Universidad Nacional Autónoma de México

FACULTAD DE CIENCIAS

Un depurador para lenguaje C en la PDP/11

T E S I S

Que para obtener el título de

A C T U A R I O

p r e s e n t a

CECILIA PEREZ COLIN

México, D. F. 1988

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

I N D I C E

INTRODUCCION

C es un lenguaje de programación de empleo general; aunque se desarrolló junto con el sistema operativo UNIX, posteriormente se ha implementado en una amplia variedad de minis y microcomputadoras.

Se le considera como "un lenguaje de programación de sistemas", dado que C ha demostrado su utilidad en el diseño de sistemas operativos. Por otro lado se ha utilizado en cualquier tipo de programas de manera exitosa.

C es un lenguaje de tipo estructurado, tiene características poderosas que le dan una gran flexibilidad.

Como otros lenguajes estructurados, C soporta tipos de datos complejos, hace un amplio uso de los apuntadores y tiene un abundante conjunto de operadores para cálculo y manejo de datos.

Debido a la flexibilidad de su sintaxis, la gente que empieza a programar en C tiene muchos errores que podrían considerarse como lógicos, pues no aparecen en la compilación como errores de sintaxis, algunos de los cuales son:

- el empleo de puntos y comas,
- el empleo de llaves,
- el empleo de operadores.

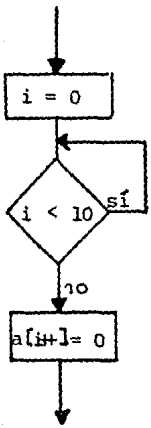
Un ejemplo del primer tipo de error es el siguiente:

Las proposiciones

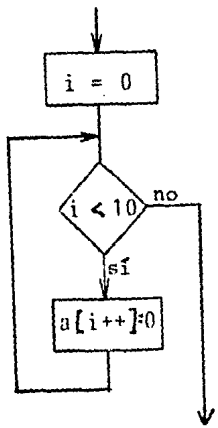
```
i = 0;                          i = 0;
while (i < 10);      y          while (i < 10)
a [ i + + ] = 0;                a [ i + + ] = 0;
```

aunque pareciera que hacen lo mismo, son muy distintas.

La primera queda representada por el siguiente diagrama de flujo:



mientras la segunda se vería así:

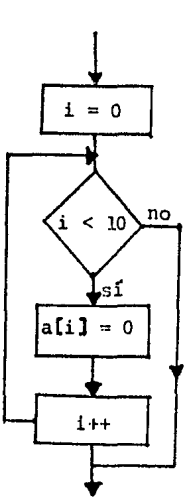


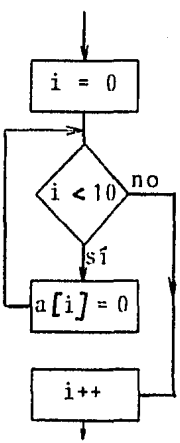
En los diagramas se ve que en el primer caso se haría un ciclo infinito, mientras que en el segundo caso, al incrementarse la variable i, en algún momento se saldría del ciclo.

Un ejemplo del segundo tipo de error, que también es muy común, es el usar o dejar de usar llaves, por ejemplo:

```
i = 0;                          i = 0;
while (i < 10).                 while (i < 10)
{  a [ i ] = 0;        y        a [ i ] = 0;
     i + + ;                      i + + ;
}
```

que quedan representados por los siguientes diagramas de flujo





En el primer caso las llaves encierran una lista de proposiciones o un bloque que se realizará siempre y cuando sea cierta la condición del while; en el segundo caso sólo -- una proposición es la que se ejecuta si es cierta la condición y lleva a un ciclo infinito.

El tercer tipo común de error es sobretodo el uso del operador de asignación " = " y el operador de igualdad " == ".

Ejemplo:

```
if (c = '\n')          y          if (c = = '\n')
    + + n l;                              + + n l;
```

en la primera proposición a la variable c se le asignará - siempre el valor numérico de '\n' (retorno de carro) el -- cual es siempre distinto de cero y por lo tanto ejecuta -- siempre el incremento de nl, mientras que en la segunda -- proposición se compara el valor numérico de la variable c y el '\n' dando cero si son iguales y distinta de cero si son diferentes.

Otro error común es rebasar los límites de los arreglos, - lo cual puede hacer que se interrumpa el programa.

Por estas razones fue que se decidió hacer un depurador para lenguaje C, enfocado a gente que comienza a programar - en este lenguaje.

En el caso concreto de la Facultad de Ciencias, la computadora PDP/11 venía siendo la más utilizada por los alumnos para cursos como los de lenguajes de programación y progra

mación de sistemas, esto no quiere decir que el depurador - fue pensado exclusivamente para este sistema, sino por el - contrario, la idea es que con pocas modificaciones se pueda utilizar y transportarse a otros sistemas.

El depurador es fácil de usar y ahorra mucho tiempo al programador, puesto que se mostrará instrucción por instruc- - ción el valor de las variables, lo que ayudará a visualizar rápidamente los errores lógicos.

CAPITULO I

TECNICAS DE PROGRAMACION

1. Ciclo de vida de un programa

La depuración viene a ser una parte importante de lo que llamaremos el ciclo de vida de un programa.

El ciclo de vida comprende todo lo relacionado con la elaboración y uso del programa.

Para tener un mejor control del desarrollo de un proyecto de programación, es conveniente poseer un método general para llevar a cabo dicho proyecto. Se ha convenido en una serie de pasos comunes en los que se divide el ciclo de vida.

Algunos teóricos dividen el ciclo en más pasos que otros; sin embargo, las ideas inherentes a todos ellos son las mismas.

La siguiente división se basa en lo que aparece en (Y) :

1) Definición del problema.
2) Disponibilidad de recursos.
3) Selección del algoritmo y las estructuras de datos.
4) Selección del lenguaje (o lenguajes) de programación.
5) Planteamiento lógico y estructura del programa.
6) Codificación.
7) Implementación, prueba y depuración.

8) Documentación.
9) Mantenimiento.

A continuación se dará una cierta explicación de los pasos anteriores basándonos en (Y) .

1) Definición del problema

Esto es, evidentemente, el primer paso; aún cuando parece muy obvio, puede ocurrir que el programador no tenga claro el problema en cuestión. Esto se puede deber a diversas causas:

a. Poco conocimiento del tema.
b. Surgimiento del problema a partir de una persona distinta al programador.
c. Falta de comunicación, etc.

De preferencia debería de especificarse el problema por escrito, incluyendo el tipo de datos, el formato de salida, lo que se entiende por una solución o entrada razonable, y otros elementos. Esto será importante en distintas etapas del proceso de programación.

2) Disponibilidad de recursos

En esta etapa hay que ver cuales son nuestros recursos, posibilidades, restricciones, tiempo con el que se cuenta, la posibilidad o necesidad de cambiar el sistema a otro tipo de máquinas, el software relacionado con el problema.

3) Selección del algoritmo y estructura de datos

Una vez que el problema ha sido identificado, hay que determinar el método o métodos a utilizar para resolverlo, además esto debe depender de los recursos y restricciones que tenemos.

A veces el problema es ya conocido y, con base en la experiencia, se pueden escoger algoritmos y estructuras de datos adecuados.

Se pueden buscar los algoritmos en libros, revistas, manuales, artículos de investigación, etc. Por supuesto, puede haber ciertos procesos dentro de un programa que puede llevarse a cabo por medio de algoritmos modificados adecuadamente.

En esta etapa hay que escoger las estructuras de datos que sean convenientes. También puede consultarse la literatura relacionada con el tema. Desde el punto de vista de la eficiencia y para facilitar la depuración, se debe escoger la estructura de datos con el fin de reducir la complejidad del programa.

Dependiendo del problema se escogerá primero el algoritmo o la estructura de datos; por ejemplo, en un problema de los promedios de los alumnos de un grupo, se ocurre inmediatamente que los datos vendrán en un arreglo o estructura y con base en esto se diseña el algoritmo.

Por otro lado hay casos en los que en la misma definición del problema se especifica el algoritmo, como en el caso de un programa que integre una función por la regla de Simpson. Aquí ya se tiene el algoritmo y después hay que pensar en qué estructura de datos se utilizará.

Citando a Wirth (W) "no se pueden tomar decisiones sobre - la estructura de datos de un programa sin conocer los algo-- ritmos que van a aplicarse a los mismos y, recíprocamente, - la estructura y la elección de algoritmos a menudo dependen, de manera importante, de la estructura de los datos en que - se apoyen. En resumen, los temas de composición de programas y estructura de datos están inseparablemente ligados".

Es claro que estos dos elementos determinarán de manera im-- portante toda la estructura del programa.

4) Selección del lenguaje (o lenguajes) de programación

En la actualidad hay varios lenguajes diseñados para diver-- sos fines; además del lenguaje ensamblador o lenguaje de má- quina, existen otros muchos de los cuales citaremos algunos con sus orientaciones, algunas ventajas y a veces desventa-- jas. Lo siguiente está basado en (V) y (P).

COBOL: Cuyas aplicaciones principales están en los negocios. Hay una gran eficiencia en el manejo de archivos. Es muy cla ro y sirve para manejar grandes cantidades de datos. Ejemplo: nóminas. Entre sus desventajas se encuentra el no ser conci- so.

FORTRAN: Se aplica sobretodo en el área científico-técnica; se pueden definir subprogramas y rutinas externas al progra- ma principal. Sin embargo carece del soporte directo de cons trucciones estructuradas, tiene una pobre tipificación de da tos, no maneja fácilmente cadenas de caracteres y tiene otras deficiencias.

SNOBOL: Sus funciones predefinidas representan concisa y fácilmente manipulaciones de cadenas de caracteres como puede ser obtener la raíz gramatical de una palabra.

LISP: En su área de aplicación están las expresiones simbólicas y el manejo de listas. Es utilizado casi exclusivamente en inteligencia artifical. Usa como principal elemento de control de ejecución la recursividad.

ALGOL: Es de uso general. Fue el primer lenguaje estructurado, es muy claro y tiene recursividad. El usuario puede definir sus propios tipos de datos, es decir, existe tipificación y se pueden declarar estructuras de datos complejos.

APL: Un lenguaje extremadamente conciso y poderoso, diseñado para manejar matrices y vectores. Tiene funciones elementales entre arreglos, pero necesita de un teclado especial, lo lo cual limita mucho su uso.

BASIC: Es un lenguaje interactivo que fue diseñado originalmente para enseñar a programar en poco tiempo. Ahora es el lenguaje de base en las microcomputadoras. Hay muchas versiones de este lenguaje que se considera de bajo nivel, por lo cual no pueden decirse fácilmente sus ventajas y desventajas.

PASCAL: Es un lenguaje de programación estructurada ideal para la enseñanza; es también de uso general aunque se utiliza sobre todo en el área científico-técnica y en programación de sistemas. Es descendiente de ALGOL por lo que tiene las mismas ventajas como lo es la estructura por bloques, fuerte tipificación de datos, recursividad, etc.

LOGO: Surge con la idea de enseñar a los niños el proceso de programación, es muy fácil de usar.

C: Su área de aplicación es primordialmente la de sistemas - operativos. Es un lenguaje estructurado que permite definir tipos de datos complejos, hace extensivo el uso de operado-- res.

A pesar de tantos lenguajes existentes, a veces ocurre que - ninguno parece tener la capacidad necesaria en un proyecto - dado o simplemente no contamos con él; en estos casos se ve lógico escoger el lenguaje más cercanamente apropiado y ex-- tenderlo; una técnica para hacer esto es escribiendo rutinas o funciones que ejecuten las tareas necesarias.

5) Planteamiento lógico y estructura del programa

Esta fase es la más importante dentro del ciclo de vida, - - puesto que de éste dependen la facilidad de codificación y - de la depuración; además de la misma claridad del programa - final.

Es por esto que en las siguientes secciones de este capítulo se discutirán algunas de las técnicas de programación más -- utilizadas con sus ventajas y desventajas, entre las cuales están:

- programación estructurada.
- programación modular.
- programación de arriba-abajo.
- programación de abajo-arriba.

Independientemente de la técnica de programación, un método común de expresión de la lógica y estructura de un programa es utilizando diagramas de flujo. Este método ayuda a visualizar mejor nuestro programa, sin embargo tiene la desventaja de que puede que el código que se genere no esté estructurado.

6) Codificación

En esta parte del proceso de programación hay que elegir adecuadamente los nombres de las distintas subrutinas, variables, etc. Siempre hay que tener en cuenta que el programa puede estar a disposición de más gente además del programador.

Cuando en el lenguaje usado se permitan nombres comunes para las subrutinas, es conveniente ponerles nombres de acuerdo con la función que realizan. Aún cuando esto no sea posible, se debe indicar de alguna manera su jerarquía dentro del programa o la rutina en el cual se encuentra contenida la subrutina.

Otra parte importante es poner nombres a las variables de acuerdo con su significado dentro del programa. Siempre que sea posible se usarán los nombres que se usaron al hacer el algoritmo y los diagramas de flujo. El hecho de poner nombres adecuados a las variables puede redundar en una mayor claridad de programa.

El formato del programa fuente debe también dejar ver la jerarquía; la identación es aquí un elemento importante.

Si el lenguaje permite utilizar renglones en blanco o comentarios, esto puede ser de mucha utilidad. Los comentarios no deben ser simples traducciones de los enunciados, sino que deben usarse para separar bloques lógicos del programa; además deben colocarse al principio o final de éstos. El programa debe documentarse por sí solo.

7) Implementación, prueba y depuración

La implementación es el proceso de editar, compilar y correr el programa fuente en la computadora. La técnica de implementación, prueba y depuración más usada es la de abajo-arriba: primero se codifican y prueban los módulos del nivel más bajo, puesto que no necesitan de los otros componentes del sistema. Luego se implementan los módulos que llaman a estos módulos, etc., hasta llegar al programa principal. Una ventaja de esto es que son relativamente pocos los enunciados que habría que poner al probar el sistema.

Este método es bueno cuando se entiende muy bien el problema por resolver; en caso contrario, pueden ocurrir cosas como que el programador no vería como operaría el programa hasta probar el programa principal; es decir, hasta probar todo el sistema. En ese momento, sería muy difícil y caro hacer cambios. En el caso en que no esté muy claro el problema por resolver, se puede usar una técnica de implementación, prueba y depuración arriba-abajo. Así, el programador vería el desempeño del programa principal haciendo los cambios necesarios. Sin embargo, para probar un módulo se necesitarían los módulos llamados por el primero. Esto se maneja mediante falsas rutinas que simularían los módulos no escritos y regresarían valores de prueba. Al utilizar esta técnica, siempre hay que recordar que se supone válida la correctez del programa, ésta no se demuestra sino hasta que sustituye la última rutina falsa.

La prueba es el proceso de ejecutar un programa intentando encontrar la mayor parte de los errores que tenga éste. El objetivo de esta etapa es ir descubriendo sistemáticamente diferentes clases de errores.

Si varios errores requieren modificar el diseño y éstos son encontrados con regularidad, entonces la calidad del programa así como su seguridad está en duda; de aquí se indica -- pruebas adicionales.

Por otro lado, si tanto las funciones como expresiones parecen trabajar correctamente y existen pocos errores que son fáciles de corregir, existen dos posibilidades o (1) la calidad y seguridad del programa es aceptada, o (2) las pruebas son inadecuadas para descubrir errores.

Si en esta fase parece no haber ninguna clase de error, el usuario se encargará de descubrirlos y en la fase de mantenimiento se corregirán.

Una buena prueba empieza por un buen conjunto de datos de - prueba, éstos deben escogerse de manera tal que se trate de probar con ellos todos los caminos de un programa.

De la depuración, dada la importancia que tiene para esta - tesis, se hablará en el siguiente capítulo. Así como de algunas técnicas con sus ventajas y desventajas.

8) Documentación

Casi toda la documentación ya debe estar preparada en este momento, pues esto se hace a la par con los pasos anteriores; sólo hay que ponerla explícitamente.

La documentación debe incluir:

1) enunciado del problema,
2) descripción del algoritmo y estructura de datos,
3) descripción de la lógica y estructura del programa incluyendo diagramas de flujo,
4) un listado del programa limpio, comentado y de ser posible con secuencia numerada,
5) una lista de datos de prueba junto con las secciones probadas.

Además puede hacerse un resumen de todo lo anterior, extrayendo los elementos de interés del usuario.

La documentación debe probarse, por ejemplo, dándolas a otra persona para ver si la entiende.

9) Mantenimiento

En un programa ya revisado y depurado pueden aparecer errores mucho tiempo después, hasta que salgan cierto tipo de circunstancias antes no dadas. Esto hace que el programador tenga que corregir estos errores. Si no se dispone de una buena documentación éste puede ser muy difícil. La existencia y corrección del error también debe entrar en la documentación.

El mantenimiento incluye también el adaptar un programa a cambios de ambiente en el que se encuentra, por ejemplo, nuevo sistema operativo, nuevos equipos periféricos o cualquier elemento del sistema que frecuentemente son degradados o modificados.

También se puede perfeccionar el programa en esta fase, por ejemplo, que tenga nuevas capacidades, modificar algunas -- funciones, etc.

Con esta fase termina el ciclo de vida de un programa. En - las secciones que siguen se describirán algunas técnicas de programación basado en (P) y en (BR).

2. Programación estructurada

Esta técnica la formaliza Edsger Dijkstra.

La programación estructurada propone un número limitado de construcciones lógicas que tienden a minimizar la complejidad del flujo de un programa.

El principio de la programación estructurada se basa en - - tres construcciones lógicas que incluye un programa estructurado; éstas son:

1) Secuencia.- La ejecución de una tarea seguida inmediatamente por otra tarea, es decir, se implementan los pasos del proceso esenciales para la especificación de un algoritmo.

2) Condición.- (if-then-else) provee la facilidad de seleccionar un proceso basándose en una ocurrencia lógica; -- "realiza esta tarea" cuando la condición es verdadera, o la alternativa "entonces realiza esta otra" cuando la -- condición es falsa.

3) Repetición.- Una tarea es ejecutada repetitivamente hasta que una condición predefinida se dé. Provee de iteraciones o "ciclos" a la programación.

Entre sus ventajas está sobre todo la claridad en la que se expresa el programa, por lo que es fácil de probar y mantener.

Una de sus desventajas es el hecho de que utilizar únicamente construcciones estructuradas puede a veces introducir -- complicaciones en el flujo lógico. Ejemplo:

Dado un cierto ciclo con una cierta condición, a veces es - necesario salir de éste con una condición diferente a la dada; si usáramos la idea de programación estructurada se incrementa la posibilidad de error, atentando también a la seguridad del programa.

Una situación así podría arreglarse rediseñando el procedimiento de tal manera que la condición no se requiera exactamente dentro del ciclo. Ejemplo: cuando se usa el método de la bisección para encontrar raíces de funciones, o bien, en el caso de un programa de búsqueda binaria.

3. Programación modular

La programación modular consiste en dividir un sistema o un programa en módulos activos. Hay un módulo principal que podría ser único y módulos comunes que reemplazan líneas de código que se repiten.

Muchos módulos son subrutinas que requieren de accesar datos globales y datos que fueron calculados en invocaciones anteriores.

La idea de modularización es simple, pero no así su implementación, la mayor dificultad consiste en determinar el conjunto de subtareas para cada tarea.

Para esto hay que tomar en cuenta que para que una parte del programa sea un buen candidato a ser un módulo, esto deberá repetirse en distintos lugares del programa.

Dentro de la estructura de un programa, un módulo puede categorizarse como:

a) Un módulo secuencial que se ejecuta y se referencíe sin ninguna interrupción en el programa. Por ejemplo, subrutinas, llamadas a función, procedimientos, etc.

b) Un módulo que se interrumpe antes de completarse y subsecuentemente se restablece en el punto de interrupción. A este tipo de módulos se les llama "corutinas", mantienen un apuntador que permite restablecerse en el punto de interrupción.

c) Módulos paralelos o corutinas, que requieren de dos o más unidades de procesamiento trabajando en paralelo. Las corutinas se ejecutan simultáneamente.

4. Programación de arriba-abajo y de abajo arriba

En realidad, estas técnicas se pueden usar tanto para el diseño del programa como para su implementación; este último aspecto es poco conocido y lo mencionamos en la parte que habla de la implementación, prueba y depuración.

En el diseño de arriba-abajo, se diseña en primer lugar lo que podríamos llamar la parte fundamental del sistema; y después en orden de jerarquías las partes secundarias. Esta técnica recibe también el nombre de programación por refinamiento sucesivos.

Para el diseño del depurador, consideramos como parte fundamental el centro del control del sistema; es decir, el programa principal.

Si una persona conoce el desarrollo de arriba-abajo de un sistema, puede ver rápidamente cuáles son las interacciones dentro de éste y así pueden hacerse cambios de manera relativamente clara y rápida. Otro método con la misma intención es el de valoración iterativa, que básicamente consiste en probar subconjuntos cada vez más grandes del sistema.

Este método puede servir para que el usuario vaya definiendo con calma sus requerimientos sin presionar indebidamente al programador.

En el diseño de abajo-arriba generalmente se diseñan primero las rutinas de los niveles bajos. Por ejemplo, si se requiere de ciertas propiedades o resultados en una rutina de un nivel alto, se diseñarían antes las rutinas con estas propiedades o que dieran estos resultados. Este proceso continuaría hasta lograr que todo el sistema tenga ciertas propiedades o resultados.

Es claro que este tipo de diseño puede tener ciertas limitaciones en un programa que no sea de tiempo real, puesto que al empezar desde abajo el programa del nivel más alto puede no encajar y resultar un sistema que no se puede implementar.

En el caso de los programas de tiempo real, la parte de nivel más abajo es el hardware, otro nivel puede accesar las páginas del hardware y controlar la memoria (con lo cual se pueden implementar niveles más altos sin preocuparse por controlar la memoria), etc.

En ciertos casos sí es práctico el diseño de abajo-arriba, como en el caso de que una parte de un sistema se parezca a otra desarrollando con anterioridad y se conozca ya su estructura.

Una forma conveniente de programar sería la de combinar estas dos técnicas: se puede usar un diseño de arriba-abajo hasta llegar a ciertos componentes pequeños; después se puede seguir abajo-arriba para terminar.

CAPITULO II

TECNICAS DE DEPURACION

La depuración es el acto de buscar y remover errores lógicos en un sistema de computación.

La depuración se considera un verdadero problema puesto que - para encontrar errores se requiere de mucho esfuerzo y tiem-- po. En una sola sesión de depuración es muy posible que no se encuentren todos los errores de un programa.

En general, existen dos fuentes principales que causan todo - el esfuerzo que se lleva depurar un programa:

a) Un mal diseño de programación y

b) Una mala técnica de depuración.

Esto se debe a que aparecen muchos errores por un mal diseño (en el capítulo I se dieron algunas técnicas de programación que pueden ayudar a salvar este problema) pero con una mala - técnica de depuración puede ocurrir que se detecten muy pocos errores.

La depuración se lleva a cabo mediante corridas de prueba, -- donde la ejecución del programa se realiza seleccionando cui- dadosamente la entrada o datos de prueba.

Independientemente de la técnica de depuración que se escoja, hay dos formas de llevarla a cabo, según Lauesen (L). La primera se llama de abajo-arriba y la otra de arriba-abajo.

Como se vio en el capítulo I, estas técnicas aparecen dentro de las técnicas de programación más utilizadas. El sentido de éstas es el mismo en cuanto a depuración se refiere, pues en la depuración de abajo arriba se separa un programa por módulos y se prueban éstos separadamente, para después juntarlos y probar el programa como en su forma final. En la depuración de arriba abajo, el programa se prueba como un todo, es decir, como en su forma final.

La diferencia básica entre estas dos formas de depuración es que, en la forma de arriba abajo, todos los datos de prueba deben tener la forma de datos reales, mientras que en la forma de abajo arriba esto sólo sucede cuando se juntan los módulos.

Un programa de la forma:

```
Program A;
Begin
  .
  .
  .
     B;
  .
  .
  .
End;
```

```
Program B;
Begin
  .
  .
  .
     C;
  .
  .
  .
End;

Program C;
Begin
  .
  .
  .
End;
```

sería depurado mediante la técnica de abajo arriba como sigue:

```
Depura (C);
Depura (B);
Depura (A);
```

y mediante la técnica de arriba abajo sería:

```
Depura (A);
```

entendiéndose que dentro del proceso de depuración de A también se depurarían B y C.

Algunos teóricos, como Yohe, Baker y el mismo Lauesen aseguran que la depuración arriba abajo es mejor, mientras que Brinch, Hansen y Naur se inclinan por la otra técnica.

En seguida discutiremos algunos métodos de depuración, con -- sus ventajas y desventajas.

1. Seguimiento con lápiz y papel

Este método es tal vez el más natural cuando uno desea saber los errores lógicos de un programa. Consiste en construir datos de prueba, e ir siguiendo éstos a lo largo del programa, en un escritorio. Por supuesto, hay que saber escoger los datos según el tipo de errores que podrían ocurrir; sería ocioso usar datos con los que, de manera natural, funcionaría el programa. También hay que ver si con los datos el programa -- funciona de acuerdo con lo que se pretende.

Esta corrida de prueba a mano es casi obligatoria, aunque se cuente con una técnica de depuración más avanzada.

Entre sus ventajas están:

a) Es una técnica que se hace en escritorio y no en computadora, ahorrando tiempo de máquina.

b) Ayuda a repasar y revisar el programa.

c) Suele suceder que los errores más comunes que se cometen son los más fáciles de corregir: cambiar una variable por otra, poner un fin de instrucción en una proposición equivocada, etc. Con este método es fácil darse cuenta de es-- tos errores pues se tiene el listado completo y por lo tanto, una visión completa del programa.

Su principal desventaja consiste en que si nos encontramos ante un programa muy grande y con muchísimos datos, será casi - imposible seguir éstos en el programa; fácilmente uno se perdería y habría que dedicarle mucho tiempo para encontrar los errores.

2. Vaciado de memoria.

El vaciado se produce después de que un programa falla y muestra el contenido de toda o de partes de la memoria; ésta es - tal vez la única ventaja que tiene.

Esta técnica es ineficiente por varias razones:

a) El error ocurre, en la mayoría de los casos, antes de que el programa se descontinúe, además de que queda muy escondido.

b) Encontrar un error es difícil, pues es tanta la información vertida y disponible en un vaciado de memoria, que el programador se lleva mucho tiempo en estudiarlo.

c) Si se trabaja en un lenguaje de alto nivel, la diferencia entre los resultados con dicho lenguaje y los del vaciado (que trabaja a nivel de bits) es muy grande.

Hay casos en que el vaciado de memoria es la única manera de encontrar un error, por ejemplo, cuando los errores se producen por fallas en alguna parte de la máquina (en el "hard- -- ware").

3. Mensajes.

El método de depuración más utilizado es el de imprimir variables, claves o enunciados que nos ayuden, por ejemplo, a ver si los valores de ciertas variables son lógicos en nuestro -- programa, si una condición se cumplió o la ejecución "llegó" a un cierto enunciado de nuestro programa. Los principiantes tienden a insertar muchos enunciados de impresión para asegurarse de la más esencial. Si el programa es estructurado, se pueden poner pocos enunciados en lugares estratégicos, aunque el programa sea muy largo.

Por ejemplo, si tenemos una llamada a función, bastará con saber si lo que nos devolvió como resultado la función es lo que queríamos realmente, y no habría necesidad de poner enunciados de impresión en todo el cuerpo de la función.

En programas muy largos conviene insertar los enunciados de impresión al principio de los módulos para saber si el programa sigue la secuencia que deseamos.

Se tienen las siguientes ventajas:

a) El programador ve únicamente resultados significativos de la corrida de prueba en forma fácil y accesible.

b) El error habrá ocurrido poco antes del resultado incorrecto, lo que hace fácil aislar el error.

c) El planteamiento de depuración, es decir, donde vayamos a colocar nuestros enunciados de prueba y el aislamiento del error se hace en un escritorio y no en la computadora.

d) Se puede dejar una versión del programa con nuestros enunciados de prueba.

Para que esta técnica sea completamente benéfica sería bueno que el planteamiento de depuración se haga dentro del diseño del programa y no después.

En el sistema de depuración propuesto en esta tesis se utiliza esta última técnica automáticamente, pues el usuario puede seleccionar la parte del programa que desee depurar.

Otra forma muy importante de encontrar todos los posible errores es saber escoger los datos, es decir, éstos deben revelar todos los posibles errores y deben estar disponibles al programador. Es bien conocido que al probar un programa se puede mostrar únicamente la presencia de errores y no su ausencia. Si el programa requiere de muchos y diferentes datos parecerá imposible construir datos de prueba que nos revelen todos los errores.

CAPITULO III

LA RUTINA DEL ANALIZADOR LEXICO

El depurador de esta tesis tiene como dato un programa; un problema fundamental es el de reconocer los distintos elementos que conforman a dicho programa dato: cuáles son las instrucciones, las variables, etc. En otras palabras, es necesario hacer lo que se llama un análisis léxico.

Un analizador léxico ("scaner") es una subrutina llamada repetitivamente por sistemas operativos, tales como un compilador o un depurador. Básicamente su función es ir combinando caracteres para formar unidades reconocibles llamadas elementos (tokens) como son las variables, las palabras reservadas, los operadores, etc.; los examina para ver si son de interés para el programa que lo llama, y si es así, regresa el elemento con un código asociado a éste.

Ejemplos de elementos que podrían no ser de interés a algún programa serían los comentarios o los espacios en blanco.

Recordemos que en el analizador léxico de la tesis se quiere analizar líneas de un programa escrito en lenguaje C en la PDP/11 y donde no habrá posibilidad de tener errores de sintaxis ya que el depurador trabajará sobre un programa ya compilado.

Un analizador léxico es parte fundamental de un sistema operativo del tipo de los mencionados, puesto que sólo a partir del análisis léxico se pueden reconocer las instrucciones de cualquier programa.

En este capítulo se trata de explicar la lógica seguida en una rutina de analizador léxico basada en el artículo de John M. Deudorek (DGM)

El diseño del analizador léxico se hizo mediante cuatro pasos:

Paso 1: Precisar los elementos (tokens) que debe reconocer el analizador léxico.

Paso 2: Se hace una representación de los elementos mediante un diagrama de estado de las clases de caracteres que conforman cada elemento.

Paso 3: Se usa este diagrama de estado para formar dos tablas, una llamada "la tabla del próximo estado" y la otra "la tabla de salida" que se utilizan para determinar si un elemento está completo.

Paso 4: Se escribe la rutina del analizador léxico ayudándose por estas tablas con algunas auxiliares.

Detallaremos cada paso con un ejemplo pequeño y esbozaremos lo que ocurre en el analizador léxico de la tesis.

Paso 1. Los elementos

C es un lenguaje bien definido en el sentido de que su sintaxis tiene reglas muy claras. Por ejemplo, para construir una variable, los caracteres permitidos para esto son alfanuméricos y el caracter "_" (guión bajo); además tiene que comenzar con un alfabético.

Otra condición no precisamente del lenguaje C, pero sí de la PDP/11 es que no se permiten números diferentes a enteros.

En un ejemplo pequeño, los elementos podrían ser:

1) identificadores
2) números
3) comentarios
4) blancos
5) operadores

Todo elemento que no esté en esta lista se considera "blanco" y se ignora.

Paso 2. Diagrama de estado

Los elementos se conforman de la siguiente manera: se procede a analizar una línea caracter por caracter, partiendo de un estado llamado "estado inicial". Se pueden presentar varias posibilidades acerca del caracter inicial, así como de los subsecuentes. Estas posibilidades se agrupan en una tabla de clases de caracteres que es un autómata finito determinado que en el analizador léxico del depurador se obtiene mediante la subrutina TRAE.

En el ejemplo, la tabla de clases de caracteres sería:

1) Alfabéticos: A,B,...,Z,a,b,...,z
2) Dígitos: 1,2,...,9,0.
3) Operadores: +,-
4) Asterisco: *
5) Diagonal: /
6) Blanco:

A partir de estas posibilidades se va creando el diagrama de estado mediante las siguientes especificaciones:

a) El estado inicial en el estado 1.

b) Las aristas que unen los estados tienen encima una o varias clases de caracteres. Si la arista no tiene una clase encima, ello representa una arista con todas las clases de caracteres que no aparezcan en otra arista que emane de ese estado.

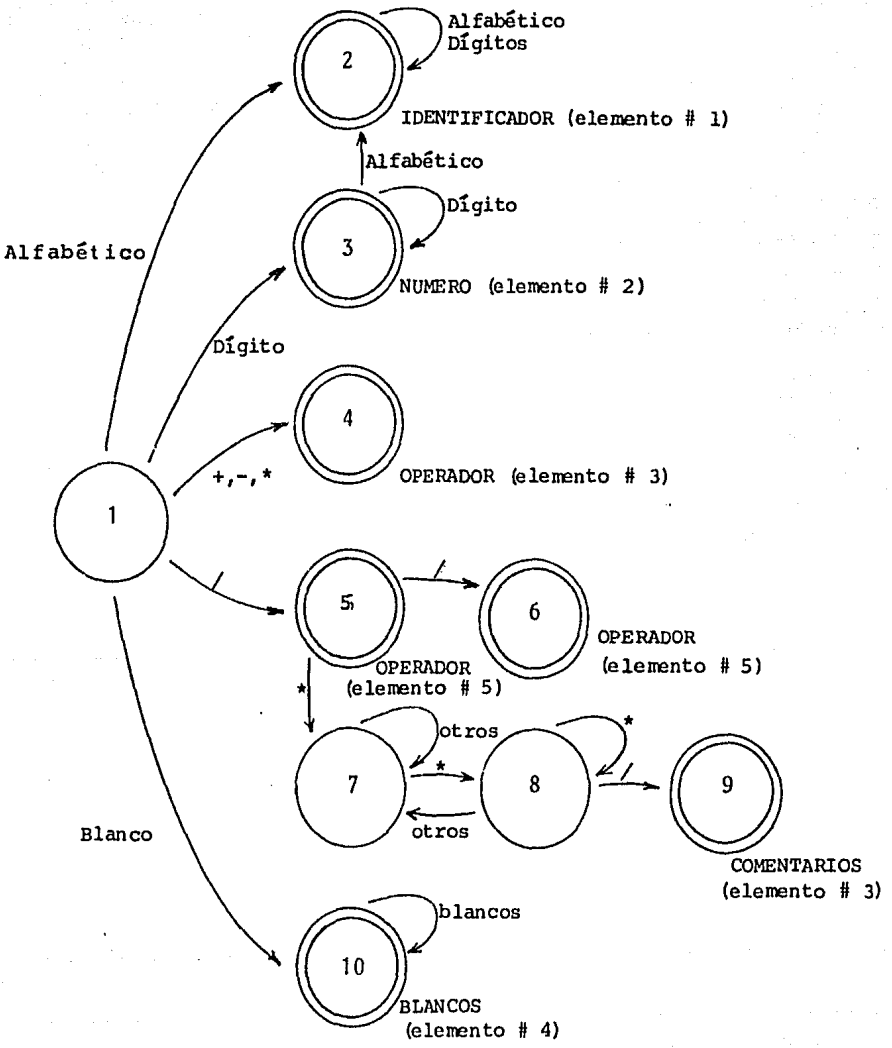
c) Para que el diagrama quede bien definido, no debe haber dos aristas con la misma clase de caracteres que prevenga de un mismo estado y vaya a dos estados distintos.

d) Las flechas indican el flujo. Si uno se halla en el estado n y hay una arista con la clase c uniendo este estado con el estado m, se pasa del estado n al m si y solo si, el caracter siguiente pertenece a la clase c.

e) Algunos estados, señalados mediante dobles círculos, son los estados finales. Estos aparecen cuando en ellos es posible que se complete un elemento de cierto tipo, señalado en el diagrama mediante el símbolo #. Una cadena de caracteres es un elemento de cierto tipo si y solo si existe un camino desde el estado inicial hasta un estado final del tipo deseado.

f) Una vez que se tiene un estado final y que se ha reconocido el tipo del elemento se regresa al estado inicial

Diagrama de Estado



Paso 3. Tablas de próximo estado y salida

A partir del diagrama de estado se conforman dos tablas para codificar la información.

La primera NEXT-STATE, especifica el estado al cual se debe pasar dados dos parámetros, el estado actual y la clase del siguiente caracter de la línea.

La segunda tabla OUTPUT, especifica si el elemento en cuestión está completo (con una entrada nula o no). El número de renglones de las tablas es igual al número de estados, mientras que el número de columnas es igual al número de clases de caracteres.

Esta matriz se forma siguiendo las reglas:

a) Si una arista con la clase de caracter c lleva del estado n al estado m, se pone m en el renglón n columna c en la tabla de NEXT-STATE; las entradas correspondientes en la tabla de OUTPUT se ponen en cero para indicar que las aristas en cuestión tienen un caracter que hay que incluir en el elemento que se está formando, es decir, la arista no apuntó a un estado final.

b) Hay que observar que se considera un programa lógico (no tiene errores de sintaxis) en el sentido que no puede haber cadenas de caracteres que no estén permitidos. Por ello en la tabla de NEXT-STATE no hay una columna de error como lo indica el artículo.

c) En los demás casos, puesto que se considera que un elemento está ya completo, en NEXT-STATE se regresa al estado inicial 1 y en la tabla de OUTPUT se pone el número de elemento correspondiente (código del elemento)

Tablas del estado próximo

estado \ clase	Alfabético	Dígito	Operador	Asterisco	Diagonal	Blanco
1	2	3	4	4	5	10
2	2	2	4	4	5	10
3	2	3	4	4	5	10
4	2	3	4	4	5	10
5	2	3	4	7	6	10
6	2	3	3	3	5	10
7	7	7	7	8	7	7
8	7	7	7	8	9	7
9	2	3	4	4	5	10
10	2	3	4	4	5	10

Tabla de salida

estado \ clase	Alfabético	Dígito	Operador	Asterisco	Diagonal	Blanco
1	0	0	0	0	0	0
2	0	0	1	1	1	1
3	0	0	2	2	2	2
4	5	5	5	5	5	5
5	5	5	5	0	0	5
6	5	5	5	5	5	5
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	3	3	3	3	3	3
10	4	4	4	4	4	0

Paso 4. La rutina del analizador léxico

Ahora sí se procederá a escribir la rutina Scaner ayudándonos de las tablas NEXT-STATE y OUTPUT.

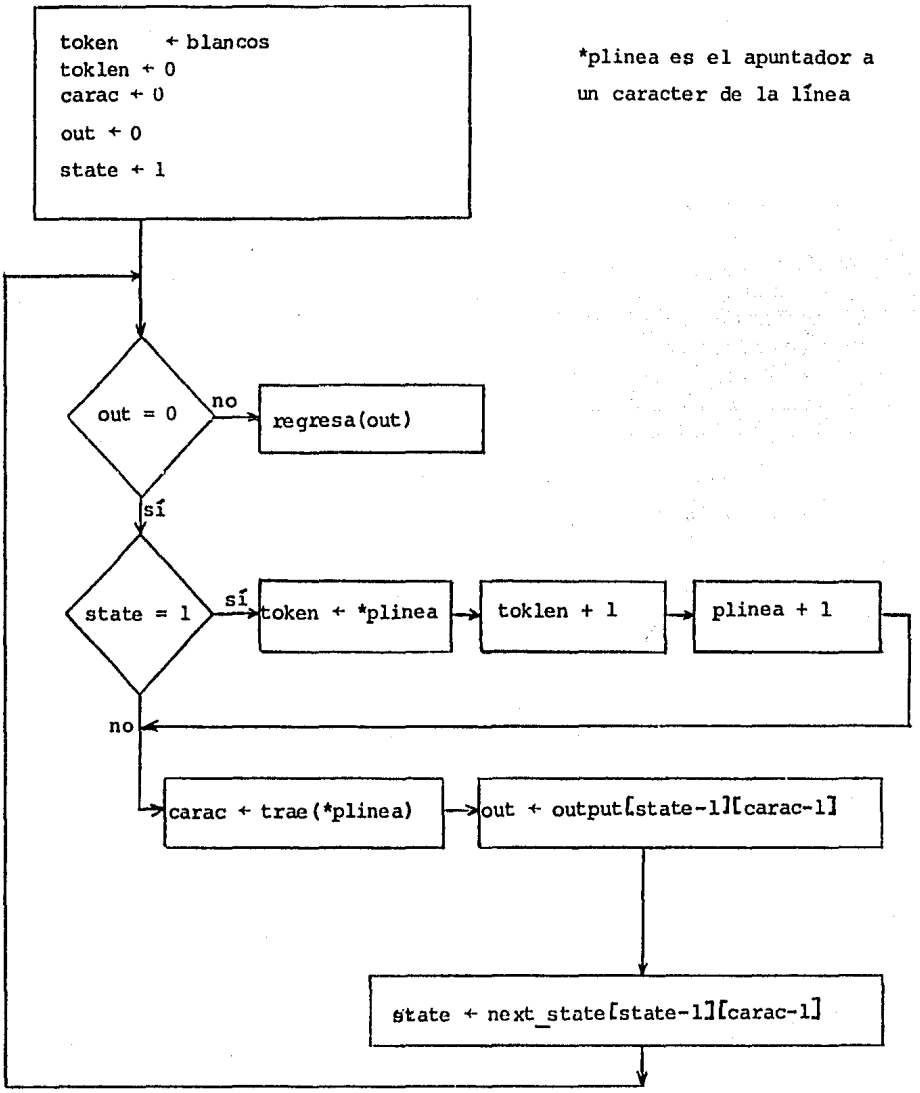
La rutina Scaner tiene el siguiente propósito:

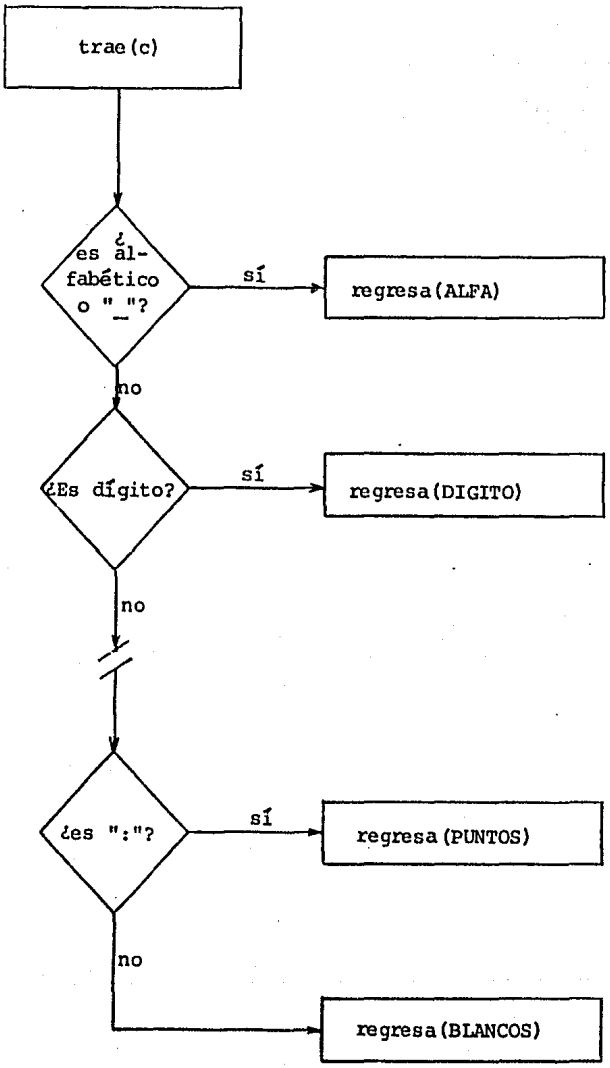
Dada una línea van analizando los elementos de ella obteniendo la siguiente información: el elemento dado como una cadena de caracteres; su tipo, dado mediante un número llamado - TOKCODE y su longitud TOKLEN.

En el programa SCANER.C se incluyó la rutina identifica que nos dice si un identificador es palabra reservada, si de ésta es una declaración; y si es una variable.

En el depurador SCANER e IDENTIFICA son dos rutinas independientes.

DIAGRAMA DE FLUJO DE LA RUTINA SCANER




trae(c)
¿es alfabético o "_"?
sí
regresa(ALFA)
no
¿Es dígito?
sí
regresa(DIGITO)
no
¿es ":"?
sí
regresa(PUNTOS)
no
regresa(BLANCOS)

En el analizador léxico que se utiliza para el depurador de la tesis, los elementos que se reconocen son:

1.- Identificadores (variables o palabras reservadas)
2.- Números enteros
3.- Operadores aritméticos
4.- Operadores unitarios
5.- Operadores de asignación
6.- Comentarios
7.- Operadores de relación
8.- Operadores de igualdad
9.- Operadores lógicos
10.- Operadores de condición
11.- Blancos
12.- Cadenas de caracteres
13.- Constantes de caracter
14.- Fin de archivo
15.- Secuencia de escape
16.- Operadores entre bits
17.- Llamada a preprocesador
18.- Llamada a función
19.- Llamada a arreglo
20.- Llamada a estructura
21.- Fin de función
22.- Fin de arreglo
23.- Comienzo de rutina
24.- Fin de instrucción

TABLA DE CLASES DE CARACTERES

1.- Alfabéticos y "_": A,B,C,..,Z, a,b,..,z y "_"
2.- Dígito: 0,1,...,9
3.- + (suma)
4.- - (resta)
5.- * (asterisco)
6.- % (módulo)
7.- / (división)
8.- > (mayor)
9.- < (menor)
10.- = (igual)
11.- ! (signo de admiración)
12.- & (amper)
13.- | (línea vertical)
14.- ^ (apuntador)
15.- ? (signo de interrogación)
16.- (blanco)
17.- # (gato)
18.- { (llave que abre)
19.- } (llave que cierra)
20.- ((paréntesis redondo que abre)
21.-) (paréntesis redondo que cierra)
22.- [(paréntesis cuadrado o corchete que abre)
23.-] (paréntesis cuadrado o corchete que cierra)
24.- \ (diagonal invertida)
25.- " (comillas)
26.- ' (apóstrofe)
27.- EOF (lo reconocerá como -1)
28.- . (punto)
29.- : (dos puntos)
30.- ; (punto y coma)

CAPITULO IV

ESTRUCTURA GENERAL DEL DEPURADOR

El depurador propuesto en esta tesis introduce una serie de mensajes que indican al usuario el valor de las variables de su programa, para que con esta información pueda ver si el programa corre de acuerdo con sus requerimientos. La técnica usada es la de insertar en el código del programa instrucciones para que, al correr éste, se impriman los mensajes acerca del valor de las variables. Así, la depuración se hace en programas ya compilados (es decir que, en particular, no tienen errores de sintaxis) y los resultados de dicha depuración, se ven al correr el programa del cual se obtendrá.

En realidad, el depurador produce, a partir del programa dato (o programa por depurar) un nuevo programa (el cual contiene las instrucciones de impresión de mensajes) que se compila y corre sin que el usuario intervenga.

A continuación daremos unos ejemplos de cómo vería el usuario los valores de las variables al correr el programa final.

En el caso de la asignación

```
i = j + k + 5;
```

se imprimen los valores de j y k antes de entrar a la asignación y el valor de i después de la asignación.

Para mayor comprensión del usuario, él vería el resultado como sigue:

```
/* j = = 10 */
/* k = = 12 */
i = j + k + 5 /
/* i = = 27 */
```

Para las asignaciones

```
i + +;    y    + + i,
```

si i llega con un valor de 8, los resultados son:

```
i + +;              /* i = = 8 */
                    y
/* i = = 9 */      + + i;
```

En varios casos de instrucciones con condición, se imprimen las variables dentro de la condición. Veamos un IF:

```
IF (i < 0)
   j = k + + ;
```

el resultado sería:

```
/*  i = = 3 */
IF (i < 0)
   /* k = = 5 */
   j = k + + ;
   /* j = = 5 */
```

En las instrucciones con condición, a veces será necesario - que se introduzcan corchetes para así incluir los mensajes - de impresión de comentarios. En el siguiente ejemplo

```
WHILE (i > 0)
   j = - - i;
```

hay que imprimir el renglón del WHILE cada vez que se entre al ciclo.

La corrida del programa final sería:

```
/* i = = 2 */
WHILE (i > 0)
   /* i = = 2 */
   j = - - i;
   /* j = = 1 */
WHILE (i > 0)
   /* i = = 1 */
   j = - - i;
   /* j = = 0 */
```

Para imprimir todos estos mensajes, hubo de insertar en el - programa del usuario varias instrucciones dentro del WHILE.

La impresión de variables es más directa en otros casos, como las llamadas a función o el RETURN:

```
RETURN (i);          o          FUNCION (j);
```

aparecen así:

```
/* i = = 3 */                    /* j = = 2 */
RETURN (i);                      FUNCION (j);
```

En el capítulo V aparecen más ejemplos, un poco más detallados.

En lo que se refiere a la implementación del depurador, éste analiza línea por línea el programa, haciendo uso de la rutina del analizador léxico, la cual se explicó con detalle en el capítulo anterior.

Mediante el analizador léxico se clasifica el primer elemento de la línea dada; dependiendo de este elemento, el programa realiza distintas subrutinas:

Por ejemplo:

- Si es una cadena de blancos, se cuenta su longitud para que se use como margen y alinear los mensajes que se imprimen en el proceso de depuración. Después de esto, el segundo elemento se considera como el primero.

- Si es una llamada a preprocesador (#), el programa se va a la rutina correspondiente (las rutinas se analizarán por separado más adelante).

- Otro sería cuando es un comentario, se imprime la línea tal cual, con la restricción de que no se pongan instrucciones después de un comentario en la misma línea.

- Si es un comienzo de rutina ({), se imprime tal cual. - Si sabemos que está dentro de la rutina WHILE, FOR o IF, indicamos que no va a ser necesario imprimir un cierre de rutina.

- Cuando hay un cierre de rutina (}) se imprime la línea tal cual, aunque si es el cierre de rutina de una declaración de estructura, tomaremos el identificador siguiente para guardarlo en la tabla de identificadores (guarda-tabla) con el número 22 que indica estructuras.

- En el caso de un operador unitario, se indica (mediante la palabra "asign") que estamos en un renglón cuya instrucción es una asignación, para posteriormente pasar a la rutina de asignación.

- El caso faltante es el principal: cuando el primer elemento es un identificador. Aquí hay que saber si el renglón anterior fue un comienzo de ciclo o rutina (do, while o for), para entonces englobar las instrucciones del programa-dato, junto con las generadas por el depurador, en un solo bloque.

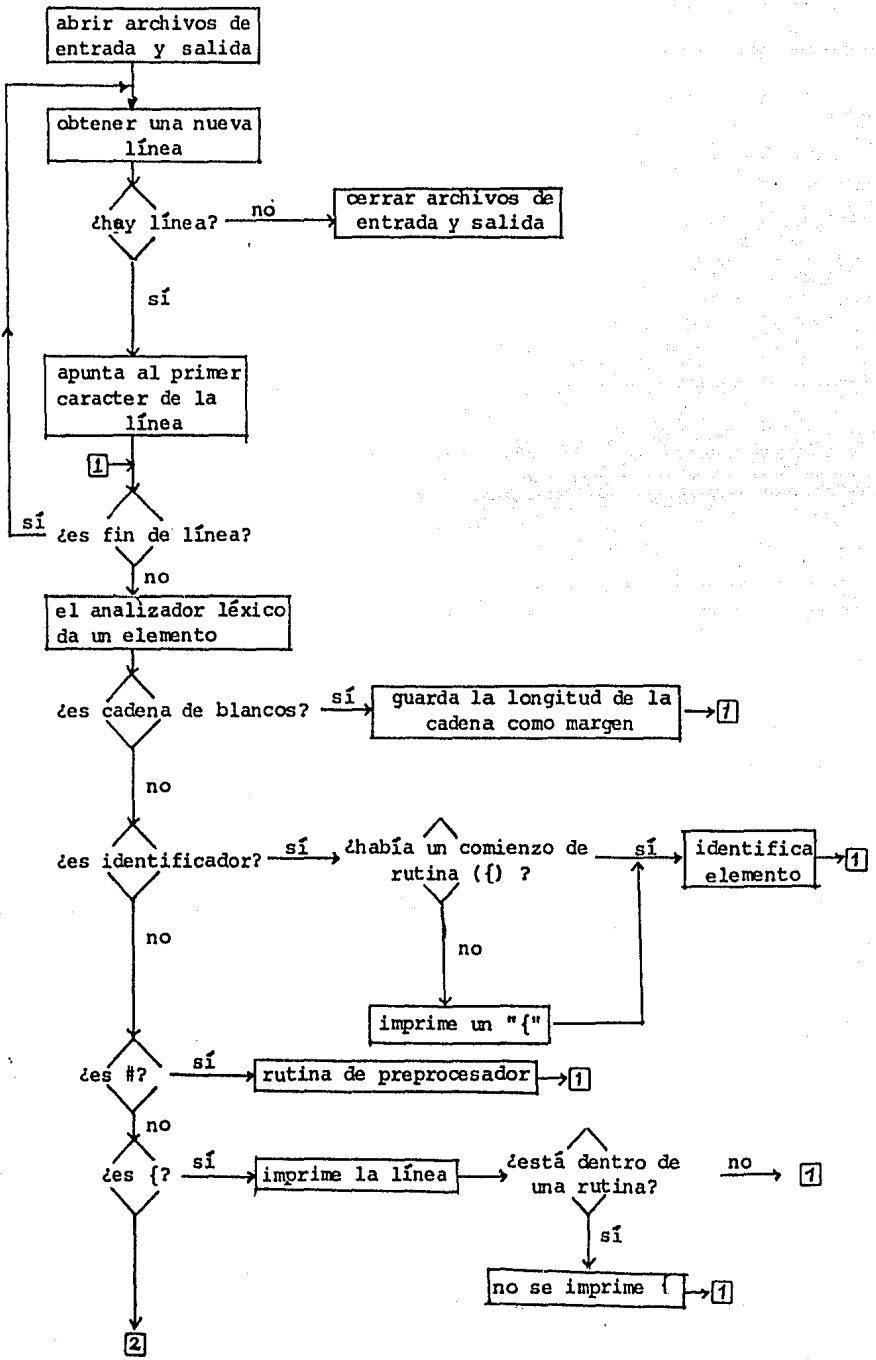
Después de esto se pasa a clasificar el identificador, por medio de la rutina identifica, en alguno de estos tipos:

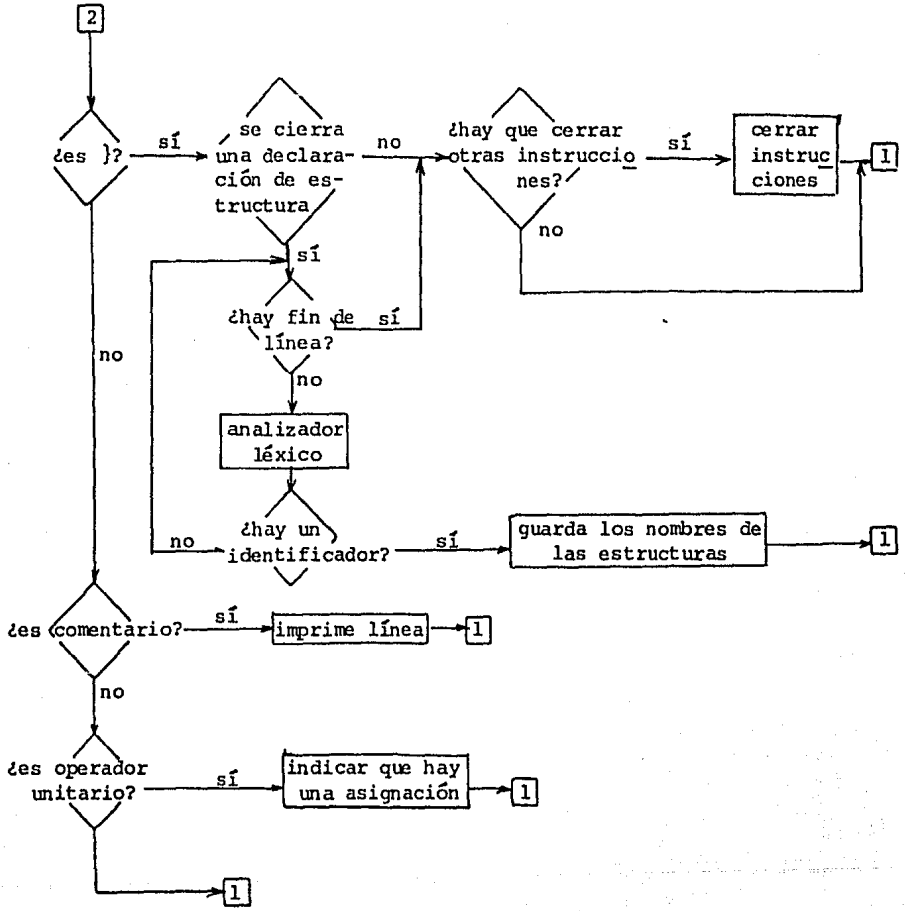
- Llamada a función
- Variable
- Constante
- Palabra reservada

La rutina identifica es la que, por medio de una búsqueda binaria, nos dice si una cadena de caracteres es palabra reservada, variable o llamada a función. Esta rutina se analizará más adelante.

En la figura aparece el diagrama de flujo del programa principal.

DIAGRAMA DEL PROGRAMA PRINCIPAL




2
¿es }?
sí
se cierra una declaración de estructura
no
¿hay que cerrar otras instrucciones?
sí
cerrar instrucciones
1
no
sí
¿hay fin de línea?
sí
no
no
analizador léxico
¿hay un identificador?
no
sí
guarda los nombres de las estructuras
1
¿es comentario?
sí
imprime línea
1
no
¿es operador unitario?
sí
indicar que hay una asignación
1
1

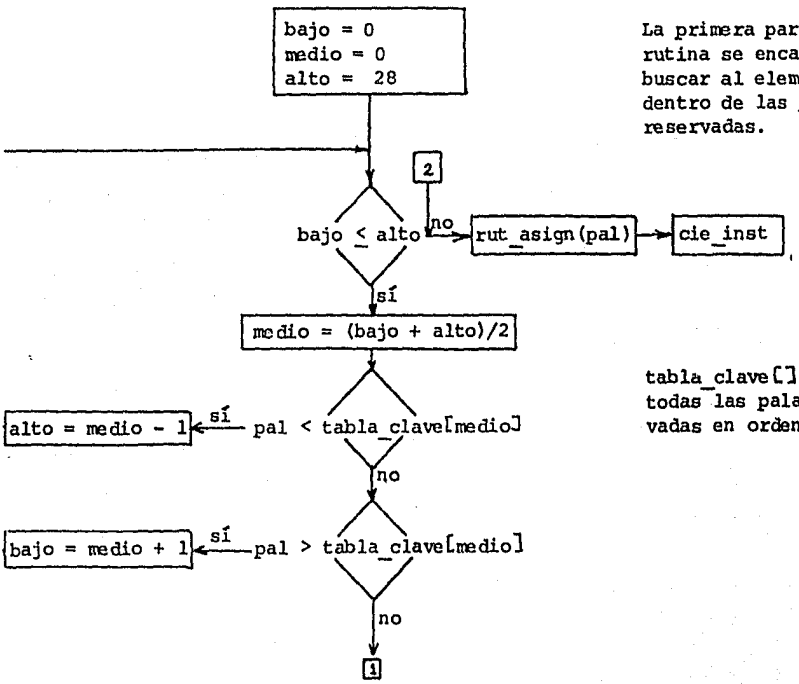
Algunas rutinas particulares.

1. Rutina de identificación de palabras (identifica)

Esta rutina tiene como parámetro un identificador, el cual - se busca en la tabla de palabras reservadas (tabla-clave). - Si el identificador es una palabra reservada del lenguaje -- "C", de los tipos SWITCH, WHILE, FOR, DO, IF, RETURN, INT, - CHAR, STRUCT, CASE, ELSE, entonces el programa se va a la rutina correspondiente. Si es una palabra reservada que no sea de las anteriores, simplemente se pasa la línea tal cual al programa final.

En caso de que el identificador no se encuentre en la tabla de palabras reservadas, el programa se va a la rutina de - - asignación.

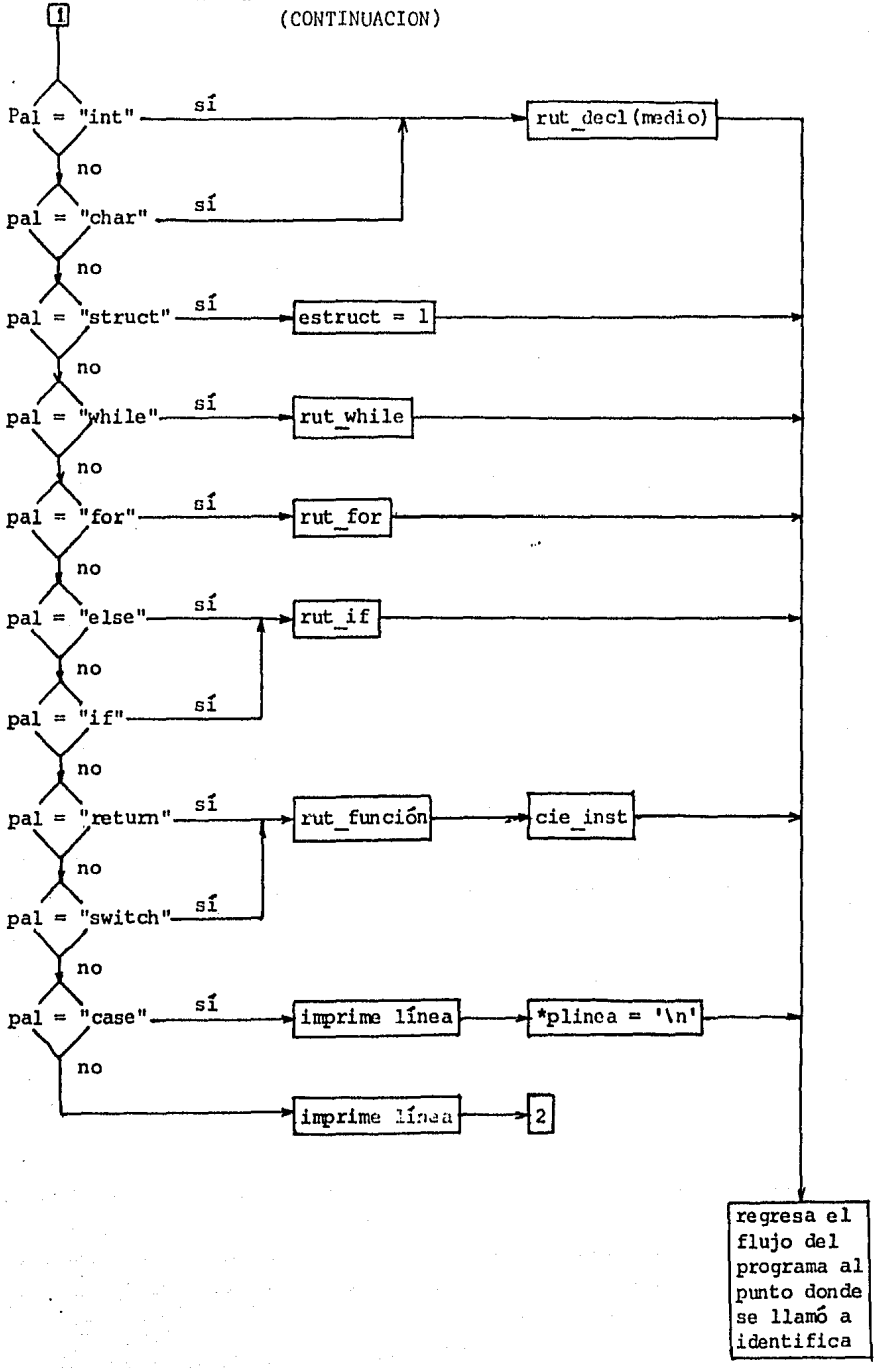
A continuación se muestra el diagrama de flujo de tal rutina:



La primera parte de la rutina se encarga de buscar al elemento "pal" dentro de las palabras reservadas.

tabla_clave[] contiene a todas las palabras reservadas en orden alfabético.

DIAGRAMA DE FLUJO DE LA RUTINA IDENTIFICA
(CONTINUACION)



2. Rutinas de declaración de constantes y variables (rut-decl y rut-prep).

Por medio de estas rutinas y de la rutina auxiliar guarda-tabla se van guardando las variables enteras y las de caracter, así como las constantes; éstas irán acompañadas por un número que indica de qué tipo son.

Los posibles tipos son: variables enteras (15), variables de caracter (3) y constantes (2).

En el caso de la declaración de arreglos, los nombres de éstos se guardan como variables, cuyo tipo aumenta dependiendo de la dimensión del arreglo. Así, por ejemplo, el tipo 16 es para arreglos unidimensionales de enteros, 5 para arreglos - bidimensionales de caracteres, etc. Para evitar confusiones, se pide aquí que el usuario no declare arreglos de caracteres cuya dimensión sea mayor que 12.

En el caso de las estructuras, el nombre de éstas se guarda en la tabla junto con el número 22, aunque las variables dentro del cuerpo de la estructura se declaran por medio de los números ya mencionados.

3. Rutinas de impresión de mensajes (pri-var y pri-stmt)

Con estas rutinas se imprimen los mensajes que ve el usuario después de depurar su programa.

pri-var imprime en forma de comentario el valor de la variable que se desee. Para poder imprimir la variable, se hace - uso de la rutina "buscatipo" para saber si es entera, caracter, arreglo o estructura.

Si es constante, no se imprime el valor. Por ejemplo, si se desea imprimir la variable entera "i" cuyo valor es 10, buscatipo regresa el número 15, correspondiente a una variable entera, con el cual ya se puede imprimir el siguiente mensaje:

```
/* i = = 10 */
```

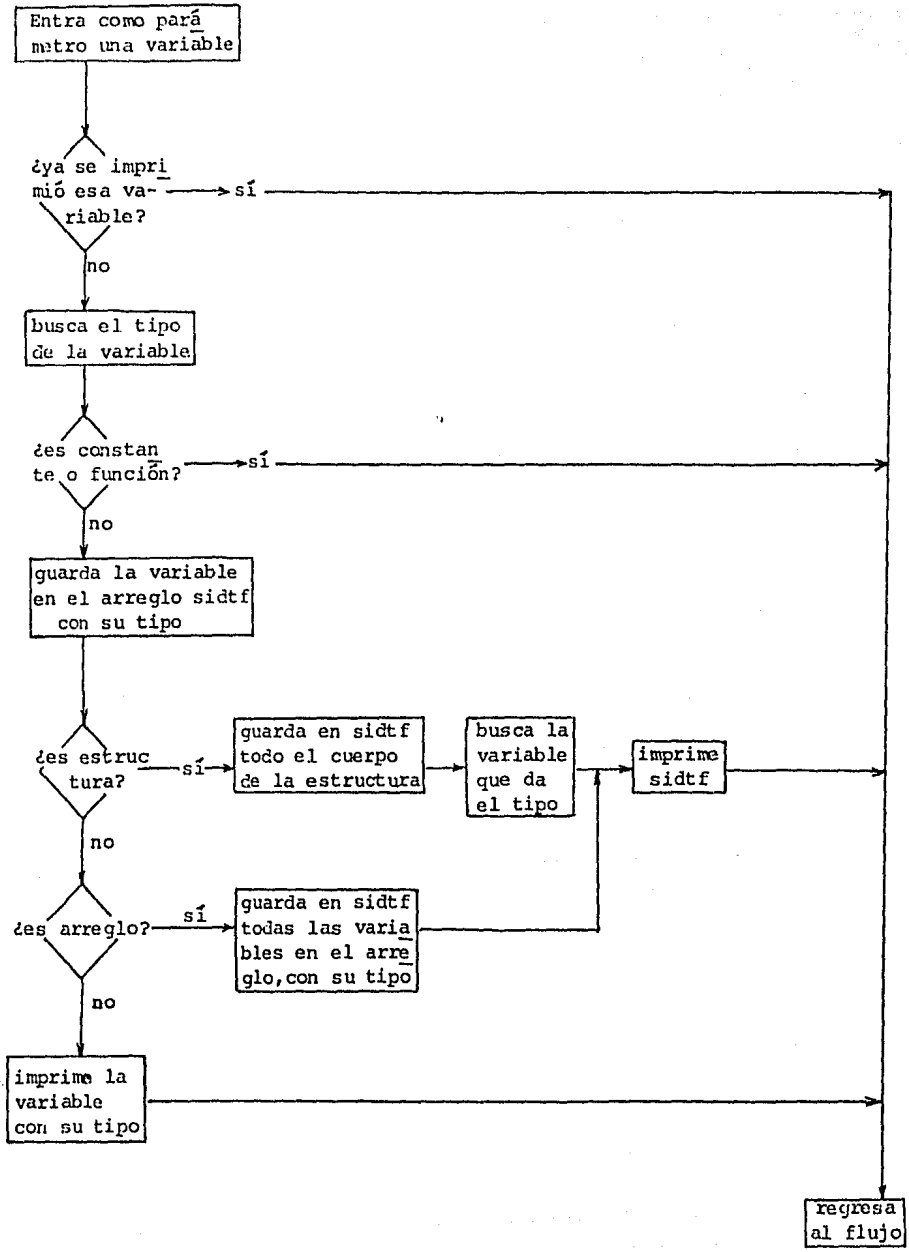
Para que el usuario sepa que instrucción se está realizando, las instrucciones se imprimen como mensaje por medio de pristmt (print statement).

Para seguir con nuestro ejemplo, la instrucción

```
i + +;
```

la vería el usuario por medio de los mensajes

```
i + +;
/* i = = 10 */
```


Entra como parámetro una variable
¿ya se imprimió esa variable?
sí
no
busca el tipo de la variable
¿es constante o función?
sí
no
guarda la variable en el arreglo sidtf con su tipo
¿es estructura?
sí
guarda en sidtf todo el cuerpo de la estructura
busca la variable que da el tipo
imprime sidtf
no
¿es arreglo?
sí
guarda en sidtf todas las variables en el arreglo, con su tipo
no
imprime la variable con su tipo
regresa al flujo

4. Rutina de evaluación (rut-evalua)

Esta rutina imprime todas las variables o arreglos de variables que se encuentran dentro de cierta expresión. rut-evalua es llamada tanto en casos de asignaciones (para evaluar el - lado derecho de una asignación) como en las rutinas para las instrucciones for, if, while, return y también para las llamadas a función. Veamos algunos ejemplos de mensajes impresos por medio de esta rutina y las rutinas rut-asign, rut-if, rut-for, rut-while y rut-do.

Para la asignación i = j + k;

rut-evalua imprime los mensajes que indican los valores de j y k.

Para la expresión

```
if (j! = 0)
```

rut-evalua imprime el mensaje que indica el valor de j.

Para la expresión

```
for (i = 0; i < n; i + +)
```

rut-evalua imprime únicamente el mensaje que indica los valores de i y n, es decir, los valores de las variables que aparecen en la condición.

5. Rutina de asignación (rut-asign)

Esta rutina analiza un renglón en donde hay una instrucción del tipo asignación; esto ocurre cuando la primera palabra -

no es una palabra reservada. Primero se determina si es una variable o una llamada a función; en este último caso, sólo se manda a imprimir el renglón. En el caso en que sea una variable, su valor se manda a imprimir después de la asignación, excepto en el caso del tipo ++i, en donde se imprime antes de hacer la instrucción. Aquí, el término "variable" incluye a arreglos y estructuras.

Para una asignación de la forma

```
i = j + k;
```

la rutina de asignación (ayudándose de rut-evalua) manda imprimir primero todas las variables del lado derecho, después se hace la asignación y a continuación se manda imprimir el lado izquierdo, entendiéndose éste como la colección de símbolos que se encuentran antes del signo igual (=); esto es debido a que del lado izquierdo puede haber expresiones más o menos complicadas como a[i] , a.i, etc.

En la siguiente página se muestra el diagrama de flujo correspondiente a esta rutina.

6. Rutina de la proposición while (rut-while)

Llama a la rutina de evaluación para que imprima las variables de la condición del while.

7. Rutina de la proposición if (rut-if).

Evalúa las variables de la condición del if y además manda - imprimir un mensaje en caso de que la condición sea verdadera.

8. Rutina de función (rut-función)

Manda a imprimir los valores de los parámetros de la función.

9. Rutina de la proposición for (rut-for)

Manda a imprimir la inicialización del ciclo for; después, - con rut-evalua imprime las variables de la condición.

Existe también una subrutina que cierra cuerpos de instruc-- ción, que se llama cie-inst(n), donde n es el parámetro que dice cuántas llaves (}) hay que imprimir.

CAPITULO V

EJEMPLOS

El siguiente ejemplo usa asignaciones y la instrucción for:

```
#define MPOT 5
main()
{
  int x, p, i;
  x = 3;
  p = 1;
  for(i = 1; i <= MPOT; i++)
  {
    p = p * x;
    printf("la potencia de 3 a la %d es %d\n",i,p);
  }
}
```

El programa ya depurado se vería así:

```
  x = 3;
  /* x == 3 3  */

  p = 1;
  /* p == 1 1  */

  /* i = 1; */
  for(i = 1; i <= MPOT; i++)

    /* p == 1 1  */
    /* x == 3 3  */
    p = p * x;
    /* p == 3 3  */

la potencia de 3 a la 1 es 3
  for(i = 1; i <= MPOT; i++)

    /* p == 3 3  */
    /* x == 3 3  */
    p = p * x;
    /* p == 9 9  */

la potencia de 3 a la 2 es 9
  for(i = 1; i <= MPOT; i++)

    /* p == 9 9  */
    /* x == 3 3  */
    p = p * x;
    /* p == 27 27  */
```

```
la potencia de 3 a la 3 es 27
  for(i = 1; i <= MPOT; i++)

    /* p == 27 1b  */
    /* x == 3 3  */
    p = p * x;
    /* p == 81 51  */

la potencia de 3 a la 4 es 81
  for(i = 1; i <= MPOT; i++)

    /* p == 81 51  */
    /* x == 3 3  */
    p = p * x;
    /* p == 243 f3  */

la potencia de 3 a la 5 es 243
```

El siguiente ejemplo además de la rutina for, hace uso de la rutina if y de los arreglos.

```
#define MAX 5
int v[7] = { 8, 5, 2, 0, 6, 3, 9 };
main()  /* busca x en el arreglo v */

{
  int i, x, y;

  x = 6;
  for(i = 0; i <= MAX; i++)
  {
    if(v[i] == x)
      y = i;
  }
  printf("el numero %d esta en el lugar %d del arreglo v\n",x,i);
}
```

En este caso el depurador insertó las llaves "{" y "}" para formar un bloque de instrucciones en la rutina if. El programa depurado queda de la siguiente manera:

```
  /* x  */

  /* i = 0; */
  for(i = 0; i<= MAX; i++)

    /* i == 0 0  */
    /* v[i] == 9 9  */
    /* x == 6 6  */
    if(v[i] == x)

  for(i = 0; i<= MAX; i++)

    /* i == 1 1  */
    /* v[i] == 3 3  */
    /* x == 6 6  */
    if(v[i] == x)

  for(i = 0; i<= MAX; i++)

    /* i == 2 2  */
    /* v[i] == 2 2  */
    /* x == 6 6  */
    if(v[i] == x)

  for(i = 0; i<= MAX; i++)

    /* i == 3 3  */
    /* v[i] == 0 0  */
    /* x == 6 6  */
    if(v[i] == x)

  for(i = 0; i<= MAX; i++)

    /* i == 4 4  */
    /* v[i] == 6 6  */
    /* x == 6 6  */
    if(v[i] == x)

sobra e if
      /* i == 4 4  */
      x = 2;
      /* x == 4 4  */

  for(i = 0; i<= MAX; i++)

    /* i == 5 5  */
    /* v[i] == 3 3  */
    /* x == 6 6  */
    if(v[i] == x)

el numero 6 esta en el lugar 6 del arreglo v
```

En el siguiente ejemplo se usan los apuntadores:

```
main() /* intercambio de *px y *py */
{
  int *px, *py, x, y, temp;

  scanf("%d %d",&x,&y);
  px = &x;
  py = &y;
  temp = *px;
  *px = *py;
  *py = temp;
  printf("%d %d\n",x,y);
}
```

El programa depurado de este ejemplo es:

```
. . .. . . . . .
TT7>
4  8
  /* *px == 4 4  */
  temp = *px;
  /* temp == 4 4  */

  /* *py == 8 8  */
  *px = *py;
  /* *px == 8 8  */

  /* temp == 4 4  */
  *py = temp;
  /* *py == 4 4  */

8 4

>
```

En el siguiente ejemplo se hace uso de estructuras y arreglos:

```
struct fecha
{
  int dia;
  int mes;
} a;
int dias[13] ={ 0,31,29,31,30,31,30,31,31,30,31,30,31 };
main()
{
  int i, dia;
  a.dia = 8;
  a.mes = 4;
  dia = a.dia;
  for(i = 1; i < a.mes; i++)
    dia += dias[i];
  printf("el dia %d corresponde al %d dia del %d mes\n",dia,a.dia,a.mes
}
```

El programa depurado es:

```
TT?>
  a.dia = 8;
  /* a.dia == 8 8  */

  a.mes = 4;
  /* a.mes == 4 4  */

  /* a.dia == 9 8  */
  dia = a.dia;
  /* dia == 8 8  */

  /* i = 1; */
  for(i = 1; i < a.mes; i++)

    /* i == 1 1  */
    /* dias[i] == 31 1f  */
    dia += dias[i];
    /* dia == 39 27  */

  for(i = 1; i < a.mes; i++)

    /* i == 2 2  */
    /* dias[i] == 29 1d  */
    dia += dias[i];
    /* dia == 68 44  */

  for(i = 1; i < a.mes; i++)

    /* i == 3 3  */
    /* dias[i] == 31 1f  */
    dia += dias[i];
    /* dia == 99 63  */

el dia 99 corresponde al 8 dia del 4 mes
```

CAPITULO VI

COMENTARIOS FINALES

Aquí se señalan algunas de las principales ventajas y limitaciones del sistema desarrollado en esta tesis.

Evidentemente, puesto que el sistema es un depurador, es necesario que éste ahorre tiempo al usuario en la búsqueda de errores. Generalmente, si un programa se interrumpe, el error se encuentra poco antes de la interrupción y puede ser fácilmente localizable.

El usuario tiene un fácil acceso al sistema, simplemente con una llamada a preprocesador (# debug). Se puede hacer una mejora de forma tal que se depuren solamente las rutinas que se deseen.

Otra característica del depurador es que, con pocas modificaciones, es posible trasladarlo a otra máquina.

Por ejemplo, la PDP/11 trabaja sólo con números enteros, por lo que la tabla de estados (ver capítulo III) únicamente considera este caso. En (Y) se indica cómo se podría ampliar esto para admitir números decimales con notación decimal.

El depurador se desarrolló pensando en un primer nivel; es decir, para ayudar a encontrar errores a gente que comienza a programar con el lenguaje C. Esto quiere decir que el depurador cubre algunas de las posibles construcciones en C; los

datos van desde constantes y variables hasta arreglos (de variables enteras o de caracter) de cualquier dimensión y estructuras sencillas; todas las instrucciones se pueden manejar, siempre que dentro del cuerpo de la instrucción no haya otras instrucciones con cuerpos muy complicados.

BIBLIOGRAFIA

(BE) A Guided Tour of Program Design Methodologies
G.D. Bergland (Bell Telephone Laboratories)
Computer (octubre 1981). Pags 13-36

(BR) Programming and Documenting Software Projects
P. J. Brown
Computing Surveys. Vol. 6, No. 4 (diciembre 1974)
Pags. 213-220

(DDH) Structured Programming
O.J. Dahl, E. W. Dykstra, C.A.R. Hoare
Academic Press London and New York (1972)

(DGM) Scanner Design
John M. Dedourek, Uday G. Gujar y Marion E. McIntyre
Software-Practice and Exprience. Vol. 10. (1980)
Pags 959-972

(DMS) A Survey of Debuggers
Frank Drake Jr., Arthur McCaffrey y John Sadowsky
Revista Byte (noviembre 1985. Pags. 177-184

(E) A High-Level Debugger for PL/1, Fortran and Basic
Brig Elliot
Software-Practice and Experience. Vol. 12 (1982)
Pags. 331-340

(VN) Van Nostrand's Scientific Encyclopedia
Fifth edition, edited by Douglas M. Considine
Van Nostrand Reinhold Co. (1976)

(W) Program Development by Stepwise Refinement
Niklaus Wirth
Association for Computing Machinery Inc. (1971)
Pags. 321-334

(Y) An Overview of Programming Practices
J.M. Yohe
Computing Surveys. Vol. 6, No. 4.(diciembre 1974)
Pags. 221-245