CONDICION- NI -REINICIO-.
      C                    CASO = 4 : NORMAL
      C                    CASO = 5 : NO HUBO -REINICIO-.
        650 CONTINUE
          TEMP1 = PILA( ETIQ-1 )
          TEMP2 = PILA( ETIQ )
          CASO = NCASOF(NFOR)
          NFOR = NFOR - 1
      C                    VARIAS POSIBILIDADES, GENERAR O NO EL
      C                    CODIGO COMPLETO, DEPENDIENDO DEL CASO ...
          IF( CASO .EQ. 0 ) GO TO 680
          IF( CASO .EQ. 2 .OR. CASO .EQ. 4 ) CALL CODFR2
          IF( CASO .EQ. 2 .OR. CASO .EQ. 3 ) GO TO 670
      C                  NO IMPRIMIR DOS  GOTO  SEGUIDOS.
          IF( NGOTO .NE. 0 ) GO TO 653
          CALL MUEVE(6)
          CALL SAGOTO( TEMP1 )
          NGOTO = 1
        653 CONTINUE
        670 CALL SACONT( TEMP2 )
        680 CALL QUITET(2,2)
          GO TO 2100
      C     ----------------CASO 'REPEAT'-----------------------
        700 CONTINUE
          CALL GENET(2)
          CALL GENSAL(1)
          CALL GENSAL(2)
          TEMP1 = PILA( ETIQ-1 )
          CALL SACONT( TEMP1 )
          CALL PUSH(7)
          GO TO 50
      C                    REGRESO DE LA RECURSION 'REPEAT' ...
      C           PROBABLEMENTE SEA NECESARIO TENER QUE "REGRESAR"
      C           CARACTERES YA LEIDOS AL RENGLON DE ENTRADA.
        750 CONTINUE
          IULT = 1
        751 CALL OTRO(TOKEN,BI,1)
          IF( TOKEN .EQ. INLIN .OR. TOKEN .EQ. IPCOMA ) GO TO 751
```

```
      IF( TOKEN .EQ. IUNT .OR. TOKEN .EQ. IHASTA ) GO TO 760
      TEMP1 = PILA( ETIQ-1 )
      TEMP2 = PILA( ETIQ )
C             NO IMPRIMIR DOS  GOTO  SEGUIDOS...
      IF( NGOTO .NE. 0 ) GO TO 753
      CALL MUEVE(6)
      CALL SAGOTO( TEMP1 )
      NGOTO = 1
  753 CONTINUE
      CALL SACONT( TEMP2 )
      CALL QUITET(2,2)
      ULTIMO = ULTIMO - BI + 1
      GO TO 2100
C     ---------------CASO 'UNTIL'-------------------------
  760 CONTINUE
      IULT = 0
      TEMP1 = PILA( ETIQ-1 )
      TEMP2 = PILA( ETIQ )
      CALL CODIF( TEMP1 )
      CALL SACONT( TEMP2 )
      CALL QUITET(2,2)
      GO TO 2100
C     ---------------CASO 'DO'----------------------------
  800 CONTINUE
      CALL GENET(2)
      CALL GENSAL(1)
      CALL GENSAL(2)
C                       GENERA CODIGO 'DO' ...
      TEMP1 = PILA( ETIQ-1 )
      CALL CODDO( TEMP1 )
      CALL PUSH(8)
      GO TO 50
C                       REGRESO DE LA RECURSION 'DO' ...
  850 CONTINUE
      TEMP1 = PILA( ETIQ-1 )
      TEMP2 = PILA( ETIQ )
      CALL SACONT( TEMP1 )
      CALL SACONT( TEMP2 )
      CALL QUITET(2,2)
      GO TO 2100
C     ---------------CASO 'CASE'--------------------------
  900 CONTINUE
C     SI -CASTP- ES NEGATIVO, QUIERE DECIR QUE HUBO ERROR.
C     SI -CAS- ES NEGATIVO QUIERE DECIR QUE HUBO 'CASE' NULO.
C     -NOTHER- SIRVE PARA CONTROLAR LAS "ETIQUETAS VACIAS".
      NOTHER = 0
C                       GENERA CODIGO 'CASE' ...
C                       CUIDA LOS 'CASE' NULOS...
      CALL CODCAS(1,TOKEN,BI)
      IF( NOTHER .EQ. 1 ) CALL ERROR(22)
      IF( CASTP .GE. 0 ) GO TO 910
  905 CASTP = IABS(CASTP)
      GO TO 2100
  910 CONTINUE
      IF( CAS .GE. 0 ) GO TO 60
      CAS = IABS(CAS)
      GO TO 2100
```

```
C                       REGRESO DE LA RECURSION 'CASE' .
  950 CONTINUE
C                       ARREGLOS AUXILIARES PARA EL CASO 'CASE' .
      TEMP1 = CASAP( CAS,2 )
      TEMP2 = PILA( ETIQ )
C                 NO IMPRIMIR DOS  GOTO  SEGUIDOS...
      IF( NGOTO .NE. 0 ) GO TO 955
      CALL MUEVE(6)
      CALL SAGOTO( TEMP1 )
      NGOTO = 1
  955 CONTINUE
      CALL SACONT( TEMP2 )
      CALL QUITET(1,1)
  956 CALL OTRO(TOKEN,BI,2)
      IF( TOKEN .EQ.INLIN .OR. TOKEN .EQ. IPCOMA ) GO TO 956
      IF( TOKEN .EQ. IEND .OR. TOKEN .EQ. ITERM ) GO TO 960
      IF( NOTHER .EQ. 1 ) CALL ERROR(22)
      CALL CODCAS(2,TOKEN,BI)
C           VER SI HUBO ERROR.
      IF( CASTP .LT. 0 ) GO TO 905
      IF( CAS .GE. 0 ) GO TO 60
      CAS = IABS(CAS)
      GO TO 2100
C
  960 CONTINUE
      TEMP1 = CASAP( CAS,2 )
      CALL SACONT( TEMP1 )
      CASTP = CASAP( CAS,1 )
      CALL QUITET(1,3)
      CAS = CAS - 1
      NOTHER = 0
      GO TO 2100
C     ----------------CASO 'BREAK'-------------------------
 1100 CONTINUE
      IF( BK .GE. 1 ) GO TO 1130
      CALL ERROR(4)
      GO TO 2560
 1130 CONTINUE
C                       NO IMPRIMIR DOS  GOTO  SEGUIDOS.
      IF( NGOTO .NE. 0 ) GO TO 2100
      CALL MUEVE(6)
      TEMP1 = ETIQBK( BK )
      CALL SAGOTO( TEMP1 )
      NGOTO = 1
      GO TO 2100
C     ----------------CASO 'NEXT'--------------------------------
 1200 CONTINUE
      IF( NX .GE. 1 ) GO TO 1230
      CALL ERROR(5)
      GO TO 2560
 1230 CONTINUE
C                       NO IMPRIMIR DOS  GOTO  SEGUIDOS.
      IF( NGOTO .NE. 0 ) GO TO 2100
      CALL MUEVE(6)
      TEMP1 = ETIQNX( NX )
      CALL SAGOTO( TEMP1 )
      NGOTO = 1
```

```
      GO TO 2100
C     ----------------CASO "ETIQUETA"-----------------------------
 1300 CONTINUE
C                       IMPRIME EL 'TOKEN' A LA SALIDA...
      CALL SACTOK(BI)
      CALL OTRO(TOKEN,BI,1)
      IF( TOKEN .EQ. IFTN ) GO TO 2000
      CALL MUEVE(6)
      CALL SACONT(0)
      GO TO 60
C     ----------------CASO 'END' DE FORTRAN (FINAL)--------------
 1900 CONTINUE
      CALL OTRO(TOKEN,BI,2)
      IF( TOKEN .EQ. INLIN ) GO TO 1920
      CALL ERROR(7)
      CALL ERROR(1)
      ULTIMO = LIMOR
      GO TO 2100
C
 1920 CONTINUE
      CALL OTRO(TOKEN,BI,1)
      IF( TOKEN .EQ. IEOF ) GO TO 1950
      CALL ERROR(7)
      GO TO 60
C
 1950 CONTINUE
      CALL MUEVE(6)
      CALL SACA( LE  )
      CALL SACA( LN  )
      CALL SACA( LD  )
      CALL SIGREN
      GO TO 2600
C     ----------------CASO "FORTRAN"-----------------------------
 2000 CONTINUE
      CALL MUEVE(6)
C                       COPIA LO QUE HAYA A LA SALIDA ...
 2005 CALL COPIA(BI)
C       EXTENDER AL SIGUIENTE RENGLON DE FORTRAN SI EL
C       ACTUAL DE FOREST TERMINO CON COMA.
      IF( BI .GT. 0 ) GO TO 2100
      CALL MUEVE(5)
      CALL SACA( WPESOS )
      CALL OTRO(TOKEN,BI,2)
      GO TO 60
C
 2100 CONTINUE
C                       REGRESO DE LA RECURSION, CASO "FORTRAN" ..
      IF( P .LE. 1 ) GO TO 50
C                       SE REMUEVE EL ULTIMO ELEMENTO DE LA PILA ...
      CALL POP(R)
      GO TO ( 50,200,350,450,550,650,750,850,950 ), R
C     ----------------CASO "SIMB" -------------------------------
 2300 CONTINUE
C               CUIDADO CON LOS TABULADORES...
C     IF( BUF(1) .EQ. TAB ) GO TO 50
C       SIMBOLOS ESPECIALES PARA EL COMPILADOR (SE COPIA
C       EL RENGLON A PARTIR DE LA COLUMNA 1 .
```

```
C       NOTESE QUE SI UN SIMBOLO ESPECIAL DE COMPILADOR
C       ES UN '/', ENTONCES NO SE PODRA EXTENDER AUTO-
C       MATICAMENTE A OTRO RENGLON DE FORTRAN AL ENCON-
C       TRAR EL '/' DE UN -DATA-, POR EJEMPLO.  EN ESTOS
C       CASOS HABRA QUE TERMINAR O COMENZAR EL RENGLON
C       DEL -DATA- CON UNA COMA PARA LOGRAR LA CONTINUA-
C       CION AUTOMATICA.
        IF( BUF(1) .EQ. SESP1 ) GO TO 2005
C       IF( BUF(1) .EQ. SESP2 ) GO TO 2005
C       EXTENDER AL SIGUIENTE RENGLON DE FORTRAN SI EL NUEVO
C      RENGLON DE FOREST COMIENZA CON COMA O CON '/' (VER NOTA ARRIBA).
        IF( BUF(1) .NE. WCOMA .AND. BUF(1) .NE. WDATA )
       $                                          GO TO 2360
        CALL MUEVE(5)
        CALL SACA( WPESOS )
        GO TO 2005
 2360 CONTINUE
C                       EXPANSION DE LOS CASOS "++VAR" Y "--VAR" .
        CALL EXPR(TEMP)
        IF( TEMP .NE. 1 ) CALL ERROR(6)
        GO TO 2100
C       ----------------CASO DE PALABRA CLAVE MAL USADA-------------
 2500 CONTINUE
        CALL ERROR(7)
        GO TO 2100
C       ------------------------------------------------------
C       AVERIGUAR SI HAY ALGO MAS EN LO QUE SOBRA DEL RENGLON.
 2560 CONTINUE
        CALL OTRO(TOKEN,BI,1)
        IF( TOKEN .EQ. INLIN .OR. TOKEN .EQ. IPCOMA ) GO TO 2100
        IF( TOKEN .EQ. IEOF ) GO TO 2600
        CALL ERROR(1)
        ULTIMO = LIMDR
        GO TO 2100
C       -----------------CASO "EOF"---------------------------------
 2600 CONTINUE
        IF( P .GT. 1 ) CALL EOFPRE
        CALL FIN
        RETURN
        END
```

PROGRAM UNIT PARSER COMPILED

```
C*****************************************************************
      SUBROUTINE INICIA
C INICIALIZACIONES VARIAS PARA PARSER .
      INTEGER MREG(80),ESE
      INTEGER ADV,BK,CAS,CASAP,CASTP,CASVAR
      INTEGER ERRORS,ETIQ,ETIQBK,ETIQNX,P,PETIQ
      INTEGER PILA,REINIC,STACK,TEXOBJ,TEXTO
      INTEGER UL,ULTIMO
      COMMON /CASS/ CASVAR(100),CASAP(15,2),CAS,CASTP,NOTHER
      COMMON /CONT/ NLIN,ERRORS,ADV
      COMMON /DISCO/ IDISC
      COMMON /ETIQT/ PILA(50),ETIQ,PETIQ,ETIQBK(15),
     $               BK,ETIQNX(15),NX
      COMMON /FOR/ REINIC(150),NV
      COMMON /LEX/ ULTIMO,TEXTO(95),IULT,LIMIZ,LIMDR
      COMMON /SALE/TEXOBJ(80),JR,UL,IMP,NGOTO,NFTN
      COMMON /TORRE/ STACK(50),P
C     SI SE TRABAJA EN 'BATCH', ENTONCES REEMPLAZAR
C      =DATA MREG= POR  =DATA TEXOBJ= Y ELIMINAR  'MREG'.
      DATA MREG  / %'C',27*%' ',%'*',%'F',%'O',%'R',%'E',%'S',
     $             %'T',%'*',
     $          %' ',%' ',%'V',%'E',%'R',%'S',%'I',%'O',%'N',%' ',
     $       %'1',%'.',%'2',10*%' ',%'C',%'H',%'U',%'I',%'K',%'.',
     $             15*%' '   /
      DATA  ESE / %'S' /
C
C       ESTE -DO- NO SERA NECESARIO SI SE TRABAJA EN 'BATCH'.
      DO 100 I = 1,80
         TEXOBJ(I) = MREG(I)
  100 CONTINUE
C
      ADV = 0
      BK = 0
      CAS = 0
      CASTP = 1
      ERRORS = 0
      ETIQ = 0
      JR = 0
      IULT = 0
      NFTN = 1
      NGOTO = 0
      NLIN = 0
      NV = 1
      NX = 0
      P = 1
      PETIQ = 7700
      STACK(1) = 1
C ---- UL  ES LA UNIDAD LOGICA DONDE SE HARA LA ESCRITURA ----
C ---- IMP ES LA "PANTALLA" O TELETIPO DE COMANDOS.
      WRITE(6,500)
  500 FORMAT(/,'?QUIERES ESCRIBIR EL P. OBJETO EN DISCO? (FIN07)')
      READ(5,550) IX
  550 FORMAT(1A1)
      IF( IX .EQ. ESE) UL = 7
C
C
      ULTIMO = LIMDR
```

```
INICIA

C                       IMPRIME EL PRIMER RENGLON .
C     EN AQUELLOS COMPILADORES EN QUE SEA ESTRICTAMENTE
C     NECESARIO QUE EL PRIMER RENGLON FISICO SEA EL QUE
C     CONTIENE UN SIMBOLO ESPECIAL DEL COMPILADOR (SESP1/2),
C     HABRA QUE REEMPLAZAR  "CALL SIGREN" POR "CALL LIMPIA".
      CALL SIGREN
C
      RETURN
      END
```

PROGRAM UNIT INICIA COMPILED

```
C*********************************************************************
      SUBROUTINE OTRO(TOKEN,BI,CASO)
C RUTINA QUE OBTIENE E IDENTIFICA UN 'TOKEN' DEL TEXTO.
      INTEGER TOKEN,CLASE,COMP,NUMER,BI,SI,KWORD
      INTEGER ALFAN,PCOMA,EOF,NWLINE,SIMB
      INTEGER CASO,CLAVE
      DATA ALFAN,EOF,NWLINE,PCOMA,SI,SIMB
     $    /  4100,4110,4135,4140,4150,4155    /
      DATA INLIN,IEOF,IETIQ,ISIMB,IPCOMA,IFTN
     $    /  25,   26,   27,   28,    29,   30  /
C
  100 CLASE = COMP(BI)
C                 ELIMINAR LOS COMENTARIOS
      IF( CLASE .EQ. SI ) GO TO 100
C
      IF( CLASE .EQ. ALFAN) GO TO 300
      IF( CLASE .EQ. PCOMA) TOKEN = IPCOMA
      IF( CLASE .EQ. EOF  ) TOKEN = IEOF
      IF( CLASE .EQ. NWLINE)TOKEN = INLIN
      IF( CLASE .EQ. SIMB ) TOKEN = ISIMB
      IF( CLASE .EQ. EOF .AND. CASO .EQ. 2 )CALL EOFPRE
      RETURN
  300 CONTINUE
C CASO GENERAL (2) . NO LLAMADO DESDE  PARSER  .  NO HACE
C FALTA PREGUNTAR SI ES O NO ETIQUETA.
      IF( CASO .EQ. 2 ) GO TO 310
C                     ---CLASE = ETIQ.---
      IF( NUMER(BI) .NE. SI) GO TO 310
      TOKEN = IETIQ
      RETURN
  310 CONTINUE
C                     ---CLASE = FOREST---
      IF( CLAVE(BI,KWORD) .NE. SI) GO TO 350
      TOKEN = KWORD
      RETURN
  350 CONTINUE
C                     ---CLASE = FTN---
      TOKEN = IFTN
      RETURN
      END


PROGRAM UNIT OTRO COMPILED
```

```
C*********************************************************************
      SUBROUTINE PUSH(R)
C RUTINA QUE SIMULA RECURSIVIDAD POR MEDIO DEL "STACK".
      INTEGER P, R, STACK
      COMMON /TORRE/ STACK(50),P
      P = P + 1
      IF( P .LE. 50 ) GO TO 110
      CALL ERROR(15)
      RETURN
  110 CONTINUE
      STACK(P) = R
      RETURN
      END


PROGRAM UNIT PUSH COMPILED
```

```
C  ******************************************************
      SUBROUTINE POP(R)
C RUTINA PARA SIMULAR RECURSION POR MEDIO DEL 'STACK'.
      INTEGER P,R,STACK
      COMMON /TORRE/ STACK(50),P
      IF( P .GT. 1 ) GO TO 110
C PROTEGER EL CASO DE QUE EL 'STACK' ESTE "VACIO"; I.E., NO HACE
C FALTA SIMULAR RECURSION.
      P = 1
      R = 1
      RETURN
  110 CONTINUE
      R = STACK(P)
      P = P - 1
      RETURN
      END
```

PROGRAM UNIT POP COMPILED

```
C*************************************************************
      SUBROUTINE ERROR(NUM)
C RUTINA PARA IMPRESION DE MENSAJES DE ERROR Y DE
C ADVERTENCIA.
      INTEGER ADV,SIGNO,ERRORS,FLECHA(80),TEXOBJ
      INTEGER TEXTO,UL,ULTIMO,WBCO
      COMMON /CONT/ NLIN,ERRORS,ADV
      COMMON /LEX/ ULTIMO,TEXTO(95),IULT,LIMIZ,LIMDR
      COMMON /SALE/ TEXOBJ(80),JR,UL,IMP,NGOTO,NFTN
      DATA SIGNO, FLECHA, WBCO  / %"^", 80*%' ' , %' ' /
C
      IF( NUM .EQ. 9 .OR. NUM .EQ. 22 ) GO TO 500
      IT = ULTIMO + 1
      FLECHA( IT ) = SIGNO
C        SE IMPRIMEN HASTA 70 POSICIONES POR
C        LIMITACIONES DE LA PANTALLA...
      WRITE(IMP,400)  ( FLECHA(I), I=1,70)
  400 FORMAT(70R1)
      FLECHA( IT ) = WBCO
C
  500 CONTINUE
      ERRORS = ERRORS + 1
      GO TO ( 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,
     $        12,13,14,15,16,17,18,19,20,21,22  ), NUM
    1 CONTINUE
      WRITE(IMP,501)
  501 FORMAT(45H  **CUIDADO: SE IGNORA EL RESTO DEL RENGLON**)
      ERRORS = ERRORS - 1
      ADV = ADV + 1
      RETURN
    2 CONTINUE
      WRITE(IMP,502)
  502 FORMAT(48H  **ERROR: MAS DE 15 'FOR' ANIDADOS, SE IGNORA**)
      RETURN
    3 CONTINUE
      WRITE(IMP,503)
  503 FORMAT(48H  **ERROR: MALA SINTAXIS EN EL 'CASE', SE IGNORA,
     $       11H COMPLETO**)
      RETURN
    4 CONTINUE
      WRITE(IMP,504)
  504 FORMAT(26H  **ERROR: 'BREAK' ILICITO**)
      RETURN
    5 CONTINUE
      WRITE(IMP,505)
  505 FORMAT(27H  **ERROR: 'NEXT' ILICITO**)
      RETURN
    6 CONTINUE
      WRITE(IMP,506)
  506 FORMAT(34H  **ERROR: SIMBOLO IRRECONOCIBLE**)
      RETURN
    7 CONTINUE
      WRITE(IMP,507)
  507 FORMAT(41H  **ERROR: USO ILICITO DE PALABRA CLAVE**)
      RETURN
    8 CONTINUE
      WRITE(IMP,508)
```

```
508 FORMAT(43H  **ERROR FATAL: FIN DE ARCHIVO PREMATURO**)
    RETURN
  9 CONTINUE
    WRITE(IMP,509)
509 FORMAT(36H  **ERROR: HACE FALTA UN APOSTROFO**)
    RETURN
 10 CONTINUE
    WRITE(IMP,510)
510 FORMAT(46H  **ERROR: 'FOR' INCORRECTO, SE IGNORA TODO EL,
   $       10H RENGLON**)
    RETURN
 11 CONTINUE
    WRITE(IMP,511)
511 FORMAT(49H  **ERROR: DEMASIADOS 'FOR' ANIDADOS, SE IGNORA**)
    RETURN
 12 CONTINUE
    WRITE(IMP,512)
512 FORMAT(49H  **CUIDADO: COMBINACION IRRECONOCIBLE DE SIMBOLO,
   $       25HS, SE TOMARA EL PRIMERO**)
    ERRORS = ERRORS - 1
    ADV = ADV + 1
    RETURN
 13 CONTINUE
    WRITE(IMP,513)
513 FORMAT(42H  **ERROR: MALA SINTAXIS EN LA CONDICION**)
    RETURN
 14 CONTINUE
    WRITE(IMP,514)
514 FORMAT(49H  **ERROR: MALA SINTAXIS EN LA CONSTRUCCION, SE I,
   $       23HGNORA TODO EL RENGLON**)
    RETURN
 15 CONTINUE
    WRITE(IMP,515)
515 FORMAT(32H  ****'PARSE STACK OVERFLOW'****)
    RETURN
 16 CONTINUE
    WRITE(IMP,516)
516 FORMAT(49H  **ERROR: ETIQUETA CON MAS DE CINCO CARACTERES**)
    RETURN
 17 CONTINUE
    WRITE(IMP,517)
517 FORMAT(43H  **ERROR: POSIBLE CONFLICTO DE ETIQUETAS**)
    RETURN
 18 CONTINUE
    WRITE(IMP,518)
518 FORMAT(49H  **ERROR: MALA SINTAXIS EN EL 'DO', SE IGNORA EL,
   $       10H RENGLON**)
    RETURN
 19 CONTINUE
    WRITE(IMP,519)
519 FORMAT(45H  **ERROR: SOBREFLUJO EN LA PILA DE 'BREAK'**)
    RETURN
 20 CONTINUE
    WRITE(IMP,520)
520 FORMAT(44H  **ERROR: SOBREFLUJO EN LA PILA DE 'NEXT'**)
    RETURN
 21 CONTINUE
```

```
      WRITE(IMP,521)
  521 FORMAT(37H  **ERROR: DEMASIADOS 'CASE' ANIDADOS,
     $       58H O DEMASIADAS VARIABLES DE CONTROL; SE,
     $       18H IGNORA COMPLETO**)
      RETURN
   22 CONTINUE
      WRITE(IMP,522)
  522 FORMAT(43H  **CUIDADO: LA "ETIQUETA VACIA" DEL 'CASE',
     $       42H  ESTA MAL POSICIONADA, O HAY MAS DE UNA**)
      ERRORS = ERRORS + 1
      ADV = ADV + 1
      RETURN
      END
```

PROGRAM UNIT ERROR COMPILED

```
C*************************************************************
      SUBROUTINE GENET(N)
C RUTINA QUE GENERA  N  ETIQUETAS EN LA PILA
C ESTAS ETIQUETAS SE PRODUCEN A PARTIR DEL NUMERO 77700 .
C COMO ESTA MAQUINA (Z80) ES DE 8 BITS NO ACEPTA NUMEROS
C MAYORES A 32767.   LO QUE HAREMOS SERA PETIQ=7700 Y LUEGO
C LE PONDREMOS EL 7 MAS SIGNIFICATIVO, EN LA RUTINA  CARDEC  .
      INTEGER N,PILA,ETIQ,PETIQ,ETIQBK,ETIQNX,BK,NX
      COMMON /ETIQT/ PILA(50),ETIQ,PETIQ,ETIQBK(15),BK,
     $               ETIQNX(15),NX
      DO 150 I = 1,N
      ETIQ = ETIQ + 1
      PILA(ETIQ) = PETIQ
      PETIQ = PETIQ + 1
  150 CONTINUE
      RETURN
      END


PROGRAM UNIT GENET COMPILED
```

```
C*************************************************************
      SUBROUTINE QUITET(N,CASO)
C RUTINA QUE QUITA  N  ETIQUETAS DE LA PILA .
C ADEMAS, LLEVA CONTROL DE LAS ETIQUETAS (ESPECIALES) QUE
C PUEDEN REQUERIR LAS INSTRUCCIONES BREAK  Y  NEXT  .
C CASO = 1 : SOLO ELIMINA ETIQUETAS DE LA PILA GENERAL.
C CASO = 2 : ELIMINA LAS ETIQUETAS DE  BREAK  Y  NEXT , Y DE PILA.
C CASO = 3 : SOLO ELIMINA LAS ETIQUETAS DE PILA Y  BREAK .
      INTEGER N,PILA,ETIQ,PETIQ,ETIQBK,ETIQNX
      INTEGER BK,NX,CASO,ERRORS,ADV
      COMMON /ETIQT/ PILA(50),ETIQ,PETIQ,ETIQBK(15),BK,
     $               ETIQNX(15),NX
      COMMON /CONT/ NLIN,ERRORS,ADV
C
      DO 150 I = 1,N
         ETIQ = ETIQ -1
         IF( ETIQ .LT. 0 ) GO TO 200
  150 CONTINUE
  160 IF( CASO .EQ. 1 ) RETURN
      BK = BK - 1
      IF( CASO .EQ. 3 ) GO TO 170
      NX = NX - 1
C HAY DOS 'IF' PORQUE EL COMPILADOR Z80 ASI LO REQUIERE...
  170 IF( BK .LT. 0 ) GO TO 170
      IF( NX .LT. 0 ) GO TO 260
      RETURN
C
  200 CONTINUE
      ETIQ = 0
      WRITE(6,500)
  500 FORMAT(49H  **ERROR: -UNDERFLOW- EN LA PILA DE ETIQUETAS.**)
      ERRORS = ERRORS + 1
      GO TO 160
  250 CONTINUE
      WRITE(6,550)
  550 FORMAT(44H   **ERROR: -UNDERFLOW- EN  BK  O EN  NX  **)
      ERRORS = ERRORS + 1
      BK = 0
      RETURN
  260 CONTINUE
      WRITE(6,550)
      NX = 0
      RETURN
      END


PROGRAM UNIT QUITET COMPILED
```

```
C*************************************************************
      SUBROUTINE CODIF(LBL)
C  RUTINA PARA PROCESAR Y GENERAR EL CODIGO IF DE FORTST.
      INTEGER BI,TOKEN,BUF,WPARIZ,WPARDE,LBL,CASO
      INTEGER ULTIMO,TEXTO
      COMMON /ITEXT/ BUF(80)
      COMMON /LEX/ ULTIMO,TEXTO(95),IULT,LIMIZ,LIMDR
      DATA WPARIZ,WPARDE / %'(', %')' /
      DATA ISIMB / 28 /
C
      CALL OTRO(TOKEN,BI,2)
C       PREGUNTAR SI EL 'TOKEN' FUE UN SIMBOLO SUELTO...
      IF( TOKEN .EQ. ISIMB) GO TO 110
  105 CONTINUE
      CALL ERROR(14)
      ULTIMO = LIMDR
      CALL LIMPIA
      RETURN
C
  110 CONTINUE
      IF( BUF(1) .NE. WPARIZ )GO TO 105
C                 GENERA CODIGO 'IF' EN UN NUEVO RENGLON.
      CALL SACAIF
C       GENERA LA CONDICION .
C       LA VARIABLE CASO AVISA SI LA COND. FUE CORRECTA O NO .
C       ESTA VARIABLE TEMPORAL ES REQUERIDA POR EL Z2D ....
      ITEMP = 1
      CALL COND( ITEMP,CASO )
      IF( CASO .EQ. 2 ) GO TO 105
      CALL SACA(WPARDE)
C       GENERA FINALMENTE "IF(.NOT.(--CONDICION--))GOTO LBL"
      CALL SAGOTO(LBL)
      RETURN
      END


PROGRAM UNIT CODIF COMPILED
```

```
C*************************************************************
      SUBROUTINE SACONT(LBL)
C     ROTINA PARA IMPRIMIR   "   777XX CONTINUE   "
      INTEGER LBL,WBCO
      DATA WBCO, LC, LO,  LN,  LT,  LI,  LU,  LE
     $  / %' ',%'C',%'O',%'N',%'T',%'I',%'U',%'E'/
C
      IF( LBL .GT. 0 ) CALL CARDEC(LBL)
      CALL SACA( WBCO )
      CALL SACA( LC   )
      CALL SACA( LO   )
      CALL SACA( LN   )
      CALL SACA( LT   )
      CALL SACA( LI   )
      CALL SACA( LN   )
      CALL SACA( LU   )
      CALL SACA( LE   )
      CALL SIGREN
      RETURN
      END
```

PROGRAM UNIT SACONT COMPILED

```
C*************************************************************
      SUBROUTINE MUEVE(N)
C  RUTINA QUE MUEVE EL APUNTADOR AL 'BUFFER' DEL
C  TEXTO DE SALIDA.
C  SI  N = 6 , EQUIVALE A COLOCARSE EN LA COLUMNA 7 DE FORTRAN.
      INTEGER TEXOBJ,JR,UL,IMP
      COMMON /SALE/ TEXOBJ(80),JR,UL,IMP,NGOTO,NFTN
C
      JR = N
      RETURN
      END
```

PROGRAM UNIT MUEVE COMPILED

```
C*************************************************************
      SUBROUTINE SAGOTO(LBL)
C  RUTINA QUE IMPRIME "  GO TO -LBL-  " ...
      INTEGER LBL,WBCO
      DATA WBCO, LG, LO, LT   / %' ',%'G',%'O',%'T'  /
C
      CALL SACA( LG  )
      CALL SACA( LO  )
      CALL SACA( LT  )
      CALL SACA( LO )
      CALL SACA( WBCO )
C  LLAMA A RUTINA PARA CONVERTIR A CARACTERES LA ETIQUETA NUMERICA.
      CALL CARDEC(LBL)
      CALL SIGREN
      RETURN
      END
```

PROGRAM UNIT SAGOTO COMPILED

```
C*************************************************************
      SUBROUTINE GENSAL(N)
C  RUTINA QUE GUARDA ETIQUETAS PARA LOS CASOS  BREAK (N=1)
C   Y NEXT (N=2) EN DOS PILAS.
      INTEGER PILA,ETIQ,PETIQ,ETIQBK,BK,ETIQNX
      INTEGER TEMP,TEMP1
      COMMON /ETIQT/ PILA(50),ETIQ,PETIQ,ETIQBK(15),BK,
     $               ETIQNX(15),NX
C
      IF( N .EQ. 2 ) GO TO 200
C          CASO BREAK  ...
      BK = BK + 1
      IF( BK .LE. 15 ) GO TO 120
      CALL ERROR(19)
      RETURN
C
  120 CONTINUE
      TEMP = PILA( ETIQ )
      ETIQBK( BK ) = TEMP
      RETURN
C          CASO NEXT  ...
  200 CONTINUE
      NX = NX + 1
      IF( NX .LE. 15 ) GO TO 220
      CALL ERROR(20)
      RETURN
C
  220 CONTINUE
      TEMP1 = PILA( ETIQ-1 )
      ETIQNX( NX ) = TEMP1
      RETURN
      END


PROGRAM UNIT GENSAL COMPILED
```

```
C*************************************************************
      SUBROUTINE CODFOR(L1,L2,CASO)
C  RUTINA QUE IMPRIME LA PRIMERA SECCION DEL CODIGO DE  FOR  .
C  IMPRIME                          " -INICIALIZA- "
C                  " -LBL-            CONTINUE   "
C                                   " IF(.NOT.(-CONDICION-))GOTO -LBL2. "
C  LA VARIABLE CASO DESCRIBE QUE TIPO DE FOR ENCONTRAMOS.
C  CASO = 0 :  SE IGNORO EL RENGLON, DEFECTUOSO.
C  CASO = 2 :  NO HUBO -CONDICION- EN EL CODIGO FOR  .
C  CASO = 3 :  NO HUBO -CONDICION- NI -REINICIALIZACION-.
C  CASO = 4 :  NORMAL.
C  CASO = 5 :  NO HUBO -REINICIALIZACION-.
C  ESTA RUTINA LLAMA A: COND, COPIA, ERROR, EXPR, LIMPIA, MUEVE, OTRO,
C                       SACA, SALAIF, SACONT, SAGOTO, SACTOK,
C                       SIGREN.
      INTEGER TOKEN,BUF,APARIZ,BI,TEMP,NV,L1,L2
      INTEGER REINIC,ULTIMO,TEXTO,BAL,NV1,APARDE
      INTEGER K1,BJ,CASO,APOS,APCOMA
      COMMON /TEXT/ BUF(80)
      COMMON /FOR/ REINIC(150),NV
      COMMON /LEX/ ULTIMO,TEXTO(95),IULT,LIMIZ,LIMDR
      DATA APARIZ,APARDE / %'(', %')' /
      DATA INLIN,ISIMB,IPCOMA,IFIN / 25,28,29,30 /
      DATA APOS / %"'" /
      DATA APCOMA  / %';' /
C
      CALL OTRO(TOKEN,BI,2)
      IF( TOKEN .EQ. ISIMB ) GO TO 110
  105 CONTINUE
      CALL ERROR(10)
      ULTIMO = LIMDR
      CALL LIMPIA
      CASO = 0
      RETURN
C
  110 CONTINUE
      IF( BUF(1) .NE. APARIZ ) GO TO 105
      CALL OTRO(TOKEN,BI,2)
      IF( TOKEN .EQ. IPCOMA ) GO TO 205
      CALL MUEVE(6)
C                     IMPRIME LA PARTE --INICIALIZA--.
  120 CONTINUE
      IF( TOKEN .NE. IFIN ) GO TO 125
      CALL COPIA(BI)
      GO TO 205
  125 CONTINUE
      CALL EXPR( IX1 )
      IF( IX1 .NE. 1 ) GO TO 105
      IF ( BUF(1) .NE. APCOMA ) CALL OTRO(TOKEN,BI,2)
      IF( BUF(1) .NE. APCOMA ) GO TO 105
C                  IMPRIME     "   -LBL1-   CONTINUE   " ...
  205 CONTINUE
      CALL SACONT(L1)
C          IMPRIME   "  IF(.NOT.(-CONDICION-))GOTO -LBL2-  " ...
      CALL SALAIF
C       EL 2000 SIRVE PARA EVITAR FALSO RETORNO DE LA RUTINA COND ...
      TEMP = 2000
```

```
      CALL COND(TEMP,CASO)
      IF( CASO .NE. 2 ) GO TO 220
      CALL LIMPIA
      GO TO 280
  220 CALL SACA( WPARDE )
      CALL SACA( WPARDE )
      CALL SAGOTO(L2)
  280 BAL = 1
      CALL OTRO(TOKEN,BI,2)
      IF( TOKEN .EQ. INLIN ) CALL OTRO(TOKEN,BI,2)
      NV1 = NV
C  PARTE DE LA -REINICIALIZACION-.
C  ES NECESARIO GUARDARLA EN UN 'STACK' PARA IMPRIMIRLA
C  EN LA SEGUNDA PARTE DEL PROCESO LUEGO.
  310 CONTINUE
      IF( TOKEN .EQ. INLIN ) GO TO 400
      IF( TOKEN .EQ. IFTN ) GO TO 320
      TEMP = BUF(1)
      IF( TEMP .EQ. WPARIZ ) BAL = BAL + 1
      IF( TEMP .EQ. WPARDE) BAL = BAL - 1
      IF( BAL .EQ. 0 ) GO TO 400
C           MANEJA NOTACION HOLLERIT Y METELA AL 'STACK'.
      IF( TEMP .NE. APOS ) GO TO 320
      CALL HOLLER(M)
      M = M + NV - 1
      KI = 0
      DO 315 I = NV,M
        KI = KI + 1
        REINIC(I) = BUF(KI)
  315 CONTINUE
      NV = M + 1
      CALL OTRO(TOKEN,BI,2)
      GO TO 310
C
  320 CONTINUE
C  METER EL CODIGO AL 'STACK'...
      BJ = BI + NV - 2
      IF( BJ .GT. 150 ) GO TO 370
      KI = 0
      DO 330 I = NV,BJ
        KI = KI + 1
        REINIC(I) = BUF(KI)
  330 CONTINUE
      NV = NV + BI - 1
      CALL OTRO(TOKEN,BI,2)
      GO TO 310
C
  370 CONTINUE
      CALL ERROR(11)
      GO TO 105
C
  400 CONTINUE
C  VERIFICAR SI HUBO O NO -REINICIALIZACION- EN EL FOR.
      IF( NV .NE. NV1 ) GO TO 410
      CASO = CASO + 1
      GO TO 450
  410 CONTINUE
```

```
            REINIC(NV) = NV1
            NV = NV + 1
        450 IF( TOKEN .EQ. INLIN ) GO TO 105
            RETURN
            END
```

PROGRAM UNIT CODFR1 COMPILED

```
C*************************************************************
      SUBROUTINE COUFR2
C  RUTINA QUE IMPRIME EL CODIGO  " REINICIALIZA "  DEL 'STACK'
C  PARA EL CASO  - FOR - ...
C  ESTA RUTINA LLAMA A  EXPR, MUEVE, SACA, SIGREN, TIPO
      INTEGER REINIC,T1,T2,T3,TEMP,TEMP1,TIPO,UNO,WBCO
      COMMON /FOR/ REINIC(150), NV
      DATA IGUAL,MAS,MENOS,UNO,WBCO /%'=', %'+', %'-', %'1', %' ' /
      DATA LETRA,NUMERO  /4120,4130/
C           PREVER EL CASO DE QUE EL 'STACK' ESTE VACIO...
      IF( NV .LE. 1 ) RETURN
      CALL LIMPIA
      CALL MUEVE(6)
      NV = NV - 1
      T1 = REINIC(NV)
      T2 = NV - 1
      TEMP = REINIC(T1)
      IF( TEMP .EQ. MAS ) GO TO 104
      IF( TEMP .EQ. MENOS ) GO TO 125
      T3 = TIPO(TEMP)
      IF( T3 .EQ. LETRA .OR. T3 .EQ. NUMERO ) GO TO 150
      GO TO 128
C             EXPANSION DE ++VAR  Y  --VAR .
  104 CONTINUE
      T1 = T1 + 1
      IF( REINIC(T1) .NE. MAS ) GO TO 128
      TEMP1 = MAS
  105 T1 = T1 + 1
      DO 110 I = T1,T2
        TEMP = REINIC(I)
        CALL SACA( TEMP )
  110 CONTINUE
      CALL SACA( WBCO)
      CALL SACA( IGUAL )
      CALL SACA( WBCO )
      DO 115 I = T1,T2
        TEMP = REINIC(I)
        CALL SACA( TEMP )
  115 CONTINUE
      CALL SACA( WBCO )
      CALL SACA( TEMP1 )
      CALL SACA( WBCO )
      CALL SACA( UNO )
      GO TO 160
C
  125 CONTINUE
      T1 = T1 + 1
      IF( REINIC(T1) .EQ. MENOS) GO TO 130
  128 CALL ERROR(6)
      GO TO 160
C
  130 CONTINUE
      TEMP1 = MENOS
      GO TO 105
C
  150 CONTINUE
      DO 155 I = T1,T2
```

```
            TEMP = REINIC(I)
            CALL SACA( TEMP )
  155 CONTINUE
C
  160 CONTINUE
      NV = T1
      CALL SIGREN
      RETURN
      END
```

PROGRAM UNIT CODFR2 COMPILED

```
      C*************************************************************
            SUBROUTINE CODDO(LBL)
      C  RUTINA QUE MANEJA EL 'DO' DE FOREST .
            INTEGER BI,ULTIMO,TEXTO,TOKEN,WBCO
            COMMON /LEX/ ULTIMO,TEXTO(95),IULT,LIMIZ,LIMDR
            DATA LD, LO ,WBCO / %'D', %'O' ,%' ' /
            DATA IFTN / 30 /
      C
            CALL MUEVE(6)
            CALL SACA( LD  )
            CALL SACA( LO  )
            CALL SACA( WBCO )
            CALL CARDEC( LBL )
            CALL SACA( WBCO )
      C  ESCRIBIR LA PARTE DE LOS LIMITES DEL DO  .
            CALL OTRO(TOKEN,BI,2)
            IF( TOKEN .EQ. IFTN ) GO TO 130
            CALL ERROR(18)
            ULTIMO = LIMDR
            CALL LIMPIA
            RETURN
      C
        130 CONTINUE
            CALL COPIA(BI)
            RETURN
            END


PROGRAM UNIT CODDO COMPILED
```

```
C*************************************************************
      SUBROUTINE LODCAS(NCASO,TOKEN,BI)
C  RUTINA QUE MANEJA EL CASO 'CASE'.
      INTEGER APOS,CASVAR,CASAP,CAS,CASTP,CASBG,DOSP
      INTEGER TOKEN,BI,BJ,BUF,TEMP,TEMP1,ETIQ,PILA
      INTEGER PETIQ,ETIQBK,BK,ETIQNX,NX,TEXTO,ULTIMO
      INTEGER LI,LF,WPARIZ,LE,WPTO,WPARDE
      COMMON /ITEXT/ BUF(80)
      COMMON /ETIQT/ PILA(50),ETIQ,PETIQ,ETIQBK(15),BK,
     $               ETIQNX(15),NX
      COMMON /CASS/ CASVAR(100),CASAP(15,2),CAS,CASTP,NOTHER
      COMMON /LEX/ ULTIMO,TEXTO(95),IULT,LIMIZ,LIMDR
      DATA IBEGIN,ICOM,IEND,IFTN,INLIN,IPCOMA,ISIMB,ITERM
     $   /   15,   23,    4,   30,   25,   29,   28,  21  /
      DATA APOS, DOSP  / %'"', %':' /
      DATA LE, LF, LI, LN,WPARIZ,WPARDE,WPTO
     $   /%'E',%'F',%'I',%'N',%'(', %')',  %'.'    /
C
      IF( NCASO .EQ.  2 ) GO TO 130
      CALL OTRO(TOKEN,BI,2)
      CAS = CAS + 1
      CALL GENET(1)
      CALL GENSAL(1)
      IF( CAS .GE. 14 ) GO TO 109
C GUARDA LA VARIABLE DE CONTROL Y SU ETIQUETA ASOCIADA EN UNA PILA.
      CASAP(CAS,1) = CASTP
      TEMP = PILA(ETIQ)
      CASAP(CAS,2) = TEMP
      IF( TOKEN .NE. IFTN ) GO TO 106
  100 CONTINUE
      BJ = BI + 2 + CASTP
      IF( BJ .GE. 100 ) GO TO 109
      J = 0
      DO 105 I = CASTP,BJ
         J = J + 1
         CASVAR(I) = BUF(J)
  105 CONTINUE
      CASTP = BJ + 1
      IF(TOKEN .EQ. IFTN .OR. TOKEN .EQ. ISIMB )GO TO 110
C        ERROR:  IGNORAR TODO EL 'CASE' MAS INTERNO.
  106 CONTINUE
      NOTHER = 0
      CASBG = 1
      CALL ERROR(3)
  108 CALL OTRO(TOKEN,BI,2)
      IF( TOKEN .EQ. IBEGIN .OR. TOKEN .EQ. ICOM )
     $                                        CASBG = CASBG + 1
      IF( TOKEN .EQ. IEND .OR. TOKEN .EQ. ITERM )
     $                                        CASBG = CASBG - 1
      IF( CASBG .NE. 0 ) GO TO 108
      CASTP = CASAP( CAS,1 )
      CAS = IABS(CAS)
      CALL LIMPIA
      TEMP = CASAP(CAS,2)
      CALL SACONT(TEMP)
      CALL QUITET(1,5)
      CAS = CAS - 1
```

```
C         AVISAR QUE HUBO ERROR
          CASTP = -CASTP
          RETURN
C         DEMASIADOS 'CASE' O DEMASIADAS VARIABLES DE CONTROL...
  109     CONTINUE
          CALL ERROR(21)
          GO TO 108
C
  110     CONTINUE
          CALL OTRO(TOKEN,BI,2)
          IF( TOKEN .EQ. INLIN ) GO TO 106
          IF( TOKEN .NE. IPCOMA ) GO TO 100
          CALL OTRO(TOKEN,BI,2)
          IF( TOKEN .EQ. INLIN .OR. TOKEN .EQ. IPCOMA )
         $                              CALL OTRO(TOKEN,BI,2)
C PROCESA LA "ETIQUETA" DEL CASE ...
  130     CONTINUE
          IT = BUF(1)
          IF( TOKEN .EQ. IFTN ) GO TO 135
C           MANEJA LA "ETIQUETA VACIA" DEL CASE, QUE
C           SIRVE COMO "SUMIDERO" (I.E., AHI VA A DAR
C           EL CONTROL SI NO SE CUMPLIO NINGUNA DE LAS
C           ANTERIORES "ETIQUETAS").
C           SOLO SE PERMITE UNA POR 'CASE'.
          IF( IT .NE. DOSP ) GO TO 133
          NOTHER = NOTHER + 1
          IF( NOTHER .EQ. 1 ) GO TO 157
          CALL ERROR(22)
          GO TO 106
  133     CONTINUE
          IF( IT .NE. APOS ) GO TO 106
  135     CONTINUE
          CALL MUEVE(6)
          CALL SACA( LI  )
          CALL SACA( LF  )
          CALL SACA( WPARIZ )
          TEMP = CASAP(CAS,1)
          TEMP1 = CASTP - 1
          DO 140 I = TEMP, TEMP1
             NNR = CASVAR(I)
             CALL SACA( NNR )
  140     CONTINUE
          CALL SACA( WPTO )
          CALL SACA( LN  )
          CALL SACA( LE  )
          CALL SACA( WPTO )
          IF( IT .EQ. APOS ) CALL HOLLER(1)
          IF( IT .EQ. APOS ) GO TO 155
  150     CONTINUE
          CALL SACTOK(BI)
  155     CONTINUE
          CALL OTRO(TOKEN,BI,2)
          IF( TOKEN .EQ. INLIN .OR. TOKEN .EQ. IPCOMA ) GO TO 106
          IF( BUF(1) .NE. DOSP )GO TO 150
          CALL SACA( WPARDE )
  157     CALL GENE1(1)
          TEMP = PILA(ETIQ)
```

```
      CALL PUSH(9)
C                 CUIDADO CON UN 'CASE' NULO...
      CALL OTRO(TOKEN,BI,1)
      IF( TOKEN .EQ. INLIN ) GO TO 106
      IF( IT .NE. DOSP ) CALL SAGOTO( TEMP )
      IF( TOKEN .EQ. IPCOMA ) GO TO 160
      RETURN
C
  160 CONTINUE
      CALL OTRO(TOKEN,BI,2)
      IF( TOKEN .NE. INLIN ) GO TO 106
      CAS = -CAS
      RETURN
      END
```

PROGRAM UNIT CODCAS COMPILED

```
C*************************************************************
      SUBROUTINE SACTOK(BI)
C RUTINA PARA IMPRIMIR UN 'TOKEN' COMPLETO .
      INTEGER BI, BJ,BUF
      COMMON /ITEXT/ BUF(80)
C
      BJ = BI - 1
      DO 110 I = 1,BJ
         NRR = BUF(I)
         CALL SACA( NRR  )
  110 CONTINUE
      RETURN
      END


PROGRAM UNIT SACTOK COMPILED
```

```
C*************************************************************
      SUBROUTINE COPIA(BI)
C RUTINA PARA COPIAR A LA SALIDA TODO EL RESTO DEL RENGLON,
C HASTA ENCONTRAR  ';'  O  'NWLINE' .
C CONVIERTE DE NOTACION CON APOSTROFOS A NOTACION HOLLERIT.
      INTEGER APOS,BI,BUF,REL(3),SESP1,SESP2,TEMP
      INTEGER TOKEN,WBCO,WCOMA,WPCOMA
      COMMON /ITEXT/ BUF(80)
      DATA APOS  / %"'" /
      DATA INLIN,IPCOMA,ISIMB  / 25,29,28 /
      DATA WBCO,WCOMA,WPCOMA  / %' ' ,  %',' ,  %';'  /
      DATA SESP1  / %'$' /
C     DATA SESP2  / %'/' /
C ---EQUIVALEN A    .OR.  .AND.  .NOT.
      DATA REL  / %'?', %'&',  %'^' /
C                 SACA EL PRIMER 'TOKEN' YA LEIDO...
C                 AYUDA EN EL CASO  ++VAR  O  --VAR...
      IT = 0
      IF( BI .GT. 0 ) GO TO 90
      IT = 1
      GO TO 100
C
   90 CONTINUE
      CALL SACTOK(BI)
C        CUIDADO CON LOS SIMBOLOS ESPECIALES DEL COMPILADOR.
      IF( BUF(1) .EQ. SESP1 ) GO TO 100
C     IF( BUF(2) .EQ. SESP2 ) GO TO 100
      CALL SACA( WBCO )
C
  100 CONTINUE
      CALL OTRO(TOKEN,BI,2)
      IF( TOKEN .EQ. ISIMB ) GO TO 200
      IF( TOKEN .EQ. INLIN .OR. TOKEN .EQ. IPCOMA ) GO TO 300
      TEMP = WBCO
  115 CONTINUE
      CALL SACTOK(BI)
      CALL SACA(WBCO)
      GO TO 100
C
  200 CONTINUE
      TEMP = BUF(1)
      IF( TEMP .EQ. APOS ) GO TO 270
      DO 220 I = 1,3
         IF( BUF(1) .NE. REL(I) ) GO TO 220
         J = I + 1
         CALL SACREL( J )
         GO TO 100
  220 CONTINUE
      GO TO 115
C
  270 CONTINUE
      CALL HOLLER(I)
      GO TO 100
C
  300 CONTINUE
      IF( TEMP .EQ. WCOMA ) BI = -BI
      IF( TOKEN .EQ. IPCOMA ) BUF(1) = WPCOMA
```

```
      IF( IT .EQ. 1 ) RETURN
      CALL SIGREN
      RETURN
      END
```

PROGRAM UNIT COPIA COMPILED

```
C*************************************************************
      SUBROUTINE EOFPRE
C RUTINA QUE SE ENCARGA DE LA TERMINACION ANORMAL
C POR ENCONTRARSE UN FIN DE ARCHIVO INESPERADO...
      CALL ERROR(8)
      CALL FIN
C     CALL EXIT
      STOP
      END
```

PROGRAM UNIT EOFPRE COMPILED

```
C*************************************************************
      SUBROUTINE SACA(CAR)
C RUTINA QUE CONTROLA LA SALIDA DEL PROGRAMA "OBJETO" YA PROCESADO
C POR FOREST . CUIDA TAMBIEN LAS LINEAS DE CONTINUACION...
      INTEGER CAR,TEXOBJ,UL,JR,WPESOS
      COMMON /SALE/ TEXOBJ(80),JR,UL,IMP,NGOTO,NFIN
      DATA WPESOS / %'$' /
C
      JR = JR + 1
      IF( JR .LE. 71 ) GO TO 130
C TARJETAS DE CONTINUACION ...
      CALL SIGREN
      TEXOBJ(6) = WPESOS
      JR = 7
  130 CONTINUE
      TEXOBJ( JR ) = CAR
      RETURN
      END


PROGRAM UNIT SACA COMPILED
```

```
C*************************************************************
      SUBROUTINE EXPR( TEMP )
C RUTINA PARA CONVERTIR DE LA NOTACION  ++VAR  A  VAR = VAR + 1
C Y DE  --VAR  A  VAR = VAR - 1 .
      INTEGER BI,BUF,TEMP,TEMP1,TEXOBJ,TOKEN,UL,UNO,WBCO
      COMMON /ITEXT/ BUF(80)
      COMMON /SALE/ TEXOBJ(80),JR,UL,IMP,NGOTO,NFTN
      DATA IGUAL,MAS,MENOS,UNO,WBCO  / %'=', %'+', %'-', %'1', %' ' /
C
      TEMP = 0
      IF( BUF(1) .NE. MAS ) GO TO 120
      CALL OTRO(TOKEN,BI,1)
      IF( BUF(1) .NE. MAS ) RETURN
C                        CASO  ++VAR .
      TEMP1 = MAS
  105 BI = -BI
      CALL MUEVE(6)
      CALL COPIA(BI)
      CALL SACA( IGUAL )
      CALL SACA( WBCO )
  110 CONTINUE
      IJR = JR - 2
      DO 115 I = 7,IJR
         IT = TEXOBJ(I)
         CALL SACA( IT )
  115 CONTINUE
      CALL SACA( TEMP1 )
      CALL SACA( WBCO )
      CALL SACA( UNO )
      CALL SIGREN
      TEMP = 1
      RETURN
C
  120 CONTINUE
      IF( BUF(1) .NE. MENOS ) RETURN
      CALL OTRO(TOKEN,BI,1)
      IF(BUF(1) .NE. MENOS ) RETURN
      TEMP1 = MENOS
      GO TO 105
      END
```

PROGRAM UNIT EXPR COMPILED

```
C*******************************************************
      SUBROUTINE HOLLER(I)
C CONVIERTE LA NOTACION CON APOSTROFOS A NOTACION HOLLERIT
C SE SIGUE LA CONVENCION DE REPRESENTAR UN APOSTROFO "REAL"
C POR DOS JUNTOS.
      INTEGER  APOS,ARR(5),BUF,CAR,HACHE,TEXTO
      INTEGER TEMP,TENCAR,ULTIMO,WBCO,WNLIN,WPCOMA
      COMMON /ITEXT/ BUF(80)
      COMMON /LEX/ ULTIMO,TEXTO(95),IULT,LIMIZ,LIMDR
      DATA APOS, HACHE / %"'",%'H' /
      DATA WBCO,WNLIN,WPCOMA /  %' ', %'\', %';' /
C
C     EVITAMOS QUE SE LEA UN NUEVO RENGLON EN CASO DE ERROR.
      IULT = 1
      I = 3
  100 CONTINUE
      CAR = TENCAR(TEMP)
      I = I + 1
      IF( I .GT. 80 ) GO TO 105
      IF( CAR .NE. WNLIN .OR. CAR .NE. WPCOMA ) GO TO 110
  105 CONTINUE
      CALL ERROR(9)
      CALL LIMPIA
      RETURN
  110 CONTINUE
      BUF(I) = CAR
      IF( CAR .NE. APOS ) GO TO 100
C                     ES UN APOSTROFO; VER SI HAY OTRO PEGADO..
      CAR = TENCAR(TEMP)
C                     DOS APOSTROFOS PEGADOS = UNO SENCILLO ...
      IF( CAR .EQ. APOS ) GO TO 100
C                     SI ERA UNO SOLO ; FIN .
      CALL REGRES
      J = I - 4
      K = J
      CALL CONV( J,ARR )
C                GENERAR LOS NUMEROS DEL HOLLERIT.
      IF( K .GE. 10 ) GO TO 190
      BUF(1) = WBCO
      BUF(2) = ARR(1)
      GO TO 195
  190 BUF(1) = ARR(1)
      BUF(2) = ARR(2)
  195 BUF(3) = HACHE
      CALL SACTOK(I)
      I = I - 1
      RETURN
      END


PROGRAM UNIT HOLLER COMPILED
```

```
C*****************************************************
      INTEGER FUNCTION COMP(BI)
C SEPARA EL TEXTO DE ENTRADA EN SUS COMPONENTES SINTACTICOS SIMPLES.
C REGRESA EN BUF(I) UN 'TOKEN' POR CADA LLAMADA.  BI  APUNTA AL FINAL.
C HAY COMPONENTES ALFANUMERICOS Y CARACTERES SENCILLOS ESPECIALES,
C SEGUIDOS TODOS DE UN TERMINADOR.          IGNORA BLANCOS Y COMENTARIOS.
C SI SE TRABAJA POR PANTALLA ENTONCES ES NECESARIO ELIMINAR
C LOS 'CAR RETURN', ESO SE HACE POR MEDIO DEL =DATA= WCR .
      INTEGER NUMERO,LETRA,WEOW,WCOMEN,NWLINE,TEMP,CAR,TEXTO
      INTEGER BUF,TENCAR,TIPO,EOF,ULTIMO,BI,WEOF,PCOMA,WPCOMA
      INTEGER NADA,WBCO,ALFAN,SI,SIMB,WNLIN,WCR
      COMMON /LEX/ ULTIMO,TEXTO(95),IULT,LIMIZ,LIMDR
      COMMON /ITEXT/ BUF(80)
      DATA WCOMEN,WBCO,WEOW,WNLIN,WEOF,WPCOMA
     $   /  %'!', %' ',%'?',%'\', %'@', %';' /
      DATA ALFAN,EOF,LETRA,NUMERO,NWLINE,PCOMA,SI, SIMB
     $   / 4100, 4110,4120, 4130, 4135,  4140,4150, 4155  /
C  -------- DEPENDIENTE DE LA MAQUINA EN CUESTION --------
C  -------- ESTO FUNCIONA PARA LA Z2D CON PANTALLA "INTEL" ---
C     DATA WCR / Z'020OD' /
C  .......Y ESTO PARA HEWLETT-PACKARD 3000.........
      DATA  WCR  / %15C /
C
   50 CONTINUE
      I = 0
C         METE A BUF(I) EL COMPONENTE
  100 I = I + 1
      CAR = TENCAR(TEMP)
      IF( TEMP .EQ. WCR .AND. I .GT. 1 ) GO TO 105
      BUF(I) = CAR
      CAR = TIPO(TEMP)
      IF( CAR .EQ. NUMERO .OR. CAR .EQ. LETRA)GO TO 100
      IF( I .LE. 1)GO TO 110
C        CASO ALFANUMERICO.      INSERTA TERMINADOR.
      CALL REGRES
  105 BUF(I) = WEOW
      BI = I
      COMP = ALFAN
      RETURN
C        CASO DE UN SIMBOLO SUELTO (NO ALFANUMERICO).
  110 CONTINUE
      IF( CAR .EQ. WCR ) GO TO 125
      IF( CAR .NE. WCOMEN ) GO TO 130
C        ELIMINA COMENTARIOS, HASTA PUNTO Y COMA, O FIN DE RENGLON.
  120 CONTINUE
      NADA = TENCAR(TEMP)
      IF( NADA .EQ. WNLIN ) GO TO 127
      IF( NADA .NE. WPCOMA ) GO TO 120
  122 NADA = TENCAR(TEMP)
      IF( NADA .EQ. WBCO ) GO TO 122
      IF( NADA .EQ. WNLIN ) GO TO 127
      CALL REGRES
      COMP = SI
      RETURN
  125 ULTIMO = LIMDR
      NADA = TENCAR(TEMP)
  127 COMP = NWLINE
```

```
            BI = 1
            RETURN
      C        ELIMINA BLANCOS INTERMEDIOS
        130 IF( CAR .NE. WBCO)GO TO 150
        140 CONTINUE
            IF( TENCAR(TEMP) .EQ. WBCO ) GO TO 140
      C
        145 CONTINUE
            COMP = NWLINE
            BI = 2
            IF( TEMP .EQ. WNLIN) RETURN
      C        FORZAR A LEER EL SIGUIENTE 'TOKEN'.
            CALL REGRES
            GO TO 50
      C        INSERTA TERMINADOR EN SIMBOLO SUELTO.
        150 CONTINUE
            BUF(I+1)= WEOW
            BI=I+1
            COMP=SIMB
            IF( CAR .EQ. WEOF) COMP = EOF
            IF( CAR .EQ. WNLIN) COMP = NWLINE
            IF( CAR .EQ. WPCOMA ) COMP = PCOMA
            RETURN
            END


PROGRAM UNIT COMP COMPILED
```

```
C*******************************************************
      SUBROUTINE COND(BAL,CASO)
C  RUTINA QUE AISLA E IMPRIME EL TEXTO DE CONDICION DE UN  IF  O DE
C  UN WHILE, FOR, UNTIL.
C  CONVERTE LOS SIMBOLOS "RELACIONALES" ( >,<,=,<=, ETC.) A SUS
C  EQUIVALENTES EN FORTRAN ( .GT., .LT., .EQ., .LE., ETC)
C  LA VARIABLE CASO AVISA EL RESULTADO ,
C  CASO = 2 : NO HUBO CONDICION.   CASO = 4 : COND. NORMAL.
      INTEGER BAL,TOKEN,TEMP,BUF,WPARDE,WPARIZ,REL(6)
      INTEGER APOS,BI,BJ,CNUM,CASO,COMA,IGUAL,MAYOR
      COMMON /ITEXT/ BUF(80)
      DATA WPARDE,WPARIZ / %')', %'(' /
C ---EQUIVALEN A .EQ. .OR. .AND. .NOT. .LT. .GT.
      DATA REL /%'=',%'?',%'&', %'^', %'<',%'>' /
      DATA INLIN,ISIMB,IPCOMA,IFTN  / 25,28,29,30 /
      DATA APOS,COMA,IGUAL,MAYOR / %"'", %',' , %'=', %'>' /
      N1 = -1
      CASO = 2
C                DIFERENTES CASOS POSIBLES EN LA CONDICION.
  100 CONTINUE
      CNUM = 0
  101 CONTINUE
      CALL OTRO(TOKEN,BI,2)
      N1 = N1 + 1
  102 CONTINUE
      IF( TOKEN .EQ. IPCOMA .AND. N1 .EQ. 0 ) RETURN
      CASO = 4
      IF( TOKEN .EQ. IPCOMA ) RETURN
      IF( TOKEN .EQ. INLIN ) GO TO 101
      TEMP = BUF(1)
      IF( TOKEN .NE. ISIMB ) GO TO 250
      IF( TEMP .EQ. WPARIZ ) GO TO 200
      IF( TEMP .EQ. WPARDE ) GO TO 200
C              DETECTA SI SE TRATA DE UN SIMBOLO RELACIONAL.
      DO 105 I = 1,3
        IF( TEMP .NE. REL(I) ) GO TO 105
        CNUM = CNUM + 1
        IF( CNUM .GT. 1 ) GO TO 270
        CALL SACREL(I)
        GO TO 101
  105 CONTINUE
      DO 110 I = 4,6
        IF( TEMP .NE. REL(I) ) GO TO 110
        TEMP = I
        GO TO 115
  110 CONTINUE
C          ES UN SIMBOLO NO-RELACIONAL, PUEDE SER UN APOSTROFO.
C        ...CONVERTIRLO A NOTACION HOLLERIT.
      IF( TEMP .NE. APOS ) CALL SACA(TEMP)
      IF( TEMP .EQ. APOS ) CALL HOLLER(1)
      GO TO 100
  115 CONTINUE
      CALL OTRO(TOKEN,BI,2)
      IF( TOKEN .EQ. ISIMB ) GO TO 130
C              CASO DE SIMBOLO  , <, O > SOLOS.
      CALL SACREL(TEMP)
      IF( TOKEN .NE. INLIN ) GO TO 102
```

```
      TEMP = REL(TEMP)
      GO TO 251
  130 CONTINUE
C        EL PRIMER SIMBOLO FUE ,<,> Y HAY OTRO SIMBOLO PEGADO...
      IF( BUF(1) .NE. APOS ) GO TO 132
      CALL SACREL(TEMP)
      CALL HOLLER(1)
      GO TO 100
  132 CONTINUE
      IF( BUF(1) .NE. IGUAL ) GO TO 140
      DO 135 I = 4,6
         IF( TEMP .NE. I ) GO TO 135
         CALL SACREL( I+3 )
         GO TO 100
  135 CONTINUE
  140 CONTINUE
C                              CASO <> ... (.NE.)
      IF( BUF(1) .NE. MAYOR ) GO TO 150
      IF( TEMP .NE. 5 ) GO TO 150
      CALL SACREL(7)
      GO TO 100
  150 CONTINUE
C   HUBO UN ?,<,> PEGADO A OTRO SIMBOLO RARO, SOLO SE TOMA EL 1ERO.
      CALL ERROR(12)
      CALL SACREL(TEMP)
      GO TO 100
C                 YA NO ESTAMOS EN EL CASO RELACIONAL...
  200 CONTINUE
C                 CASO ) O ( ...
      IF( TEMP .EQ. RPARDE ) BAL = BAL-1
      IF( TEMP .EQ. RPARIZ ) BAL = BAL + 1
      CALL SACA(TEMP)
      IF( BAL .EQ. 0 ) RETURN
      GO TO 100
C    COPIAR LO QUE HAYA Y VERIFICAR CONDICION -ANORMAL- DE FIN...
C    MANEJA TAMBIEN RENGLONES MULTIPLES...
  250 CONTINUE
      IF( TOKEN .NE. INLIN ) GO TO 260
  251 IF ( TEMP .EQ. RPARIZ ) GO TO 100
      IF( TEMP .EQ. RPARDE ) GO TO 100
      DO 253 I = 1,6
         IF( TEMP .EQ. REL(I) ) GO TO 101
  253 CONTINUE
      IF( TEMP .NE. COMA ) GO TO 270
  260 CONTINUE
C                 COPIA LA CONDICION...
      IF( TOKEN .EQ. IFIN ) GO TO 300
  270 CONTINUE
      CALL ERROR(13)
      CASO = 2
      RETURN
C
  300 CONTINUE
      CALL SACTOK(BI)
      GO TO 100
      END
```

PROGRAM UNIT COND COMPILED

```
C*************************************************************
      SUBROUTINE REGRES
C  "REGRESA" UN CARACTER A LA CADENA DE ENTRADA.
C  DE ESTA MANERA PERMITE VOLVER A LEER UN CARACTER.
C   LO HACE DE MANERA SIMBOLICA, MOVIENDO EL APUNTADOR
C   AL 'BUFFER' DE ENTRADA.
      INTEGER ULTIMO,TEXTO
      COMMON /LEX/ ULTIMO,TEXTO(95),IULT,LIMIZ,LIMDR
C
      ULTIMO = ULTIMO - 1
      RETURN
      END


PROGRAM UNIT REGRES COMPILED
```

```
C*************************************************
      INTEGER FUNCTION TIPO(C)
C  AVERIGUA SI UN CARACTER ES ALFANUMERICO O NO .
      INTEGER C,CAR,LETRA,NUMERO,LET(26),NUMRS
      COMMON /NUMR/ NUMRS(10)
      DATA LET /%'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' /
      DATA NUMERO,LETRA /4130,4120/
C
C
  100 CONTINUE
      DO 110 I=1,26
         IF( C .EQ. LET(I) ) GO TO 150
  110 CONTINUE
      DO 120 I=1,10
         IF( C .EQ. NUMRS(I) ) GO TO 170
  120 CONTINUE
C  TIPO = SIMBOLO O BLANCO.
      CAR = C
      TIPO = CAR
      RETURN
C
  150 CAR = LETRA
      TIPO = CAR
      RETURN
C
  170 CAR = NUMERO
      TIPO = CAR
      RETURN
      END
```

PROGRAM UNIT TIPO COMPILED

```
C*************************************************
      INTEGER FUNCTION TENCAR(TEMP)
C  OBTIENE EL SIGUIENTE CARACTER DEL RENGLON.
C  LEE UNO NUEVO SI ES NECESARIO.
C  AVISA FIN DE RENGLON Y EOF .
      INTEGER ARR(5),ADV,DOBLE(80),ERRORS
      INTEGER TEMP,TEXTO,TEXOBJ,ULTIMO,UL
      INTEGER WBCO,WEOF,WNLIN
      COMMON /LEX/ ULTIMO,TEXTO(95),IULT,LIMIZ,LIMDR
      COMMON /CONT/NLIN,ERRORS,ADV
      COMMON /DISCO/IDISC
      COMMON /NUMN/NUMRS(10)
      COMMON /SALE/TEXOBJ(80),JR,UL,IMP,NGOTO,NFTN
C        -WNLIN- Y -WEOF- SERAN SIMBOLOS NO UTILIZADOS REGULARMENTE.
      DATA WNLIN,WEOF,WBCO/%'\',%'@',%' ' /
C
      ULTIMO = ULTIMO + 1
      IF( ULTIMO .LE. LIMDR ) GO TO 120
C
C                    SI  IULT = 1  NO SE LEE UN NUEVO RENGLON.
C                    AYUDA SI HAY QUE DEVOLVERSE EN EL CASO
C                    LIMITE CUANDO  ULTIMO = LIMDR .
      IF( IULT .EQ. 0 ) GO TO 100
      TEMP = WNLIN
      TENCAR = TEMP
      IULT = 0
      ULTIMO = LIMDR + 1
      RETURN
C
  100 CONTINUE
      IF( ULTIMO .GE. 1000 ) GO TO 220
C             -------- TIPO "BATCH" O TIPO INTERACTIVO --------
      IF( IDISC .EQ. 5 ) WRITE(6,555)
  555 FORMAT('  ENTRA RENGLON',/)
      READ(IDISC,500,END=200) (TEXTO(I),I=LIMIZ,LIMDR)
C    ***NOTA: EL COMPILADOR HEWLETT-PACKARD 3000, COMO
C    TRABAJA CON 16 BITS, TRANSFIERE UN CARACTER ASCII
C    A LA PARTE DERECHA DE LA PALABRA CON EL FORMATO
C    "R", Y A LA PARTE IZQUIERDA CON EL FORMATO "A".
C    ESTO VARIA DE MAQUINA A MAQUINA***
C    EN OTRO CASO, USAR EL FORMATO "A" E INCLUIR LA
C    VARIABLE "FCOR" DE LA RUTINA 'CLAVE'.
  500 FORMAT(72R1)
      IULT = 0
C
C    GENERA LOS NUMEROS SECUENCIALES DE RENGLON DE SALIDA ...
      NLIN = NLIN + 1
      NL = NLIN
      CALL CONV(NL,ARR)
      DO 102 I=2,6
         TEXTO(I) = ARR(I-1)
  102 CONTINUE
C          ESCRIBIR LOS NUMEROS DE REFERENCIA CRUZADA AL
C          "CODIGO" FORTRAN GENERADO .
      NF = NFTN
      CALL CONV( NF,ARR )
      DO 103 I=13,17
```

```
      TENCAR

            TEXTO(I) = ARR( I-12 )
  103 CONTINUE
      WRITE(IMP,558) (TEXTO(I),I=1,95)
  558 FORMAT(1X,95R1)
C
C  ENCUENTRA EL PRIMER CARACTER NO BLANCO DEL NUEVO RENGLON.
      DO 105 I=LIMIZ,LIMDR
         IU = I
         IF( TEXTO(I) .NE. WBCO ) GO TO 110
  105 CONTINUE
C  EL RENGLON ERA BLANCO
      GO TO 100
C
  110 CONTINUE
      ULTIMO = IU - 1
      TEMP = WNLIN
      TENCAR = TEMP
      RETURN
C
  120 CONTINUE
      TEMP = TEXTO(ULTIMO)
      TENCAR = TEMP
      RETURN
C
  200 CONTINUE
      ULTIMO = 1000
      TEMP = WNLIN
      TENCAR = TEMP
      RETURN
C
  220 CONTINUE
      TEMP = WEOF
      TENCAR = TEMP
      NLIN = NLIN + 1
      RETURN
      END


PROGRAM UNIT TENCAR COMPILED
```

```
C*************************************************
      INTEGER FUNCTION NUMER(BI)
C  AVERIGUA SI BUF(I) CONTIENE SOLO CARACTERES NUMERICOS. (SI/NO).
C  AVISA SI LA ETIQUETA ES > 5 CARACTERES.
C  AVISA SI LA ETIQUETA LEIDA ENTRA EN CONFLICTO CON LAS GENERADAS
C   POR EL SISTEMA (777XX).
      INTEGER BI,BJ,BUF,SI,SIETE,TIPO
      COMMON/ITEXT/ BUF(80)
      DATA NO,NUMERO,SI,SIETE /4127,4130,4150,%'7'/
C
      NUMER = NO
      IF( BI .EQ. 1 ) RETURN
      BJ = BI - 1
      DO 150 I=1,BJ
         IF( TIPO(BUF(I)) .NE. NUMERO) RETURN
  150 CONTINUE
C                          TENER CUIDADO CON EL CASO 'CASE'...
      CALL POP(NR)
      CALL PUSH(NR)
      IF(NR .EQ. 9) RETURN
C                          CUIDADO CON LAS ETIQUETAS DEMAS DE 5 NUMS...
      IF( BI .LE. 6 ) GO TO 170
      CALL ERROR(16)
      RETURN
C
  170 CONTINUE
      NUMER = SI
      IF(BI .LE. 5) RETURN
C        POSIBLE CONFLICTO CON ETIQUETAS GENERADAS INTERNAMENTE...
      IF( BUF(1) .NE. SIETE .OR. BUF(2) .NE. SIETE ) RETURN
      CALL ERROR(17)
      RETURN
      END


PROGRAM UNIT NUMER COMPILED
```

```
C***********************************************
      INTEGER FUNCTION CLAVE(BI,KWORD)
C  RUTINA PARA SABER SI EL CONTENIDO DE BUF(I) ES UNA PALABRA
C  CLAVE PREDEFINIDA DE FORTEST.
C  BUSCA EN EL DICCIONARIO DIC , APUNTADO POR APDIC .
C  SE HACE UNA BUSQUEDA BINARIA, GUIADA SOBRE EL ARREGLO CLDIC .
C    VER KNUTH, VOL, 3, P.407 .
      INTEGER BI,BJ,BUF,KWORD,DIC,APDIC
      INTEGER T1,NO,SI,U,FCOR,VAL,CLDIC
      COMMON /DIC1/ DIC(113),CLDIC(24),APDIC(24)
      COMMON /ITEXT/ BUF(80)
      DATA NO,SI / 4127,4150 /
C  FCOR ES UNA CTTE. QUE EL Z2D SUMA A LOS CARACTERES ASCII .
C      DATA FCOR / 8192 /
C    EL COMPILADOR HEWLETT-PACKARD 3000, POR MEDIO DEL
C    FORMATO "R" EVITA LA NECESIDAD DEL VALOR "FCOR".
C  K ES EL VALOR NUMERICO DE LA CADENA QUE SE BUSCA .
C  EL VALOR NUMERICO DE LA CADENA ASCII SE OBTIENE DE RESTAR A
C  CADA CARACTER LA CONSTANTE FCOR (CASO PARTICULAR DEL COMPILADOR
C  "Z2D") Y LUEGO MULTIPLICARLA POR 1,3,5,7,9,..., PARA
C    ASI OBTENER UN NUMERO -NO NECESARIAMENTE UNICO- QUE GUIARA
C    LA BUSQUEDA BINARIA.
      BJ = BI - 1
      K = 0
      VAL = 1
      DO 110 J=1,BJ
C    ESTA SERIA LA EXPRESION PARA EL COMPILADOR "Z2D".
C       K = K + (( BUF(J) - FCOR ) * VAL )
C    PERO ESTA ES LA EXPRESION SIN EL VALOR "FCOR"...
        K = K + ( BUF(J)  * VAL )
        VAL = VAL + 2
  110 CONTINUE
C  INICIALIZACIONES...
      L = 1
      U = 24
  120 CONTINUE
C  LLEGAR AL PUNTO MEDIO DE LA TABLA ...
      IF(U .GE. L) GO TO 125
  122 CONTINUE
      CLAVE = NO
      RETURN
  125 CONTINUE
C  HACER LAS COMPARACIONES ...
      I = ( L+U ) / 2
      IF( K .LT. CLDIC(I) ) GO TO 140
      IF( K .GT. CLDIC(I) ) GO TO 150
C  SE ENCONTRO UNA CADENA CON EL MISMO VALOR NUMERICO.
C  HAY QUE COMPARAR SUS CARACTERES.
      T1 = APDIC(I)
      DO 127 J=1,BJ
        IF( BUF(J) .NE. DIC(T1) ) GO TO 122
        T1 = T1 + 1
  127 CONTINUE
      CLAVE = SI
      KWORD = I
      RETURN
  140 CONTINUE
```

```
            U = I - 1
            GO TO 120
        150 CONTINUE
            L = I + 1
            GO TO 120
            END
```

PROGRAM UNIT CLAVE COMPILED

```
C*******************************************************
      SUBROUTINE CONV(NUM,ARR)
C  RUTINA QUE CONVIERTE UN NUMERO POSITIVO
C  SUS CARACTERES NUMERICOS.
      INTEGER ARR(5),TEMP,BLCO
      COMMON/NUMR/NUMRS(10)
      DATA BLCO / %" "  /
C
      DO 110 I=1,5
         ARR(I)=BLCO
  110 CONTINUE
      IF( NUM .LE. 0 ) RETURN
      J = 0
      I = 0
  120 IF( NUM .LE. 0 ) GO TO 130
      TEMP = MOD(NUM,10) + 1
      I = I + 1
      ARR(I) = NUMRS(TEMP)
      NUM = NUM / 10
      GO TO 120
  130 CONTINUE
C              INVERTIR EL ORDEN DEL ARREGLO DE CARACTERES..
      J = J + 1
      TEMP = ARR(I)
      ARR(I) = ARR(J)
      ARR(J) = TEMP
      I = I - 1
      IF( J .LT. I ) GO TO 130
      RETURN
      END


PROGRAM UNIT CONV COMPILED
```

```
C*******************************************************
      SUBROUTINE LIMPIA
C  RUTINA PARA PONER EN BLANCO EL RENGLON DE SALIDA.
      INTEGER TEXOBJ,JR,UL,WBCO
      COMMON/SALE/ TEXOBJ(80),JR,UL,IMP,NGUTO,NFTN
      DATA WBCO / %' ' /
C
      DO 120 I=1,80
        TEXOBJ(I) = WBCO
  120 CONTINUE
      JR = 0
      RETURN
      END
```

PROGRAM UNIT LIMPIA COMPILED

```
C*********************************************************
      SUBROUTINE CARDEC(LBL)
C  CONVIERTE UN NUMERO A SUS CARACTERES NUMERICOS Y "SACALO",
C  COMO EL Z2D NO ACEPTA NUMEROS DE 5 CIFRAS HAY QUE
C  INSERTAR UN '7' EN LA PRIMERA POSICION DE LAS
C  ETIQUETAS GENERADAS POR GENET  .
      INTEGER ARR(5),SIETE,TEMP
      DATA SIETE  / %'7' /
C
      CALL CONV(LBL,ARR)
      CALL SACA( SIETE )
      DO 100 I=1,4
        TEMP = ARR(I)
        CALL SACA(TEMP)
  100 CONTINUE
      RETURN
      END
```

PROGRAM UNIT CARDEC COMPILED

```
C*************************************************************
      SUBROUTINE SACAIF
C  RUTINA QUE IMPRIME "  IF(.NOT.(  " .
      INTEGER IFCOD(9)
      DATA IFCOD /%'I',%'F',%'(',%'.',%'N',%'O',%'T',%'.',%'(' /
C
      CALL MUEVE(6)
      DO 110 I = 1,9
         IFCO = IFCOD(I)
         CALL SALA( IFCO )
  110 CONTINUE
      RETURN
      END
```

PROGRAM UNIT SACAIF COMPILED

```
C*********************************************************
      SUBROUTINE SIGREN
C  RUTINA PARA OBLIGAR LA IMPRESION DEL RENGLON CORRIENTE A LA
C  SALIDA Y LA GENERACION DE UNO NUEVO (LIMPIO).
C  LA VARIABLE UL CONTROLA HACIA DONDE SE HARA LA IMPRESION.
      INTEGER JR,TEXOBJ,UL
      COMMON/SALE/TEXOBJ(80),JR,UL,IMP,NGOTO,NFTN
C
      WRITE(UL,500)(TEXOBJ(I),I=1,72)
C   HEWLETT-PACKARD ACEPTA EL FORMATO "R".  SI NO LO HUBIERA,
C   HAY QUE CAMBIAR AL FORMATO "A" Y MODIFICAR LA RUTINA "CLAVE".
  500 FORMAT(72R1)
C  "LIMPIA" EL RENGLON ...
      CALL LIMPIA
C             NGOTO  EVITA QUE SE GENEREN DOS O MAS GOTO'S
C             SEGUIDOS, LO QUE PODRIA DAR LUGAR A UNA
C             ADVERTENCIA DEL COMPILADOR FORTRAN .
      NGOTO = 0
C             NFTN  CUENTA EL NUM. DE RENGLONES DE
C             "CODIGO" FORTRAN YA GENERADOS.
      NFTN = NFTN + 1
      RETURN
      END


PROGRAM UNIT SIGREN COMPILED
```

```
C*********************************************************
      SUBROUTINE SACREL(NUM)
C  RUTINA QUE IMPRIME LOS CONDICIONALES DE FORTRAN
C  EN LUGAR DE LOS DE FOREST.
      INTEGER WBCO,WPTO
      DATA WBCO,WPTO, LE, LG, LO,   LR, LA, LN, LD, LT, LL, LG
     $  /'(' ',','.','E','G','D','R','A','N','D','T','L','G'/
C
      CALL SACA( WBCO )
      CALL SACA( WPTO )
      GO TO ( 100,200,300,400,500,600,700,800,900 ) ,NUM
C
  100 CALL SACA( LE )
      CALL SACA( LG  )
      GO TO 1000
  200 CALL SACA( LO  )
      CALL SACA( LR  )
      GO TO 1000
  300 CALL SACA( LA  )
      CALL SACA( LN  )
      CALL SACA( LD  )
      GO TO 1000
  400 CALL SACA( LN  )
      CALL SACA( LO  )
      CALL SACA( LT  )
      GO TO 1000
  500 CALL SACA( LL  )
      CALL SACA( LT  )
      GO TO 1000
  600 CALL SACA( LG  )
      CALL SACA( LT  )
      GO TO 1000
  700 CALL SACA( LN  )
      CALL SACA( LE  )
      GO TO 1000
  800 CALL SACA( LL  )
      CALL SACA( LE  )
      GO TO 1000
  900 CALL SACA( LG  )
      CALL SACA( LE  )
C
 1000 CALL SACA( WPTO )
      CALL SACA( WBCO )
      RETURN
      END


PROGRAM UNIT SACREL COMPILED
```

```
C*******************************************************
      SUBROUTINE FIN
      INTEGER ERRORS,ADV,TEXOBJ,UL
C
      COMMON /CONT/ NLIN,ERRORS,ADV
      COMMON /SALE/ TEXOBJ(80),JR,UL,IMP,NGOTO,NFTN
C
      IT = 6
  100 WRITE(IMP,500) ADV
  500 FORMAT(29H  ##NUMERO DE ADVERTENCIAS = ,I6)
      WRITE(IMP,510) ERRORS
  510 FORMAT(24H  ##NUMERO DE ERRORES = ,I6)
      IF( ERRORS .LE. 0 ) GO TO 1200
      WRITE(IMP,520)
  520 FORMAT(27H  ***EL CODIGO NO SIRVE*** )
C
 1200 CONTINUE
C     IF( NLIN .LE. 0 ) NLIN = 1
      NLIN = NLIN - 1
      WRITE(IMP,550) NLIN
  550 FORMAT(41H  ##NUMERO DE LINEAS (TARJETAS) LEIDAS = ,I6)
      WRITE(IMP,570)
  570 FORMAT(10H  ##FIN.##)
C      DAR LOS DIAGNOSTICOS POR PANTALLA SI NO HAN SIDO DADOS.
      IF( IMP .EQ. 6 ) GO TO 600
      IT = IMP
      NLIN = NLIN + 1
      IMP = 6
      GO TO 100
  600 CONTINUE
      IF( IT .EQ. 6 ) RETURN
      IMP = IT
      RETURN
      END


PROGRAM UNIT FIN COMPILED
```

```
****        GLOBAL STATISTICS        ****
****    NO ERRORS,    NO WARNINGS    ****
TOTAL COMPILATION TIME   0:00:50
TOTAL ELAPSED TIME       0:01:27
```

INDICE ALFABETICO